
hyperledger-fabricdocs Documentation

Release master

hyperledger

Apr 14, 2022

Contents

1	Introduction	3
2	What's new in Hyperledger Fabric v2.x	9
3	Release notes	15
4	Key Concepts	17
5	Getting Started	117
6	Developing Applications	133
7	Tutorials	193
8	Deploying a production network	375
9	Operations Guides	417
10	Upgrading to the latest release	497
11	Commands Reference	517
12	Architecture Reference	587
13	Frequently Asked Questions	613
14	Contributions Welcome!	617
15	Glossary	643
16	Releases	653
17	Still Have Questions?	655
18	Status	657

Note: Please make sure you are looking at the documentation that matches the version of the software you are using. See the version label at the top of the navigation panel on the left. You can change it using selector at the bottom of that navigation panel.



Enterprise grade permissioned distributed ledger platform that offers modularity and versatility for a broad set of industry use cases.

CHAPTER 1

Introduction

In general terms, a blockchain is an immutable transaction ledger, maintained within a distributed network of *peer nodes*. These nodes each maintain a copy of the ledger by applying transactions that have been validated by a *consensus protocol*, grouped into blocks that include a hash that bind each block to the preceding block.

The first and most widely recognized application of blockchain is the [Bitcoin](#) cryptocurrency, though others have followed in its footsteps. Ethereum, an alternative cryptocurrency, took a different approach, integrating many of the same characteristics as Bitcoin but adding *smart contracts* to create a platform for distributed applications. Bitcoin and Ethereum fall into a class of blockchain that we would classify as *public permissionless* blockchain technology. Basically, these are public networks, open to anyone, where participants interact anonymously.

As the popularity of Bitcoin, Ethereum and a few other derivative technologies grew, interest in applying the underlying technology of the blockchain, distributed ledger and distributed application platform to more innovative *enterprise* use cases also grew. However, many enterprise use cases require performance characteristics that the permissionless blockchain technologies are unable (presently) to deliver. In addition, in many use cases, the identity of the participants is a hard requirement, such as in the case of financial transactions where Know-Your-Customer (KYC) and Anti-Money Laundering (AML) regulations must be followed.

For enterprise use, we need to consider the following requirements:

- Participants must be identified/identifiable
- Networks need to be *permissioned*
- High transaction throughput performance
- Low latency of transaction confirmation
- Privacy and confidentiality of transactions and data pertaining to business transactions

While many early blockchain platforms are currently being *adapted* for enterprise use, Hyperledger Fabric has been *designed* for enterprise use from the outset. The following sections describe how Hyperledger Fabric (Fabric) differentiates itself from other blockchain platforms and describes some of the motivation for its architectural decisions.

1.1 Hyperledger Fabric

Hyperledger Fabric is an open source enterprise-grade permissioned distributed ledger technology (DLT) platform, designed for use in enterprise contexts, that delivers some key differentiating capabilities over other popular distributed ledger or blockchain platforms.

One key point of differentiation is that Hyperledger was established under the Linux Foundation, which itself has a long and very successful history of nurturing open source projects under **open governance** that grow strong sustaining communities and thriving ecosystems. Hyperledger is governed by a diverse technical steering committee, and the Hyperledger Fabric project by a diverse set of maintainers from multiple organizations. It has a development community that has grown to over 35 organizations and nearly 200 developers since its earliest commits.

Fabric has a highly **modular** and **configurable** architecture, enabling innovation, versatility and optimization for a broad range of industry use cases including banking, finance, insurance, healthcare, human resources, supply chain and even digital music delivery.

Fabric is the first distributed ledger platform to support **smart contracts authored in general-purpose programming languages** such as Java, Go and Node.js, rather than constrained domain-specific languages (DSL). This means that most enterprises already have the skill set needed to develop smart contracts, and no additional training to learn a new language or DSL is needed.

The Fabric platform is also **permissioned**, meaning that, unlike with a public permissionless network, the participants are known to each other, rather than anonymous and therefore fully untrusted. This means that while the participants may not *fully* trust one another (they may, for example, be competitors in the same industry), a network can be operated under a governance model that is built off of what trust *does* exist between participants, such as a legal agreement or framework for handling disputes.

One of the most important of the platform's differentiators is its support for **pluggable consensus protocols** that enable the platform to be more effectively customized to fit particular use cases and trust models. For instance, when deployed within a single enterprise, or operated by a trusted authority, fully byzantine fault tolerant consensus might be considered unnecessary and an excessive drag on performance and throughput. In situations such as that, a **crash fault-tolerant** (CFT) consensus protocol might be more than adequate whereas, in a multi-party, decentralized use case, a more traditional **byzantine fault tolerant** (BFT) consensus protocol might be required.

Fabric can leverage consensus protocols that **do not require a native cryptocurrency** to incent costly mining or to fuel smart contract execution. Avoidance of a cryptocurrency reduces some significant risk/attack vectors, and absence of cryptographic mining operations means that the platform can be deployed with roughly the same operational cost as any other distributed system.

The combination of these differentiating design features makes Fabric one of the **better performing platforms** available today both in terms of transaction processing and transaction confirmation latency, and it enables **privacy and confidentiality** of transactions and the smart contracts (what Fabric calls "chaincode") that implement them.

Let's explore these differentiating features in more detail.

1.2 Modularity

Hyperledger Fabric has been specifically architected to have a modular architecture. Whether it is pluggable consensus, pluggable identity management protocols such as LDAP or OpenID Connect, key management protocols or cryptographic libraries, the platform has been designed at its core to be configured to meet the diversity of enterprise use case requirements.

At a high level, Fabric is comprised of the following modular components:

- A pluggable *ordering service* establishes consensus on the order of transactions and then broadcasts blocks to peers.

- A pluggable *membership service provider* is responsible for associating entities in the network with cryptographic identities.
- An optional *peer-to-peer gossip service* disseminates the blocks output by ordering service to other peers.
- Smart contracts (“chaincode”) run within a container environment (e.g. Docker) for isolation. They can be written in standard programming languages but do not have direct access to the ledger state.
- The ledger can be configured to support a variety of DBMSs.
- A pluggable endorsement and validation policy enforcement that can be independently configured per application.

There is fair agreement in the industry that there is no “one blockchain to rule them all”. Hyperledger Fabric can be configured in multiple ways to satisfy the diverse solution requirements for multiple industry use cases.

1.3 Permissioned vs Permissionless Blockchains

In a permissionless blockchain, virtually anyone can participate, and every participant is anonymous. In such a context, there can be no trust other than that the state of the blockchain, prior to a certain depth, is immutable. In order to mitigate this absence of trust, permissionless blockchains typically employ a “mined” native cryptocurrency or transaction fees to provide economic incentive to offset the extraordinary costs of participating in a form of byzantine fault tolerant consensus based on “proof of work” (PoW).

Permissioned blockchains, on the other hand, operate a blockchain amongst a set of known, identified and often vetted participants operating under a governance model that yields a certain degree of trust. A permissioned blockchain provides a way to secure the interactions among a group of entities that have a common goal but which may not fully trust each other. By relying on the identities of the participants, a permissioned blockchain can use more traditional crash fault tolerant (CFT) or byzantine fault tolerant (BFT) consensus protocols that do not require costly mining.

Additionally, in such a permissioned context, the risk of a participant intentionally introducing malicious code through a smart contract is diminished. First, the participants are known to one another and all actions, whether submitting application transactions, modifying the configuration of the network or deploying a smart contract are recorded on the blockchain following an endorsement policy that was established for the network and relevant transaction type. Rather than being completely anonymous, the guilty party can be easily identified and the incident handled in accordance with the terms of the governance model.

1.4 Smart Contracts

A smart contract, or what Fabric calls “chaincode”, functions as a trusted distributed application that gains its security/trust from the blockchain and the underlying consensus among the peers. It is the business logic of a blockchain application.

There are three key points that apply to smart contracts, especially when applied to a platform:

- many smart contracts run concurrently in the network,
- they may be deployed dynamically (in many cases by anyone), and
- application code should be treated as untrusted, potentially even malicious.

Most existing smart-contract capable blockchain platforms follow an **order-execute** architecture in which the consensus protocol:

- validates and orders transactions then propagates them to all peer nodes,
- each peer then executes the transactions sequentially.

The order-execute architecture can be found in virtually all existing blockchain systems, ranging from public/permissionless platforms such as [Ethereum](#) (with PoW-based consensus) to permissioned platforms such as [Tendermint](#), [Chain](#), and [Quorum](#).

Smart contracts executing in a blockchain that operates with the order-execute architecture must be deterministic; otherwise, consensus might never be reached. To address the non-determinism issue, many platforms require that the smart contracts be written in a non-standard, or domain-specific language (such as [Solidity](#)) so that non-deterministic operations can be eliminated. This hinders wide-spread adoption because it requires developers writing smart contracts to learn a new language and may lead to programming errors.

Further, since all transactions are executed sequentially by all nodes, performance and scale is limited. The fact that the smart contract code executes on every node in the system demands that complex measures be taken to protect the overall system from potentially malicious contracts in order to ensure resiliency of the overall system.

1.5 A New Approach

Fabric introduces a new architecture for transactions that we call **execute-order-validate**. It addresses the resiliency, flexibility, scalability, performance and confidentiality challenges faced by the order-execute model by separating the transaction flow into three steps:

- *execute* a transaction and check its correctness, thereby endorsing it,
- *order* transactions via a (pluggable) consensus protocol, and
- *validate* transactions against an application-specific endorsement policy before committing them to the ledger

This design departs radically from the order-execute paradigm in that Fabric executes transactions before reaching final agreement on their order.

In Fabric, an application-specific endorsement policy specifies which peer nodes, or how many of them, need to vouch for the correct execution of a given smart contract. Thus, each transaction need only be executed (endorsed) by the subset of the peer nodes necessary to satisfy the transaction's endorsement policy. This allows for parallel execution increasing overall performance and scale of the system. This first phase also **eliminates any non-determinism**, as inconsistent results can be filtered out before ordering.

Because we have eliminated non-determinism, Fabric is the first blockchain technology that **enables use of standard programming languages**.

1.6 Privacy and Confidentiality

As we have discussed, in a public, permissionless blockchain network that leverages PoW for its consensus model, transactions are executed on every node. This means that neither can there be confidentiality of the contracts themselves, nor of the transaction data that they process. Every transaction, and the code that implements it, is visible to every node in the network. In this case, we have traded confidentiality of contract and data for byzantine fault tolerant consensus delivered by PoW.

This lack of confidentiality can be problematic for many business/enterprise use cases. For example, in a network of supply-chain partners, some consumers might be given preferred rates as a means of either solidifying a relationship, or promoting additional sales. If every participant can see every contract and transaction, it becomes impossible to maintain such business relationships in a completely transparent network — everyone will want the preferred rates!

As a second example, consider the securities industry, where a trader building a position (or disposing of one) would not want her competitors to know of this, or else they will seek to get in on the game, weakening the trader's gambit.

In order to address the lack of privacy and confidentiality for purposes of delivering on enterprise use case requirements, blockchain platforms have adopted a variety of approaches. All have their trade-offs.

Encrypting data is one approach to providing confidentiality; however, in a permissionless network leveraging PoW for its consensus, the encrypted data is sitting on every node. Given enough time and computational resource, the encryption could be broken. For many enterprise use cases, the risk that their information could become compromised is unacceptable.

Zero knowledge proofs (ZKP) are another area of research being explored to address this problem, the trade-off here being that, presently, computing a ZKP requires considerable time and computational resources. Hence, the trade-off in this case is performance for confidentiality.

In a permissioned context that can leverage alternate forms of consensus, one might explore approaches that restrict the distribution of confidential information exclusively to authorized nodes.

Hyperledger Fabric, being a permissioned platform, enables confidentiality through its channel architecture and [private data](#) feature. In channels, participants on a Fabric network establish a sub-network where every member has visibility to a particular set of transactions. Thus, only those nodes that participate in a channel have access to the smart contract (chaincode) and data transacted, preserving the privacy and confidentiality of both. Private data allows collections between members on a channel, allowing much of the same protection as channels without the maintenance overhead of creating and maintaining a separate channel.

1.7 Pluggable Consensus

The ordering of transactions is delegated to a modular component for consensus that is logically decoupled from the peers that execute transactions and maintain the ledger. Specifically, the ordering service. Since consensus is modular, its implementation can be tailored to the trust assumption of a particular deployment or solution. This modular architecture allows the platform to rely on well-established toolkits for CFT (crash fault-tolerant) or BFT (byzantine fault-tolerant) ordering.

Fabric currently offers a CFT ordering service implementation based on the [etcd library](#) of the [Raft protocol](#). For information about currently available ordering services, check out our [conceptual documentation about ordering](#).

Note also that these are not mutually exclusive. A Fabric network can have multiple ordering services supporting different applications or application requirements.

1.8 Performance and Scalability

Performance of a blockchain platform can be affected by many variables such as transaction size, block size, network size, as well as limits of the hardware, etc. The Hyperledger Fabric [Performance and Scale working group](#) currently works on a benchmarking framework called [Hyperledger Caliper](#).

Several research papers have been published studying and testing the performance capabilities of Hyperledger Fabric. The latest [scaled Fabric to 20,000 transactions per second](#).

1.9 Conclusion

Any serious evaluation of blockchain platforms should include Hyperledger Fabric in its short list.

Combined, the differentiating capabilities of Fabric make it a highly scalable system for permissioned blockchains supporting flexible trust assumptions that enable the platform to support a wide range of industry use cases ranging from government, to finance, to supply-chain logistics, to healthcare and so much more.

Hyperledger Fabric is the most active of the Hyperledger projects. The community building around the platform is growing steadily, and the innovation delivered with each successive release far out-paces any of the other enterprise blockchain platforms.

1.10 Acknowledgement

The preceding is derived from the peer reviewed “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains” - Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, Jason Yellick

What's new in Hyperledger Fabric v2.x

2.1 What's New in Hyperledger Fabric v2.3

Hyperledger Fabric v2.3 introduces two new features for improved orderer and peer operations.

2.1.1 Orderer channel management without a system channel

To simplify the channel creation process and enhance the privacy and scalability of channels, it is now possible to create application channels without first creating a “system channel” managed by the ordering service. This process allows ordering nodes to join (or leave) any number of channels as needed, similar to how peers can participate in multiple channels.

Benefits of the new process:

- **Increased privacy:** Because all ordering nodes used to be joined to the system channel, every ordering node in a network knew about the existence of every channel on that ordering service. Now, an ordering node only knows about the channels it is joined to.
- **Scalability:** When there is a large number of ordering nodes and channels defined on the system channel, it can take a long time for ordering nodes to reach consensus on the membership of all the channels. Now, an ordering service can scale horizontally in a decentralized fashion by independently joining ordering nodes to specific channels.
- **Operational benefits**
 - Simple process to join an ordering node to a channel.
 - Ability to list the channels that the ordering node is a consenter on.
 - Simple process to remove a channel from an ordering node, which automatically cleans up the blocks associated with that channel.
 - Peer organizations do not need to coordinate with an admin of the system channel to create or update its MSP.

For more information, see the [Create a channel](#) topic.

2.1.2 Ledger snapshot

It is now possible to take a snapshot of a peer's channel information, including its state database, and join new peers (in the same organization or different organizations) to the channel based on the snapshot.

Using ledger snapshots has the following advantages:

- **Peers don't need to process all blocks since genesis block:** Peers can join a channel without processing all previous blocks since the genesis block, greatly reducing the time it takes to join a peer to an existing channel.
- **Peers can join channels using latest channel configuration:** Because snapshots include the latest channel configuration, peers can now join a channel using the latest channel configuration. This is especially important if critical channel configuration such as orderer endpoints or TLS CA certificates have been updated since the genesis block.
- **Reduced storage costs:** Peers that join by snapshot do not incur the storage cost of maintaining all blocks since the genesis block.
- **State checkpoints:** Peer administrators can snapshot current channel state and compare with other peers, in the same organization or different organizations, to verify the consistency and integrity of the ledger on each peer. Agreed upon snapshots can be used as a checkpoint and basis for newly joining peers.

For more information, see the *Taking ledger snapshots and using them to join channels* topic.

Note: While Fabric v2.3.0 introduces new features, Fabric v2.2.x remains the current long-term support release until the next LTS release is announced.

2.2 What's New in Hyperledger Fabric v2.0, v2.1, v2.2

The first Hyperledger Fabric major release since v1.0, Fabric v2.0 delivers important new features and changes for users and operators alike, including support for new application and privacy patterns, enhanced governance around smart contracts, and new options for operating nodes.

v2.1 and v2.2 build on the v2.0 release with minor features, improvements, and bug fixes, with v2.2 being the first long-term support (LTS) release of Fabric v2.x. Fixes will be provided on the v2.2.x release stream until after the next LTS release is announced.

Let's take a look at some of the highlights of the Fabric v2.0 release...

2.2.1 Decentralized governance for smart contracts

Fabric v2.0 introduces decentralized governance for smart contracts, with a new process for installing a chaincode on your peers and starting it on a channel. The new Fabric chaincode lifecycle allows multiple organizations to come to agreement on the parameters of a chaincode, such as the chaincode endorsement policy, before it can be used to interact with the ledger. The new model offers several improvements over the previous lifecycle:

- **Multiple organizations must agree to the parameters of a chaincode:** In the release 1.x versions of Fabric, one organization had the ability to set parameters of a chaincode (for instance the endorsement policy) for all other channel members, who only had the power to refuse to install the chaincode and therefore not take part in transactions invoking it. The new Fabric chaincode lifecycle is more flexible since it supports both centralized trust models (such as that of the previous lifecycle model) as well as decentralized models requiring a sufficient number of organizations to agree on an endorsement policy and other details before the chaincode becomes active on a channel.

- **More deliberate chaincode upgrade process:** In the previous chaincode lifecycle, the upgrade transaction could be issued by a single organization, creating a risk for a channel member that had not yet installed the new chaincode. The new model allows for a chaincode to be upgraded only after a sufficient number of organizations have approved the upgrade.
- **Simpler endorsement policy and private data collection updates:** Fabric lifecycle allows you to change an endorsement policy or private data collection configuration without having to repackage or reinstall the chaincode. Users can also take advantage of a new default endorsement policy that requires endorsement from a majority of organizations on the channel. This policy is updated automatically when organizations are added or removed from the channel.
- **Inspectable chaincode packages:** The Fabric lifecycle packages chaincode in easily readable tar files. This makes it easier to inspect the chaincode package and coordinate installation across multiple organizations.
- **Start multiple chaincodes on a channel using one package:** The previous lifecycle defined each chaincode on the channel using a name and version that was specified when the chaincode package was installed. You can now use a single chaincode package and deploy it multiple times with different names on the same channel or on different channels. For example, if you'd like to track different types of assets in their own 'copy' of the chaincode.
- **Chaincode packages do not need to be identical across channel members:** Organizations can extend a chaincode for their own use case, for example to perform different validations in the interest of their organization. As long as the required number of organizations endorse chaincode transactions with matching results, the transaction will be validated and committed to the ledger. This also allows organizations to individually roll out minor fixes on their own schedules without requiring the entire network to proceed in lock-step.

For existing Fabric deployments, you can continue to use the prior chaincode lifecycle with Fabric v2.x. The new chaincode lifecycle will become effective only when the channel application capability is updated to v2.0. See the [Fabric chaincode lifecycle](#) concept topic for an overview of the new chaincode lifecycle.

2.2.2 New chaincode application patterns for collaboration and consensus

The same decentralized methods of coming to agreement that underpin the new chaincode lifecycle management can also be used in your own chaincode applications to ensure organizations consent to data transactions before they are committed to the ledger.

- **Automated checks:** As mentioned above, organizations can add automated checks to chaincode functions to validate additional information before endorsing a transaction proposal.
- **Decentralized agreement:** Human decisions can be modeled into a chaincode process that spans multiple transactions. The chaincode may require actors from various organizations to indicate their terms and conditions of agreement in a ledger transaction. Then, a final chaincode proposal can verify that the conditions from all the individual transactors are met, and “settle” the business transaction with finality across all channel members. For a concrete example of indicating terms and conditions in private, see the asset transfer scenario in the [Private data](#) documentation.

2.2.3 Private data enhancements

Fabric v2.0 also enables new patterns for working with and sharing private data, without the requirement of creating private data collections for all combinations of channel members that may want to transact. Specifically, instead of sharing private data within a collection of multiple members, you may want to share private data across collections, where each collection may include a single organization, or perhaps a single organization along with a regulator or auditor.

Several enhancements in Fabric v2.x make these new private data patterns possible:

- **Sharing and verifying private data:** When private data is shared with a channel member who is not a member of a collection, or shared with another private data collection that contains one or more channel members (by writing a key to that collection), the receiving parties can utilize the `GetPrivateDataHash()` chaincode API to verify that the private data matches the on-chain hashes that were created from private data in previous transactions.
- **Collection-level endorsement policies:** Private data collections can now optionally be defined with an endorsement policy that overrides the chaincode-level endorsement policy for keys within the collection. This feature can be used to restrict which organizations can write data to a collection, and is what enables the new chaincode lifecycle and chaincode application patterns mentioned earlier. For example, you may have a chaincode endorsement policy that requires a majority of organizations to endorse, but for any given transaction, you may need two transacting organizations to individually endorse their agreement in their own private data collections.
- **Implicit per-organization collections:** If you'd like to utilize per-organization private data patterns, you don't even need to define the collections when deploying chaincode in Fabric v2.x. Implicit organization-specific collections can be used without any upfront definition.

To learn more about the new private data patterns, see the [Private data](#) (conceptual documentation). For details about private data collection configuration and implicit collections, see the [Private Data](#) (reference documentation).

2.2.4 External chaincode launcher

The external chaincode launcher feature empowers operators to build and launch chaincode with the technology of their choice. Use of external builders and launchers is not required as the default behavior builds and runs chaincode in the same manner as prior releases using the Docker API.

- **Eliminate Docker daemon dependency:** Prior releases of Fabric required peers to have access to a Docker daemon in order to build and launch chaincode - something that may not be desirable in production environments due to the privileges required by the peer process.
- **Alternatives to containers:** Chaincode is no longer required to be run in Docker containers, and may be executed in the operator's choice of environment (including containers).
- **External builder executables:** An operator can provide a set of external builder executables to override how the peer builds and launches chaincode.
- **Chaincode as an external service:** Traditionally, chaincodes are launched by the peer, and then connect back to the peer. It is now possible to run chaincode as an external service, for example in a Kubernetes pod, which a peer can connect to and utilize for chaincode execution. See [Chaincode as an external service](#) for more information.

See [External Builders and Launchers](#) to learn more about the external chaincode launcher feature.

2.2.5 State database cache for improved performance on CouchDB

- When using external CouchDB state database, read delays during endorsement and validation phases have historically been a performance bottleneck.
- With Fabric v2.0, a new peer cache replaces many of these expensive lookups with fast local cache reads. The cache size can be configured by using the `core.yaml` property `cacheSize`.

2.2.6 Alpine-based docker images

Starting with v2.0, Hyperledger Fabric Docker images will use Alpine Linux, a security-oriented, lightweight Linux distribution. This means that Docker images are now much smaller, providing faster download and startup times, as

well as taking up less disk space on host systems. Alpine Linux is designed from the ground up with security in mind, and the minimalist nature of the Alpine distribution greatly reduces the risk of security vulnerabilities.

2.2.7 Sample test network

The fabric-samples repository now includes a new Fabric test network. The test network is built to be a modular and user friendly sample Fabric network that makes it easy to test your applications and smart contracts. The network also supports the ability to deploy your network using Certificate Authorities, in addition to cryptogen.

For more information about this network, check out *Using the Fabric test network*.

2.2.8 Upgrading to Fabric v2.x

A major new release brings some additional upgrade considerations. Rest assured though, that rolling upgrades from v1.4.x to v2.0 are supported, so that network components can be upgraded one at a time with no downtime. You can also upgrade directly from the v1.4.x LTS release to the v2.2.x LTS release.

The upgrade docs have been significantly expanded and reworked, and now have a standalone home in the documentation: *Upgrading to the latest release*. Here you'll find documentation on *Upgrading your components* and *Updating the capability level of a channel*, as well as a specific look at the considerations for upgrading to v2.x, *Considerations for getting to v2.x*.

CHAPTER 3

Release notes

The release notes provide more details for users moving to the new release. Specifically, take a look at the changes and deprecations announced in each of the v2.x releases.

- [Fabric v2.0.0 release notes](#).
- [Fabric v2.0.1 release notes](#).
- [Fabric v2.1.0 release notes](#).
- [Fabric v2.1.1 release notes](#).
- [Fabric v2.2.0 release notes](#).
- [Fabric v2.2.1 release notes](#).
- [Fabric v2.2.2 release notes](#).
- [Fabric v2.3.0 release notes](#).
- [Fabric v2.3.1 release notes](#).
- [Fabric v2.3.2 release notes](#).
- [Fabric v2.3.3 release notes](#).

4.1 Introduction

Hyperledger Fabric is a platform for distributed ledger solutions underpinned by a modular architecture delivering high degrees of confidentiality, resiliency, flexibility, and scalability. It is designed to support pluggable implementations of different components and accommodate the complexity and intricacies that exist across the economic ecosystem.

We recommend first-time users begin by going through the rest of the introduction below in order to gain familiarity with how blockchains work and with the specific features and components of Hyperledger Fabric.

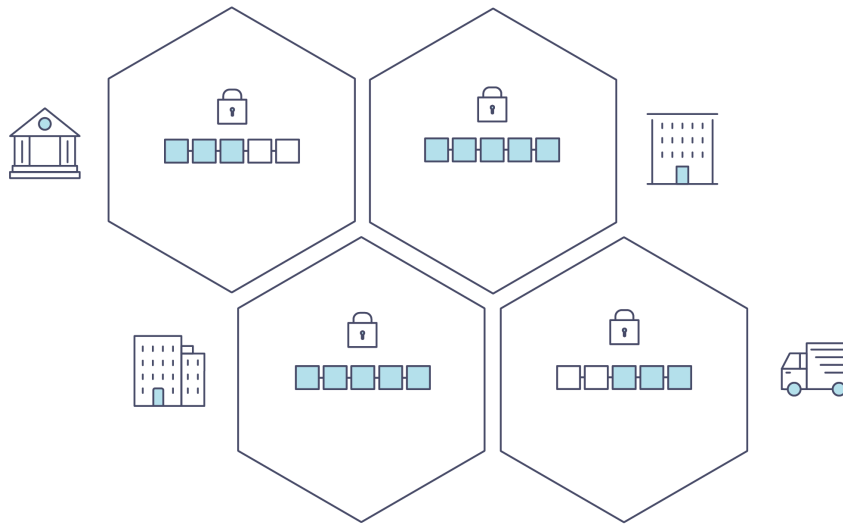
Once comfortable — or if you're already familiar with blockchain and Hyperledger Fabric — go to *Getting Started* and from there explore the demos, technical specifications, APIs, etc.

4.1.1 What is a Blockchain?

A Distributed Ledger

At the heart of a blockchain network is a distributed ledger that records all the transactions that take place on the network.

A blockchain ledger is often described as **decentralized** because it is replicated across many network participants, each of whom **collaborate** in its maintenance. We'll see that decentralization and collaboration are powerful attributes that mirror the way businesses exchange goods and services in the real world.



In addition to being decentralized and collaborative, the information recorded to a blockchain is append-only, using cryptographic techniques that guarantee that once a transaction has been added to the ledger it cannot be modified. This property of “immutability” makes it simple to determine the provenance of information because participants can be sure information has not been changed after the fact. It’s why blockchains are sometimes described as **systems of proof**.

Smart Contracts

To support the consistent update of information — and to enable a whole host of ledger functions (transacting, querying, etc) — a blockchain network uses **smart contracts** to provide controlled access to the ledger.

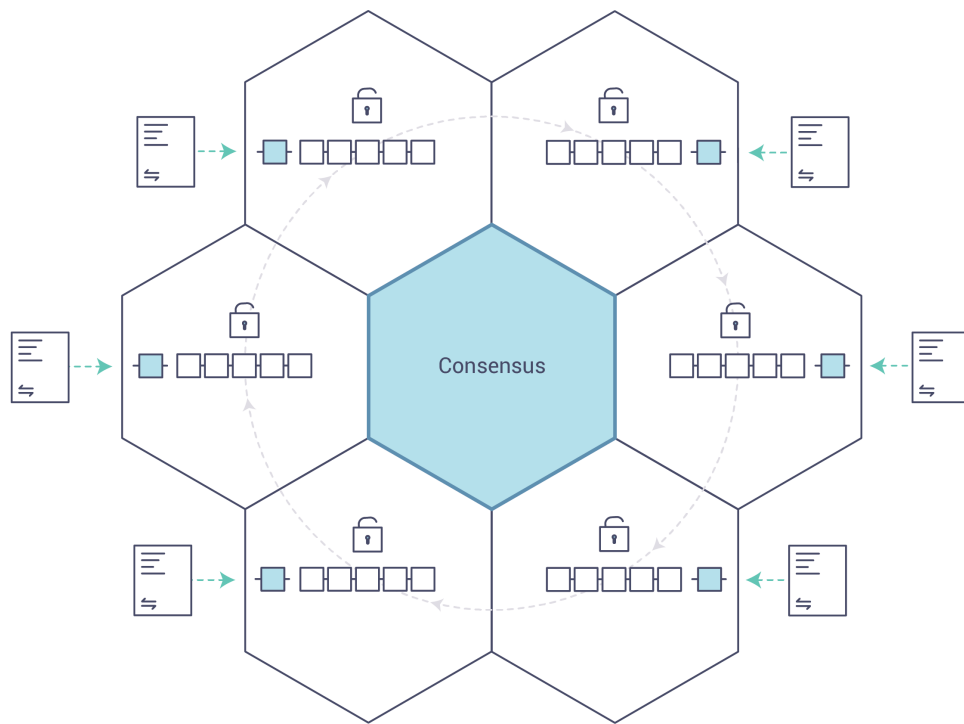


Smart contracts are not only a key mechanism for encapsulating information and keeping it simple across the network, they can also be written to allow participants to execute certain aspects of transactions automatically.

A smart contract can, for example, be written to stipulate the cost of shipping an item where the shipping charge changes depending on how quickly the item arrives. With the terms agreed to by both parties and written to the ledger, the appropriate funds change hands automatically when the item is received.

Consensus

The process of keeping the ledger transactions synchronized across the network — to ensure that ledgers update only when transactions are approved by the appropriate participants, and that when ledgers do update, they update with the same transactions in the same order — is called **consensus**.



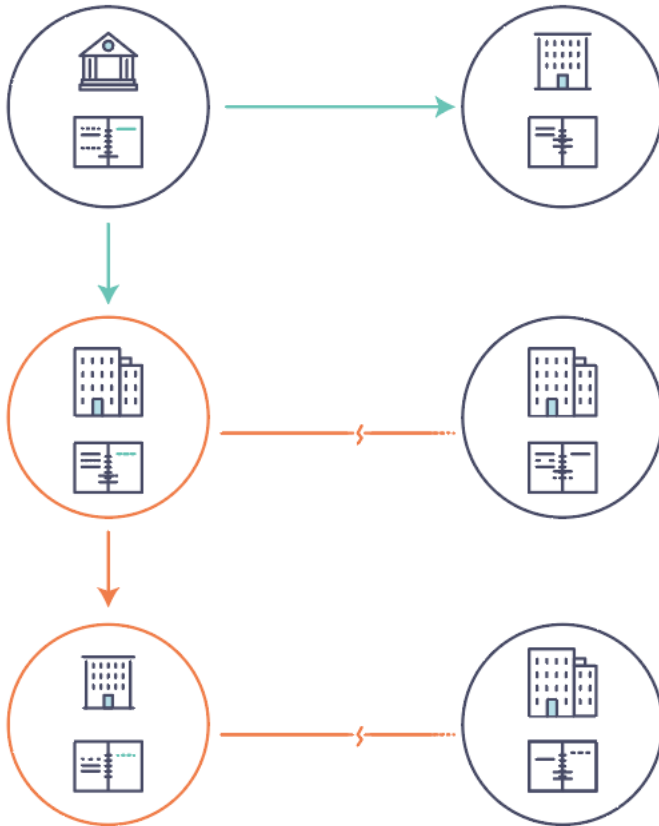
You'll learn a lot more about ledgers, smart contracts and consensus later. For now, it's enough to think of a blockchain as a shared, replicated transaction system which is updated via smart contracts and kept consistently synchronized through a collaborative process called consensus.

4.1.2 Why is a Blockchain useful?

Today's Systems of Record

The transactional networks of today are little more than slightly updated versions of networks that have existed since business records have been kept. The members of a **business network** transact with each other, but they maintain separate records of their transactions. And the things they're transacting — whether it's Flemish tapestries in the 16th century or the securities of today — must have their provenance established each time they're sold to ensure that the business selling an item possesses a chain of title verifying their ownership of it.

What you're left with is a business network that looks like this:



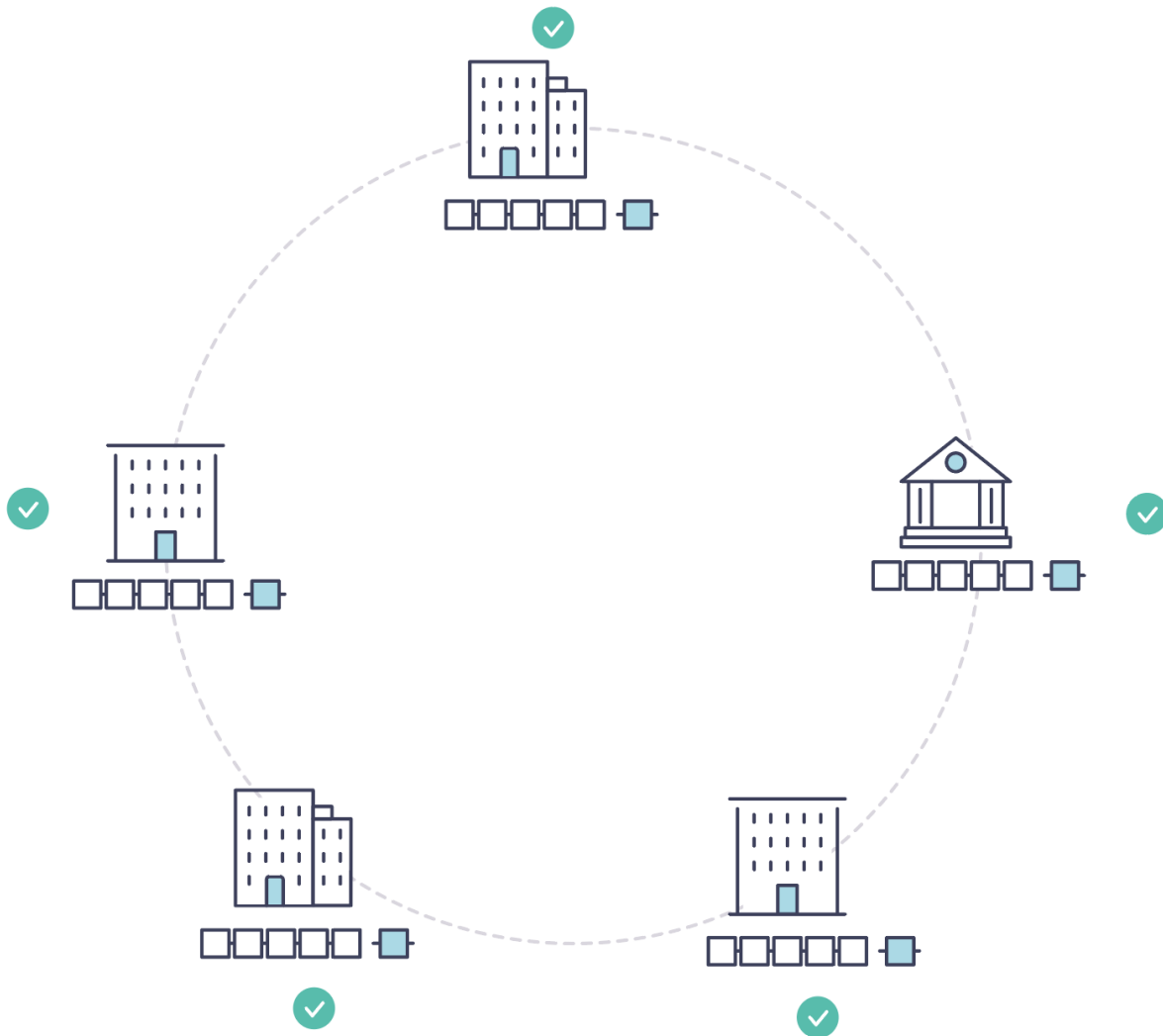
Modern technology has taken this process from stone tablets and paper folders to hard drives and cloud platforms, but the underlying structure is the same. Unified systems for managing the identity of network participants do not exist, establishing provenance is so laborious it takes days to clear securities transactions (the world volume of which is numbered in the many trillions of dollars), contracts must be signed and executed manually, and every database in the system contains unique information and therefore represents a single point of failure.

It's impossible with today's fractured approach to information and process sharing to build a system of record that spans a business network, even though the needs of visibility and trust are clear.

The Blockchain Difference

What if, instead of the rat's nest of inefficiencies represented by the "modern" system of transactions, business networks had standard methods for establishing identity on the network, executing transactions, and storing data? What if establishing the provenance of an asset could be determined by looking through a list of transactions that, once written, cannot be changed, and can therefore be trusted?

That business network would look more like this:



This is a blockchain network, wherein every participant has their own replicated copy of the ledger. In addition to ledger information being shared, the processes which update the ledger are also shared. Unlike today's systems, where a participant's **private** programs are used to update their **private** ledgers, a blockchain system has **shared** programs to update **shared** ledgers.

With the ability to coordinate their business network through a shared ledger, blockchain networks can reduce the time, cost, and risk associated with private information and processing while improving trust and visibility.

You now know what blockchain is and why it's useful. There are a lot of other details that are important, but they all relate to these fundamental ideas of the sharing of information and processes.

4.1.3 What is Hyperledger Fabric?

The Linux Foundation founded the Hyperledger project in 2015 to advance cross-industry blockchain technologies. Rather than declaring a single blockchain standard, it encourages a collaborative approach to developing blockchain technologies via a community process, with intellectual property rights that encourage open development and the adoption of key standards over time.

Hyperledger Fabric is one of the blockchain projects within Hyperledger. Like other blockchain technologies, it has a ledger, uses smart contracts, and is a system by which participants manage their transactions.

Where Hyperledger Fabric breaks from some other blockchain systems is that it is **private** and **permissioned**. Rather than an open permissionless system that allows unknown identities to participate in the network (requiring protocols like “proof of work” to validate transactions and secure the network), the members of a Hyperledger Fabric network enroll through a trusted **Membership Service Provider (MSP)**.

Hyperledger Fabric also offers several pluggable options. Ledger data can be stored in multiple formats, consensus mechanisms can be swapped in and out, and different MSPs are supported.

Hyperledger Fabric also offers the ability to create **channels**, allowing a group of participants to create a separate ledger of transactions. This is an especially important option for networks where some participants might be competitors and not want every transaction they make — a special price they’re offering to some participants and not others, for example — known to every participant. If two participants form a channel, then those participants — and no others — have copies of the ledger for that channel.

Shared Ledger

Hyperledger Fabric has a ledger subsystem comprising two components: the **world state** and the **transaction log**. Each participant has a copy of the ledger to every Hyperledger Fabric network they belong to.

The world state component describes the state of the ledger at a given point in time. It’s the database of the ledger. The transaction log component records all transactions which have resulted in the current value of the world state; it’s the update history for the world state. The ledger, then, is a combination of the world state database and the transaction log history.

The ledger has a replaceable data store for the world state. By default, this is a LevelDB key-value store database. The transaction log does not need to be pluggable. It simply records the before and after values of the ledger database being used by the blockchain network.

Smart Contracts

Hyperledger Fabric smart contracts are written in **chaincode** and are invoked by an application external to the blockchain when that application needs to interact with the ledger. In most cases, chaincode interacts only with the database component of the ledger, the world state (querying it, for example), and not the transaction log.

Chaincode can be implemented in several programming languages. Currently, Go, Node.js, and Java chaincode are supported.

Privacy

Depending on the needs of a network, participants in a Business-to-Business (B2B) network might be extremely sensitive about how much information they share. For other networks, privacy will not be a top concern.

Hyperledger Fabric supports networks where privacy (using channels) is a key operational requirement as well as networks that are comparatively open.

Consensus

Transactions must be written to the ledger in the order in which they occur, even though they might be between different sets of participants within the network. For this to happen, the order of transactions must be established and a method for rejecting bad transactions that have been inserted into the ledger in error (or maliciously) must be put into place.

This is a thoroughly researched area of computer science, and there are many ways to achieve it, each with different trade-offs. For example, PBFT (Practical Byzantine Fault Tolerance) can provide a mechanism for file replicas to communicate with each other to keep each copy consistent, even in the event of corruption. Alternatively, in Bitcoin, ordering happens through a process called mining where competing computers race to solve a cryptographic puzzle which defines the order that all processes subsequently build upon.

Hyperledger Fabric has been designed to allow network starters to choose a consensus mechanism that best represents the relationships that exist between participants. As with privacy, there is a spectrum of needs; from networks that are highly structured in their relationships to those that are more peer-to-peer.

4.2 Hyperledger Fabric Model

This section outlines the key design features woven into Hyperledger Fabric that fulfill its promise of a comprehensive, yet customizable, enterprise blockchain solution:

- *Assets* — Asset definitions enable the exchange of almost anything with monetary value over the network, from whole foods to antique cars to currency futures.
- *Chaincode* — Chaincode execution is partitioned from transaction ordering, limiting the required levels of trust and verification across node types, and optimizing network scalability and performance.
- *Ledger Features* — The immutable, shared ledger encodes the entire transaction history for each channel, and includes SQL-like query capability for efficient auditing and dispute resolution.
- *Privacy* — Channels and private data collections enable private and confidential multi-lateral transactions that are usually required by competing businesses and regulated industries that exchange assets on a common network.
- *Security & Membership Services* — Permissioned membership provides a trusted blockchain network, where participants know that all transactions can be detected and traced by authorized regulators and auditors.
- *Consensus* — A unique approach to consensus enables the flexibility and scalability needed for the enterprise.

4.2.1 Assets

Assets can range from the tangible (real estate and hardware) to the intangible (contracts and intellectual property). Hyperledger Fabric provides the ability to modify assets using chaincode transactions.

Assets are represented in Hyperledger Fabric as a collection of key-value pairs, with state changes recorded as transactions on a *Channel* ledger. Assets can be represented in binary and/or JSON form.

4.2.2 Chaincode

Chaincode is software defining an asset or assets, and the transaction instructions for modifying the asset(s); in other words, it's the business logic. Chaincode enforces the rules for reading or altering key-value pairs or other state database information. Chaincode functions execute against the ledger's current state database and are initiated through a transaction proposal. Chaincode execution results in a set of key-value writes (write set) that can be submitted to the network and applied to the ledger on all peers.

4.2.3 Ledger Features

The ledger is the sequenced, tamper-resistant record of all state transitions in the fabric. State transitions are a result of chaincode invocations ('transactions') submitted by participating parties. Each transaction results in a set of asset key-value pairs that are committed to the ledger as creates, updates, or deletes.

The ledger is comprised of a blockchain ('chain') to store the immutable, sequenced record in blocks, as well as a state database to maintain current fabric state. There is one ledger per channel. Each peer maintains a copy of the ledger for each channel of which they are a member.

Some features of a Fabric ledger:

- Query and update ledger using key-based lookups, range queries, and composite key queries
- Read-only queries using a rich query language (if using CouchDB as state database)
- Read-only history queries — Query ledger history for a key, enabling data provenance scenarios

- Transactions consist of the versions of keys/values that were read in chaincode (read set) and keys/values that were written in chaincode (write set)
- Transactions contain signatures of every endorsing peer and are submitted to ordering service
- Transactions are ordered into blocks and are “delivered” from an ordering service to peers on a channel
- Peers validate transactions against endorsement policies and enforce the policies
- Prior to appending a block, a versioning check is performed to ensure that states for assets that were read have not changed since chaincode execution time
- There is immutability once a transaction is validated and committed
- A channel’s ledger contains a configuration block defining policies, access control lists, and other pertinent information
- Channels contain *Membership Service Provider* instances allowing for crypto materials to be derived from different certificate authorities

See the ledger topic for a deeper dive on the databases, storage structure, and “query-ability.”

4.2.4 Privacy

Hyperledger Fabric employs an immutable ledger on a per-channel basis, as well as chaincode that can manipulate and modify the current state of assets (i.e. update key-value pairs). A ledger exists in the scope of a channel — it can be shared across the entire network (assuming every participant is operating on one common channel) — or it can be privatized to include only a specific set of participants.

In the latter scenario, these participants would create a separate channel and thereby isolate/segregate their transactions and ledger. In order to solve scenarios that want to bridge the gap between total transparency and privacy, chaincode can be installed only on peers that need to access the asset states to perform reads and writes (in other words, if a chaincode is not installed on a peer, it will not be able to properly interface with the ledger).

When a subset of organizations on that channel need to keep their transaction data confidential, a private data collection (collection) is used to segregate this data in a private database, logically separate from the channel ledger, accessible only to the authorized subset of organizations.

Thus, channels keep transactions private from the broader network whereas collections keep data private between subsets of organizations on the channel.

To further obfuscate the data, values within chaincode can be encrypted (in part or in total) using common cryptographic algorithms such as AES before sending transactions to the ordering service and appending blocks to the ledger. Once encrypted data has been written to the ledger, it can be decrypted only by a user in possession of the corresponding key that was used to generate the cipher text.

See the *Private Data* topic for more details on how to achieve privacy on your blockchain network.

4.2.5 Security & Membership Services

Hyperledger Fabric underpins a transactional network where all participants have known identities. Public Key Infrastructure is used to generate cryptographic certificates which are tied to organizations, network components, and end users or client applications. As a result, data access control can be manipulated and governed on the broader network and on channel levels. This “permissioned” notion of Hyperledger Fabric, coupled with the existence and capabilities of channels, helps address scenarios where privacy and confidentiality are paramount concerns.

For more information see the *Security Model* topic.

4.2.6 Consensus

In distributed ledger technology, consensus has recently become synonymous with a specific algorithm, within a single function. However, consensus encompasses more than simply agreeing upon the order of transactions, and this differentiation is highlighted in Hyperledger Fabric through its fundamental role in the entire transaction flow, from proposal and endorsement, to ordering, validation and commitment. In a nutshell, consensus is defined as the full-circle verification of the correctness of a set of transactions comprising a block.

Consensus is achieved ultimately when the order and results of a block's transactions have met the explicit policy criteria checks. These checks and balances take place during the lifecycle of a transaction, and include the usage of endorsement policies to dictate which specific members must endorse a certain transaction class, as well as system chaincodes to ensure that these policies are enforced and upheld. Prior to commitment, the peers will employ these system chaincodes to make sure that enough endorsements are present, and that they were derived from the appropriate entities. Moreover, a versioning check will take place during which the current state of the ledger is agreed or consented upon, before any blocks containing transactions are appended to the ledger. This final check provides protection against double spend operations and other threats that might compromise data integrity, and allows for functions to be executed against non-static variables.

In addition to the multitude of endorsement, validity and versioning checks that take place, there are also ongoing identity verifications happening in all directions of the transaction flow. Access control lists are implemented on hierarchical layers of the network (ordering service down to channels), and payloads are repeatedly signed, verified and authenticated as a transaction proposal passes through the different architectural components. To conclude, consensus is not merely limited to the agreed upon order of a batch of transactions; rather, it is an overarching characterization that is achieved as a byproduct of the ongoing verifications that take place during a transaction's journey from proposal to commitment.

Check out the [Transaction Flow](#) diagram for a visual representation of consensus.

4.3 How Fabric networks are structured

This topic will describe, **at a conceptual level**, how Hyperledger Fabric allows organizations to collaborate in the formation of blockchain networks. If you're an architect, administrator or developer, you can use this topic to get a solid understanding of the major structure and process components in a Hyperledger Fabric blockchain network. This topic will use a manageable example that introduces all of the major components in a blockchain network.

After reading this topic and understanding the concept of policies, you will have a solid understanding of the decisions that organizations need to make to establish the policies that control a deployed Hyperledger Fabric network. You'll also understand how organizations manage network evolution using declarative policies – a key feature of Hyperledger Fabric. In a nutshell, you'll understand the major technical components of Hyperledger Fabric and the decisions organizations need to make about them.

Note: in this topic, we'll refer to the structure of a network that does not have a "system channel", a channel run by the ordering service that ordering nodes are bootstrapped with. For a version of this topic that does use the system channel, check out [Blockchain network](#).

4.3.1 What is a blockchain network?

A blockchain network is a technical infrastructure that provides ledger and smart contract (which are packaged as part of a "chaincode") services to applications. Primarily, smart contracts are used to generate transactions which are subsequently distributed to every peer node in the network where they are immutably recorded on their copy of the ledger. The users of applications might be end users using client applications or blockchain network administrators.

In most cases, multiple [organizations](#) come together to form a **channel** on which transactions are invoked on chaincodes and where permissions are determined by a set of [policies](#) that are agreed to when the channel is originally

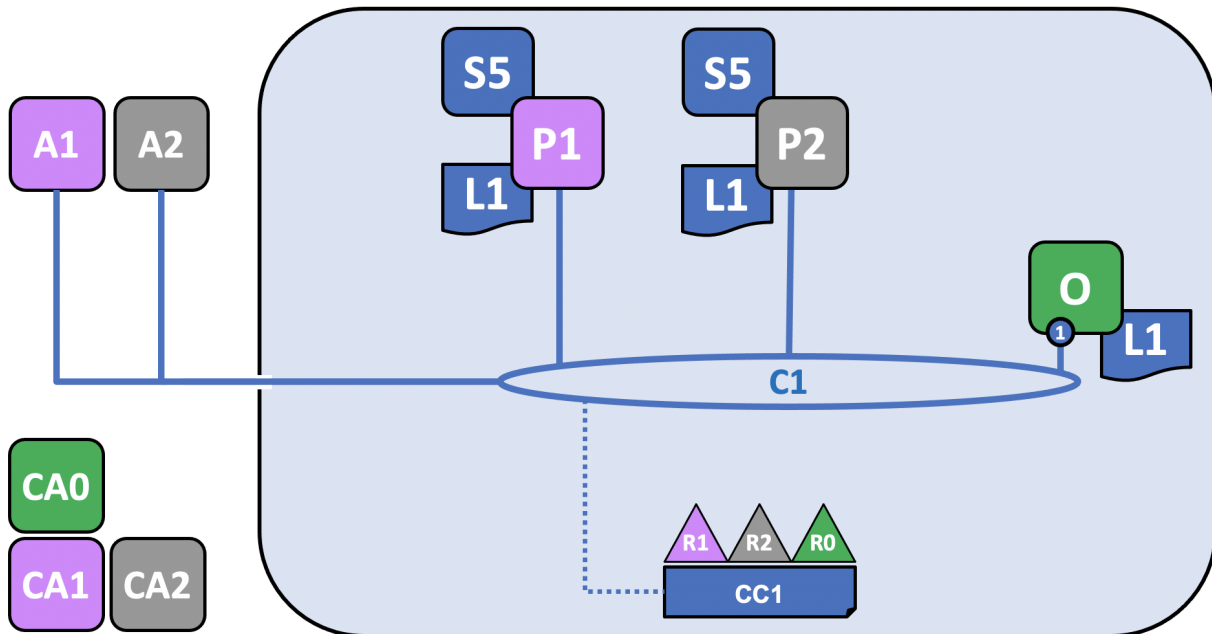
configured. Moreover, policies can change over time subject to the agreement of the organizations.

In this topic, we'll refer to both the “network” and the “channel”. In Hyperledger Fabric, these terms are effectively synonymous, as they both refer collectively to the organizations, components, policies, and processes that govern the interactions between organizations within a defined structure.

4.3.2 The sample network

Before we start, let's show you what we're aiming at! Here's a diagram representing the **final state** of our sample network.

It might look complicated right now, but as we go through this topic, we will build up the network piece by piece, so that you see how the organizations R1, R2 and R0 contribute infrastructure to the network to help form it. This infrastructure implements the blockchain network, and it is governed by policies agreed by the organizations who form the network – for example, who can add new organizations. You'll discover how applications consume the ledger and smart contract services provided by the blockchain network.

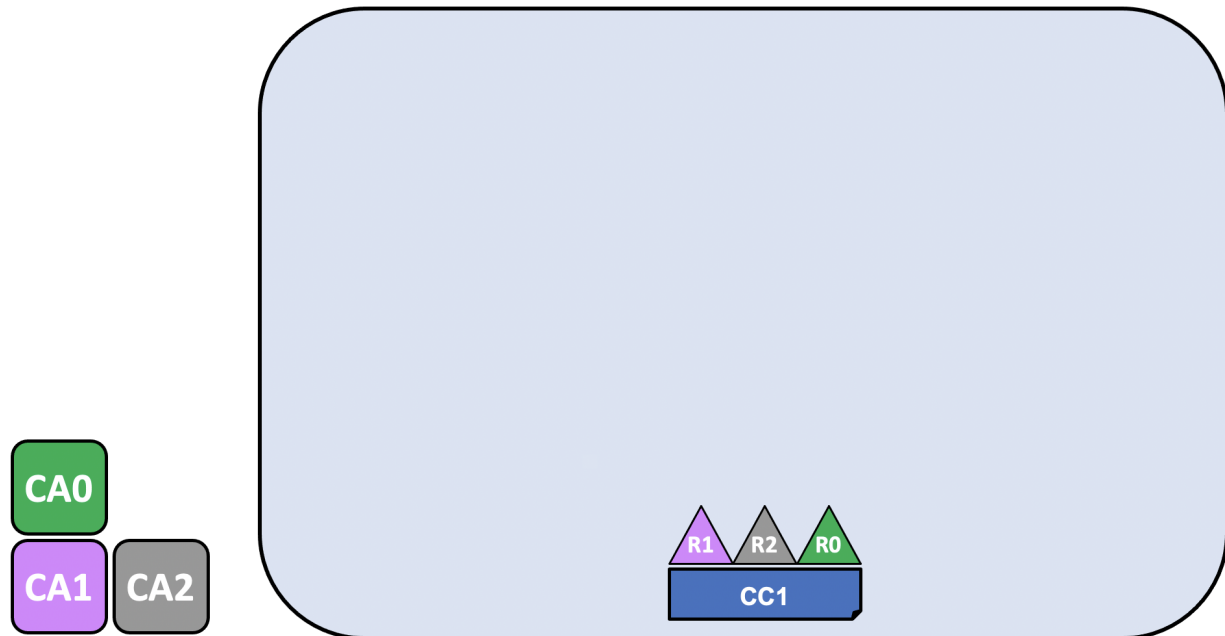


Three organizations, R1, R2, and R0 have jointly decided that they will establish a network. This network has a configuration, CC1, which all of the organizations have agreed to and which lists the definition of the organizations as well as the policies which define the roles each organization will play on the channel.

On this channel, R1 and R2 will join peers, named P1 and P2, to the channel, C1, while R0 owns O, the ordering service of the channel. All of these nodes will contain a copy of the ledger (L1) of the channel, which is where transactions are recorded. Note that the copy of the ledger kept by the ordering service does not contain a *state database*. R1 and R2 will also interact with the channel through the applications A1 and A2, which they own. All three organizations have a Certificate Authority that has generated the necessary certificates for the nodes, admins, organizations definitions, and applications of its organization.

4.3.3 Creating the network

The first step in creating a network or a channel is to agree to and then define its configuration:



The channel configuration, CC1, has been agreed to by organizations R1, R2, and R0 and is contained in a block known as a “configuration block” that is, typically, created by the `configtxgen` tool from a `configtx.yaml` file. While it is possible for one organization to create this a channel unilaterally and then invite other organizations to it (we’ll explore this in [Adding an organization to an existing channel](#)), for now we’ll assume that the organizations wish to collaborate on the channel from the beginning.

Once a configuration block exists, a channel can be said to **logically exist**, even though no components are physically joined to it. This configuration block contains a record of the organizations that can join components and interact on the channel, as well as the **policies** that define the structure for how decisions are made and specific outcomes are reached. While the peers and applications are critical actors in the network, their behavior in a channel is dictated more by the channel configuration policy than any other factor. For more information about the policies and how they are defined in a channel configuration, check out [Policies](#).

The definitions of these organizations, and the identities of their admins, must be created by a Certificate Authority (CA) associated with each organization. In our example, the organizations R1, R2, and R0 have had their certifications and organization definition created by CA1, CA2, and CA0, respectively. For information about how to create a CA, check out [Planning for a CA](#). After the CA has been created, check out [Registering and enrolling identities with a CA](#) for information about how to define an organization and create identities for admins and nodes.

For more information about using `configtxgen` to create a configuration block, check out [Using configtx.yaml to build a channel configuration](#).

Certificate Authorities

Certificate Authorities play a key role in the network because they dispense X.509 certificates that can be used to identify components as belonging to an organization. Certificates issued by CAs can also be used to sign transactions to indicate that an organization endorses the transaction result – a precondition of it being accepted onto the ledger. Let’s examine these two aspects of a CA in a little more detail.

Firstly, different components of the blockchain network use certificates to identify themselves to each other as being from a particular organization. That’s why there is usually more than one CA supporting a blockchain network – different organizations often use different CAs. We’re going to use three CAs in our channel; one for each organization. Indeed, CAs are so important that Hyperledger Fabric provides you with a built-in one (called the *Fabric-CA*) to help you get going, though in practice, organizations will choose to use their own CA.

The mapping of certificates to member organizations is achieved via a structure called a [Membership Services Provider \(MSP\)](#), which defines an organization by creating an MSP which is tied to a root CA certificate to identify that components and identities were created by the root CA. The channel configuration can then assign certain rights and permissions to the organization through a policy (which will give a particular organization, such as R1, the right to add new organizations to the channel). We don't show MSPs on these diagrams, as they would clutter them up, but because they define organizations, they are very important.

Secondly, we'll see later how certificates issued by CAs are at the heart of the [transaction](#) generation and validation process. Specifically, X.509 certificates are used in client application [transaction proposals](#) and smart contract [transaction responses](#) to digitally sign [transactions](#). Subsequently the network nodes who host copies of the ledger verify that transaction signatures are valid before accepting transactions onto the ledger.

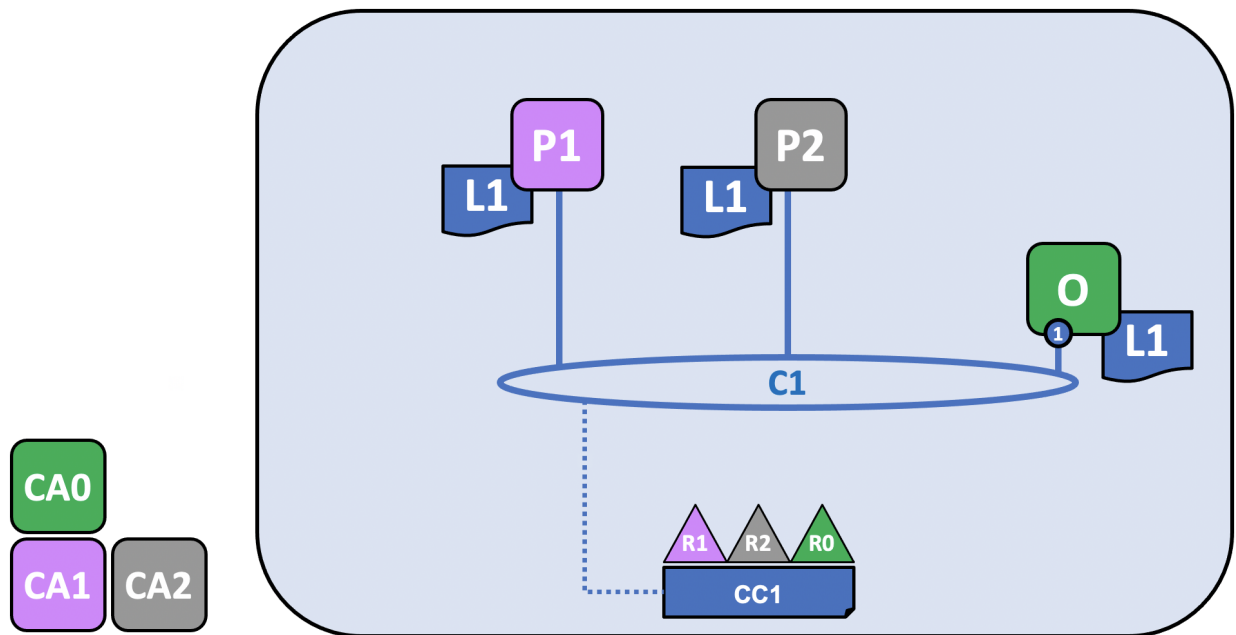
4.3.4 Join nodes to the channel

Peers are a fundamental element of the network because they host ledgers and chaincode (which contain smart contracts) and are therefore one of the physical points at which organizations that transact on a channel connect to the channel (the other being an application). A peer can belong to as many channels as an organizations deems appropriate (depending on factors like the processing limitations of the peer pod and data residency rules that exist in a particular country). For more information about peers, check out [Peers](#).

The ordering service, on the other hand, gathers endorsed transactions from applications and orders them into transaction blocks, which are subsequently distributed to every peer node in the channel. At each of these committing peers, transactions are recorded and the local copy of the ledger updated appropriately. An ordering service is unique to a particular channel, with the nodes servicing that channel also known as a "consenter set". Even if a node (or group of nodes) services multiple channels, each channel's ordering service is considered to be a distinct instance of the ordering service. For more information about the ordering service, check out [The Ordering Service](#).

For information about how to create peer and ordering nodes, check out [Deploying a production network](#).

Because R1, R2, and R0 are listed in the channel configuration, they are allowed to join peers (in the case of R1 and R2) or ordering nodes (in the case of R0) to the channel.



R1's peer, P1, and R2's peer, P2, along with R0's ordering service, O, join the channel through the process described in the [Create a channel](#) tutorial. Note that while only one ordering node, 1, is joined to this channel, in a production

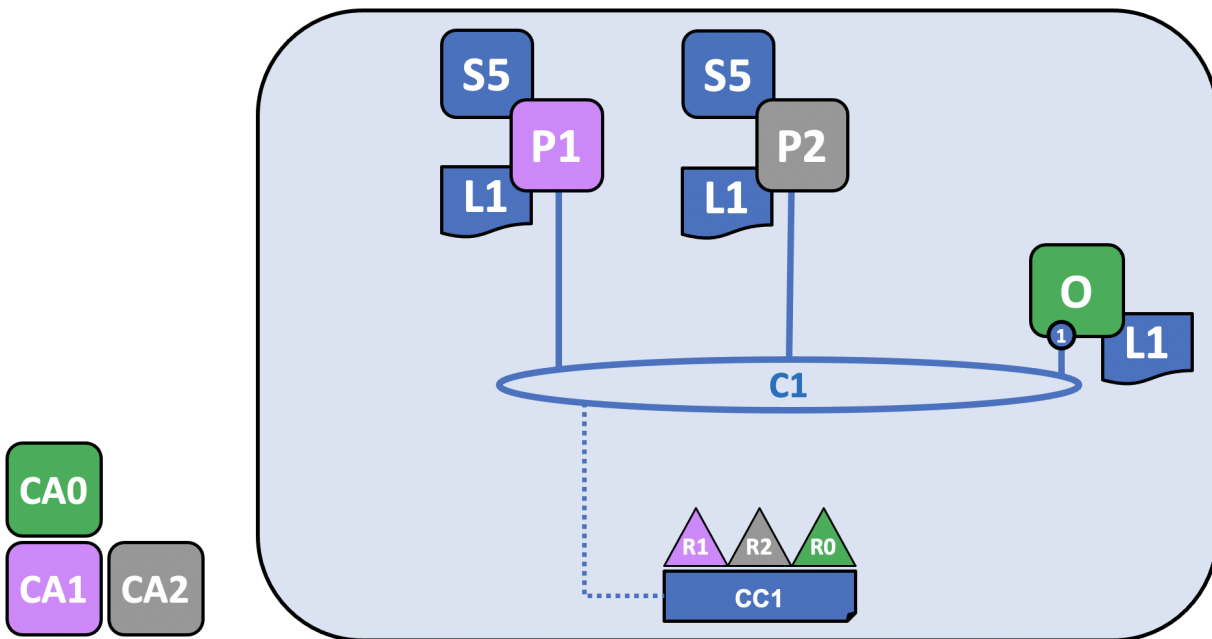
scenario, an ordering service should contain at least three nodes. For the purposes of this topic, however, it is more important to conceptualize the interactions of the ordering service and the other components of the network than it is to understand how the needs of high availability impact configuration decisions. The nodes belonging to each organization have x.509 certificates created for them by the Certificate Authority associated with that organization. P1's certificates are created by CA1, P2's certificates are created by CA2, and so on.

Every node in the channel stores a copy of the ledger of the channel, L1, which will be updated with each new block (note that the ordering service only contains the blockchain portion of a ledger and not the [state database](#)). Because of this, we can think of L1 as being **physically hosted** on P1, but **logically hosted** on the channel C1. The best practice is for R1 and R2 to make their peers, P1 and P2, [anchor peers](#), as this will bootstrap communication on the network between R1 and R2.

After the ordering service has been joined to the channel, it is possible to propose and commit updates to the channel configuration, but little else. Next, you must install, approve, and commit a chaincode on a channel.

4.3.5 Install, approve, and commit a chaincode

Chaincodes are installed on peers, and then defined and committed on a channel:



In Fabric, the business logic that defines how peer organizations interact with the ledger (for example, a transaction that changes the ownership of an asset), is contained in a smart contract. The structure that contains the smart contract, called chaincode, is installed on the relevant peers, approved by the relevant peer organizations, and committed on the channel. In this way, you can consider a chaincode to be **physically hosted** on a peer but **logically hosted** on a channel. In our example, the chaincode, S5, is installed on every peer, even though organizations are not required to install every chaincode. Note that the ordering service does not have the chaincode installed on it, as ordering nodes do not typically propose transactions. The process of installing, approving, and committing a chaincode is known as the “lifecycle” of the chaincode. For more information, check out [Fabric chaincode lifecycle](#).

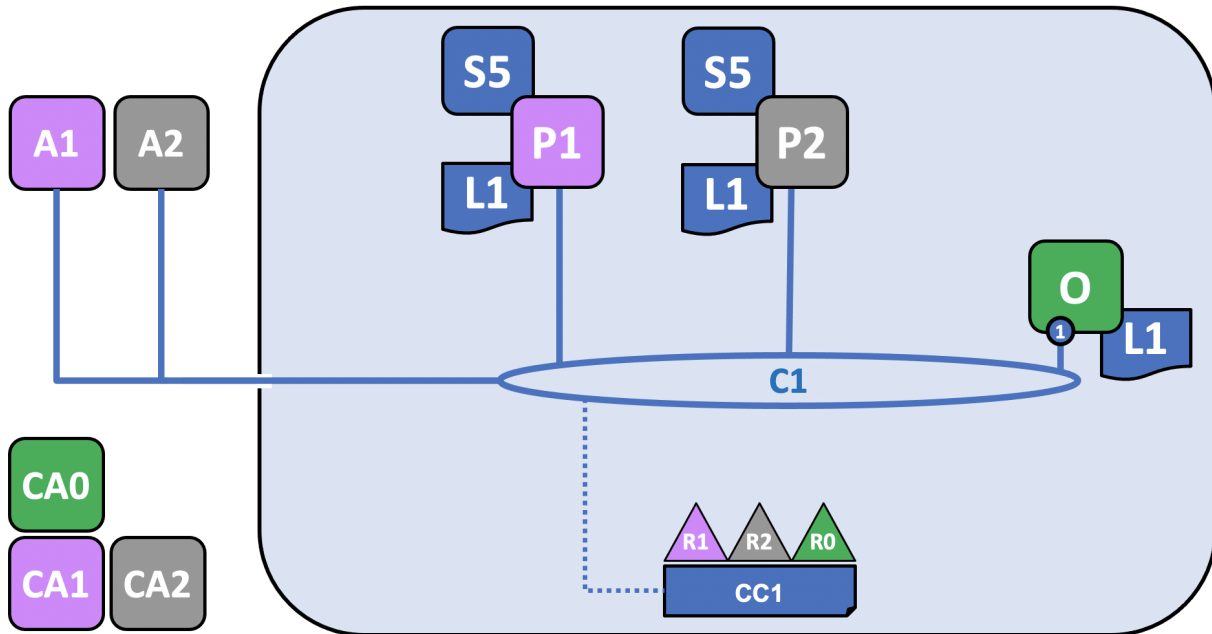
The most important piece of information supplied within the chaincode definition is the [endorsement policy](#). It describes which organizations must endorse transactions before they will be accepted by other organizations onto their copy of the ledger. An endorsement policy can be set to any combination of members in a channel, depending on the use case. If an endorsement policy is not set, it is inherited from the default endorsement policy specified in the channel configuration.

Note that while some chaincodes include the ability to create [private data transactions](#) between members on a channel, private data is outside the scope of this topic.

While it's now technically possible to drive transactions using the peer CLI, the best practice is to create an application and use it to invoke transactions on your chaincode.

4.3.6 Using an application on the channel

After a smart contract has been committed, client applications can be used to invoke transactions on a chaincode. This completes the structure we showed in the first image:



Just like peers and orderers, a client application has an identity that associates it with an organization. In our example, client application A1 is associated with organization R1 and is connected to C1.

Once a chaincode has been installed on a peer node and defined on a channel it can be [invoked](#) by a client application. Client applications do this by sending transaction proposals to peers owned by the organizations specified by the endorsement policy. The transaction proposal serves as input to the chaincode, which uses it to generate an endorsed transaction response, which is returned by the peer node to the client application.

We can see that our peer organizations, R1 and R2, are fully participating in the channel. Their applications can access the ledger L1 via smart contract S5 to generate transactions that will be endorsed by the organizations specified in the endorsement policy and written to the ledger.

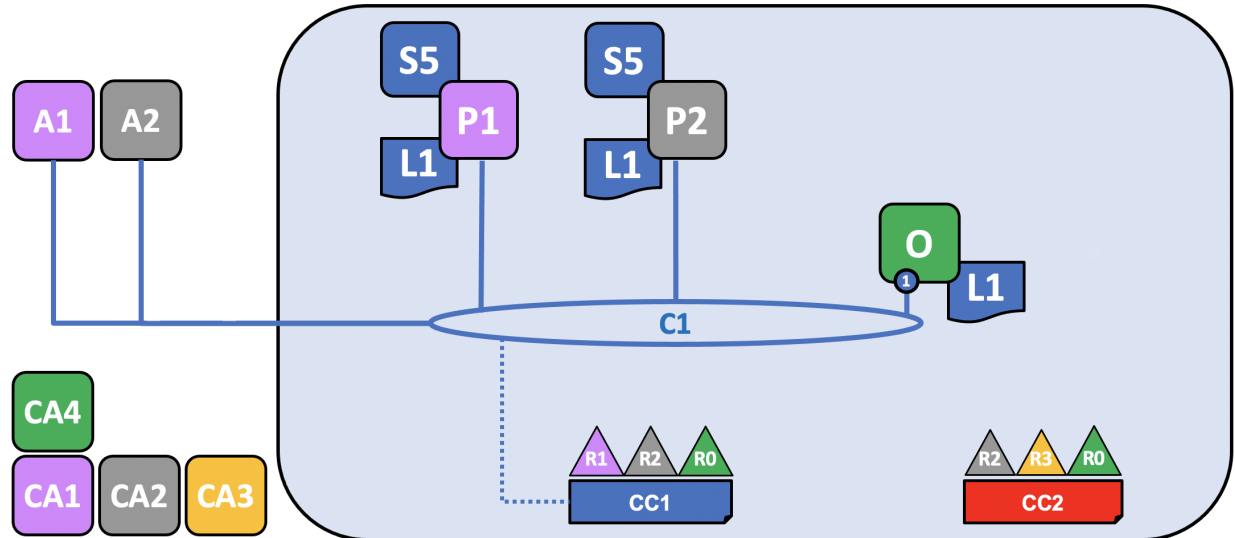
For more information about how to develop an application, check out [Developing applications](#).

4.3.7 Joining components to multiple channels

Now that we have showed the process for how a channel is created, as well as the nature of the high level interactions between organizations, nodes, policies, chaincodes, and applications, let's expand our view by adding a new organization and a new channel to our scenario. To show how Fabric components can be joined to multiple channels, we'll join R2 and its peer, P2, to the new channel, while R1 and P1 will not be joined.

Creating the new channel configuration

As we've seen, the first step in creating a channel is to create its configuration. This channel will include not just R2 and R0, but a new organization, R3, which has had its identities and certificates created for it by CA3. R1 will have no rights over this channel and will not be able to join components to it. In fact, it has no way to know it even exists!

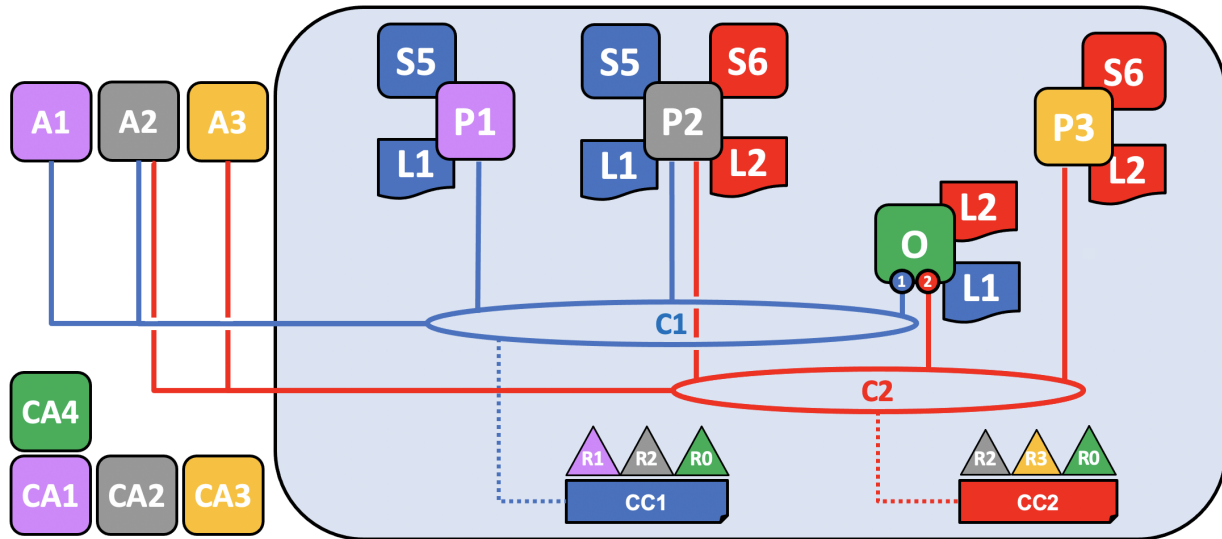


As before, now that the channel configuration, CC2, has been created, the channel can be said to **logically** exist, even though no components are joined to it.

So let's join some components to it!

Join components to the new channel

Just as we did with C1, let's join our components to C2. Because we already showed how all channels have a ledger and how chaincodes are installed on peers and committed to a channel (in this case, the chaincode is called S6), we'll skip those steps for now to show the end state of C2. Note that this channel has its own ledger, L2, which is completely separate from the ledger of C1. That's because even though R2 (and its peer, P2) are joined to both channels, the two channels are entirely separate administrative domains.



Note that while both C1 and C2 both have the same orderer organization joined to it, R0, different ordering nodes are servicing each channel. This is not a mandatory configuration because even if the same ordering nodes are joined to multiple channels, each channel has a separate instance of the ordering service, and is more common in channels in which multiple orderer organizations come together to contribute nodes to an ordering service. Note that only the ordering node joined to a particular channel has the ledger of that channel.

While it would also be possible for R2 to deploy a new peer to join to channel C2, in this case they have chosen to deploy the P2 to C2. Note that P2 has both the ledger of C1 (called L1) and the ledger of C2 (called L2) on its file system. Similarly, R2 has chosen to modify its application, A2, to be able to be used with C2, while R3's application, A3, is being used with C2.

Logically, this is all very similar to the creation of C1. Two peer organizations come together with an ordering organization to create a channel and join components and a chaincode to it.

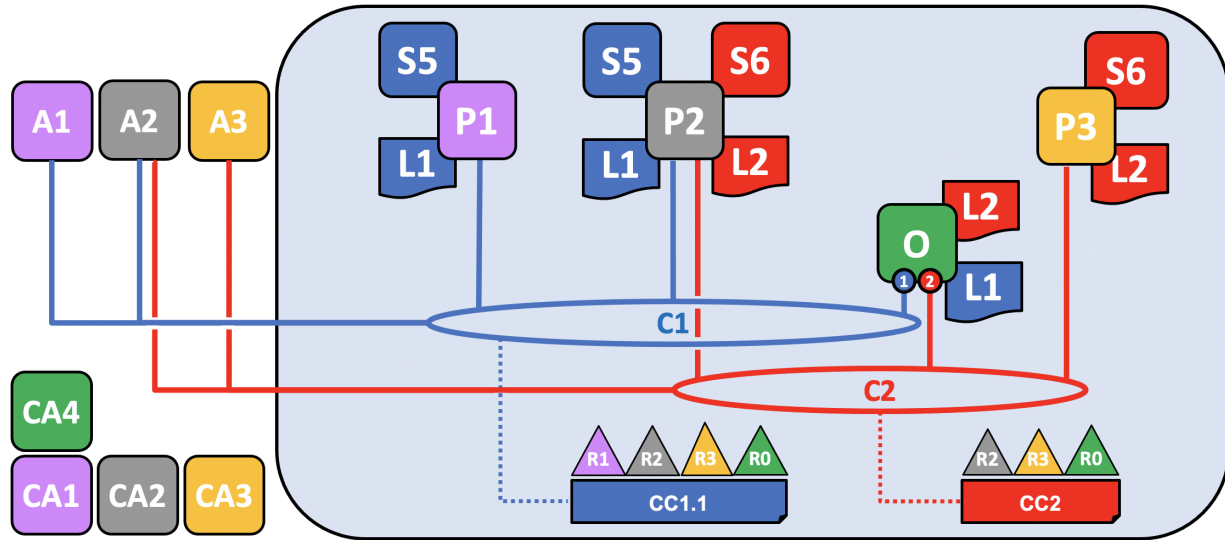
Think about this configuration from the standpoint of R2, which is joined to both channels. From their perspective, they might think about both C1 and C2, as well as the components they have joined to both, as the “network”, even though both channels are distinct from each other. In this sense, a “network” can also be seen as existing within the perspective of a particular organization as “all of the channels I am a member of and all of the components I own”.

Now that we have shown how organizations and their components can be joined to multiple channels, let's talk about how an organization and its components are added to an existing channel.

4.3.8 Adding an organization to an existing channel

As channels mature, it is natural that its configuration will also mature, reflecting changes in the world that must be reflected in the channel. One of the more common ways a channel will be modified is to add new organizations to it. While it is also possible to add more orderer organizations (who may or may not contribute their own nodes), in this example we'll describe the process of how a peer organization, R3, is added to the channel configuration CC1 of channel C1.

Note that rights and permissions are defined at a channel level. Just because an organization is an administrator of one channel does not mean it will be an administrator of a different channel. Each channel is a distinct administrative zone and fully customizable to the use case it's serving.



Although the update to the diagram looks like one simple step, adding a new organization to a channel is, at a high level, a three step process:

1. Decide on the new organization's permissions and role. The full scope of these rights must be agreed to before R3 is added to C1 and is beyond the scope of this topic, but comprise the same kinds of questions that must be answered when creating a channel in the first place. What kind of permissions and rights will R3 have on C1? Will it be an admin on the channel? Will its access to any channel resources be restricted (for example, R3 might only be able to write to C1, which means it can propose changes but not sign them)? What chaincodes will R3 install on its peers?
2. Update the channel, including the relevant chaincodes, to reflect these decisions.
3. The organization joins its peer nodes (and potentially ordering nodes) to the channel and begins participating.

In this topic, we'll assume that R3 will join C1 with the same rights and status enjoyed by R1 and R2. Similarly, R3 will also be joined as an endorser of the S5 chaincode, which means that R1 or R2 must redefine S5 (specifically, the endorsement policy section of the chaincode definition) and approve it on the channel.

Updating the channel configuration creates a new configuration block, CC1.1, which will serve as the channel configuration until it is updated again. Note that even though the configuration has changed, the channel still exists and P1 and P2 are still joined to it. There is no need to re-add organizations or peers to the channel.

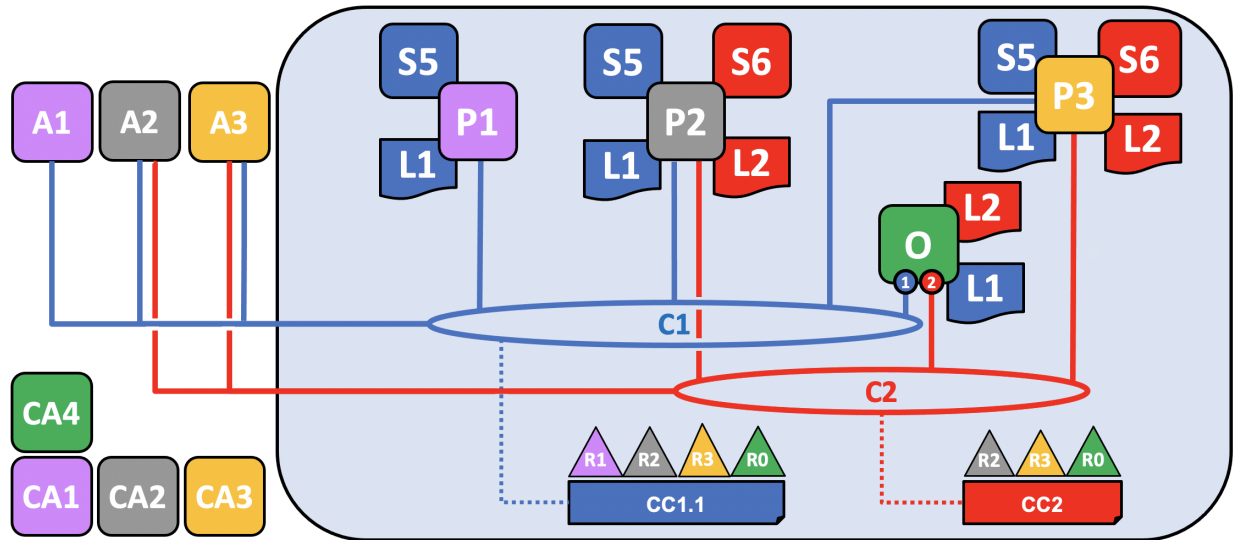
For more information about the process of adding an organization to a channel, check out [Adding an org to a channel](#).

For more information about policies (which define the roles organizations have on a channel), check out [Policies](#).

For more information about upgrading a chaincode, check out [Upgrade a chaincode](#).

Adding existing components to the newly joined channel

Now that R3 is able to fully participate in channel C2, it can add its components to the channel. Rather than do this one component at a time, let's show how its peer, its local copy of a ledger, a smart contract and a client application can be joined all at once!



In this example, R3 adds P3, which was previously joined to C2, to C1. When it does this, P3 pulls C1's ledger, L1. As we mentioned in the previous section, R3 has been added to C1 with equivalent rights as R1 and R2. Similarly, because the chaincode S5 was redefined and reapproved on the channel to include R3, R3 can now install S5 and begin transacting. Just as R2 modified its application A2 to be able to be used with channel C2, A3 is also now able to invoke transactions on C1.

4.3.9 Network recap

We've covered a lot of ground in this topic. We've gone from a simple configuration with two organizations transacting on a single channel to multiple organizations transacting on multiple channels as well as the process for joining an organization to a channel that already exists.

While this topic represents a relatively simple case, there are endless combinations of sophisticated topologies which are possible to achieve in Fabric, supporting an endless number of operational goals, and no theoretical limit to how big a network can get. The careful use of network and channel policies allow even large networks to be well-governed.

4.4 Identity

4.4.1 What is an Identity?

The different actors in a blockchain network include peers, orderers, client applications, administrators and more. Each of these actors — active elements inside or outside a network able to consume services — has a digital identity encapsulated in an X.509 digital certificate. These identities really matter because they **determine the exact permissions over resources and access to information that actors have in a blockchain network**.

A digital identity furthermore has some additional attributes that Fabric uses to determine permissions, and it gives the union of an identity and the associated attributes a special name — **principal**. Principals are just like userIDs or groupIDs, but a little more flexible because they can include a wide range of properties of an actor's identity, such as the actor's organization, organizational unit, role or even the actor's specific identity. When we talk about principals, they are the properties which determine their permissions.

For an identity to be **verifiable**, it must come from a **trusted** authority. A **membership service provider** (MSP) is that trusted authority in Fabric. More specifically, an MSP is a component that defines the rules that govern the

valid identities for this organization. The default MSP implementation in Fabric uses X.509 certificates as identities, adopting a traditional Public Key Infrastructure (PKI) hierarchical model (more on PKI later).

4.4.2 A Simple Scenario to Explain the Use of an Identity

Imagine that you visit a supermarket to buy some groceries. At the checkout you see a sign that says that only Visa, Mastercard and AMEX cards are accepted. If you try to pay with a different card — let’s call it an “ImagineCard” — it doesn’t matter whether the card is authentic and you have sufficient funds in your account. It will be not be accepted.



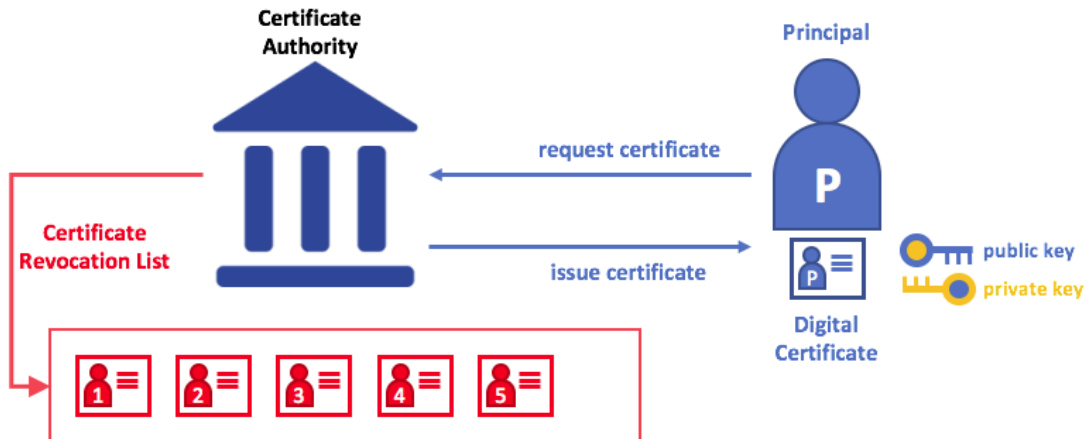
Having a valid credit card is not enough — it must also be accepted by the store! PKIs and MSPs work together in the same way — a PKI provides a list of identities, and an MSP says which of these are members of a given organization that participates in the network.

PKI certificate authorities and MSPs provide a similar combination of functionalities. A PKI is like a card provider — it dispenses many different types of verifiable identities. An MSP, on the other hand, is like the list of card providers accepted by the store, determining which identities are the trusted members (actors) of the store payment network. **MSPs turn verifiable identities into the members of a blockchain network.**

Let’s drill into these concepts in a little more detail.

4.4.3 What are PKIs?

A public key infrastructure (PKI) is a collection of internet technologies that provides secure communications in a network. It’s PKI that puts the **S** in **HTTPS** — and if you’re reading this documentation on a web browser, you’re probably using a PKI to make sure it comes from a verified source.



The elements of Public Key Infrastructure (PKI). A PKI is comprised of Certificate Authorities who issue digital certificates to parties (e.g., users of a service, service provider), who then use them to authenticate themselves in the messages they exchange in their environment. A CA's Certificate Revocation List (CRL) constitutes a reference for the certificates that are no longer valid. Revocation of a certificate can happen for a number of reasons. For example, a certificate may be revoked because the cryptographic private material associated to the certificate has been exposed.

Although a blockchain network is more than a communications network, it relies on the PKI standard to ensure secure communication between various network participants, and to ensure that messages posted on the blockchain are properly authenticated. It's therefore important to understand the basics of PKI and then why MSPs are so important.

There are four key elements to PKI:

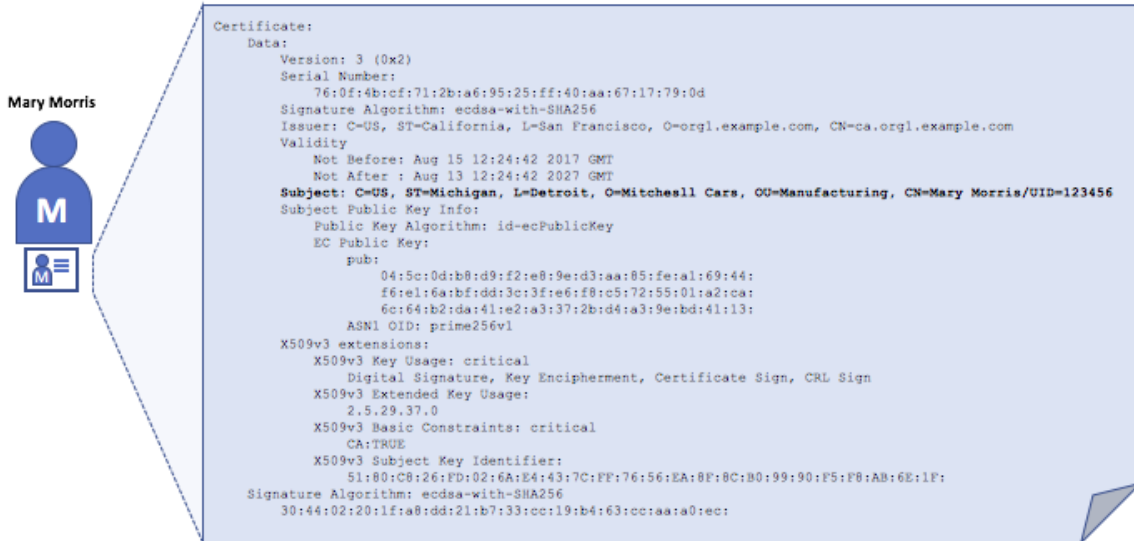
- **Digital Certificates**
- **Public and Private Keys**
- **Certificate Authorities**
- **Certificate Revocation Lists**

Let's quickly describe these PKI basics, and if you want to know more details, [Wikipedia](#) is a good place to start.

4.4.4 Digital Certificates

A digital certificate is a document which holds a set of attributes relating to the holder of the certificate. The most common type of certificate is the one compliant with the [X.509 standard](#), which allows the encoding of a party's identifying details in its structure.

For example, Mary Morris in the Manufacturing Division of Mitchell Cars in Detroit, Michigan might have a digital certificate with a SUBJECT attribute of C=US, ST=Michigan, L=Detroit, O=Mitchell Cars, OU=Manufacturing, CN=Mary Morris /UID=123456. Mary's certificate is similar to her government identity card — it provides information about Mary which she can use to prove key facts about her. There are many other attributes in an X.509 certificate, but let's concentrate on just these for now.



A digital certificate describing a party called Mary Morris. Mary is the **SUBJECT** of the certificate, and the highlighted **SUBJECT** text shows key facts about Mary. The certificate also holds many more pieces of information, as you can see. Most importantly, Mary’s public key is distributed within her certificate, whereas her private signing key is not. This signing key must be kept private.

What is important is that all of Mary’s attributes can be recorded using a mathematical technique called cryptography (literally, “*secret writing*”) so that tampering will invalidate the certificate. Cryptography allows Mary to present her certificate to others to prove her identity so long as the other party trusts the certificate issuer, known as a **Certificate Authority** (CA). As long as the CA keeps certain cryptographic information securely (meaning, its own **private signing key**), anyone reading the certificate can be sure that the information about Mary has not been tampered with — it will always have those particular attributes for Mary Morris. Think of Mary’s X.509 certificate as a digital identity card that is impossible to change.

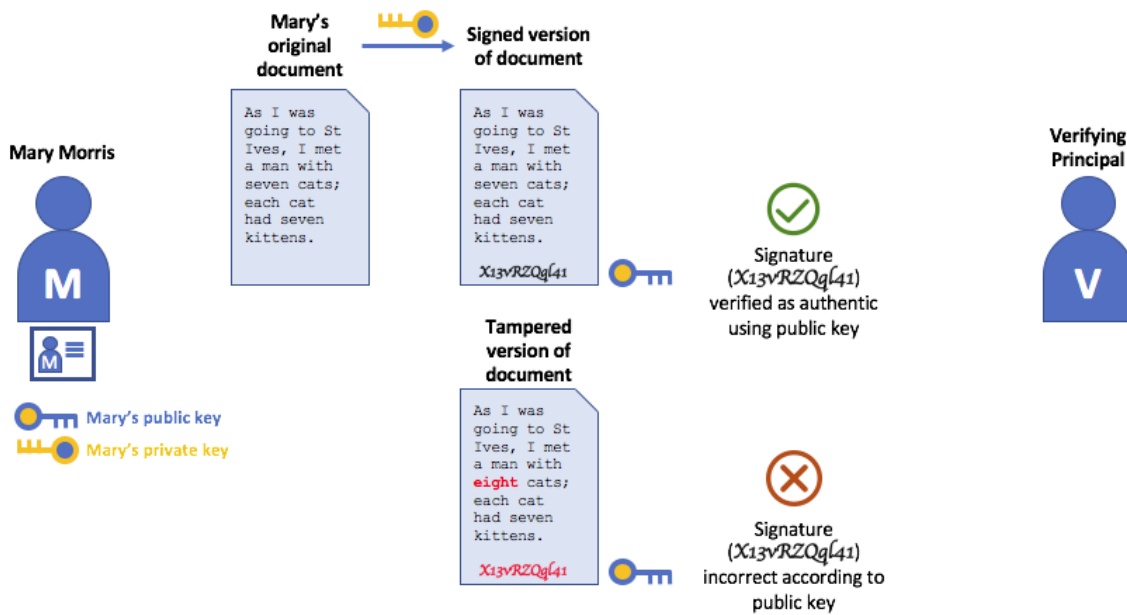
4.4.5 Authentication, Public keys, and Private Keys

Authentication and message integrity are important concepts in secure communications. Authentication requires that parties who exchange messages are assured of the identity that created a specific message. For a message to have “integrity” means that cannot have been modified during its transmission. For example, you might want to be sure you’re communicating with the real Mary Morris rather than an impersonator. Or if Mary has sent you a message, you might want to be sure that it hasn’t been tampered with by anyone else during transmission.

Traditional authentication mechanisms rely on **digital signatures** that, as the name suggests, allow a party to digitally **sign** its messages. Digital signatures also provide guarantees on the integrity of the signed message.

Technically speaking, digital signature mechanisms require each party to hold two cryptographically connected keys: a public key that is made widely available and acts as authentication anchor, and a private key that is used to produce **digital signatures** on messages. Recipients of digitally signed messages can verify the origin and integrity of a received message by checking that the attached signature is valid under the public key of the expected sender.

The unique relationship between a private key and the respective public key is the cryptographic magic that makes secure communications possible. The unique mathematical relationship between the keys is such that the private key can be used to produce a signature on a message that only the corresponding public key can match, and only on the same message.

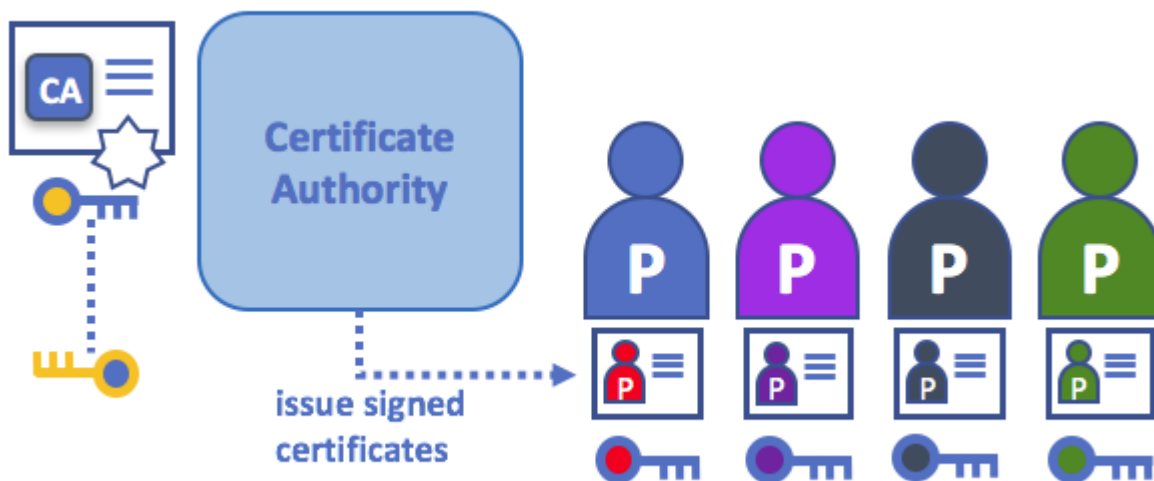


In the example above, Mary uses her private key to sign the message. The signature can be verified by anyone who sees the signed message using her public key.

4.4.6 Certificate Authorities

As you've seen, an actor or a node is able to participate in the blockchain network, via the means of a **digital identity** issued for it by an authority trusted by the system. In the most common case, digital identities (or simply **identities**) have the form of cryptographically validated digital certificates that comply with X.509 standard and are issued by a Certificate Authority (CA).

CAs are a common part of internet security protocols, and you've probably heard of some of the more popular ones: Symantec (originally Verisign), GeoTrust, DigiCert, GoDaddy, and Comodo, among others.



A Certificate Authority dispenses certificates to different actors. These certificates are digitally signed by the CA and

bind together the actor with the actor's public key (and optionally with a comprehensive list of properties). As a result, if one trusts the CA (and knows its public key), it can trust that the specific actor is bound to the public key included in the certificate, and owns the included attributes, by validating the CA's signature on the actor's certificate.

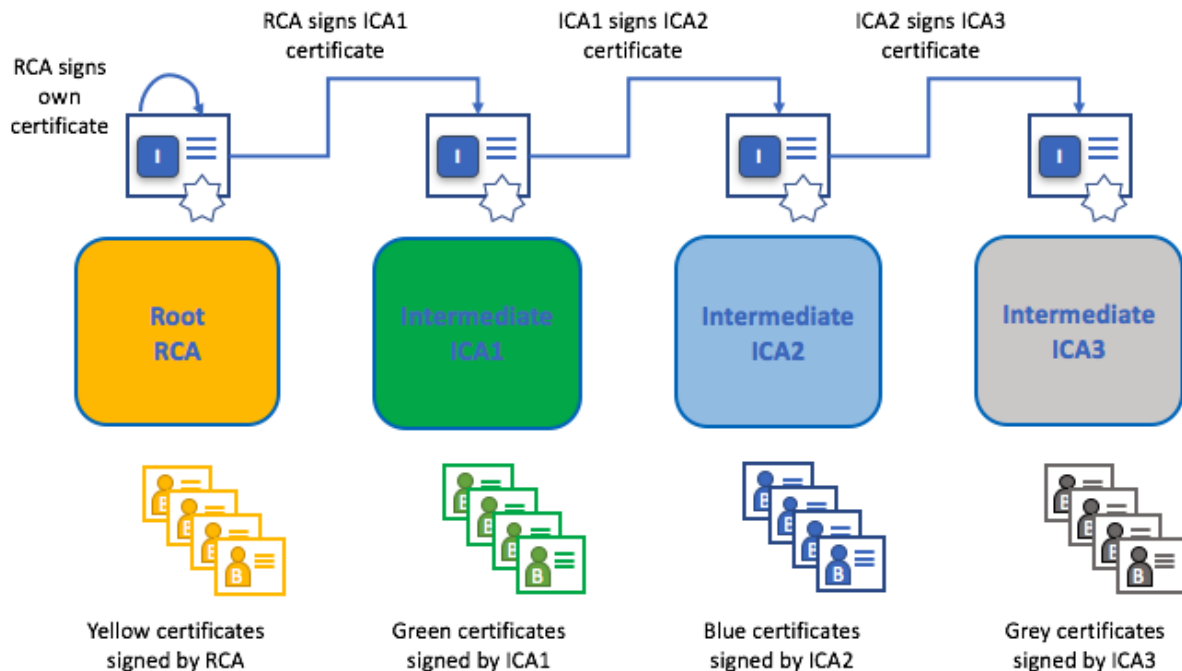
Certificates can be widely disseminated, as they do not include either the actors' nor the CA's private keys. As such they can be used as anchor of trusts for authenticating messages coming from different actors.

CAs also have a certificate, which they make widely available. This allows the consumers of identities issued by a given CA to verify them by checking that the certificate could only have been generated by the holder of the corresponding private key (the CA).

In a blockchain setting, every actor who wishes to interact with the network needs an identity. In this setting, you might say that **one or more CAs** can be used to **define the members of an organization's from a digital perspective**. It's the CA that provides the basis for an organization's actors to have a verifiable digital identity.

Root CAs, Intermediate CAs and Chains of Trust

CAs come in two flavors: **Root CAs** and **Intermediate CAs**. Because Root CAs (Symantec, Geotrust, etc) have to **securely distribute** hundreds of millions of certificates to internet users, it makes sense to spread this process out across what are called *Intermediate CAs*. These Intermediate CAs have their certificates issued by the root CA or another intermediate authority, allowing the establishment of a "chain of trust" for any certificate that is issued by any CA in the chain. This ability to track back to the Root CA not only allows the function of CAs to scale while still providing security — allowing organizations that consume certificates to use Intermediate CAs with confidence — it limits the exposure of the Root CA, which, if compromised, would endanger the entire chain of trust. If an Intermediate CA is compromised, on the other hand, there will be a much smaller exposure.



A chain of trust is established between a Root CA and a set of Intermediate CAs as long as the issuing CA for the certificate of each of these Intermediate CAs is either the Root CA itself or has a chain of trust to the Root CA.

Intermediate CAs provide a huge amount of flexibility when it comes to the issuance of certificates across multiple organizations, and that's very helpful in a permissioned blockchain system (like Fabric). For example, you'll see that different organizations may use different Root CAs, or the same Root CA with different Intermediate CAs — it really does depend on the needs of the network.

Fabric CA

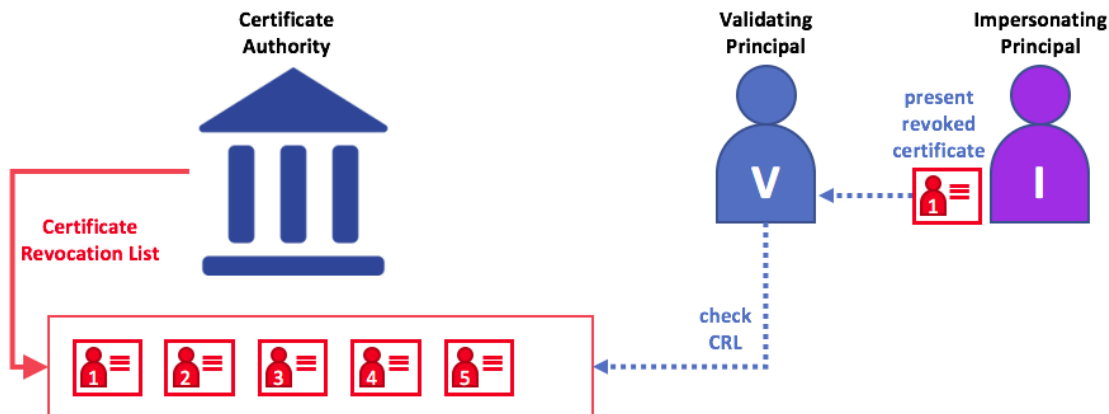
It's because CAs are so important that Fabric provides a built-in CA component to allow you to create CAs in the blockchain networks you form. This component — known as **Fabric CA** is a private root CA provider capable of managing digital identities of Fabric participants that have the form of X.509 certificates. Because Fabric CA is a custom CA targeting the Root CA needs of Fabric, it is inherently not capable of providing SSL certificates for general/automatic use in browsers. However, because **some** CA must be used to manage identity (even in a test environment), Fabric CA can be used to provide and manage certificates. It is also possible — and fully appropriate — to use a public/commercial root or intermediate CA to provide identification.

If you're interested, you can read a lot more about Fabric CA in the [CA documentation section](#).

4.4.7 Certificate Revocation Lists

A Certificate Revocation List (CRL) is easy to understand — it's just a list of references to certificates that a CA knows to be revoked for one reason or another. If you recall the store scenario, a CRL would be like a list of stolen credit cards.

When a third party wants to verify another party's identity, it first checks the issuing CA's CRL to make sure that the certificate has not been revoked. A verifier doesn't have to check the CRL, but if they don't they run the risk of accepting a compromised identity.



Using a CRL to check that a certificate is still valid. If an impersonator tries to pass a compromised digital certificate to a validating party, it can be first checked against the issuing CA's CRL to make sure it's not listed as no longer valid.

Note that a certificate being revoked is very different from a certificate expiring. Revoked certificates have not expired — they are, by every other measure, a fully valid certificate. For more in-depth information about CRLs, click [here](#).

Now that you've seen how a PKI can provide verifiable identities through a chain of trust, the next step is to see how these identities can be used to represent the trusted members of a blockchain network. That's where a Membership Service Provider (MSP) comes into play — **it identifies the parties who are the members of a given organization in the blockchain network**.

To learn more about membership, check out the conceptual documentation on [MSPs](#).

4.5 Membership Service Provider (MSP)

Note: this topic describes a network that does not use a “system channel”, a channel that the ordering service is bootstrapped with and the ordering service exclusively controls. Since the release of v2.3, using system channel is now considered the legacy process as compared to the process to [Create a channel](#) without a system channel. For a version of this topic that includes information about the system channel, check out [Membership Service Provider \(MSP\)](#) from the v2.2 documentation.

4.5.1 Why do I need an MSP?

Because Fabric is a permissioned network, blockchain participants need a way to prove their identity to the rest of the network in order to transact on the network. If you’ve read through the documentation on [Identity](#) you’ve seen how a Public Key Infrastructure (PKI) can provide verifiable identities through a chain of trust. How is that chain of trust used by the blockchain network?

Certificate Authorities issue identities by generating a public and private key which forms a key-pair that can be used to prove identity. Because a private key can never be shared publicly, a mechanism is required to enable that proof which is where the MSP comes in. For example, a peer uses its private key to digitally sign, or endorse, a transaction. The MSP on the ordering service contains the peer’s public key which is then used to verify that the signature attached to the transaction is valid. The private key is used to produce a signature on a transaction that only the corresponding public key, that is part of an MSP, can match. Thus, the MSP is the mechanism that allows that identity to be trusted and recognized by the rest of the network without ever revealing the member’s private key.

Recall from the credit card scenario in the Identity topic that the Certificate Authority is like a card provider — it dispenses many different types of verifiable identities. An MSP, on the other hand, determines which credit card providers are accepted at the store. In this way, the MSP turns an identity (the credit card) into a role (the ability to buy things at the store).

This ability to turn verifiable identities into roles is fundamental to the way Fabric networks function, since it allows organizations, nodes, and channels the ability establish MSPs that determine who is allowed to do what at the organization, node, and channel level.



Identities are similar to your credit cards that are used to prove you can pay. The MSP is similar to the list of accepted credit cards.

Consider a group of banks that operate a blockchain network. Each bank operates peer and ordering nodes, and the peers endorse transactions submitted to the network. However, each bank would also have departments and account holders. The account holders would belong to each organization, but would not run nodes on the network. They would only interact with the system from their mobile or web application. So how does the network recognize and differentiate these identities? A CA was used to create the identities, but like the card example, those identities can’t

just be issued, they need to be recognized by the network. MSPs are used to define the organizations that are trusted by the network members. MSPs are also the mechanism that provide members with a set of roles and permissions within the network. Because the MSPs defining these organizations are known to the members of a network, they can then be used to validate that network entities that attempt to perform actions are allowed to.

Finally, consider if you want to join an *existing* network, you need a way to turn your identity into something that is recognized by the network. The MSP is the mechanism that enables you to participate on a permissioned blockchain network. To transact on a Fabric network a member needs to:

1. Have an identity issued by a CA that is trusted by the network.
2. Become a member of an *organization* that is recognized and approved by the network members. The MSP is how the identity is linked to the membership of an organization. Membership is achieved by adding the member's public key (also known as certificate, signing cert, or signcert) to the organization's MSP.
3. Add the MSP to a channel.
4. Ensure the MSP is included in the [policy](#) definitions on the network.

4.5.2 What is an MSP?

Despite its name, the Membership Service Provider does not actually provide anything. Rather, the implementation of the MSP requirement is a set of folders that are added to the configuration of the network and is used to define an organization both inwardly (organizations decide who its admins are) and outwardly (by allowing other organizations to validate that entities have the authority to do what they are attempting to do). Whereas Certificate Authorities generate the certificates that represent identities, the MSP contains a list of permissioned identities.

The MSP identifies which Root CAs and Intermediate CAs are accepted to define the members of a trust domain by listing the identities of their members, or by identifying which CAs are authorized to issue valid identities for their members.

But the power of an MSP goes beyond simply listing who is a network participant or member of a channel. It is the MSP that turns an identity into a **role** by identifying specific privileges an actor has on a node or channel. Note that when a user is registered with a Fabric CA, a role of admin, peer, client, orderer, or member must be associated with the user. For example, identities registered with the “peer” role should, naturally, be given to a peer. Similarly, identities registered with the “admin” role should be given to organization admins. We'll delve more into the significance of these roles later in the topic.

In addition, an MSP can allow for the identification of a list of identities that have been revoked — as discussed in the [Identity](#) documentation — but we will talk about how that process also extends to an MSP.

4.5.3 MSP domains

MSPs occur in two domains in a blockchain network:

- Locally on an actor's node (**local MSP**)
- In channel configuration (**channel MSP**)

The key difference between local and channel MSPs is not how they function – both turn identities into roles – but their **scope**. Each MSP lists roles and permissions at a particular level of administration.

Local MSPs

Local MSPs are defined for clients and for nodes (peers and orderers). Local MSPs define the permissions for a node (who are the peer admins who can operate the node, for example). The local MSPs of clients (the account holders in the banking scenario above), allow the user to authenticate itself in its transactions as a member of a channel (e.g. in

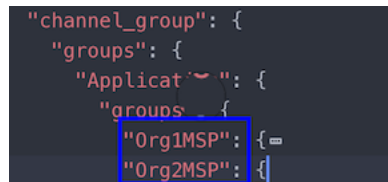
chaincode transactions), or as the owner of a specific role into the system such as an organization admin, for example, in configuration transactions.

Every node must have a local MSP defined, as it defines who has administrative or participatory rights at that level (peer admins will not necessarily be channel admins, and vice versa). This allows for authenticating member messages outside the context of a channel and to define the permissions over a particular node (who has the ability to install chaincode on a peer, for example). Note that one or more nodes can be owned by an organization. An MSP defines the organization admins. And the organization, the admin of the organization, the admin of the node, and the node itself should all have the same root of trust.

An orderer local MSP is also defined on the file system of the node and only applies to that node. Like peer nodes, orderers are also owned by a single organization and therefore have a single MSP to list the actors or nodes it trusts.

Channel MSPs

In contrast, **channel MSPs define administrative and participatory rights at the channel level**. Peers and ordering nodes on an application channel share the same view of channel MSPs, and will therefore be able to correctly authenticate the channel participants. This means that if an organization wishes to join the channel, an MSP incorporating the chain of trust for the organization's members would need to be included in the channel configuration. Otherwise transactions originating from this organization's identities will be rejected. Whereas local MSPs are represented as a folder structure on the file system, channel MSPs are described in a channel configuration.



```
"channel_group": {
  "groups": {
    "Application": {
      "groups": {
        "Org1MSP": {
          ...
        },
        "Org2MSP": {
          ...
        }
      }
    }
  }
}
```

Snippet from a channel config.json file that includes two organization MSPs.

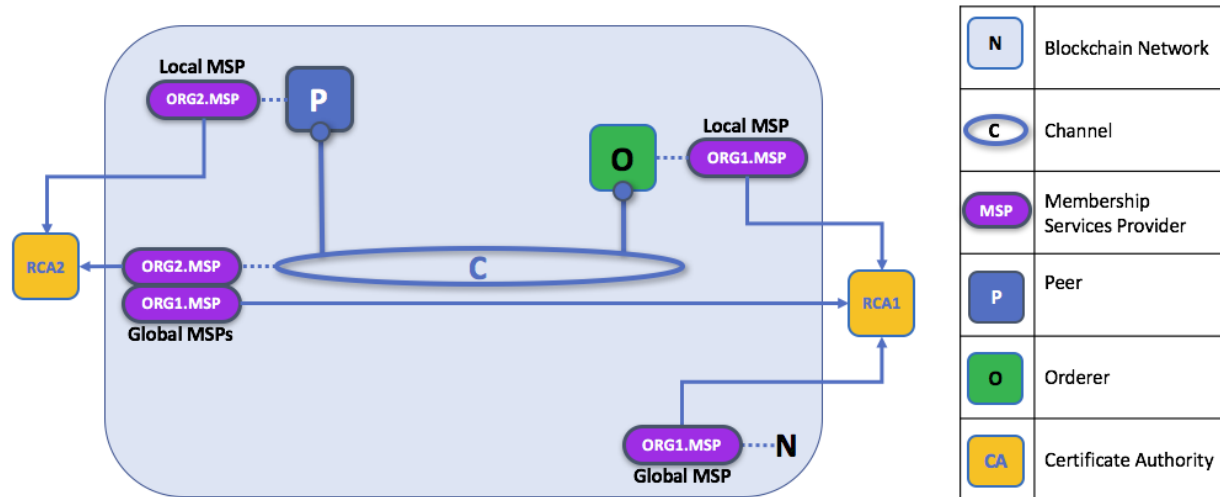
Channel MSPs identify who has authorities at a channel level. The channel MSP defines the *relationship* between the identities of channel members (which themselves are MSPs) and the enforcement of channel level policies. Channel MSPs contain the MSPs of the organizations of the channel members.

Every organization participating in a channel must have an MSP defined for it. In fact, it is recommended that there is a one-to-one mapping between organizations and MSPs. The MSP defines which members are empowered to act on behalf of the organization. This includes configuration of the MSP itself as well as approving administrative tasks that the organization has role, such as adding new members to a channel. If all network members were part of a single organization or MSP, data privacy is sacrificed. Multiple organizations facilitate privacy by segregating ledger data to only channel members. If more granularity is required within an organization, the organization can be further divided into organizational units (OUs) which we describe in more detail later in this topic.

The channel MSP includes the MSPs of all the organizations on a channel. This includes not just “peer organizations”, which own peers and invoke chaincodes, but the organizations that own and run the ordering service.

Local MSPs are only defined on the file system of the node or user to which they apply. Therefore, physically and logically there is only one local MSP per node. However, as channel MSPs are available to all nodes in the channel, they are logically defined once in the channel configuration. However, **a channel MSP is also instantiated on the file system of every node in the channel and kept synchronized via consensus**. So while there is a copy of each channel MSP on the local file system of every node, logically a channel MSP resides on and is maintained by the channel or the network.

The following diagram illustrates how local and channel MSPs coexist on the network:



In this figure, ORG1, owns the ordering node joined to the channel. The MSPs related to ORG1, the local MSP of the node and the global MSP that formally represents ORG1 on the channel, have been created by RCA1, the CA for ORG1. The peer organization, ORG2, also has a local MSP for its peer and another global MSP that represents ORG2 on the channel. Both ORG1 and ORG2 are channel members, and manage the channel in their areas of administration, and trust identities created by each other's CA. Note that in a production scenario, it is likely that there will be several peer organizations who collaborate in the administration of the channel, and potentially more than one orderer organization as well.

4.5.4 What role does an organization play in an MSP?

An **organization** is a logical managed group of members. This can be something as big as a multinational corporation or as small as a flower shop. What's most important about organizations (or **orgs**) is that they manage their members under a single MSP. The MSP allows an identity to be linked to an organization. Note that this is different from the organization concept defined in an X.509 certificate, which we mentioned above.

The exclusive relationship between an organization and its MSP makes it sensible to name the MSP after the organization, a convention you'll find adopted in most policy configurations. For example, organization ORG1 would likely have an MSP called something like ORG1-MSP. In some cases an organization may require multiple membership groups — for example, where channels are used to perform very different business functions between organizations. In these cases it makes sense to have multiple MSPs and name them accordingly, e.g., ORG2-MSP-NATIONAL and ORG2-MSP-GOVERNMENT, reflecting the different membership roots of trust within ORG2 in the NATIONAL sales channel compared to the GOVERNMENT regulatory channel.

Organizational Units (OUs) and MSPs

An organization can also be divided into multiple **organizational units**, each of which has a certain set of responsibilities, also referred to as **affiliations**. Think of an OU as a department inside an organization. For example, the ORG1 organization might have both ORG1.MANUFACTURING and ORG1.DISTRIBUTION OUs to reflect these separate lines of business. When a CA issues X.509 certificates, the OU field in the certificate specifies the line of business to which the identity belongs. A benefit of using OUs like this is that these values can then be used in policy definitions in order to restrict access or in smart contracts for attribute-based access control. Otherwise, separate MSPs would need to be created for each organization.

Specifying OUs is optional. If OUs are not used, all of the identities that are part of an MSP — as identified by the Root CA and Intermediate CA folders — will be considered members of the organization.

Node OU Roles and MSPs

Additionally, there is a special kind of OU, sometimes referred to as a `Node OU`, that can be used to confer a role onto an identity. These Node OU roles are defined in the `$FABRIC_CFG_PATH/msp/config.yaml` file and contain a list of organizational units whose members are considered to be part of the organization represented by this MSP. This is particularly useful when you want to restrict the members of an organization to the ones holding an identity (signed by one of MSP designated CAs) with a specific Node OU role in it. For example, with node OU's you can implement a more granular endorsement policy that requires `Org1` peers to endorse a transaction, rather than any member of `Org1`.

In order to use the Node OU roles, the “identity classification” feature must be enabled for the network. When using the folder-based MSP structure, this is accomplished by enabling “Node OUs” in the `config.yaml` file which resides in the root of the MSP folder:

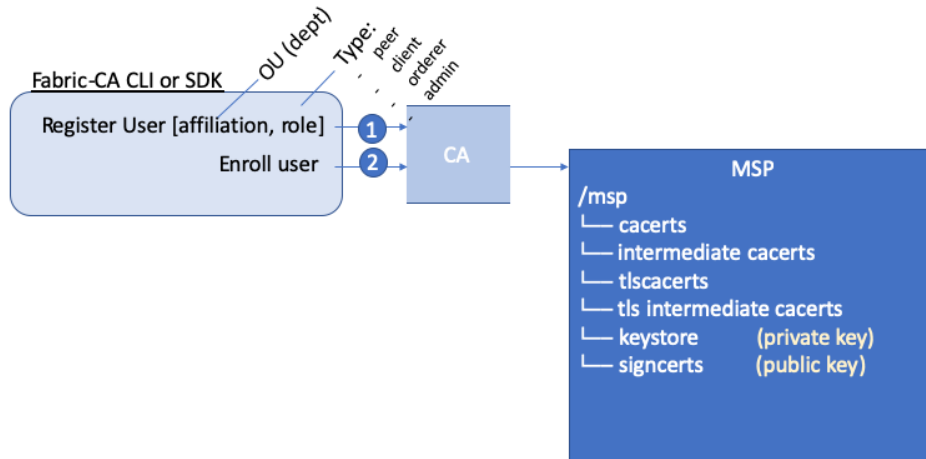
```
NodeOUs:
  Enable: true
  ClientOUIdentifier:
    Certificate: cacerts/ca.sampleorg-cert.pem
    OrganizationalUnitIdentifier: client
  PeerOUIdentifier:
    Certificate: cacerts/ca.sampleorg-cert.pem
    OrganizationalUnitIdentifier: peer
  AdminOUIdentifier:
    Certificate: cacerts/ca.sampleorg-cert.pem
    OrganizationalUnitIdentifier: admin
  OrdererOUIdentifier:
    Certificate: cacerts/ca.sampleorg-cert.pem
    OrganizationalUnitIdentifier: orderer
```

In the example above, there are 4 possible Node OU ROLES for the MSP:

- client
- peer
- admin
- orderer

This convention allows you to distinguish MSP roles by the OU present in the `CommonName` attribute of the X509 certificate. The example above says that any certificate issued by `cacerts/ca.sampleorg-cert.pem` in which `OU=client` will be identified as a client, `OU=peer` as a peer, etc. Starting with Fabric v1.4.3, there is also an OU for the orderer and for admins. The new `admins` role means that you no longer have to explicitly place certs in the `admincerts` folder of the MSP directory. Rather, the `admin` role present in the user's `signcert` qualifies the identity as an admin user.

These Role and OU attributes are assigned to an identity when the Fabric CA or SDK is used to `register` a user with the CA. It is the subsequent `enroll` user command that generates the certificates in the users' `/msp` folder.



The resulting `ROLE` and `OU` attributes are visible inside the X.509 signing certificate located in the `/signcerts` folder. The `ROLE` attribute is identified as `hf.Type` and refers to an actor's role within its organization, (specifying, for example, that an actor is a `peer`). See the following snippet from a signing certificate shows how the Roles and OUs are represented in the certificate.

```

Certificate:
  Data
    Version: 3 (0x2)
    Serial Number:
      45:6a:4f:01:dc:fj:5d:b2:94:18:79:91:26:31:d8:0e:b0:9b:6b:88
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: C=US, ST=New York, O=Hyperledger, OU=Fabric, CN=fabric-ca-server
    Validity
      Not Before: Nov 20 22:13:00 2019 GMT
      Not After : Nov 19 22:18:00 2020 GMT
    Subject: OU=peer, OU=ORG1, OU=DISTRIBUTION, CN=user1
           ROLE ORGANIZATIONAL UNIT ENROLL ID
           (Node OU)
    .
    .
    X509v3 extensions:
      X509v3 Key Usage: critical
        Digital Signature
      X509v3 Basic Constraints: critical
        CA:FALSE
      X509v3 Subject Key Identifier:
        17:B0:9B:29:42:F6:44:E0:7D:02:C6:78:96:2D:97:14:7A:D7:FC:CA
      X509v3 Authority Key Identifier:
        keyid:DC:91:B7:85:A4:37:66:D0:D2:B7:62:A9:3F:59:83:D6:EB:01:E8:80
    1.2.3.4.5.6.7.8.1:
           ORGANIZATIONAL UNIT ENROLL ID ROLE (Node OU)
    {"attrs":{"hf.Affiliation":"ORG1.DISTRIBUTION","hf.EnrollmentID":"user1","hf.Type":"peer"}}
  
```

Note: For Channel MSPs, just because an actor has the role of an administrator it doesn't mean that they can administer particular resources. The actual power a given identity has with respect to administering the system is determined by the *policies* that manage system resources. For example, a channel policy might specify that `ORG1-MANUFACTURING` administrators, meaning identities with a role of `admin` and a

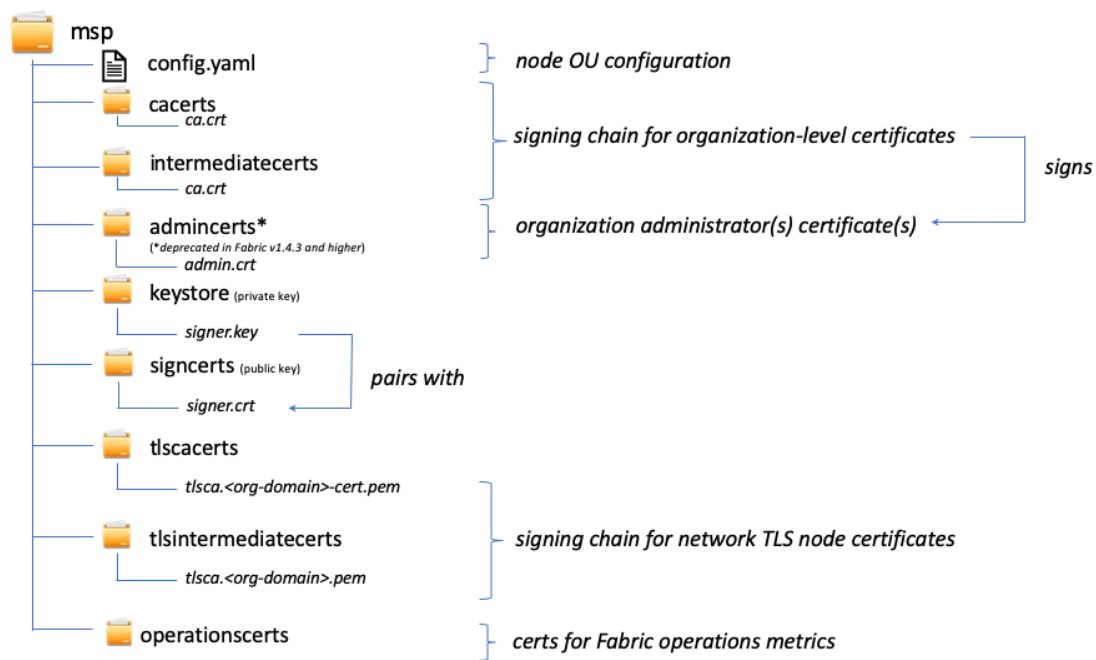
Node OU of `ORG1-MANUFACTURING`, have the rights to add new organizations to the channel, whereas the `ORG1-DISTRIBUTION` administrators have no such rights.

Finally, OUs could be used by different organizations to distinguish each other. But in such cases, the different organizations have to use the same Root CAs and Intermediate CAs for their chain of trust, and assign the OU field to identify members of each organization. When every organization has the same CA or chain of trust, this makes the system more centralized than what might be desirable and therefore deserves careful consideration on a blockchain network.

4.5.5 MSP Structure

Let's explore the MSP elements that render the functionality we've described so far.

A local MSP folder contains the following sub-folders:



The figure above shows the subfolders in a local MSP on the file system

- **config.yaml**: Used to configure the identity classification feature in Fabric by enabling “Node OUs” and defining the accepted roles.
- **cacerts**: This folder contains a list of self-signed X.509 certificates of the Root CAs trusted by the organization represented by this MSP. There must be at least one Root CA certificate in this MSP folder.

This is the most important folder because it identifies the CAs from which all other certificates must be derived to be considered members of the corresponding organization to form the chain of trust.

- **intermediatecerts**: This folder contains a list of X.509 certificates of the Intermediate CAs trusted by this organization. Each certificate must be signed by one of the Root CAs in the MSP or by any Intermediate CA whose issuing CA chain ultimately leads back to a trusted Root CA.

An intermediate CA may represent a different subdivision of the organization (like `ORG1-MANUFACTURING` and `ORG1-DISTRIBUTION` do for `ORG1`), or the organization itself (as may be the case if a commercial CA is leveraged for the organization's identity management). In the latter case intermediate CAs can be used to represent organization subdivisions. [Here](#) you may find more information on best practices for MSP configuration.

Notice, that it is possible to have a functioning network that does not have an Intermediate CA, in which case this folder would be empty.

Like the Root CA folder, this folder defines the CAs from which certificates must be issued to be considered members of the organization.

- **admincerts (Deprecated from Fabric v1.4.3 and higher):** This folder contains a list of identities that define the actors who have the role of administrators for this organization. In general, there should be one or more X.509 certificates in this list.

Note: Prior to Fabric v1.4.3, admins were defined by explicitly putting certs in the `admincerts` folder in the local MSP directory of your peer. **With Fabric v1.4.3 or higher, certificates in this folder are no longer required.** Instead, it is recommended that when the user is registered with the CA, that the `admin` role is used to designate the node administrator. Then, the identity is recognized as an `admin` by the Node OU role value in their `signcert`. As a reminder, in order to leverage the `admin` role, the “identity classification” feature must be enabled in the `config.yaml` above by setting “Node OUs” to `Enable: true`. We’ll explore this more later.

And as a reminder, for Channel MSPs, just because an actor has the role of an administrator it doesn’t mean that they can administer particular resources. The actual power a given identity has with respect to administering the system is determined by the *policies* that manage system resources. For example, a channel policy might specify that `ORG1-MANUFACTURING` administrators have the rights to add new organizations to the channel, whereas the `ORG1-DISTRIBUTION` administrators have no such rights.

- **keystore: (private Key)** This folder is defined for the local MSP of a peer or orderer node (or in a client’s local MSP), and contains the node’s private key. This key is used to sign data — for example to sign a transaction proposal response, as part of the endorsement phase.

This folder is mandatory for local MSPs, and must contain exactly one private key. Obviously, access to this folder must be limited only to the identities of users who have administrative responsibility on the peer.

The **channel MSP** configuration does not include this folder, because channel MSPs solely aim to offer identity validation functionalities and not signing abilities.

Note: If you are using a [Hardware Security Module\(HSM\)](#) for key management, this folder is empty because the private key is generated by and stored in the HSM.

- **signcert:** For a peer or orderer node (or in a client’s local MSP) this folder contains the node’s certificate issued by CA. The certificate represents the node’s identity, and this certificate’s corresponding **private key** can be used to generate signatures which may be verified by anyone with a copy of this certificate.

This folder is mandatory for local MSPs, and must contain exactly one **public key**. Obviously, access to this folder must be limited only to the identities of users who have administrative responsibility on the peer.

Configuration of a **channel MSP** does not include this folder, as channel MSPs solely aim to offer identity validation functionalities and not signing abilities.

- **tlscacerts:** This folder contains a list of self-signed X.509 certificates of the Root CAs trusted by this organization **for secure communications between nodes using TLS**. An example of a TLS communication would be when a peer needs to connect to an orderer so that it can receive ledger updates.

MSP TLS information relates to the nodes inside the network — the peers and the orderers, in other words, rather than the applications and administrations that consume the network.

There must be at least one TLS Root CA certificate in this folder. For more information about TLS, see [Securing Communication with Transport Layer Security \(TLS\)](#).

- **tlsintermediatecacerts:** This folder contains a list intermediate CA certificates CAs trusted by the organization represented by this MSP **for secure communications between nodes using TLS**. This folder is specifically useful when commercial CAs are used for TLS certificates of an organization. Similar to membership intermediate CAs, specifying intermediate TLS CAs is optional.

- **operationscerts:** This folder contains the certificates required to communicate with the [Fabric Operations Service API](#).

A channel MSP includes the following additional folder:

- **Revoked Certificates:** If the identity of an actor has been revoked, identifying information about the identity — not the identity itself — is held in this folder. For X.509-based identities, these identifiers are pairs of strings known as Subject Key Identifier (SKI) and Authority Access Identifier (AKI), and are checked whenever the certificate is being used to make sure the certificate has not been revoked.

This list is conceptually the same as a CA’s Certificate Revocation List (CRL), but it also relates to revocation of membership from the organization. As a result, the administrator of a channel MSP can quickly revoke an actor or node from an organization by advertising the updated CRL of the CA. This “list of lists” is optional. It will only become populated as certificates are revoked.

If you’ve read this doc as well as our doc on [Identity](#), you should now have a pretty good grasp of how identities and MSPs work in Hyperledger Fabric. You’ve seen how a PKI and MSPs are used to identify the actors collaborating in a blockchain network. You’ve learned how certificates, public/private keys, and roots of trust work, in addition to how MSPs are physically and logically structured.

4.6 Policies

Audience: Architects, application and smart contract developers, administrators

In this topic, we’ll cover:

- *What is a policy*
- *Why are policies needed*
- *How are policies implemented*
- *How do you write a policy in Fabric*
- *Fabric chaincode lifecycle*
- *Overriding policy definitions*

Note: this topic describes a network that does not use a “system channel”, a channel that the ordering service is bootstrapped with and the ordering service exclusively controls. Since the release of v2.3, using system channel is now considered the legacy process as compared to the process to [Create a channel](#) without a system channel. For a version of this topic that includes information about the system channel, check out [Policies](#) from the v2.2 documentation.

4.6.1 What is a policy

At its most basic level, a policy is a set of rules that define the structure for how decisions are made and specific outcomes are reached. To that end, policies typically describe a **who** and a **what**, such as the access or rights that an individual has over an **asset**. We can see that policies are used throughout our daily lives to protect assets of value to us, from car rentals, health, our homes, and many more.

For example, an insurance policy defines the conditions, terms, limits, and expiration under which an insurance payout will be made. The policy is agreed to by the policy holder and the insurance company, and defines the rights and responsibilities of each party.

Whereas an insurance policy is put in place for risk management, in Hyperledger Fabric, policies are the mechanism for infrastructure management. Fabric policies represent how members come to agreement on accepting or rejecting changes to the network, a channel, or a smart contract. Policies are agreed to by the channel members when the channel is originally configured, but they can also be modified as the channel evolves. For example, they describe the

criteria for adding or removing members from a channel, change how blocks are formed, or specify the number of organizations required to endorse a smart contract. All of these actions are described by a policy which defines who can perform the action. Simply put, everything you want to do on a Fabric network is controlled by a policy.

4.6.2 Why are policies needed

Policies are one of the things that make Hyperledger Fabric different from other blockchains like Ethereum or Bitcoin. In those systems, transactions can be generated and validated by any node in the network. The policies that govern the network are fixed at any point in time and can only be changed using the same process that governs the code. Because Fabric is a permissioned blockchain whose users are recognized by the underlying infrastructure, those users have the ability to decide on the governance of the network before it is launched, and change the governance of a running network.

Policies allow members to decide which organizations can access or update a Fabric network, and provide the mechanism to enforce those decisions. Policies contain the lists of organizations that have access to a given resource, such as a user or system chaincode. They also specify how many organizations need to agree on a proposal to update a resource, such as a channel or smart contracts. Once they are written, policies evaluate the collection of signatures attached to transactions and proposals and validate if the signatures fulfill the governance agreed to by the network.

4.6.3 How are policies implemented

Policies are defined within the relevant administrative domain of a particular action defined by the policy. For example, the policy for adding a peer organization to a channel is defined within the administrative domain of the peer organizations (known as the `Application` group). Similarly, adding ordering nodes in the consenter set of the channel is controlled by a policy inside the `Orderer` group. Actions that cross both the peer and orderer organizational domains are contained in the `Channel` group.

Typically, these policies default to the “majority of admins” of the group they fall under (a majority of peer organization admins for example, or in the case of `Channel` policies, a majority of both peer organizations and orderer organizations), though they can be specified to any rule a user wishes to define. Check out [Signature policies](#) for more information.

Access control lists (ACLs)

Network administrators will be especially interested in the Fabric use of ACLs, which provide the ability to configure access to resources by associating those resources with existing policies. These “resources” could be functions on system chaincode (e.g., “`GetBlockByNumber`” on the “`qsc`” system chaincode) or other resources (e.g., who can receive Block events). ACLs refer to policies defined in an application channel configuration and extends them to control additional resources. The default set of Fabric ACLs is visible in the `configtx.yaml` file under the `Application: &ApplicationDefaults` section but they can and should be overridden in a production environment. The list of resources named in `configtx.yaml` is the complete set of all internal resources currently defined by Fabric.

In that file, ACLs are expressed using the following format:

```
# ACL policy for chaincode to chaincode invocation
peer/ChaincodeToChaincode: /Channel/Application/Writers
```

Where `peer/ChaincodeToChaincode` represents the resource being secured and `/Channel/Application/Writers` refers to the policy which must be satisfied for the associated transaction to be considered valid.

For a deeper dive into ACLs, refer to the topic in the Operations Guide on [ACLs](#).

Smart contract endorsement policies

Every smart contract inside a chaincode package has an endorsement policy that specifies how many peers belonging to different channel members need to execute and validate a transaction against a given smart contract in order for the transaction to be considered valid. Hence, the endorsement policies define the organizations (through their peers) who must “endorse” (i.e., approve of) the execution of a proposal.

Modification policies

There is one last type of policy that is crucial to how policies work in Fabric, the `Modification` policy. Modification policies specify the group of identities required to sign (approve) any configuration *update*. It is the policy that defines how the policy is updated. Thus, each channel configuration element includes a reference to a policy which governs its modification.

4.6.4 How do you write a policy in Fabric

If you want to change anything in Fabric, the policy associated with the resource describes **who** needs to approve it, either with an explicit sign off from individuals, or an implicit sign off by a group. In the insurance domain, an explicit sign off could be a single member of the homeowners insurance agents group. And an implicit sign off would be analogous to requiring approval from a majority of the managerial members of the homeowners insurance group. This is particularly useful because the members of that group can change over time without requiring that the policy be updated. In Hyperledger Fabric, explicit sign offs in policies are expressed using the `Signature` syntax and implicit sign offs use the `ImplicitMeta` syntax.

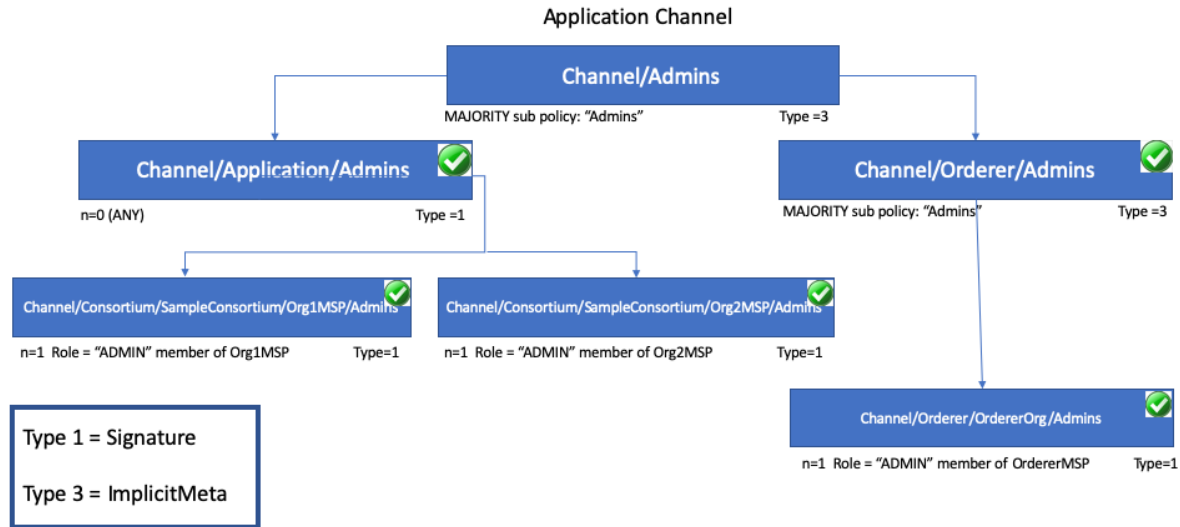
Signature policies

Signature policies define specific types of users who must sign in order for a policy to be satisfied such as `OR('Org1.peer', 'Org2.peer')`. These policies are considered the most versatile because they allow for the construction of extremely specific rules like: “An admin of org A and 2 other admins, or 5 of 6 organization admins”. The syntax supports arbitrary combinations of `AND`, `OR` and `NOutOf`. For example, a policy can be easily expressed by using `AND('Org1.member', 'Org2.member')` which means that a signature from at least one member in Org1 AND one member in Org2 is required for the policy to be satisfied.

ImplicitMeta policies

`ImplicitMeta` policies are only valid in the context of channel configuration which is based on a tiered hierarchy of policies in a configuration tree. `ImplicitMeta` policies aggregate the result of policies deeper in the configuration tree that are ultimately defined by `Signature` policies. They are `Implicit` because they are constructed implicitly based on the current organizations in the channel configuration, and they are `Meta` because their evaluation is not against specific MSP principals, but rather against other sub-policies below them in the configuration tree.

The following diagram illustrates the tiered policy structure for an application channel and shows how the `ImplicitMeta` channel configuration admins policy, named `/Channel/Admins`, is resolved when the sub-policies named `Admins` below it in the configuration hierarchy are satisfied where each check mark represents that the conditions of the sub-policy were satisfied.



In order for the Channel/Admins policy to be satisfied, every sub-policy under it in the configuration hierarchy must be satisfied.

As you can see in the diagram above, ImplicitMeta policies, Type = 3, use a different syntax, "<ANY|ALL|MAJORITY> <SubPolicyName>", for example:

```
`MAJORITY sub policy: Admins`
```

The diagram shows a sub-policy Admins, which refers to all the Admins policy below it in the configuration tree. You can create your own sub-policies and name them whatever you want and then define them in each of your organizations.

As mentioned above, a key benefit of an ImplicitMeta policy such as MAJORITY Admins is that when you add a new admin organization to the channel, you do not have to update the channel policy. Therefore ImplicitMeta policies are considered to be more flexible as organizations are added. Recall that ImplicitMeta policies ultimately resolve the Signature sub-policies underneath them in the configuration tree as the diagram shows.

You can also define an application level implicit policy to operate across organizations, in a channel for example, and either require that ANY of them are satisfied, that ALL are satisfied, or that a MAJORITY are satisfied. This format lends itself to much better, more natural defaults, so that each organization can decide what it means for a valid endorsement.

Further granularity and control can be achieved if you include NodeOUs in your organization definition. Organization Units (OUs) are defined in the Fabric CA client configuration file and can be associated with an identity when it is created. In Fabric, NodeOUs provide a way to classify identities in a digital certificate hierarchy. For instance, an organization having specific NodeOUs enabled could require that a 'peer' sign for it to be a valid endorsement, whereas an organization without any might simply require that any member can sign.

4.6.5 An example: channel configuration policy

Understanding policies begins with examining the configtx.yaml where the channel policies are defined. We can use the configtx.yaml file in the Fabric test network to see examples of both policy syntax types. We are going to examine the configtx.yaml file used by the fabric-samples/test-network sample.

The first section of the file defines the organizations that will be members of the channel. Inside each organization definition are the default policies for that organization, Readers, Writers, Admins, and Endorsement, although

you can name your policies anything you want. Each policy has a `Type` which describes how the policy is expressed (`Signature` or `ImplicitMeta`) and a `Rule`.

The test network example below shows the `Org1` organization definition in the channel, where the policy `Type` is `Signature` and the `Endorsement` : policy rule is defined as `"OR('Org1MSP.peer')"`. This policy specifies that a peer that is a member of `Org1MSP` is required to sign. It is these signature policies that become the sub-policies that the `ImplicitMeta` policies point to.

Click here to see an example of an organization defined with signature policies

```
- &Org1
  # DefaultOrg defines the organization which is used in the sampleconfig
  # of the fabric.git development environment
  Name: Org1MSP

  # ID to load the MSP definition as
  ID: Org1MSP

  MSPDir: ../organizations/peerOrganizations/org1.example.com/msp

  # Policies defines the set of policies at this level of the config tree
  # For organization policies, their canonical path is usually
  # /Channel/<Application|Orderer>/<OrgName>/<PolicyName>
  Policies:
    Readers:
      Type: Signature
      Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client')"
    Writers:
      Type: Signature
      Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
    Admins:
      Type: Signature
      Rule: "OR('Org1MSP.admin')"
    Endorsement:
      Type: Signature
      Rule: "OR('Org1MSP.peer')"
```

The next example shows the `ImplicitMeta` policy type used in the `Application` section of the `configtx.yaml`. These set of policies lie on the `/Channel/Application/` path. If you use the default set of Fabric ACLs, these policies define the behavior of many important features of application channels, such as who can query the channel ledger, invoke a chaincode, or update a channel config. These policies point to the sub-policies defined for each organization. The `Org1` defined in the section above contains `Reader`, `Writer`, and `Admin` sub-policies that are evaluated by the `Reader`, `Writer`, and `Admin` `ImplicitMeta` policies in the `Application` section. Because the test network is built with the default policies, you can use the example `Org1` to query the channel ledger, invoke a chaincode, and approve channel updates for any test network channel that you create.

Click here to see an example of `ImplicitMeta` policies

```
#####
#
# SECTION: Application
#
# - This section defines the values to encode into a config transaction or
# genesis block for application related parameters
#
#####
Application: &ApplicationDefaults
```

(continues on next page)

(continued from previous page)

```

# Organizations is the list of orgs which are defined as participants on
# the application side of the network
Organizations:

# Policies defines the set of policies at this level of the config tree
# For Application policies, their canonical path is
# /Channel/Application/<PolicyName>
Policies:
  Readers:
    Type: ImplicitMeta
    Rule: "ANY Readers"
  Writers:
    Type: ImplicitMeta
    Rule: "ANY Writers"
  Admins:
    Type: ImplicitMeta
    Rule: "MAJORITY Admins"
  LifecycleEndorsement:
    Type: ImplicitMeta
    Rule: "MAJORITY Endorsement"
  Endorsement:
    Type: ImplicitMeta
    Rule: "MAJORITY Endorsement"

```

4.6.6 Fabric chaincode lifecycle

In the Fabric 2.0 release, a new chaincode lifecycle process was introduced, whereby a more democratic process is used to govern chaincode on the network. The new process allows multiple organizations to vote on how a chaincode will be operated before it can be used on a channel. This is significant because it is the combination of this new lifecycle process and the policies that are specified during that process that dictate the security across the network. More details on the flow are available in the [Fabric chaincode lifecycle](#) concept topic, but for purposes of this topic you should understand how policies are used in this flow. The new flow includes two steps where policies are specified: when chaincode is **approved** by organization members, and when it is **committed** to the channel.

The Application section of the configtx.yaml file includes the default chaincode lifecycle endorsement policy. In a production environment you would customize this definition for your own use case.

```

#####
#
# SECTION: Application
#
# - This section defines the values to encode into a config transaction or
# genesis block for application related parameters
#
#####
Application: &ApplicationDefaults

# Organizations is the list of orgs which are defined as participants on
# the application side of the network
Organizations:

# Policies defines the set of policies at this level of the config tree
# For Application policies, their canonical path is
# /Channel/Application/<PolicyName>
Policies:

```

(continues on next page)

(continued from previous page)

```

Readers:
  Type: ImplicitMeta
  Rule: "ANY Readers"
Writers:
  Type: ImplicitMeta
  Rule: "ANY Writers"
Admins:
  Type: ImplicitMeta
  Rule: "MAJORITY Admins"
LifecycleEndorsement:
  Type: ImplicitMeta
  Rule: "MAJORITY Endorsement"
Endorsement:
  Type: ImplicitMeta
  Rule: "MAJORITY Endorsement"

```

- The `LifecycleEndorsement` policy governs who needs to *approve a chaincode definition*.
- The `Endorsement` policy is the *default endorsement policy for a chaincode*. More on this below.

4.6.7 Chaincode endorsement policies

The endorsement policy is specified for a **chaincode** when it is approved and committed to the channel using the Fabric chaincode lifecycle (that is, one endorsement policy covers all of the state associated with a chaincode). The endorsement policy can be specified either by reference to an endorsement policy defined in the channel configuration or by explicitly specifying a Signature policy.

If an endorsement policy is not explicitly specified during the approval step, the default `Endorsement` policy `"MAJORITY Endorsement"` is used which means that a majority of the peers belonging to the different channel members (organizations) need to execute and validate a transaction against the chaincode in order for the transaction to be considered valid. This default policy allows organizations that join the channel to become automatically added to the chaincode endorsement policy. If you don't want to use the default endorsement policy, use the Signature policy format to specify a more complex endorsement policy (such as requiring that a chaincode be endorsed by one organization, and then one of the other organizations on the channel).

Signature policies also allow you to include `principals` which are simply a way of matching an identity to a role. Principals are just like user IDs or group IDs, but they are more versatile because they can include a wide range of properties of an actor's identity, such as the actor's organization, organizational unit, role or even the actor's specific identity. When we talk about principals, they are the properties which determine their permissions. Principals are described as `'MSP.ROLE'`, where `MSP` represents the required MSP ID (the organization), and `ROLE` represents one of the four accepted roles: Member, Admin, Client, and Peer. A role is associated to an identity when a user enrolls with a CA. You can customize the list of roles available on your Fabric CA.

Some examples of valid principals are:

- `'Org0.Admin'`: an administrator of the Org0 MSP
- `'Org1.Member'`: a member of the Org1 MSP
- `'Org1.Client'`: a client of the Org1 MSP
- `'Org1.Peer'`: a peer of the Org1 MSP
- `'OrdererOrg.Orderer'`: an orderer in the OrdererOrg MSP

There are cases where it may be necessary for a particular state (a particular key-value pair, in other words) to have a different endorsement policy. This **state-based endorsement** allows the default chaincode-level endorsement policies to be overridden by a different policy for the specified keys.

For a deeper dive on how to write an endorsement policy refer to the topic on [Endorsement policies](#) in the Operations Guide.

Note: Policies work differently depending on which version of Fabric you are using:

- In Fabric releases prior to 2.0, chaincode endorsement policies can be updated during chaincode instantiation or by using the chaincode lifecycle commands. If not specified at instantiation time, the endorsement policy defaults to “any member of the organizations in the channel”. For example, a channel with “Org1” and “Org2” would have a default endorsement policy of “OR(‘Org1.member’, ‘Org2.member’)”.
- Starting with Fabric 2.0, Fabric introduced a new chaincode lifecycle process that allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. The new process requires that organizations agree to the parameters that define a chaincode, such as name, version, and the chaincode endorsement policy.

4.6.8 Overriding policy definitions

Hyperledger Fabric includes default policies which are useful for getting started, developing, and testing your blockchain, but they are meant to be customized in a production environment. You should be aware of the default policies in the `configtx.yaml` file. Channel configuration policies can be extended with arbitrary verbs, beyond the default Readers, Writers, Admins in `configtx.yaml`.

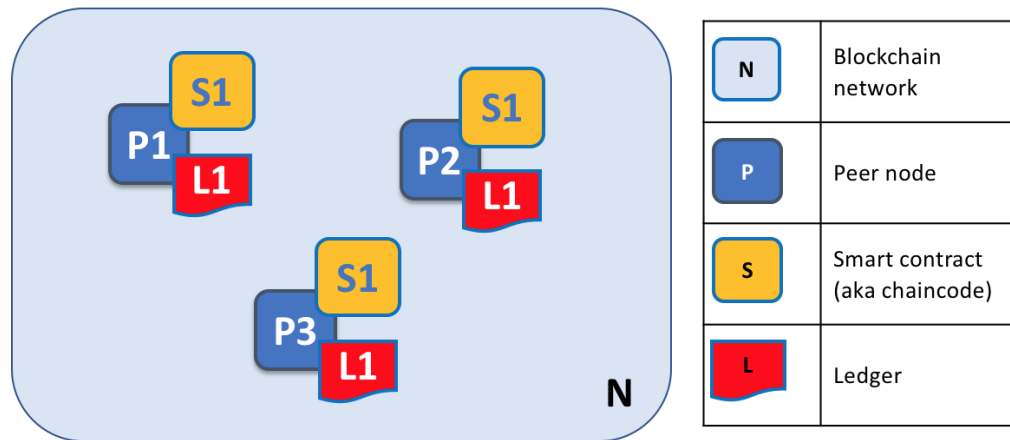
For more information on overriding policy definitions when creating a channel, check out [Channel policies](#) and [Creating a channel without a system channel](#).

For information about how to update a channel, check out [Updating a channel configuration](#) for more information.

4.7 Peers

A blockchain network is comprised primarily of a set of *peer nodes* (or, simply, *peers*). Peers are a fundamental element of the network because they host ledgers and smart contracts. Recall that a ledger immutably records all the transactions generated by smart contracts (which in Hyperledger Fabric are contained in a *chaincode*, more on this later). Smart contracts and ledgers are used to encapsulate the shared *processes* and shared *information* in a network, respectively. These aspects of a peer make them a good starting point to understand a Fabric network.

Other elements of the blockchain network are of course important: ledgers and smart contracts, orderers, policies, channels, applications, organizations, identities, and membership, and you can read more about them in their own dedicated sections. This section focusses on peers, and their relationship to those other elements in a Fabric network.



A blockchain network is comprised of peer nodes, each of which can hold copies of ledgers and copies of smart contracts. In this example, the network *N* consists of peers *P1*, *P2* and *P3*, each of which maintain their own instance of the distributed ledger *L1*. *P1*, *P2* and *P3* use the same chaincode, *S1*, to access their copy of that distributed ledger.

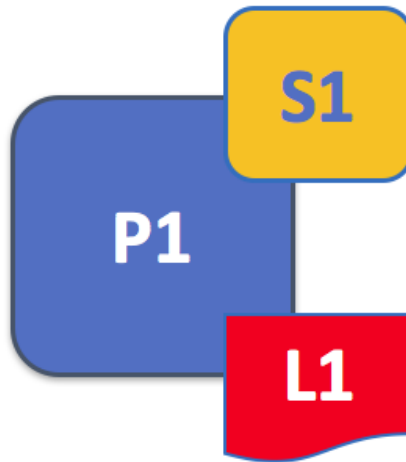
Peers can be created, started, stopped, reconfigured, and even deleted. They expose a set of APIs that enable administrators and applications to interact with the services that they provide. We'll learn more about these services in this section.

4.7.1 A word on terminology

Fabric implements **smart contracts** with a technology concept it calls **chaincode** — simply a piece of code that accesses the ledger, written in one of the supported programming languages. In this topic, we'll usually use the term **chaincode**, but feel free to read it as **smart contract** if you're more used to that term. It's the same thing! If you want to learn more about chaincode and smart contracts, check out our [documentation on smart contracts and chaincode](#).

4.7.2 Ledgers and Chaincode

Let's look at a peer in a little more detail. We can see that it's the peer that hosts both the ledger and chaincode. More accurately, the peer actually hosts *instances* of the ledger, and *instances* of chaincode. Note that this provides a deliberate redundancy in a Fabric network — it avoids single points of failure. We'll learn more about the distributed and decentralized nature of a blockchain network later in this section.

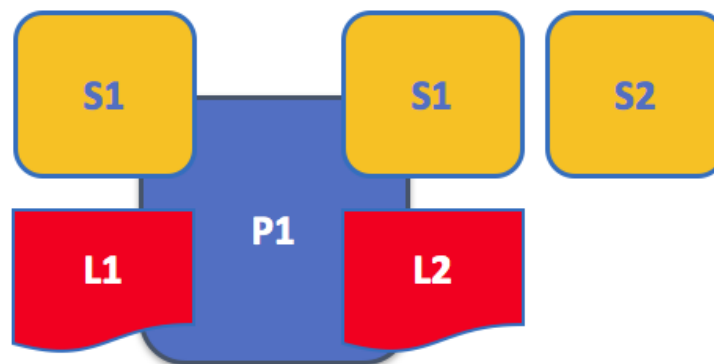


A peer hosts instances of ledgers and instances of chaincodes. In this example, P1 hosts an instance of ledger L1 and an instance of chaincode S1. There can be many ledgers and chaincodes hosted on an individual peer.

Because a peer is a *host* for ledgers and chaincodes, applications and administrators must interact with a peer if they want to access these resources. That's why peers are considered the most fundamental building blocks of a Fabric network. When a peer is first created, it has neither ledgers nor chaincodes. We'll see later how ledgers get created, and how chaincodes get installed, on peers.

Multiple Ledgers

A peer is able to host more than one ledger, which is helpful because it allows for a flexible system design. The simplest configuration is for a peer to manage a single ledger, but it's absolutely appropriate for a peer to host two or more ledgers when required.

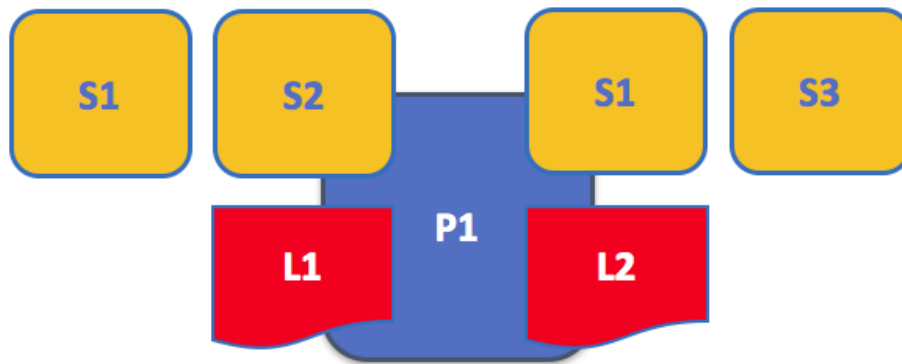


A peer hosting multiple ledgers. Peers host one or more ledgers, and each ledger has zero or more chaincodes that apply to them. In this example, we can see that the peer P1 hosts ledgers L1 and L2. Ledger L1 is accessed using chaincode S1. Ledger L2 on the other hand can be accessed using chaincodes S1 and S2.

Although it is perfectly possible for a peer to host a ledger instance without hosting any chaincodes which access that ledger, it's rare that peers are configured this way. The vast majority of peers will have at least one chaincode installed on it which can query or update the peer's ledger instances. It's worth mentioning in passing that, whether or not users have installed chaincodes for use by external applications, peers also have special **system chaincodes** that are always present. These are not discussed in detail in this topic.

Multiple Chaincodes

There isn't a fixed relationship between the number of ledgers a peer has and the number of chaincodes that can access that ledger. A peer might have many chaincodes and many ledgers available to it.



An example of a peer hosting multiple chaincodes. Each ledger can have many chaincodes which access it. In this example, we can see that peer P1 hosts ledgers L1 and L2, where L1 is accessed by chaincodes S1 and S2, and L2 is accessed by S1 and S3. We can see that S1 can access both L1 and L2.

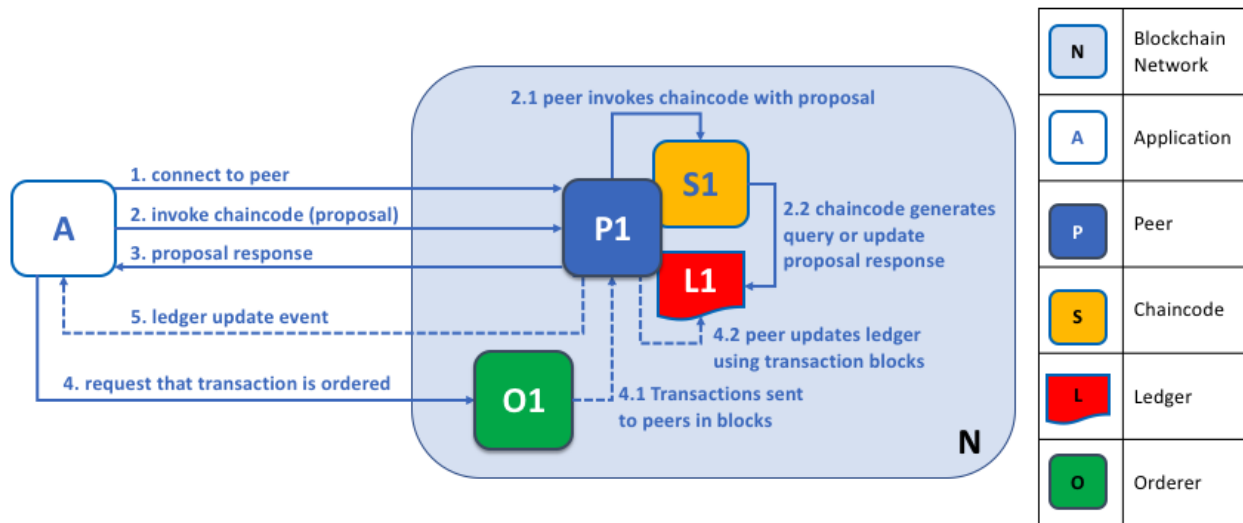
We'll see a little later why the concept of **channels** in Fabric is important when hosting multiple ledgers or multiple chaincodes on a peer.

4.7.3 Applications and Peers

We're now going to show how applications interact with peers to access the ledger. Ledger-query interactions involve a simple three-step dialogue between an application and a peer; ledger-update interactions are a little more involved, and require two extra steps. We've simplified these steps a little to help you get started with Fabric, but don't worry — what's most important to understand is the difference in application-peer interactions for ledger-query compared to ledger-update transaction styles.

Applications always connect to peers when they need to access ledgers and chaincodes. The Fabric Software Development Kit (SDK) makes this easy for programmers — its APIs enable applications to connect to peers, invoke chaincodes to generate transactions, submit transactions to the network that will get ordered, validated and committed to the distributed ledger, and receive events when this process is complete.

Through a peer connection, applications can execute chaincodes to query or update a ledger. The result of a ledger query transaction is returned immediately, whereas ledger updates involve a more complex interaction between applications, peers and orderers. Let's investigate this in a little more detail.



Peers, in conjunction with orderers, ensure that the ledger is kept up-to-date on every peer. In this example, application A connects to P1 and invokes chaincode S1 to query or update the ledger L1. P1 invokes S1 to generate a proposal response that contains a query result or a proposed ledger update. Application A receives the proposal response and, for queries, the process is now complete. For updates, A builds a transaction from all of the responses, which it sends to O1 for ordering. O1 collects transactions from across the network into blocks, and distributes these to all peers, including P1. P1 validates the transaction before committing to L1. Once L1 is updated, P1 generates an event, received by A, to signify completion.

A peer can return the results of a query to an application immediately since all of the information required to satisfy the query is in the peer's local copy of the ledger. Peers never consult with other peers in order to respond to a query from an application. Applications can, however, connect to one or more peers to issue a query; for example, to corroborate a result between multiple peers, or retrieve a more up-to-date result from a different peer if there's a suspicion that information might be out of date. In the diagram, you can see that ledger query is a simple three-step process.

An update transaction starts in the same way as a query transaction, but has two extra steps. Although ledger-updating applications also connect to peers to invoke a chaincode, unlike with ledger-querying applications, an individual peer cannot perform a ledger update at this time, because other peers must first agree to the change — a process called **consensus**. Therefore, peers return to the application a **proposed** update — one that this peer would apply subject to other peers' prior agreement. The first extra step — step four — requires that applications send an appropriate set of matching proposed updates to the entire network of peers as a transaction for commitment to their respective ledgers. This is achieved by the application by using an **orderer** to package transactions into blocks, and distributing them to the entire network of peers, where they can be verified before being applied to each peer's local copy of the ledger. As this whole ordering processing takes some time to complete (seconds), the application is notified asynchronously, as shown in step five.

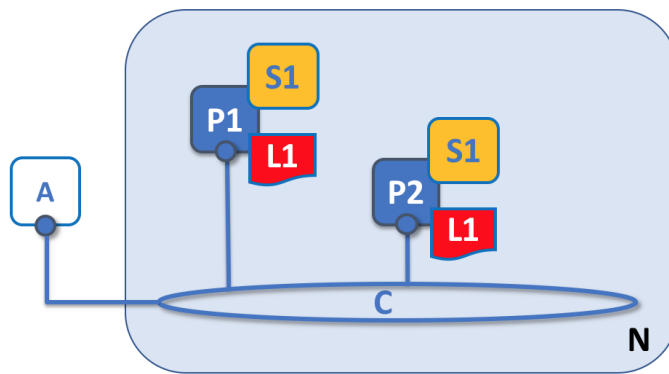
Later in this section, you'll learn more about the detailed nature of this ordering process — and for a really detailed look at this process see the [Transaction Flow](#) topic.

4.7.4 Peers and Channels

Although this section is about peers rather than channels, it's worth spending a little time understanding how peers interact with each other, and with applications, via *channels* — a mechanism by which a set of components within a blockchain network can communicate and transact *privately*.

These components are typically peer nodes, orderer nodes and applications and, by joining a channel, they agree to collaborate to collectively share and manage identical copies of the ledger associated with that channel. Conceptually, you can think of channels as being similar to groups of friends (though the members of a channel certainly don't need

to be friends!). A person might have several groups of friends, with each group having activities they do together. These groups might be totally separate (a group of work friends as compared to a group of hobby friends), or there can be some crossover between them. Nevertheless, each group is its own entity, with “rules” of a kind.



N	Blockchain Network	L	Ledger
C	Channel	A	Application
P	Peer	PA	Principal PA (e.g. A, P1) communicates via channel C.
S	Chaincode		

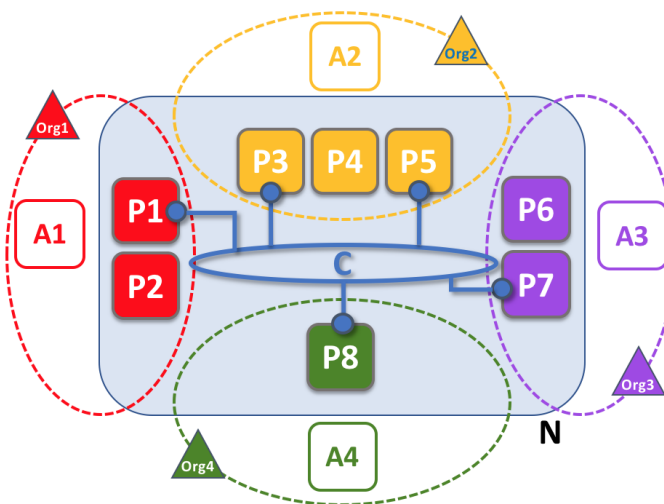
Channels allow a specific set of peers and applications to communicate with each other within a blockchain network. In this example, application A can communicate directly with peers P1 and P2 using channel C. You can think of the channel as a pathway for communications between particular applications and peers. (For simplicity, orderers are not shown in this diagram, but must be present in a functioning network.)

We see that channels don’t exist in the same way that peers do — it’s more appropriate to think of a channel as a logical structure that is formed by a collection of physical peers. It is vital to understand this point — peers provide the control point for access to, and management of, channels.

4.7.5 Peers and Organizations

Now that you understand peers and their relationship to ledgers, chaincodes and channels, you’ll be able to see how multiple organizations come together to form a blockchain network.

Blockchain networks are administered by a collection of organizations rather than a single organization. Peers are central to how this kind of distributed network is built because they are owned by — and are the connection points to the network for — these organizations.



N	Blockchain Network	L	Ledger
C	Channel	A	Application
P	Peer	PA	Principal PA (e.g. A1, P5) communicates via channel C.
		Org	Organization
		Organization R owns application A1 and peers P1, P2.	

Peers in a blockchain network with multiple organizations. The blockchain network is built up from the peers owned and contributed by the different organizations. In this example, we see four organizations contributing eight peers to form a network. The channel C connects five of these peers in the network N — P1, P3, P5, P7 and P8. The other peers owned by these organizations have not been joined to this channel, but are typically joined to at least one other channel. Applications that have been developed by a particular organization will connect to their own organization's peers as well as those of different organizations. Again, for simplicity, an orderer node is not shown in this diagram.

It's really important that you can see what's happening in the formation of a blockchain network. *The network is both formed and managed by the multiple organizations who contribute resources to it.* Peers are the resources that we're discussing in this topic, but the resources an organization provides are more than just peers. There's a principle at work here — the network literally does not exist without organizations contributing their individual resources to the collective network. Moreover, the network grows and shrinks with the resources that are provided by these collaborating organizations.

You can see that (other than the ordering service) there are no centralized resources — in the *example above*, the network, N, would not exist if the organizations did not contribute their peers. This reflects the fact that the network does not exist in any meaningful sense unless and until organizations contribute the resources that form it. Moreover, the network does not depend on any individual organization — it will continue to exist as long as one organization remains, no matter which other organizations may come and go. This is at the heart of what it means for a network to be decentralized.

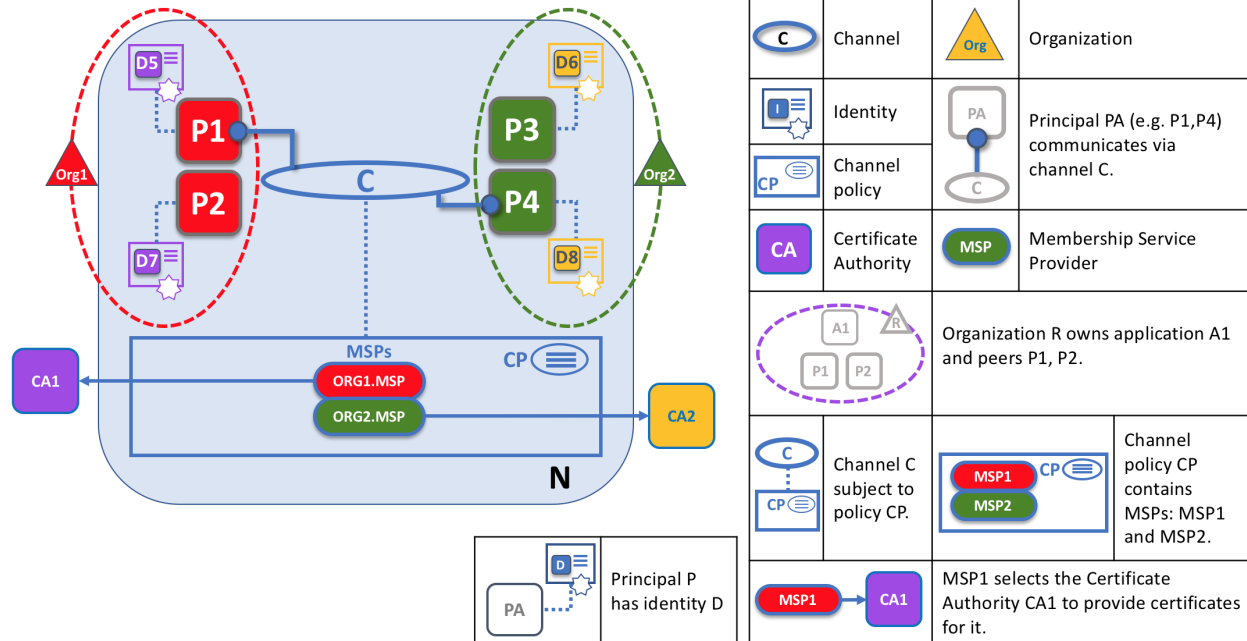
Applications in different organizations, as in the *example above*, may or may not be the same. That's because it's entirely up to an organization as to how its applications process their peers' copies of the ledger. This means that both application and presentation logic may vary from organization to organization even though their respective peers host exactly the same ledger data.

Applications connect either to peers in their organization, or peers in another organization, depending on the nature of the ledger interaction that's required. For ledger-query interactions, applications typically connect to their own organization's peers. For ledger-update interactions, we'll see later why applications need to connect to peers representing *every* organization that is required to endorse the ledger update.

4.7.6 Peers and Identity

Now that you've seen how peers from different organizations come together to form a blockchain network, it's worth spending a few moments understanding how peers get assigned to organizations by their administrators.

Peers have an identity assigned to them via a digital certificate from a particular certificate authority. You can read lots more about how X.509 digital certificates work elsewhere in this guide but, for now, think of a digital certificate as being like an ID card that provides lots of verifiable information about a peer. *Each and every peer in the network is assigned a digital certificate by an administrator from its owning organization.*



When a peer connects to a channel, its digital certificate identifies its owning organization via a channel MSP. In this example, P1 and P2 have identities issued by CA1. Channel C determines from a policy in its channel configuration that identities from CA1 should be associated with Org1 using ORG1.MSP. Similarly, P3 and P4 are identified by ORG2.MSP as being part of Org2.

Whenever a peer connects using a channel to a blockchain network, a policy in the channel configuration uses the peer's identity to determine its rights. The mapping of identity to organization is provided by a component called a *Membership Service Provider (MSP)* — it determines how a peer gets assigned to a specific role in a particular organization and accordingly gains appropriate access to blockchain resources. Moreover, a peer can be owned only by a single organization, and is therefore associated with a single MSP. We'll learn more about peer access control later in this section, and there's an entire section on MSPs and access control policies elsewhere in this guide. But for now, think of an MSP as providing linkage between an individual identity and a particular organizational role in a blockchain network.

To digress for a moment, peers as well as *everything that interacts with a blockchain network acquire their organizational identity from their digital certificate and an MSP*. Peers, applications, end users, administrators and orderers must have an identity and an associated MSP if they want to interact with a blockchain network. We give a name to every entity that interacts with a blockchain network using an identity — a *principal*. You can learn lots more about principals and organizations elsewhere in this guide, but for now you know more than enough to continue your understanding of peers!

Finally, note that it's not really important where the peer is physically located — it could reside in the cloud, or in a data centre owned by one of the organizations, or on a local machine — it's the digital certificate associated with it that identifies it as being owned by a particular organization. In our example above, P3 could be hosted in Org1's data center, but as long as the digital certificate associated with it is issued by CA2, then it's owned by Org2.

4.7.7 Peers and Orderers

We've seen that peers form the basis for a blockchain network, hosting ledgers and smart contracts which can be queried and updated by peer-connected applications. However, the mechanism by which applications and peers interact

with each other to ensure that every peer's ledger is kept consistent with each other is mediated by special nodes called *orderers*, and it's to these nodes we now turn our attention.

An update transaction is quite different from a query transaction because a single peer cannot, on its own, update the ledger — updating requires the consent of other peers in the network. A peer requires other peers in the network to approve a ledger update before it can be applied to a peer's local ledger. This process is called *consensus*, which takes much longer to complete than a simple query. But when all the peers required to approve the transaction do so, and the transaction is committed to the ledger, peers will notify their connected applications that the ledger has been updated. You're about to be shown a lot more detail about how peers and orderers manage the consensus process in this section.

Specifically, applications that want to update the ledger are involved in a 3-phase process, which ensures that all the peers in a blockchain network keep their ledgers consistent with each other.

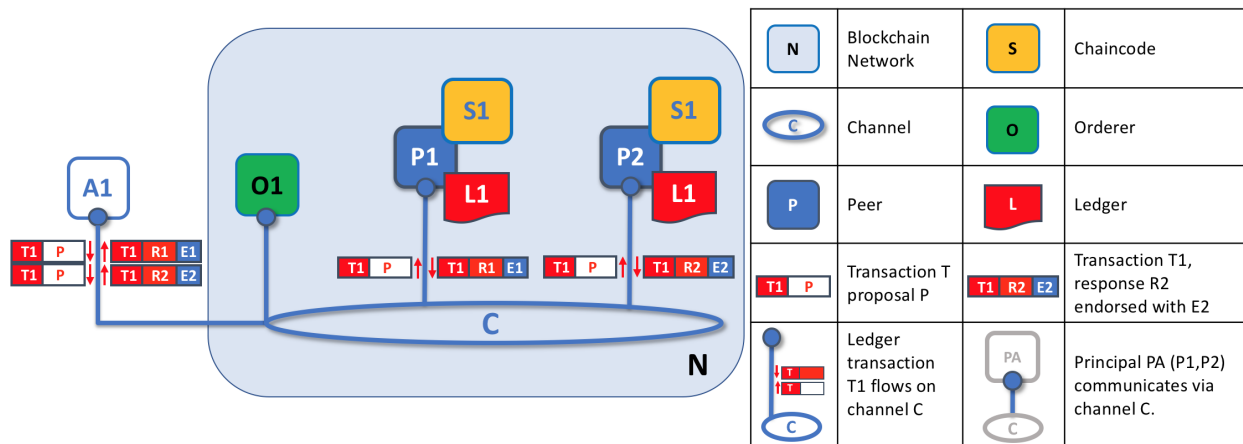
- In the first phase, applications work with a subset of *endorsing peers*, each of which provide an endorsement of the proposed ledger update to the application, but do not apply the proposed update to their copy of the ledger.
- In the second phase, these separate endorsements are collected together as transactions and packaged into blocks.
- In the third and final phase, these blocks are distributed back to every peer where each transaction is validated before being committed to that peer's copy of the ledger.

As you will see, orderer nodes are central to this process, so let's investigate in a little more detail how applications and peers use orderers to generate ledger updates that can be consistently applied to a distributed, replicated ledger.

Phase 1: Proposal

Phase 1 of the transaction workflow involves an interaction between an application and a set of peers — it does not involve orderers. Phase 1 is only concerned with an application asking different organizations' endorsing peers to agree to the results of the proposed chaincode invocation.

To start phase 1, applications generate a transaction proposal which they send to each of the required set of peers for endorsement. Each of these *endorsing peers* then independently executes a chaincode using the transaction proposal to generate a transaction proposal response. It does not apply this update to the ledger, but rather simply signs it and returns it to the application. Once the application has received a sufficient number of signed proposal responses, the first phase of the transaction flow is complete. Let's examine this phase in a little more detail.



Transaction proposals are independently executed by peers who return endorsed proposal responses. In this example, application A1 generates transaction T1 proposal P which it sends to both peer P1 and peer P2 on channel C. P1 executes S1 using transaction T1 proposal P generating transaction T1 response R1 which it endorses with E1. Independently, P2 executes S1 using transaction T1 proposal P generating transaction T1 response R2 which it endorses with E2. Application A1 receives two endorsed responses for transaction T1, namely E1 and E2.

Initially, a set of peers are chosen by the application to generate a set of proposed ledger updates. Which peers are chosen by the application? Well, that depends on the *endorsement policy* (defined for a chaincode), which defines the set of organizations that need to endorse a proposed ledger change before it can be accepted by the network. This is literally what it means to achieve consensus — every organization who matters must have endorsed the proposed ledger change *before* it will be accepted onto any peer's ledger.

A peer endorses a proposal response by adding its digital signature, and signing the entire payload using its private key. This endorsement can be subsequently used to prove that this organization's peer generated a particular response. In our example, if peer P1 is owned by organization Org1, endorsement E1 corresponds to a digital proof that "Transaction T1 response R1 on ledger L1 has been provided by Org1's peer P1!".

Phase 1 ends when the application receives signed proposal responses from sufficient peers. We note that different peers can return different and therefore inconsistent transaction responses to the application *for the same transaction proposal*. It might simply be that the result was generated at different times on different peers with ledgers at different states, in which case an application can simply request a more up-to-date proposal response. Less likely, but much more seriously, results might be different because the chaincode is *non-deterministic*. Non-determinism is the enemy of chaincodes and ledgers and if it occurs it indicates a serious problem with the proposed transaction, as inconsistent results cannot, obviously, be applied to ledgers. An individual peer cannot know that their transaction result is non-deterministic — transaction responses must be gathered together for comparison before non-determinism can be detected. (Strictly speaking, even this is not enough, but we defer this discussion to the transaction section, where non-determinism is discussed in detail.)

At the end of phase 1, the application is free to discard inconsistent transaction responses if it wishes to do so, effectively terminating the transaction workflow early. We'll see later that if an application tries to use an inconsistent set of transaction responses to update the ledger, it will be rejected.

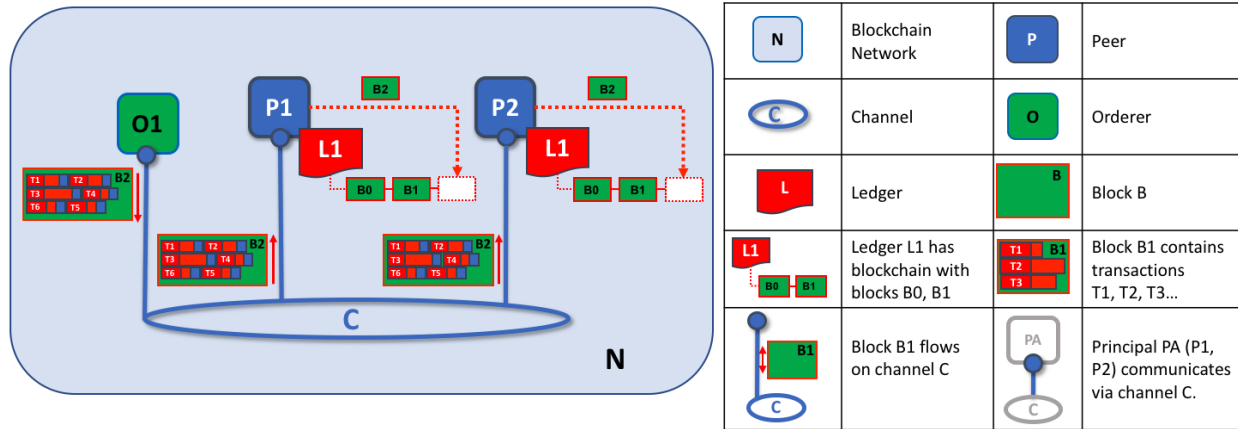
Phase 2: Ordering and packaging transactions into blocks

The second phase of the transaction workflow is the packaging phase. The orderer is pivotal to this process — it receives transactions containing endorsed transaction proposal responses from many applications, and orders the transactions into blocks. For more details about the ordering and packaging phase, check out our [conceptual information about the ordering phase](#).

Phase 3: Validation and commit

At the end of phase 2, we see that orderers have been responsible for the simple but vital processes of collecting proposed transaction updates, ordering them, and packaging them into blocks, ready for distribution to the peers.

The final phase of the transaction workflow involves the distribution and subsequent validation of blocks from the orderer to the peers, where they can be committed to the ledger. Specifically, at each peer, every transaction within a block is validated to ensure that it has been consistently endorsed by all relevant organizations before it is committed to the ledger. Failed transactions are retained for audit, but are not committed to the ledger.



The second role of an orderer node is to distribute blocks to peers. In this example, orderer O1 distributes block B2 to peer P1 and peer P2. Peer P1 processes block B2, resulting in a new block being added to ledger L1 on P1. In parallel, peer P2 processes block B2, resulting in a new block being added to ledger L1 on P2. Once this process is complete, the ledger L1 has been consistently updated on peers P1 and P2, and each may inform connected applications that the transaction has been processed.

Phase 3 begins with the orderer distributing blocks to all peers connected to it. Peers are connected to orderers on channels such that when a new block is generated, all of the peers connected to the orderer will be sent a copy of the new block. Each peer will process this block independently, but in exactly the same way as every other peer on the channel. In this way, we'll see that the ledger can be kept consistent. It's also worth noting that not every peer needs to be connected to an orderer — peers can cascade blocks to other peers using the **gossip** protocol, who also can process them independently. But let's leave that discussion to another time!

Upon receipt of a block, a peer will process each transaction in the sequence in which it appears in the block. For every transaction, each peer will verify that the transaction has been endorsed by the required organizations according to the *endorsement policy* of the chaincode which generated the transaction. For example, some transactions may only need to be endorsed by a single organization, whereas others may require multiple endorsements before they are considered valid. This process of validation verifies that all relevant organizations have generated the same outcome or result. Also note that this validation is different than the endorsement check in phase 1, where it is the application that receives the response from endorsing peers and makes the decision to send the proposal transactions. In case the application violates the endorsement policy by sending wrong transactions, the peer is still able to reject the transaction in the validation process of phase 3.

If a transaction has been endorsed correctly, the peer will attempt to apply it to the ledger. To do this, a peer must perform a ledger consistency check to verify that the current state of the ledger is compatible with the state of the ledger when the proposed update was generated. This may not always be possible, even when the transaction has been fully endorsed. For example, another transaction may have updated the same asset in the ledger such that the transaction update is no longer valid and therefore can no longer be applied. In this way, the ledger is kept consistent across each peer in the channel because they each follow the same rules for validation.

After a peer has successfully validated each individual transaction, it updates the ledger. Failed transactions are not applied to the ledger, but they are retained for audit purposes, as are successful transactions. This means that peer blocks are almost exactly the same as the blocks received from the orderer, except for a valid or invalid indicator on each transaction in the block.

We also note that phase 3 does not require the running of chaincodes — this is done only during phase 1, and that's important. It means that chaincodes only have to be available on endorsing nodes, rather than throughout the blockchain network. This is often helpful as it keeps the logic of the chaincode confidential to endorsing organizations. This is in contrast to the output of the chaincodes (the transaction proposal responses) which are shared with every peer in the channel, whether or not they endorsed the transaction. This specialization of endorsing peers is designed to help scalability and confidentiality.

Finally, every time a block is committed to a peer's ledger, that peer generates an appropriate *event*. *Block events* include the full block content, while *block transaction events* include summary information only, such as whether each transaction in the block has been validated or invalidated. *Chaincode* events that the chaincode execution has produced can also be published at this time. Applications can register for these event types so that they can be notified when they occur. These notifications conclude the third and final phase of the transaction workflow.

In summary, phase 3 sees the blocks which are generated by the orderer consistently applied to the ledger. The strict ordering of transactions into blocks allows each peer to validate that transaction updates are consistently applied across the blockchain network.

Orderers and Consensus

This entire transaction workflow process is called *consensus* because all peers have reached agreement on the order and content of transactions, in a process that is mediated by orderers. Consensus is a multi-step process and applications are only notified of ledger updates when the process is complete — which may happen at slightly different times on different peers.

We will discuss orderers in a lot more detail in a future orderer topic, but for now, think of orderers as nodes which collect and distribute proposed ledger updates from applications for peers to validate and include on the ledger.

That's it! We've now finished our tour of peers and the other components that they relate to in Fabric. We've seen that peers are in many ways the most fundamental element — they form the network, host chaincodes and the ledger, handle transaction proposals and responses, and keep the ledger up-to-date by consistently applying transaction updates to it.

4.8 Ledger

Audience: Architects, Application and smart contract developers, administrators

A **ledger** is a key concept in Hyperledger Fabric; it stores important factual information about business objects; both the current value of the attributes of the objects, and the history of transactions that resulted in these current values.

In this topic, we're going to cover:

- *What is a Ledger?*
- *Storing facts about business objects*
- *A blockchain ledger*
- *The world state*
- *The blockchain data structure*
- *How blocks are stored in a blockchain*
- *Transactions*
- *World state database options*
- *The **Basic** example ledger*
- *Ledgers and namespaces*
- *Ledgers and channels*

4.8.1 What is a Ledger?

A ledger contains the current state of a business as a journal of transactions. The earliest European and Chinese ledgers date from almost 1000 years ago, and the Sumerians had [stone ledgers](#) 4000 years ago – but let’s start with a more up-to-date example!

You’re probably used to looking at your bank account. What’s most important to you is the available balance – it’s what you’re able to spend at the current moment in time. If you want to see how your balance was derived, then you can look through the transaction credits and debits that determined it. This is a real life example of a ledger – a state (your bank balance), and a set of ordered transactions (credits and debits) that determine it. Hyperledger Fabric is motivated by these same two concerns – to present the current value of a set of ledger states, and to capture the history of the transactions that determined these states.

4.8.2 Ledgers, Facts, and States

A ledger doesn’t literally store business objects – instead it stores **facts** about those objects. When we say “we store a business object in a ledger” what we really mean is that we’re recording the facts about the current state of an object, and the facts about the history of transactions that led to the current state. In an increasingly digital world, it can feel like we’re looking at an object, rather than facts about an object. In the case of a digital object, it’s likely that it lives in an external datastore; the facts we store in the ledger allow us to identify its location along with other key information about it.

While the facts about the current state of a business object may change, the history of facts about it is **immutable**, it can be added to, but it cannot be retrospectively changed. We’re going to see how thinking of a blockchain as an immutable history of facts about business objects is a simple yet powerful way to understand it.

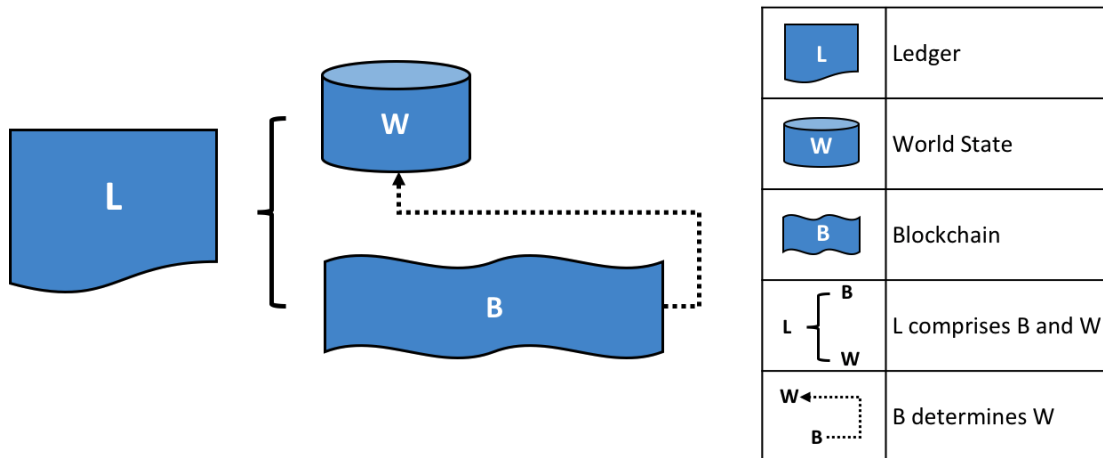
Let’s now take a closer look at the Hyperledger Fabric ledger structure!

4.8.3 The Ledger

In Hyperledger Fabric, a ledger consists of two distinct, though related, parts – a world state and a blockchain. Each of these represents a set of facts about a set of business objects.

Firstly, there’s a **world state** – a database that holds **current values** of a set of ledger states. The world state makes it easy for a program to directly access the current value of a state rather than having to calculate it by traversing the entire transaction log. Ledger states are, by default, expressed as **key-value** pairs, and we’ll see later how Hyperledger Fabric provides flexibility in this regard. The world state can change frequently, as states can be created, updated and deleted.

Secondly, there’s a **blockchain** – a transaction log that records all the changes that have resulted in the current the world state. Transactions are collected inside blocks that are appended to the blockchain – enabling you to understand the history of changes that have resulted in the current world state. The blockchain data structure is very different to the world state because once written, it cannot be modified; it is **immutable**.



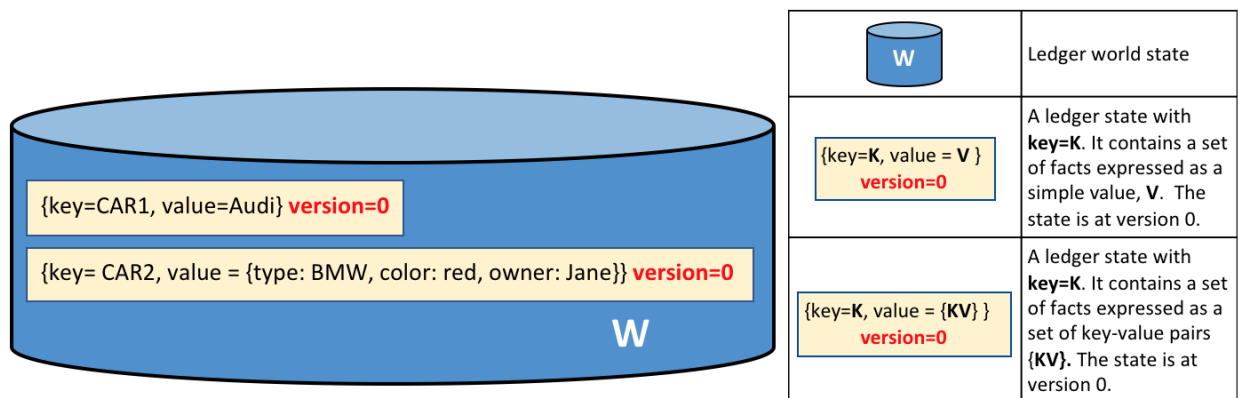
A Ledger L comprises blockchain B and world state W , where blockchain B determines world state W . We can also say that world state W is derived from blockchain B .

It's helpful to think of there being one **logical** ledger in a Hyperledger Fabric network. In reality, the network maintains multiple copies of a ledger – which are kept consistent with every other copy through a process called **consensus**. The term **Distributed Ledger Technology (DLT)** is often associated with this kind of ledger – one that is logically singular, but has many consistent copies distributed throughout a network.

Let's now examine the world state and blockchain data structures in more detail.

4.8.4 World State

The world state holds the current value of the attributes of a business object as a unique ledger state. That's useful because programs usually require the current value of an object; it would be cumbersome to traverse the entire blockchain to calculate an object's current value – you just get it directly from the world state.



A ledger world state containing two states. The first state is: $\text{key}=\text{CAR1}$ and $\text{value}=\text{Audi}$. The second state has a more complex value: $\text{key}=\text{CAR2}$ and $\text{value}=\{\text{model:BMW, color:red, owner:Jane}\}$. Both states are at version 0.

A ledger state records a set of facts about a particular business object. Our example shows ledger states for two cars,

CAR1 and CAR2, each having a key and a value. An application program can invoke a smart contract which uses simple ledger APIs to **get**, **put** and **delete** states. Notice how a state value can be simple (Audi...) or compound (type:BMW...). The world state is often queried to retrieve objects with certain attributes, for example to find all red BMWs.

The world state is implemented as a database. This makes a lot of sense because a database provides a rich set of operators for the efficient storage and retrieval of states. We'll see later that Hyperledger Fabric can be configured to use different world state databases to address the needs of different types of state values and the access patterns required by applications, for example in complex queries.

Applications submit transactions which capture changes to the world state, and these transactions end up being committed to the ledger blockchain. Applications are insulated from the details of this [consensus](#) mechanism by the Hyperledger Fabric SDK; they merely invoke a smart contract, and are notified when the transaction has been included in the blockchain (whether valid or invalid). The key design point is that only transactions that are **signed** by the required set of **endorsing organizations** will result in an update to the world state. If a transaction is not signed by sufficient endorsers, it will not result in a change of world state. You can read more about how applications use [smart contracts](#), and how to [develop applications](#).

You'll also notice that a state has a version number, and in the diagram above, states CAR1 and CAR2 are at their starting versions, 0. The version number is for internal use by Hyperledger Fabric, and is incremented every time the state changes. The version is checked whenever the state is updated to make sure the current states matches the version at the time of endorsement. This ensures that the world state is changing as expected; that there has not been a concurrent update.

Finally, when a ledger is first created, the world state is empty. Because any transaction which represents a valid change to world state is recorded on the blockchain, it means that the world state can be re-generated from the blockchain at any time. This can be very convenient – for example, the world state is automatically generated when a peer is created. Moreover, if a peer fails abnormally, the world state can be regenerated on peer restart, before transactions are accepted.

4.8.5 Blockchain

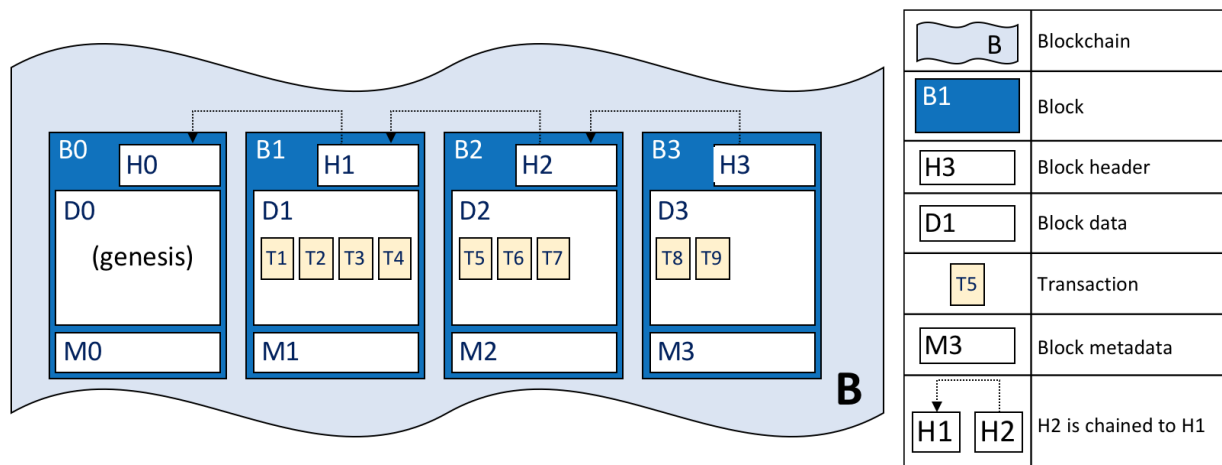
Let's now turn our attention from the world state to the blockchain. Whereas the world state contains a set of facts relating to the current state of a set of business objects, the blockchain is an historical record of the facts about how these objects arrived at their current states. The blockchain has recorded every previous version of each ledger state and how it has been changed.

The blockchain is structured as sequential log of interlinked blocks, where each block contains a sequence of transactions, each transaction representing a query or update to the world state. The exact mechanism by which transactions are ordered is discussed [elsewhere](#); what's important is that block sequencing, as well as transaction sequencing within blocks, is established when blocks are first created by a Hyperledger Fabric component called the **ordering service**.

Each block's header includes a hash of the block's transactions, as well a hash of the prior block's header. In this way, all transactions on the ledger are sequenced and cryptographically linked together. This hashing and linking makes the ledger data very secure. Even if one node hosting the ledger was tampered with, it would not be able to convince all the other nodes that it has the 'correct' blockchain because the ledger is distributed throughout a network of independent nodes.

The blockchain is always implemented as a file, in contrast to the world state, which uses a database. This is a sensible design choice as the blockchain data structure is heavily biased towards a very small set of simple operations. Appending to the end of the blockchain is the primary operation, and query is currently a relatively infrequent operation.

Let's have a look at the structure of a blockchain in a little more detail.



A blockchain *B* containing blocks *B0*, *B1*, *B2*, *B3*. *B0* is the first block in the blockchain, the genesis block.

In the above diagram, we can see that **block B2** has a **block data D2** which contains all its transactions: T5, T6, T7.

Most importantly, B2 has a **block header H2**, which contains a cryptographic **hash** of all the transactions in D2 as well as a hash of H1. In this way, blocks are inextricably and immutably linked to each other, which the term **blockchain** so neatly captures!

Finally, as you can see in the diagram, the first block in the blockchain is called the **genesis block**. It's the starting point for the ledger, though it does not contain any user transactions. Instead, it contains a configuration transaction containing the initial state of the network channel (not shown). We discuss the genesis block in more detail when we discuss the blockchain network and [channels](#) in the documentation.

4.8.6 Blocks

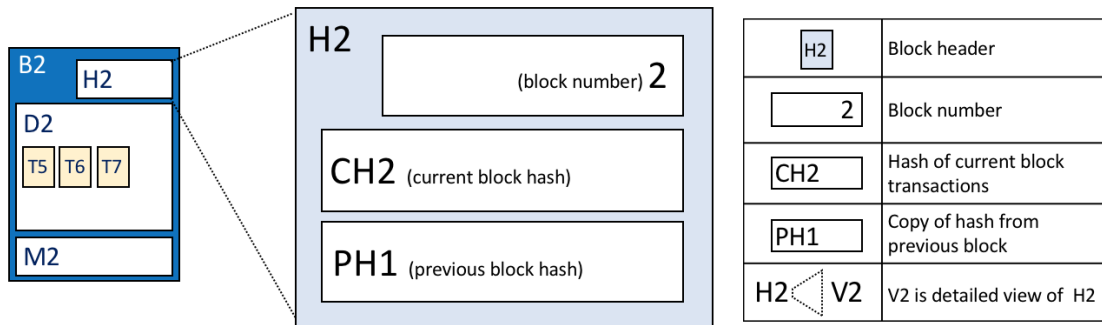
Let's have a closer look at the structure of a block. It consists of three sections

- **Block Header**

This section comprises three fields, written when a block is created.

- **Block number**: An integer starting at 0 (the genesis block), and increased by 1 for every new block appended to the blockchain.
- **Current Block Hash**: The hash of all the transactions contained in the current block.
- **Previous Block Header Hash**: The hash from the previous block header.

These fields are internally derived by cryptographically hashing the block data. They ensure that each and every block is inextricably linked to its neighbour, leading to an immutable ledger.



Block header details. The header H2 of block B2 consists of block number 2, the hash CH2 of the current block data D2, and the hash of the prior block header H1.

- **Block Data**

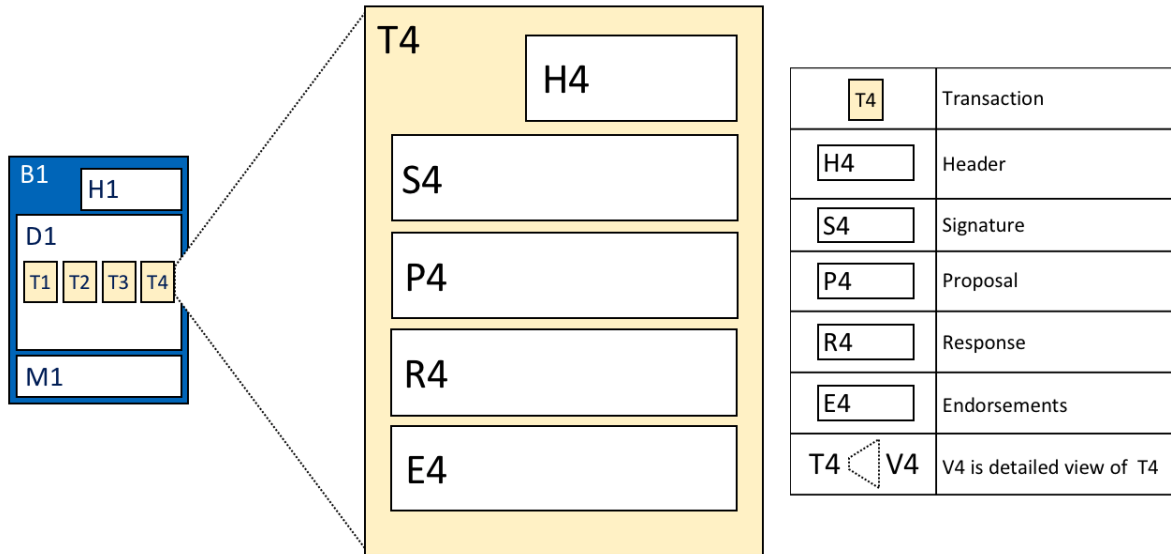
This section contains a list of transactions arranged in order. It is written when the block is created by the ordering service. These transactions have a rich but straightforward structure, which we describe *later* in this topic.

- **Block Metadata**

This section contains the certificate and signature of the block creator which is used to verify the block by network nodes. Subsequently, the block committer adds a valid/invalid indicator for every transaction into a bitmap that also resides in the block metadata, as well as a hash of the cumulative state updates up until and including that block, in order to detect a state fork. Unlike the block data and header fields, this section is not an input to the block hash computation.

4.8.7 Transactions

As we've seen, a transaction captures changes to the world state. Let's have a look at the detailed **blockdata** structure which contains the transactions in a block.



Transaction details. Transaction T4 in blockdata D1 of block B1 consists of transaction header, H4, a transaction signature, S4, a transaction proposal P4, a transaction response, R4, and a list of endorsements, E4.

In the above example, we can see the following fields:

- **Header**

This section, illustrated by H4, captures some essential metadata about the transaction – for example, the name of the relevant chaincode, and its version.

- **Signature**

This section, illustrated by S4, contains a cryptographic signature, created by the client application. This field is used to check that the transaction details have not been tampered with, as it requires the application's private key to generate it.

- **Proposal**

This field, illustrated by P4, encodes the input parameters supplied by an application to the smart contract which creates the proposed ledger update. When the smart contract runs, this proposal provides a set of input parameters, which, in combination with the current world state, determines the new world state.

- **Response**

This section, illustrated by R4, captures the before and after values of the world state, as a **Read Write set** (RW-set). It's the output of a smart contract, and if the transaction is successfully validated, it will be applied to the ledger to update the world state.

- **Endorsements**

As shown in E4, this is a list of signed transaction responses from each required organization sufficient to satisfy the endorsement policy. You'll notice that, whereas only one transaction response is included in the transaction, there are multiple endorsements. That's because each endorsement effectively encodes its organization's particular transaction response – meaning that there's no need to include any transaction response that doesn't match sufficient endorsements as it will be rejected as invalid, and not update the world state.

That concludes the major fields of the transaction – there are others, but these are the essential ones that you need to understand to have a solid understanding of the ledger data structure.

4.8.8 World State database options

The world state is physically implemented as a database, to provide simple and efficient storage and retrieval of ledger states. As we've seen, ledger states can have simple or compound values, and to accommodate this, the world state database implementation can vary, allowing these values to be efficiently implemented. Options for the world state database currently include LevelDB and CouchDB.

LevelDB is the default and is particularly appropriate when ledger states are simple key-value pairs. A LevelDB database is co-located with the peer node – it is embedded within the same operating system process.

CouchDB is a particularly appropriate choice when ledger states are structured as JSON documents because CouchDB supports the rich queries and update of richer data types often found in business transactions. Implementation-wise, CouchDB runs in a separate operating system process, but there is still a 1:1 relation between a peer node and a CouchDB instance. All of this is invisible to a smart contract. See [CouchDB as the StateDatabase](#) for more information on CouchDB.

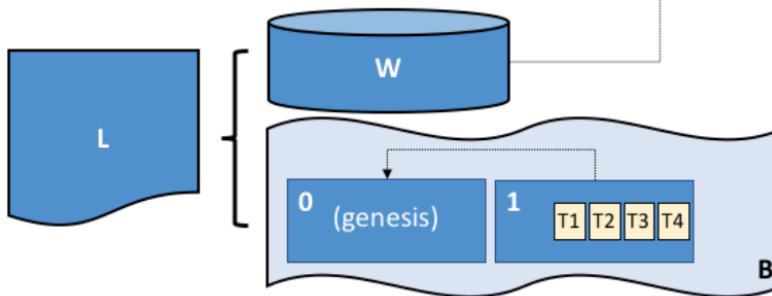
In LevelDB and CouchDB, we see an important aspect of Hyperledger Fabric – it is *pluggable*. The world state database could be a relational data store, or a graph store, or a temporal database. This provides great flexibility in the types of ledger states that can be efficiently accessed, allowing Hyperledger Fabric to address many different types of problems.

4.8.9 Example Ledger: Basic Asset Transfer

As we end this topic on the ledger, let's have a look at a sample ledger. If you've run the [basic asset transfer sample application](#), then you've created this ledger.

The basic sample app creates a set of 6 assets each with a unique identity; a different color, size, owner, and appraised value. Here's what the ledger looks like after the first four assets have been created.

key=ASSET4, value={color:yellow, size: 10, owner: Max, appraisedValue: 600}	version=0
key=ASSET3, value={color:green, size: 10, owner: Jin Soo, appraisedValue: 500}	version=0
key=ASSET2, value={color:red, size: 5, owner: Brad, appraisedValue: 400}	version=0
key=ASSET1, value={color:blue, size: 5, owner: Tomoko, appraisedValue: 300}	version=0



The ledger, *L*, comprises a world state, *W* and a blockchain, *B*. *W* contains four states with keys: *ASSET1*, *ASSET2*, *ASSET3*, and *ASSET4*. *B* contains two blocks, 0 and 1. Block 1 contains four transactions: *T1*, *T2*, *T3*, *T4*.

We can see that the world state contains states that correspond to *ASSET1*, *ASSET2*, *ASSET3*, and *ASSET4*. *ASSET1* has a value which indicates that it is a blue with size 5, currently owned by Tomoko, and we can see similar states and values for the other cars. Moreover, we can see that all car states are at version number 0, indicating that this is their starting version number – they have not been updated since they were created.

We can also see that the blockchain contains two blocks. Block 0 is the genesis block, though it does not contain any transactions that relate to cars. Block 1 however, contains transactions *T1*, *T2*, *T3*, *T4* and these correspond to

transactions that created the initial states for ASSET1 to ASSET4 in the world state. We can see that block 1 is linked to block 0.

We have not shown the other fields in the blocks or transactions, specifically headers and hashes. If you're interested in the precise details of these, you will find a dedicated reference topic elsewhere in the documentation. It gives you a fully worked example of an entire block with its transactions in glorious detail – but for now, you have achieved a solid conceptual understanding of a Hyperledger Fabric ledger. Well done!

4.8.10 Namespaces

Even though we have presented the ledger as though it were a single world state and single blockchain, that's a little bit of an over-simplification. In reality, each chaincode has its own world state that is separate from all other chaincodes. World states are in a namespace so that only smart contracts within the same chaincode can access a given namespace.

A blockchain is not namespaced. It contains transactions from many different smart contract namespaces. You can read more about chaincode namespaces in this [topic](#).

Let's now look at how the concept of a namespace is applied within a Hyperledger Fabric channel.

4.8.11 Channels

In Hyperledger Fabric, each [channel](#) has a completely separate ledger. This means a completely separate blockchain, and completely separate world states, including namespaces. It is possible for applications and smart contracts to communicate between channels so that ledger information can be accessed between them.

You can read more about how ledgers work with channels in this [topic](#).

4.8.12 More information

See the [Transaction Flow](#), [Read-Write set semantics](#) and [CouchDB as the StateDatabase](#) topics for a deeper dive on transaction flow, concurrency control, and the world state database.

4.9 The Ordering Service

Audience: Architects, ordering service admins, channel creators

This topic serves as a conceptual introduction to the concept of ordering, how orderers interact with peers, the role they play in a transaction flow, and an overview of the currently available implementations of the ordering service, with a particular focus on the recommended **Raft** ordering service implementation.

4.9.1 What is ordering?

Many distributed blockchains, such as Ethereum and Bitcoin, are not permissioned, which means that any node can participate in the consensus process, wherein transactions are ordered and bundled into blocks. Because of this fact, these systems rely on **probabilistic** consensus algorithms which eventually guarantee ledger consistency to a high degree of probability, but which are still vulnerable to divergent ledgers (also known as a ledger “fork”), where different participants in the network have a different view of the accepted order of transactions.

Hyperledger Fabric works differently. It features a node called an **orderer** (it's also known as an “ordering node”) that does this transaction ordering, which along with other orderer nodes forms an **ordering service**. Because Fabric's design relies on **deterministic** consensus algorithms, any block validated by the peer is guaranteed to be final and correct. Ledgers cannot fork the way they do in many other distributed and permissionless blockchain networks.

In addition to promoting finality, separating the endorsement of chaincode execution (which happens at the peers) from ordering gives Fabric advantages in performance and scalability, eliminating bottlenecks which can occur when execution and ordering are performed by the same nodes.

4.9.2 Orderer nodes and channel configuration

Orderers also enforce basic access control for channels, restricting who can read and write data to them, and who can configure them. Remember that who is authorized to modify a configuration element in a channel is subject to the policies that the relevant administrators set when they created the consortium or the channel. Configuration transactions are processed by the orderer, as it needs to know the current set of policies to execute its basic form of access control. In this case, the orderer processes the configuration update to make sure that the requestor has the proper administrative rights. If so, the orderer validates the update request against the existing configuration, generates a new configuration transaction, and packages it into a block that is relayed to all peers on the channel. The peers then process the configuration transactions in order to verify that the modifications approved by the orderer do indeed satisfy the policies defined in the channel.

4.9.3 Orderer nodes and identity

Everything that interacts with a blockchain network, including peers, applications, admins, and orderers, acquires their organizational identity from their digital certificate and their Membership Service Provider (MSP) definition.

For more information about identities and MSPs, check out our documentation on [Identity](#) and [Membership](#).

Just like peers, ordering nodes belong to an organization. And similar to peers, a separate Certificate Authority (CA) should be used for each organization. Whether this CA will function as the root CA, or whether you choose to deploy a root CA and then intermediate CAs associated with that root CA, is up to you.

4.9.4 Orderers and the transaction flow

Phase one: Proposal

We've seen from our topic on [Peers](#) that they form the basis for a blockchain network, hosting ledgers, which can be queried and updated by applications through smart contracts.

Specifically, applications that want to update the ledger are involved in a process with three phases that ensures all of the peers in a blockchain network keep their ledgers consistent with each other.

In the first phase, a client application sends a transaction proposal to a subset of peers that will invoke a smart contract to produce a proposed ledger update and then endorse the results. The endorsing peers do not apply the proposed update to their copy of the ledger at this time. Instead, the endorsing peers return a proposal response to the client application. The endorsed transaction proposals will ultimately be ordered into blocks in phase two, and then distributed to all peers for final validation and commit in phase three.

For an in-depth look at the first phase, refer back to the [Peers](#) topic.

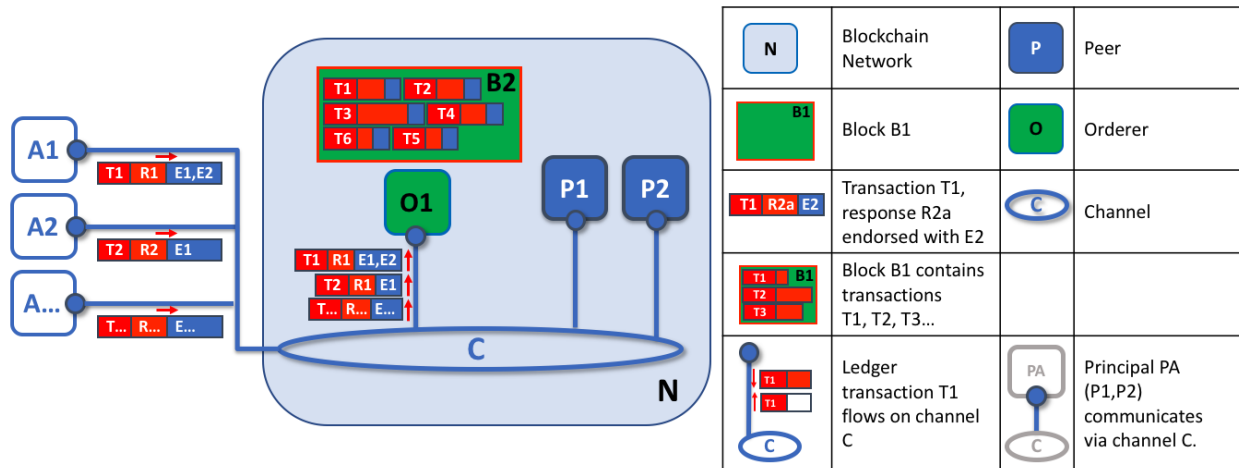
Phase two: Ordering and packaging transactions into blocks

After the completion of the first phase of a transaction, a client application has received an endorsed transaction proposal response from a set of peers. It's now time for the second phase of a transaction.

In this phase, application clients submit transactions containing endorsed transaction proposal responses to an ordering service node. The ordering service creates blocks of transactions which will ultimately be distributed to all peers on the channel for final validation and commit in phase three.

Ordering service nodes receive transactions from many different application clients concurrently. These ordering service nodes work together to collectively form the ordering service. Its job is to arrange batches of submitted transactions into a well-defined sequence and package them into *blocks*. These blocks will become the *blocks* of the blockchain!

The number of transactions in a block depends on channel configuration parameters related to the desired size and maximum elapsed duration for a block (`BatchSize` and `BatchTimeout` parameters, to be exact). The blocks are then saved to the orderer's ledger and distributed to all peers that have joined the channel. If a peer happens to be down at this time, or joins the channel later, it will receive the blocks after reconnecting to an ordering service node, or by gossiping with another peer. We'll see how this block is processed by peers in the third phase.



The first role of an ordering node is to package proposed ledger updates. In this example, application A1 sends a transaction T1 endorsed by E1 and E2 to the orderer O1. In parallel, Application A2 sends transaction T2 endorsed by E1 to the orderer O1. O1 packages transaction T1 from application A1 and transaction T2 from application A2 together with other transactions from other applications in the network into block B2. We can see that in B2, the transaction order is T1,T2,T3,T4,T6,T5 – which may not be the order in which these transactions arrived at the orderer! (This example shows a very simplified ordering service configuration with only one ordering node.)

It's worth noting that the sequencing of transactions in a block is not necessarily the same as the order received by the ordering service, since there can be multiple ordering service nodes that receive transactions at approximately the same time. What's important is that the ordering service puts the transactions into a strict order, and peers will use this order when validating and committing transactions.

This strict ordering of transactions within blocks makes Hyperledger Fabric a little different from other blockchains where the same transaction can be packaged into multiple different blocks that compete to form a chain. In Hyperledger Fabric, the blocks generated by the ordering service are **final**. Once a transaction has been written to a block, its position in the ledger is immutably assured. As we said earlier, Hyperledger Fabric's finality means that there are no **ledger forks** — validated transactions will never be reverted or dropped.

We can also see that, whereas peers execute smart contracts and process transactions, orderers most definitely do not. Every authorized transaction that arrives at an orderer is mechanically packaged in a block — the orderer makes no judgement as to the content of a transaction (except for channel configuration transactions, as mentioned earlier).

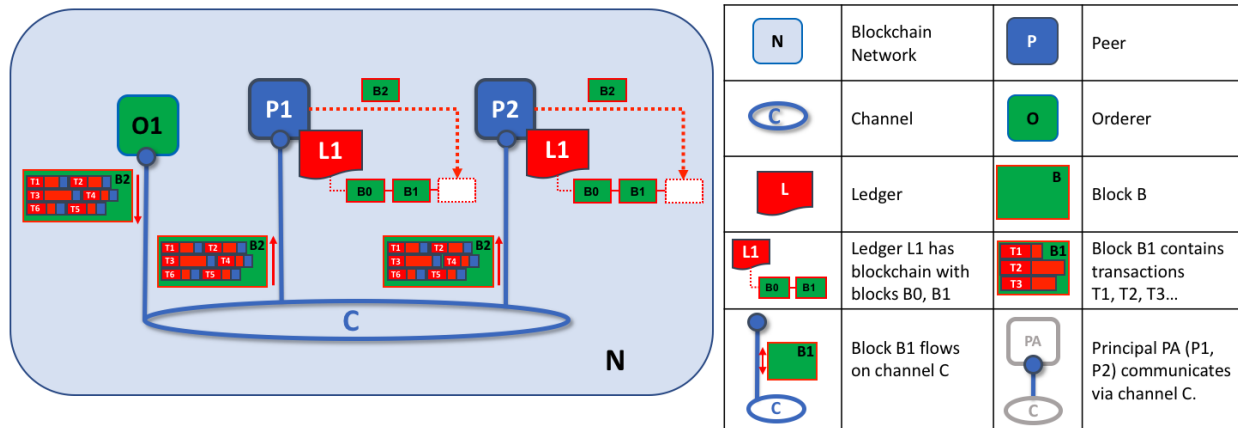
At the end of phase two, we see that orderers have been responsible for the simple but vital processes of collecting proposed transaction updates, ordering them, and packaging them into blocks, ready for distribution.

Phase three: Validation and commit

The third phase of the transaction workflow involves the distribution and subsequent validation of blocks from the orderer to the peers, where they can be committed to the ledger.

Phase 3 begins with the orderer distributing blocks to all peers connected to it. It's also worth noting that not every peer needs to be connected to an orderer — peers can cascade blocks to other peers using the [gossip](#) protocol.

Each peer will validate distributed blocks independently, but in a deterministic fashion, ensuring that ledgers remain consistent. Specifically, each peer in the channel will validate each transaction in the block to ensure it has been endorsed by the required organization's peers, that its endorsements match, and that it hasn't become invalidated by other recently committed transactions which may have been in-flight when the transaction was originally endorsed. Invalidated transactions are still retained in the immutable block created by the orderer, but they are marked as invalid by the peer and do not update the ledger's state.



The second role of an ordering node is to distribute blocks to peers. In this example, orderer O1 distributes block B2 to peer P1 and peer P2. Peer P1 processes block B2, resulting in a new block being added to ledger L1 on P1. In parallel, peer P2 processes block B2, resulting in a new block being added to ledger L1 on P2. Once this process is complete, the ledger L1 has been consistently updated on peers P1 and P2, and each may inform connected applications that the transaction has been processed.

In summary, phase three sees the blocks generated by the ordering service applied consistently to the ledger. The strict ordering of transactions into blocks allows each peer to validate that transaction updates are consistently applied across the blockchain network.

For a deeper look at phase 3, refer back to the [Peers](#) topic.

4.9.5 Ordering service implementations

While every ordering service currently available handles transactions and configuration updates the same way, there are nevertheless several different implementations for achieving consensus on the strict ordering of transactions between ordering service nodes.

For information about how to stand up an ordering node (regardless of the implementation the node will be used in), check out [our documentation on deploying a production ordering service](#).

- **Raft** (recommended)

New as of v1.4.1, Raft is a crash fault tolerant (CFT) ordering service based on an implementation of [Raft protocol](#) in [etcd](#). Raft follows a “leader and follower” model, where a leader node is elected (per channel) and its decisions are replicated by the followers. Raft ordering services should be easier to set up and manage than Kafka-based ordering services, and their design allows different organizations to contribute nodes to a distributed ordering service.

- **Kafka** (deprecated in v2.x)

Similar to Raft-based ordering, Apache Kafka is a CFT implementation that uses a “leader and follower” node configuration. Kafka utilizes a ZooKeeper ensemble for management purposes. The Kafka based ordering service has been available since Fabric v1.0, but many users may find the additional administrative overhead of managing a Kafka cluster intimidating or undesirable.

- **Solo** (deprecated in v2.x)

The Solo implementation of the ordering service is intended for test only and consists only of a single ordering node. It has been deprecated and may be removed entirely in a future release. Existing users of Solo should move to a single node Raft network for equivalent function.

4.9.6 Raft

For information on how to customize the `orderer.yaml` file that determines the configuration of an ordering node, check out the [Checklist for a production ordering node](#).

The go-to ordering service choice for production networks, the Fabric implementation of the established Raft protocol uses a “leader and follower” model, in which a leader is dynamically elected among the ordering nodes in a channel (this collection of nodes is known as the “consenter set”), and that leader replicates messages to the follower nodes. Because the system can sustain the loss of nodes, including leader nodes, as long as there is a majority of ordering nodes (what’s known as a “quorum”) remaining, Raft is said to be “crash fault tolerant” (CFT). In other words, if there are three nodes in a channel, it can withstand the loss of one node (leaving two remaining). If you have five nodes in a channel, you can lose two nodes (leaving three remaining nodes). This feature of a Raft ordering service is a factor in the establishment of a high availability strategy for your ordering service. Additionally, in a production environment, you would want to spread these nodes across data centers and even locations. For example, by putting one node in three different data centers. That way, if a data center or entire location becomes unavailable, the nodes in the other data centers continue to operate.

From the perspective of the service they provide to a network or a channel, Raft and the existing Kafka-based ordering service (which we’ll talk about later) are similar. They’re both CFT ordering services using the leader and follower design. If you are an application developer, smart contract developer, or peer administrator, you will not notice a functional difference between an ordering service based on Raft versus Kafka. However, there are a few major differences worth considering, especially if you intend to manage an ordering service.

- Raft is easier to set up. Although Kafka has many admirers, even those admirers will (usually) admit that deploying a Kafka cluster and its ZooKeeper ensemble can be tricky, requiring a high level of expertise in Kafka infrastructure and settings. Additionally, there are many more components to manage with Kafka than with Raft, which means that there are more places where things can go wrong. Kafka also has its own versions, which must be coordinated with your orderers. **With Raft, everything is embedded into your ordering node.**
- Kafka and Zookeeper are not designed to be run across large networks. While Kafka is CFT, it should be run in a tight group of hosts. This means that practically speaking you need to have one organization run the Kafka cluster. Given that, having ordering nodes run by different organizations when using Kafka (which Fabric supports) doesn’t decentralize the nodes because ultimately the nodes all go to a Kafka cluster which is under the control of a single organization. With Raft, each organization can have its own ordering nodes, participating in the ordering service, which leads to a more decentralized system.
- Raft is supported natively, which means that users are required to get the requisite images and learn how to use Kafka and ZooKeeper on their own. Likewise, support for Kafka-related issues is handled through [Apache](#), the open-source developer of Kafka, not Hyperledger Fabric. The Fabric Raft implementation, on the other hand, has been developed and will be supported within the Fabric developer community and its support apparatus.
- Where Kafka uses a pool of servers (called “Kafka brokers”) and the admin of the orderer organization specifies how many nodes they want to use on a particular channel, Raft allows the users to specify which ordering nodes will be deployed to which channel. In this way, peer organizations can make sure that, if they also own an orderer, this node will be made a part of a ordering service of that channel, rather than trusting and depending on a central admin to manage the Kafka nodes.

- Raft is the first step toward Fabric’s development of a byzantine fault tolerant (BFT) ordering service. As we’ll see, some decisions in the development of Raft were driven by this. If you are interested in BFT, learning how to use Raft should ease the transition.

For all of these reasons, support for Kafka-based ordering service is being deprecated in Fabric v2.x.

Note: Similar to Solo and Kafka, a Raft ordering service can lose transactions after acknowledgement of receipt has been sent to a client. For example, if the leader crashes at approximately the same time as a follower provides acknowledgement of receipt. Therefore, application clients should listen on peers for transaction commit events regardless (to check for transaction validity), but extra care should be taken to ensure that the client also gracefully tolerates a timeout in which the transaction does not get committed in a configured timeframe. Depending on the application, it may be desirable to resubmit the transaction or collect a new set of endorsements upon such a timeout.

Raft concepts

While Raft offers many of the same features as Kafka — albeit in a simpler and easier-to-use package — it functions substantially different under the covers from Kafka and introduces a number of new concepts, or twists on existing concepts, to Fabric.

Log entry. The primary unit of work in a Raft ordering service is a “log entry”, with the full sequence of such entries known as the “log”. We consider the log consistent if a majority (a quorum, in other words) of members agree on the entries and their order, making the logs on the various orderers replicated.

Consenter set. The ordering nodes actively participating in the consensus mechanism for a given channel and receiving replicated logs for the channel.

Finite-State Machine (FSM). Every ordering node in Raft has an FSM and collectively they’re used to ensure that the sequence of logs in the various ordering nodes is deterministic (written in the same sequence).

Quorum. Describes the minimum number of consenter that need to affirm a proposal so that transactions can be ordered. For every consenter set, this is a **majority** of nodes. In a cluster with five nodes, three must be available for there to be a quorum. If a quorum of nodes is unavailable for any reason, the ordering service cluster becomes unavailable for both read and write operations on the channel, and no new logs can be committed.

Leader. This is not a new concept — Kafka also uses leaders — but it’s critical to understand that at any given time, a channel’s consenter set elects a single node to be the leader (we’ll describe how this happens in Raft later). The leader is responsible for ingesting new log entries, replicating them to follower ordering nodes, and managing when an entry is considered committed. This is not a special **type** of orderer. It is only a role that an orderer may have at certain times, and then not others, as circumstances determine.

Follower. Again, not a new concept, but what’s critical to understand about followers is that the followers receive the logs from the leader and replicate them deterministically, ensuring that logs remain consistent. As we’ll see in our section on leader election, the followers also receive “heartbeat” messages from the leader. In the event that the leader stops sending those message for a configurable amount of time, the followers will initiate a leader election and one of them will be elected the new leader.

Raft in a transaction flow

Every channel runs on a **separate** instance of the Raft protocol, which allows each instance to elect a different leader. This configuration also allows further decentralization of the service in use cases where clusters are made up of ordering nodes controlled by different organizations. Ordering nodes can be added or removed from a channel as needed as long as only a single node is added or removed at a time. While this configuration creates more overhead in the form of redundant heartbeat messages and goroutines, it lays necessary groundwork for BFT.

In Raft, transactions (in the form of proposals or configuration updates) are automatically routed by the ordering node that receives the transaction to the current leader of that channel. This means that peers and applications do not need to know who the leader node is at any particular time. Only the ordering nodes need to know.

When the orderer validation checks have been completed, the transactions are ordered, packaged into blocks, consented on, and distributed, as described in phase two of our transaction flow.

Architectural notes

How leader election works in Raft

Although the process of electing a leader happens within the orderer’s internal processes, it’s worth noting how the process works.

Raft nodes are always in one of three states: follower, candidate, or leader. All nodes initially start out as a **follower**. In this state, they can accept log entries from a leader (if one has been elected), or cast votes for leader. If no log entries or heartbeats are received for a set amount of time (for example, five seconds), nodes self-promote to the **candidate** state. In the candidate state, nodes request votes from other nodes. If a candidate receives a quorum of votes, then it is promoted to a **leader**. The leader must accept new log entries and replicate them to the followers.

For a visual representation of how the leader election process works, check out [The Secret Lives of Data](#).

Snapshots

If an ordering node goes down, how does it get the logs it missed when it is restarted?

While it’s possible to keep all logs indefinitely, in order to save disk space, Raft uses a process called “snapshotting”, in which users can define how many bytes of data will be kept in the log. This amount of data will conform to a certain number of blocks (which depends on the amount of data in the blocks. Note that only full blocks are stored in a snapshot).

For example, let’s say lagging replica R1 was just reconnected to the network. Its latest block is 100. Leader L is at block 196, and is configured to snapshot at amount of data that in this case represents 20 blocks. R1 would therefore receive block 180 from L and then make a `Deliver` request for blocks 101 to 180. Blocks 180 to 196 would then be replicated to R1 through the normal Raft protocol.

Kafka (deprecated in v2.x)

The other crash fault tolerant ordering service supported by Fabric is an adaptation of a Kafka distributed streaming platform for use as a cluster of ordering nodes. You can read more about Kafka at the [Apache Kafka Web site](#), but at a high level, Kafka uses the same conceptual “leader and follower” configuration used by Raft, in which transactions (which Kafka calls “messages”) are replicated from the leader node to the follower nodes. In the event the leader node goes down, one of the followers becomes the leader and ordering can continue, ensuring fault tolerance, just as with Raft.

The management of the Kafka cluster, including the coordination of tasks, cluster membership, access control, and controller election, among others, is handled by a ZooKeeper ensemble and its related APIs.

Kafka clusters and ZooKeeper ensembles are notoriously tricky to set up, so our documentation assumes a working knowledge of Kafka and ZooKeeper. If you decide to use Kafka without having this expertise, you should complete, *at a minimum*, the first six steps of the [Kafka Quickstart guide](#) before experimenting with the Kafka-based ordering service. You can also consult [this sample configuration file](#) for a brief explanation of the sensible defaults for Kafka and ZooKeeper.

To learn how to bring up a Kafka-based ordering service, check out [our documentation on Kafka](#).

4.10 Smart Contracts and Chaincode

Audience: Architects, application and smart contract developers, administrators

From an application developer's perspective, a **smart contract**, together with the **ledger**, form the heart of a Hyperledger Fabric blockchain system. Whereas a ledger holds facts about the current and historical state of a set of business objects, a **smart contract** defines the executable logic that generates new facts that are added to the ledger. A **chaincode** is typically used by administrators to group related smart contracts for deployment, but can also be used for low level system programming of Fabric. In this topic, we'll focus on why both **smart contracts** and **chaincode** exist, and how and when to use them.

In this topic, we'll cover:

- *What is a smart contract*
- *A note on terminology*
- *Smart contracts and the ledger*
- *How to develop a smart contract*
- *The importance of endorsement policies*
- *Valid transactions*
- *Channels and chaincode definitions*
- *Communicating between smart contracts*
- *What is system chaincode?*

4.10.1 Smart contract

Before businesses can transact with each other, they must define a common set of contracts covering common terms, data, rules, concept definitions, and processes. Taken together, these contracts lay out the **business model** that govern all of the interactions between transacting parties.



A smart contract defines the rules between different organizations in executable code. Applications invoke a smart contract to generate transactions that are recorded on the ledger.

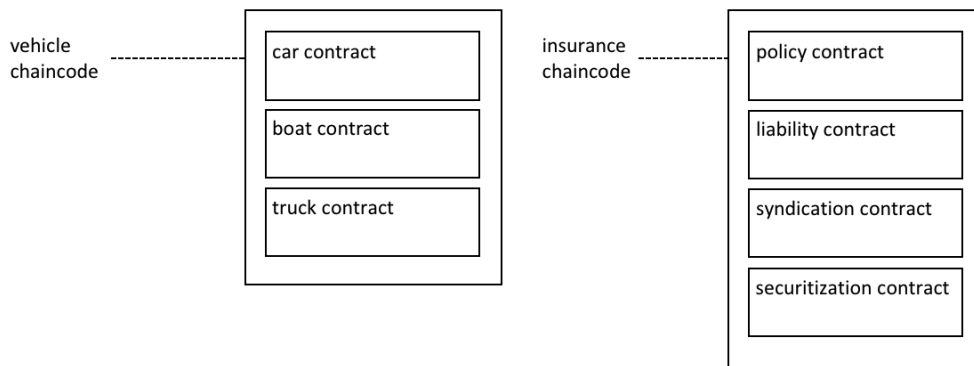
Using a blockchain network, we can turn these contracts into executable programs – known in the industry as **smart contracts** – to open up a wide variety of new possibilities. That's because a smart contract can implement the gov-

ernance rules for **any** type of business object, so that they can be automatically enforced when the smart contract is executed. For example, a smart contract might ensure that a new car delivery is made within a specified timeframe, or that funds are released according to prearranged terms, improving the flow of goods or capital respectively. Most importantly however, the execution of a smart contract is much more efficient than a manual human business process.

In the [diagram above](#), we can see how two organizations, ORG1 and ORG2 have defined a `car` smart contract to query, transfer and update cars. Applications from these organizations invoke this smart contract to perform an agreed step in a business process, for example to transfer ownership of a specific car from ORG1 to ORG2.

4.10.2 Terminology

Hyperledger Fabric users often use the terms **smart contract** and **chaincode** interchangeably. In general, a smart contract defines the **transaction logic** that controls the lifecycle of a business object contained in the world state. It is then packaged into a chaincode which is then deployed to a blockchain network. Think of smart contracts as governing transactions, whereas chaincode governs how smart contracts are packaged for deployment.



A smart contract is defined within a chaincode. Multiple smart contracts can be defined within the same chaincode. When a chaincode is deployed, all smart contracts within it are made available to applications.

In the diagram, we can see a `vehicle` chaincode that contains three smart contracts: `cars`, `boats` and `trucks`. We can also see an `insurance` chaincode that contains four smart contracts: `policy`, `liability`, `syndication` and `securitization`. In both cases these contracts cover key aspects of the business process relating to vehicles and insurance. In this topic, we will use the `car` contract as an example. We can see that a smart contract is a domain specific program which relates to specific business processes, whereas a chaincode is a technical container of a group of related smart contracts.

4.10.3 Ledger

At the simplest level, a blockchain immutably records transactions which update states in a ledger. A smart contract programmatically accesses two distinct pieces of the ledger – a **blockchain**, which immutably records the history of all transactions, and a **world state** that holds a cache of the current value of these states, as it's the current value of an object that is usually required.

Smart contracts primarily **put**, **get** and **delete** states in the world state, and can also query the immutable blockchain record of transactions.

- A **get** typically represents a query to retrieve information about the current state of a business object.
- A **put** typically creates a new business object or modifies an existing one in the ledger world state.
- A **delete** typically represents the removal of a business object from the current state of the ledger, but not its history.

Smart contracts have many [APIs](#) available to them. Critically, in all cases, whether transactions create, read, update or delete business objects in the world state, the blockchain contains an [immutable record](#) of these changes.

4.10.4 Development

Smart contracts are the focus of application development, and as we've seen, one or more smart contracts can be defined within a single chaincode. Deploying a chaincode to a network makes all its smart contracts available to the organizations in that network. It means that only administrators need to worry about chaincode; everyone else can think in terms of smart contracts.

At the heart of a smart contract is a set of `transaction` definitions. For example, look at `assetTransfer.js` [here](#), where you can see a smart contract transaction that creates a new asset:

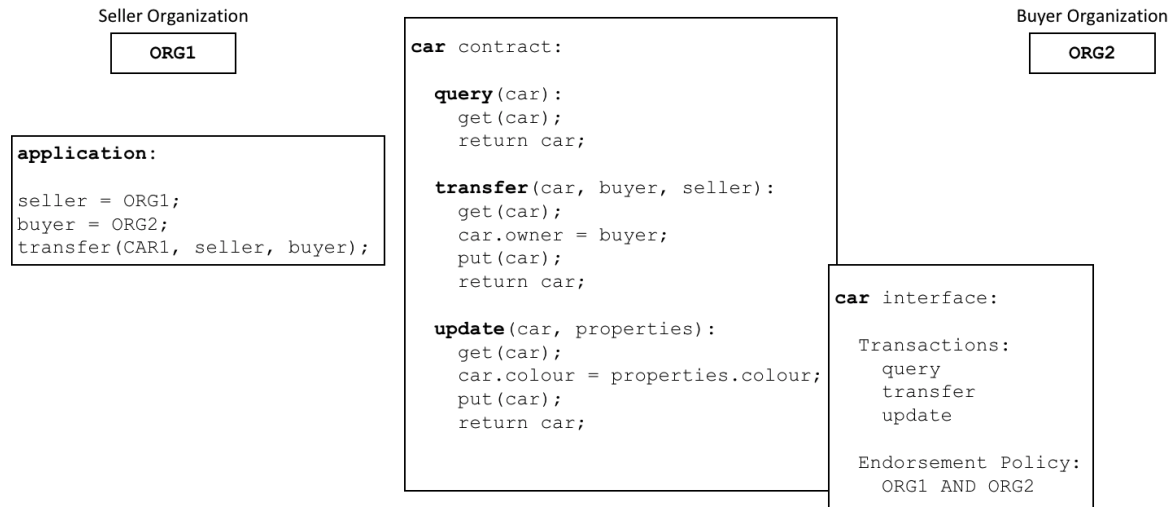
```
async CreateAsset(ctx, id, color, size, owner, appraisedValue) {
  const asset = {
    ID: id,
    Color: color,
    Size: size,
    Owner: owner,
    AppraisedValue: appraisedValue,
  };
  return ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
}
```

You can learn more about the **Basic** smart contract in the [Writing your first application](#) tutorial.

A smart contract can describe an almost infinite array of business use cases relating to immutability of data in multi-organizational decision making. The job of a smart contract developer is to take an existing business process that might govern financial prices or delivery conditions, and express it as a smart contract in a programming language such as JavaScript, Go, or Java. The legal and technical skills required to convert centuries of legal language into programming language is increasingly practiced by **smart contract auditors**. You can learn about how to design and develop a smart contract in the [Developing applications](#) topic.

4.10.5 Endorsement

Associated with every chaincode is an endorsement policy that applies to all of the smart contracts defined within it. An endorsement policy is very important; it indicates which organizations in a blockchain network must sign a transaction generated by a given smart contract in order for that transaction to be declared **valid**.



Every smart contract has an endorsement policy associated with it. This endorsement policy identifies which organizations must approve transactions generated by the smart contract before those transactions can be identified as valid.

An example endorsement policy might define that three of the four organizations participating in a blockchain network must sign a transaction before it is considered **valid**. All transactions, whether **valid** or **invalid** are added to a distributed ledger, but only **valid** transactions update the world state.

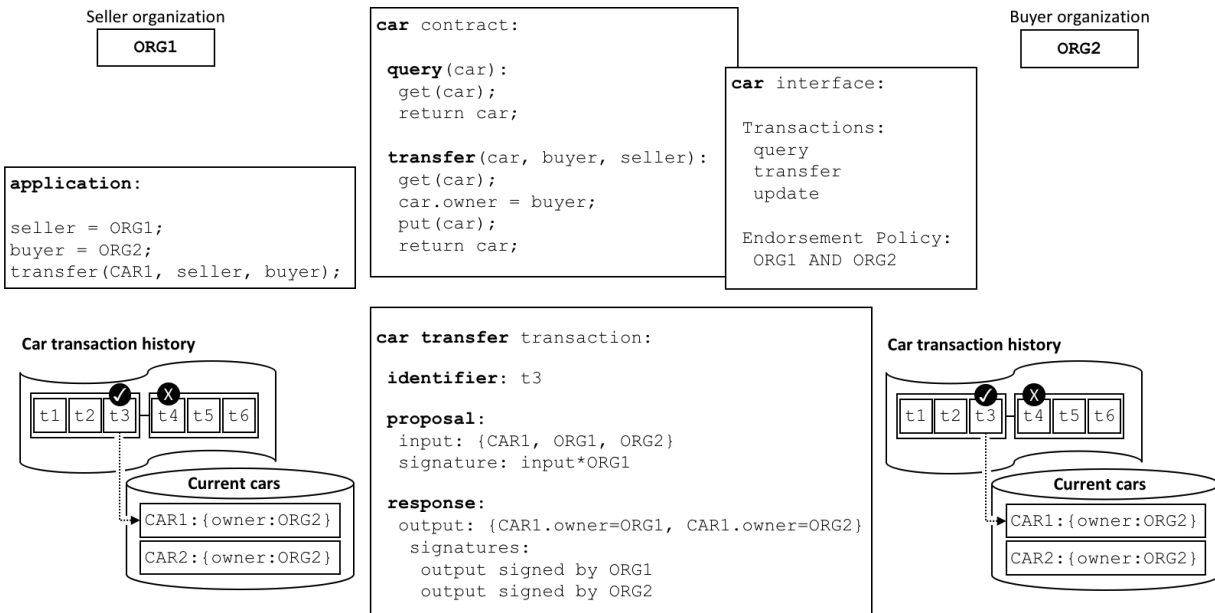
If an endorsement policy specifies that more than one organization must sign a transaction, then the smart contract must be executed by a sufficient set of organizations in order for a valid transaction to be generated. In the example [above](#), a smart contract transaction to `transfer` a car would need to be executed and signed by both ORG1 and ORG2 for it to be valid.

Endorsement policies are what make Hyperledger Fabric different to other blockchains like Ethereum or Bitcoin. In these systems valid transactions can be generated by any node in the network. Hyperledger Fabric more realistically models the real world; transactions must be validated by trusted organizations in a network. For example, a government organization must sign a valid `issueIdentity` transaction, or both the buyer and seller of a car must sign a `car transfer` transaction. Endorsement policies are designed to allow Hyperledger Fabric to better model these types of real-world interactions.

Finally, endorsement policies are just one example of [policy](#) in Hyperledger Fabric. Other policies can be defined to identify who can query or update the ledger, or add or remove participants from the network. In general, policies should be agreed in advance by the consortium of organizations in a blockchain network, although they are not set in stone. Indeed, policies themselves can define the rules by which they can be changed. And although an advanced topic, it is also possible to define [custom endorsement policy](#) rules over and above those provided by Fabric.

4.10.6 Valid transactions

When a smart contract executes, it runs on a peer node owned by an organization in the blockchain network. The contract takes a set of input parameters called the **transaction proposal** and uses them in combination with its program logic to read and write the ledger. Changes to the world state are captured as a **transaction proposal response** (or just **transaction response**) which contains a **read-write set** with both the states that have been read, and the new states that are to be written if the transaction is valid. Notice that the world state **is not updated when the smart contract is executed**!



All transactions have an identifier, a proposal, and a response signed by a set of organizations. All transactions are recorded on the blockchain, whether valid or invalid, but only valid transactions contribute to the world state.

Examine the car transfer transaction. You can see a transaction t3 for a car transfer between ORG1 and ORG2. See how the transaction has input {CAR1, ORG1, ORG2} and output {CAR1.owner=ORG1, CAR1.owner=ORG2}, representing the change of owner from ORG1 to ORG2. Notice how the input is signed by the application's organization ORG1, and the output is signed by *both* organizations identified by the endorsement policy, ORG1 and ORG2. These signatures were generated by using each actor's private key, and mean that anyone in the network can verify that all actors in the network are in agreement about the transaction details.

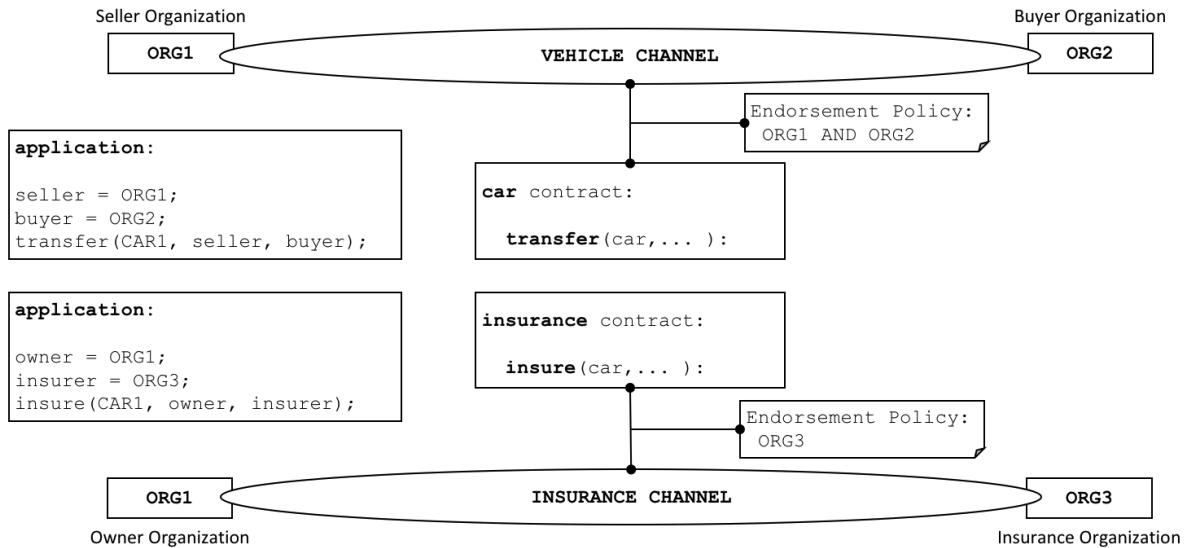
A transaction that is distributed to all peer nodes in the network is **validated** in two phases by each peer. Firstly, the transaction is checked to ensure it has been signed by sufficient organizations according to the endorsement policy. Secondly, it is checked to ensure that the current value of the world state matches the read set of the transaction when it was signed by the endorsing peer nodes; that there has been no intermediate update. If a transaction passes both these tests, it is marked as **valid**. All transactions are added to the blockchain history, whether **valid** or **invalid**, but only **valid** transactions result in an update to the world state.

In our example, t3 is a valid transaction, so the owner of CAR1 has been updated to ORG2. However, t4 (not shown) is an invalid transaction, so while it was recorded in the ledger, the world state was not updated, and CAR2 remains owned by ORG2.

Finally, to understand how to use a smart contract or chaincode with world state, read the [chaincode namespace topic](#).

4.10.7 Channels

Hyperledger Fabric allows an organization to simultaneously participate in multiple, separate blockchain networks via **channels**. By joining multiple channels, an organization can participate in a so-called **network of networks**. Channels provide an efficient sharing of infrastructure while maintaining data and communications privacy. They are independent enough to help organizations separate their work traffic with different counterparties, but integrated enough to allow them to coordinate independent activities when necessary.



A channel provides a completely separate communication mechanism between a set of organizations. When a chaincode definition is committed to a channel, all the smart contracts within the chaincode are made available to the applications on that channel.

While the smart contract code is installed inside a chaincode package on an organizations peers, channel members can only execute a smart contract after the chaincode has been defined on a channel. The **chaincode definition** is a struct that contains the parameters that govern how a chaincode operates. These parameters include the chaincode name, version, and the endorsement policy. Each channel member agrees to the parameters of a chaincode by approving a chaincode definition for their organization. When a sufficient number of organizations (a majority by default) have approved to the same chaincode definition, the definition can be committed to the channel. The smart contracts inside the chaincode can then be executed by channel members, subject to the endorsement policy specified in the chaincode definition. The endorsement policy applies equally to all smart contracts defined within the same chaincode.

In the example [above](#), a `car` contract is defined on the `VEHICLE` channel, and an `insurance` contract is defined on the `INSURANCE` channel. The chaincode definition of `car` specifies an endorsement policy that requires both `ORG1` and `ORG2` to sign transactions before they can be considered valid. The chaincode definition of the `insurance` contract specifies that only `ORG3` is required to endorse a transaction. `ORG1` participates in two networks, the `VEHICLE` channel and the `INSURANCE` network, and can coordinate activity with `ORG2` and `ORG3` across these two networks.

The chaincode definition provides a way for channel members to agree on the governance of a chaincode before they start using the smart contract to transact on the channel. Building on the example above, both `ORG1` and `ORG2` want to endorse transactions that invoke the `car` contract. Because the default policy requires that a majority of organizations approve a chaincode definition, both organizations need to approve an endorsement policy of `AND{ORG1, ORG2}`. Otherwise, `ORG1` and `ORG2` would approve different chaincode definitions and would be unable to commit the chaincode definition to the channel as a result. This process guarantees that a transaction from the `car` smart contract needs to be approved by both organizations.

4.10.8 Intercommunication

A Smart Contract can call other smart contracts both within the same channel and across different channels. In this way, they can read and write world state data to which they would not otherwise have access due to smart contract namespaces.

There are limitations to this inter-contract communication, which are described fully in the [chaincode namespace](#) topic.

4.10.9 System chaincode

The smart contracts defined within a chaincode encode the domain dependent rules for a business process agreed between a set of blockchain organizations. However, a chaincode can also define low-level program code which corresponds to domain independent *system* interactions, unrelated to these smart contracts for business processes.

The following are the different types of system chaincodes and their associated abbreviations:

- `_lifecycle` runs in all peers and manages the installation of chaincode on your peers, the approval of chaincode definitions for your organization, and the committing of chaincode definitions to channels. You can read more about how `_lifecycle` implements the Fabric chaincode lifecycle [process](#).
- Lifecycle system chaincode (LSCC) manages the chaincode lifecycle for the 1.x releases of Fabric. This version of lifecycle required that chaincode be instantiated or upgraded on channels. You can still use LSCC to manage your chaincode if you have the channel application capability set to V1_4_x or below.
- **Configuration system chaincode (CSCC)** runs in all peers to handle changes to a channel configuration, such as a policy update. You can read more about this process in the following chaincode [topic](#).
- **Query system chaincode (QSCC)** runs in all peers to provide ledger APIs which include block query, transaction query etc. You can read more about these ledger APIs in the transaction context [topic](#).
- **Endorsement system chaincode (ESCC)** runs in endorsing peers to cryptographically sign a transaction response. You can read more about how the ESCC implements this [process](#).
- **Validation system chaincode (VSCC)** validates a transaction, including checking endorsement policy and read-write set versioning. You can read more about the VSCC implements this [process](#).

It is possible for low level Fabric developers and administrators to modify these system chaincodes for their own uses. However, the development and management of system chaincodes is a specialized activity, quite separate from the development of smart contracts, and is not normally necessary. Changes to system chaincodes must be handled with extreme care as they are fundamental to the correct functioning of a Hyperledger Fabric network. For example, if a system chaincode is not developed correctly, one peer node may update its copy of the world state or blockchain differently compared to another peer node. This lack of consensus is one form of a **ledger fork**, a very undesirable situation.

4.11 Fabric chaincode lifecycle

4.11.1 What is Chaincode?

Chaincode is a program, written in [Go](#), [Node.js](#), or [Java](#) that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it may be considered as a “smart contract”. Ledger updates created by a chaincode are scoped exclusively to that chaincode and can’t be accessed directly by another chaincode. However, within the same network, given the appropriate permission a chaincode may invoke another chaincode to access its state.

In this concept topic, we will explore chaincode through the eyes of a blockchain network operator rather than an application developer. Chaincode operators can use this topic as a guide to how to use the Fabric chaincode lifecycle to deploy and manage chaincode on their network.

4.11.2 Deploying a chaincode

The Fabric chaincode lifecycle is a process that allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. A network operator would use the Fabric lifecycle to perform the following tasks:

- *Install and define a chaincode*
- *Upgrade a chaincode*
- *Deployment Scenarios*
- *Migrate to the new Fabric lifecycle*

You can use the Fabric chaincode lifecycle by creating a new channel and setting the channel capabilities to V2_0. You will not be able to use the old lifecycle to install, instantiate, or update a chaincode on channels with V2_0 capabilities enabled. However, you can still invoke chaincode installed using the previous lifecycle model after you enable V2_0 capabilities. If you are upgrading from a v1.4.x network and need to edit your channel configurations to enable the new lifecycle, check out [Enabling the new chaincode lifecycle](#).

4.11.3 Install and define a chaincode

Fabric chaincode lifecycle requires that organizations agree to the parameters that define a chaincode, such as name, version, and the chaincode endorsement policy. Channel members come to agreement using the following four steps. Not every organization on a channel needs to complete each step.

1. **Package the chaincode:** This step can be completed by one organization or by each organization.
2. **Install the chaincode on your peers:** Every organization that will use the chaincode to endorse a transaction or query the ledger needs to complete this step.
3. **Approve a chaincode definition for your organization:** Every organization that will use the chaincode needs to complete this step. The chaincode definition needs to be approved by a sufficient number of organizations to satisfy the channel's LifecycleEndorsement policy (a majority, by default) before the chaincode can be started on the channel.
4. **Commit the chaincode definition to the channel:** The commit transaction needs to be submitted by one organization once the required number of organizations on the channel have approved. The submitter first collects endorsements from enough peers of the organizations that have approved, and then submits the transaction to commit the chaincode definition.

This topic provides a detailed overview of the operations of the Fabric chaincode lifecycle rather than the specific commands. To learn more about how to use the Fabric lifecycle using the Peer CLI, see the [Deploying a smart contract to a channel tutorial](#) or the [peer lifecycle command reference](#).

Step One: Packaging the smart contract

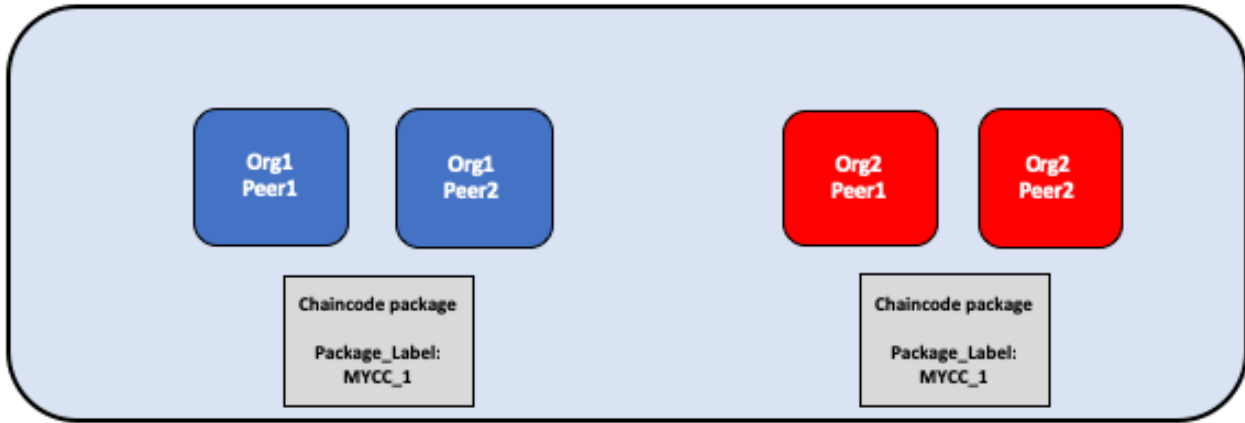
Chaincode needs to be packaged in a tar file before it can be installed on your peers. You can package a chaincode using the Fabric peer binaries, the Node Fabric SDK, or a third party tool such as GNU tar. When you create a chaincode package, you need to provide a chaincode package label to create a succinct and human readable description of the package.

If you use a third party tool to package the chaincode, the resulting file needs to be in the format below. The Fabric peer binaries and the Fabric SDKs will automatically create a file in this format.

- The chaincode needs to be packaged in a tar file, ending with a `.tar.gz` file extension.
- The tar file needs to contain two files (no directory): a metadata file “metadata.json” and another tar “code.tar.gz” containing the chaincode files.

- “metadata.json” contains JSON that specifies the chaincode language, code path, and package label. You can see an example of a metadata file below:

```
{ "Path": "fabric-samples/asset-transfer-basic/chaincode-go", "Type": "golang", "Label": "basicv1" }
```

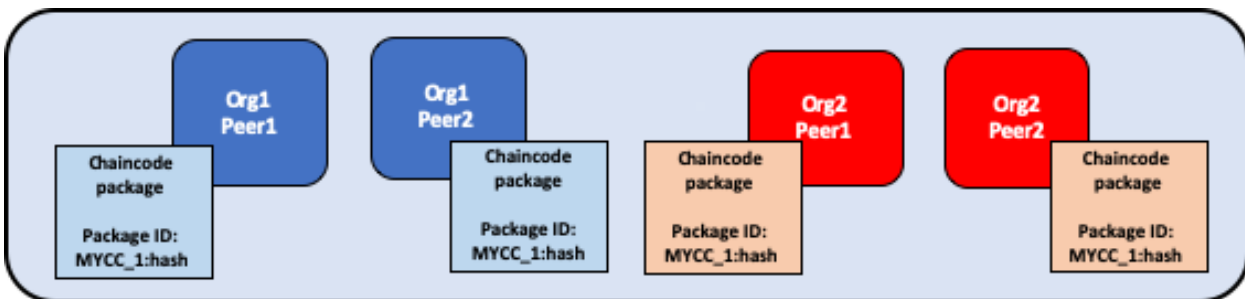


The chaincode is packaged separately by Org1 and Org2. Both organizations use MYCC_1 as their package label in order to identify the package using the name and version. It is not necessary for organizations to use the same package label.

Step Two: Install the chaincode on your peers

You need to install the chaincode package on every peer that will execute and endorse transactions. Whether using the CLI or an SDK, you need to complete this step using your **Peer Administrator**. Your peer will build the chaincode after the chaincode is installed, and return a build error if there is a problem with your chaincode. It is recommended that organizations only package a chaincode once, and then install the same package on every peer that belongs to their org. If a channel wants to ensure that each organization is running the same chaincode, one organization can package a chaincode and send it to other channel members out of band.

A successful install command will return a chaincode package identifier, which is the package label combined with a hash of the package. This package identifier is used to associate a chaincode package installed on your peers with a chaincode definition approved by your organization. **Save the identifier** for next step. You can also find the package identifier by querying the packages installed on your peer using the Peer CLI.



A peer administrator from Org1 and Org2 installs the chaincode package MYCC_1 on the peers joined to the channel. Installing the chaincode package builds the chaincode and creates a package identifier of MYCC_1:hash.

Step Three: Approve a chaincode definition for your organization

The chaincode is governed by a **chaincode definition**. When channel members approve a chaincode definition, the approval acts as a vote by an organization on the chaincode parameters it accepts. These approved organization definitions allow channel members to agree on a chaincode before it can be used on a channel. The chaincode definition includes the following parameters, which need to be consistent across organizations:

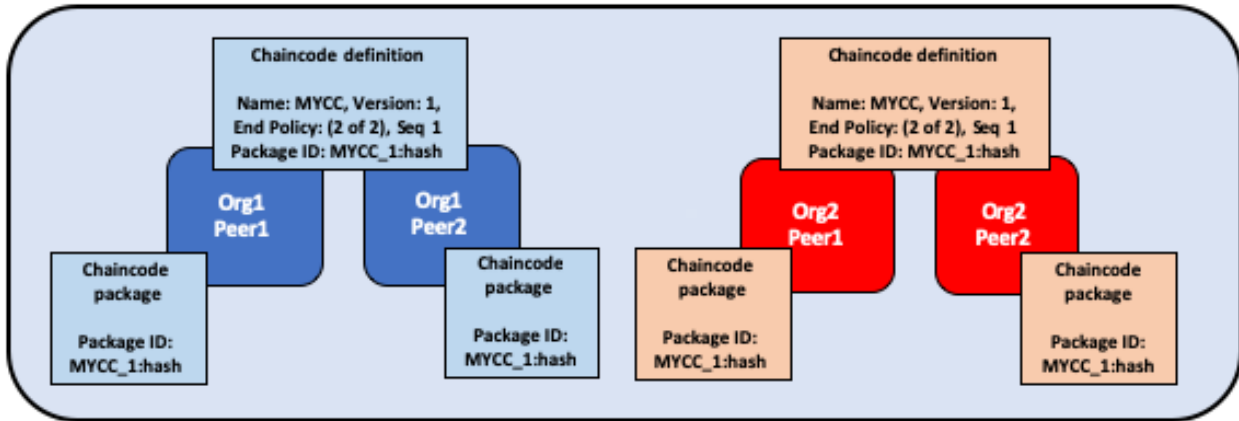
- **Name:** The name that applications will use when invoking the chaincode.
- **Version:** A version number or value associated with a given chaincodes package. If you upgrade the chaincode binaries, you need to change your chaincode version as well.
- **Sequence:** The number of times the chaincode has been defined. This value is an integer, and is used to keep track of chaincode upgrades. For example, when you first install and approve a chaincode definition, the sequence number will be 1. When you next upgrade the chaincode, the sequence number will be incremented to 2.
- **Endorsement Policy:** Which organizations need to execute and validate the transaction output. The endorsement policy can be expressed as a string passed to the CLI, or it can reference a policy in the channel config. By default, the endorsement policy is set to `Channel/Application/Endorsement`, which defaults to require that a majority of organizations in the channel endorse a transaction.
- **Collection Configuration:** The path to a private data collection definition file associated with your chaincode. For more information about private data collections, see the [Private Data architecture reference](#).
- **ESCC/VSCC Plugins:** The name of a custom endorsement or validation plugin to be used by this chaincode.
- **Initialization:** If you use the low level APIs provided by the Fabric Chaincode Shim API, your chaincode needs to contain an `Init` function that is used to initialize the chaincode. This function is required by the chaincode interface, but does not necessarily need to be invoked by your applications. When you approve a chaincode definition, you can specify whether `Init` must be called prior to `Invoke`s. If you specify that `Init` is required, Fabric will ensure that the `Init` function is invoked before any other function in the chaincode and is only invoked once. Requesting the execution of the `Init` function allows you to implement logic that is run when the chaincode is initialized, for example to set some initial state. You will need to call `Init` to initialize the chaincode every time you increment the version of a chaincode, assuming the chaincode definition that increments the version indicates that `Init` is required.

If you are using the Fabric peer CLI, you can use the `--init-required` flag when you approve and commit the chaincode definition to indicate that the `Init` function must be called to initialize the new chaincode version. To call `Init` using the Fabric peer CLI, use the `peer chaincode invoke` command and pass the `--isInit` flag.

If you are using the Fabric contract API, you do not need to include an `Init` method in your chaincode. However, you can still use the `--init-required` flag to request that the chaincode be initialized by a call from your applications. If you use the `--init-required` flag, you will need to pass the `--isInit` flag or parameter to a chaincode call in order to initialize the chaincode every time you increment the chaincode version. You can pass `--isInit` and initialize the chaincode using any function in your chaincode.

The chaincode definition also includes the **Package Identifier**. This is a required parameter for each organization that wants to use the chaincode. The package ID does not need to be the same for all organizations. An organization can approve a chaincode definition without installing a chaincode package or including the identifier in the definition.

Each channel member that wants to use the chaincode needs to approve a chaincode definition for their organization. This approval needs to be submitted to the ordering service, after which it is distributed to all peers. This approval needs to be submitted by your **Organization Administrator**. After the approval transaction has been successfully submitted, the approved definition is stored in a collection that is available to all the peers of your organization. As a result you only need to approve a chaincode for your organization once, even if you have multiple peers.



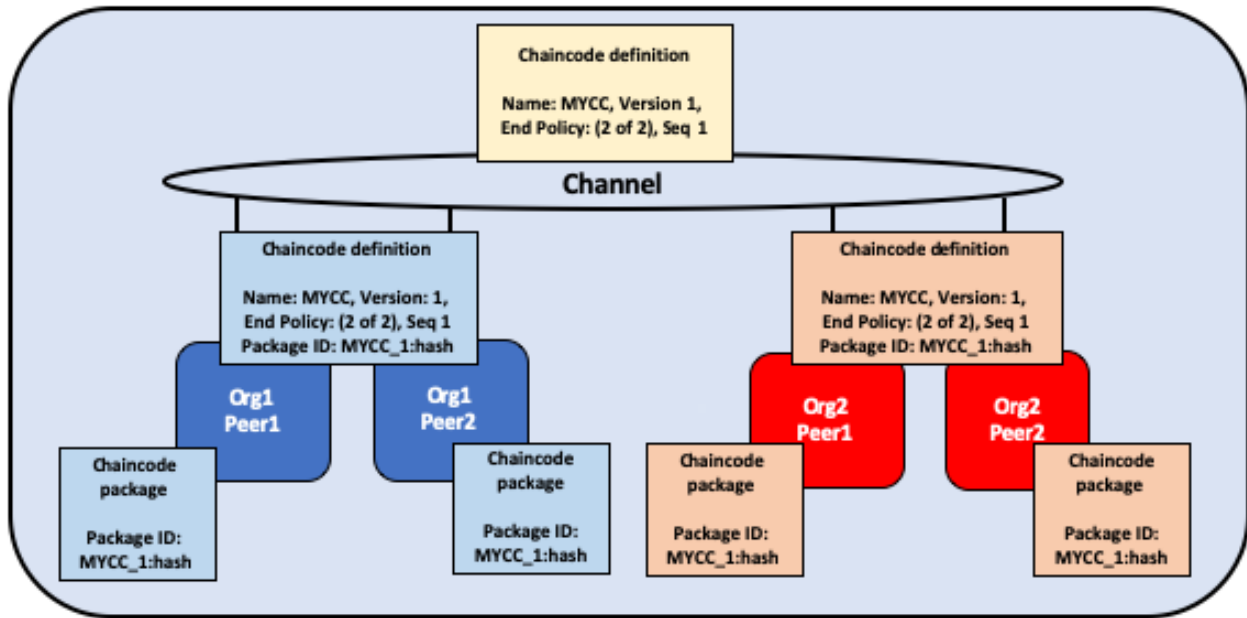
An organization administrator from Org1 and Org2 approve the chaincode definition of MYCC for their organization. The chaincode definition includes the chaincode name, version, and the endorsement policy, among other fields. Since both organizations will use the chaincode to endorse transactions, the approved definitions for both organizations need to include the packageID.

Step Four: Commit the chaincode definition to the channel

Once a sufficient number of channel members have approved a chaincode definition, one organization can commit the definition to the channel. You can use the `checkcommitreadiness` command to check whether committing the chaincode definition should be successful based on which channel members have approved a definition before committing it to the channel using the peer CLI. The commit transaction proposal is first sent to the peers of channel members, who query the chaincode definition approved for their organizations and endorse the definition if their organization has approved it. The transaction is then submitted to the ordering service, which then commits the chaincode definition to the channel. The commit definition transaction needs to be submitted as the **Organization Administrator**.

The number of organizations that need to approve a definition before it can be successfully committed to the channel is governed by the `Channel/Application/LifecycleEndorsement` policy. By default, this policy requires that a majority of organizations in the channel endorse the transaction. The `LifecycleEndorsement` policy is separate from the chaincode endorsement policy. For example, even if a chaincode endorsement policy only requires signatures from one or two organizations, a majority of channel members still need to approve the chaincode definition according to the default policy. When committing a channel definition, you need to target enough peer organizations in the channel to satisfy your `LifecycleEndorsement` policy. You can learn more about the Fabric chaincode lifecycle policies in the [Policies concept topic](#).

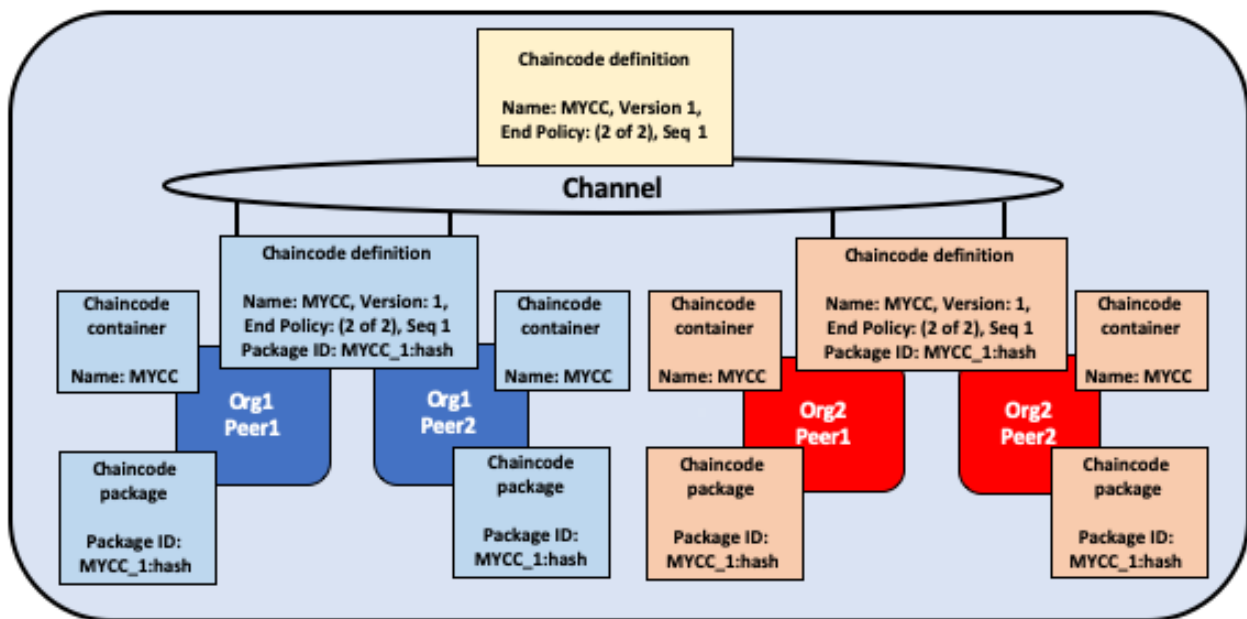
You can also set the `Channel/Application/LifecycleEndorsement` policy to be a signature policy and explicitly specify the set of organizations on the channel that can approve a chaincode definition. This allows you to create a channel where a select number of organizations act as chaincode administrators and govern the business logic used by the channel. You can also use a signature policy if your channel has a large number of Idemix organizations, which cannot approve chaincode definitions or endorse chaincode and may prevent the channel from reaching a majority as a result.



One organization administrator from Org1 or Org2 commits the chaincode definition to the channel. The definition on the channel does not include the packageID.

An organization can approve a chaincode definition without installing the chaincode package. If an organization does not need to use the chaincode, they can approve a chaincode definition without a package identifier to ensure that the Lifecycle Endorsement policy is satisfied.

After the chaincode definition has been committed to the channel, the chaincode container will launch on all of the peers where the chaincode has been installed, allowing channel members to start using the chaincode. It may take a few minutes for the chaincode container to start. You can use the chaincode definition to require the invocation of the `Init` function to initialize the chaincode. If the invocation of the `Init` function is requested, the first invoke of the chaincode must be a call to the `Init` function. The invoke of the `Init` function is subject to the chaincode endorsement policy.



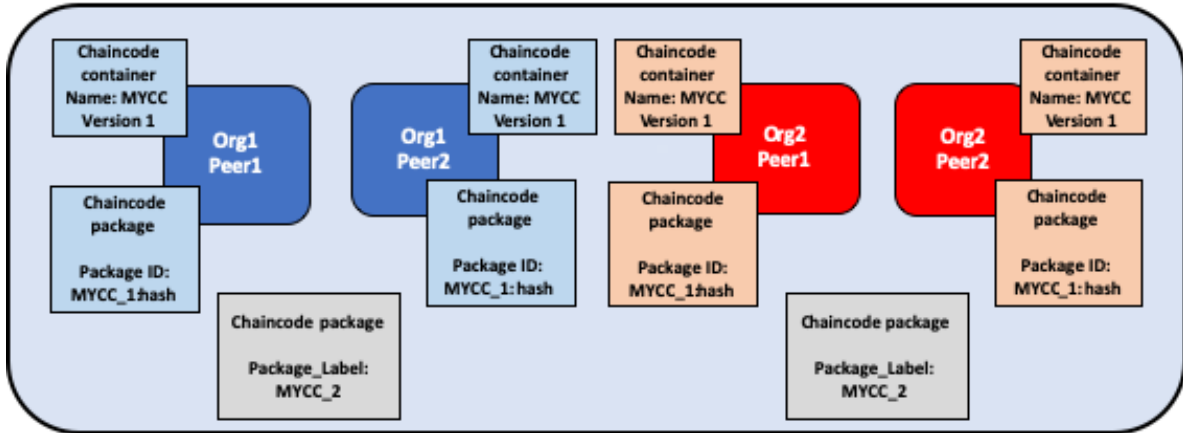
Once MYCC is defined on the channel, Org1 and Org2 can start using the chaincode. The first invoke of the chaincode

on each peer starts the chaincode container on that peer.

4.11.4 Upgrade a chaincode

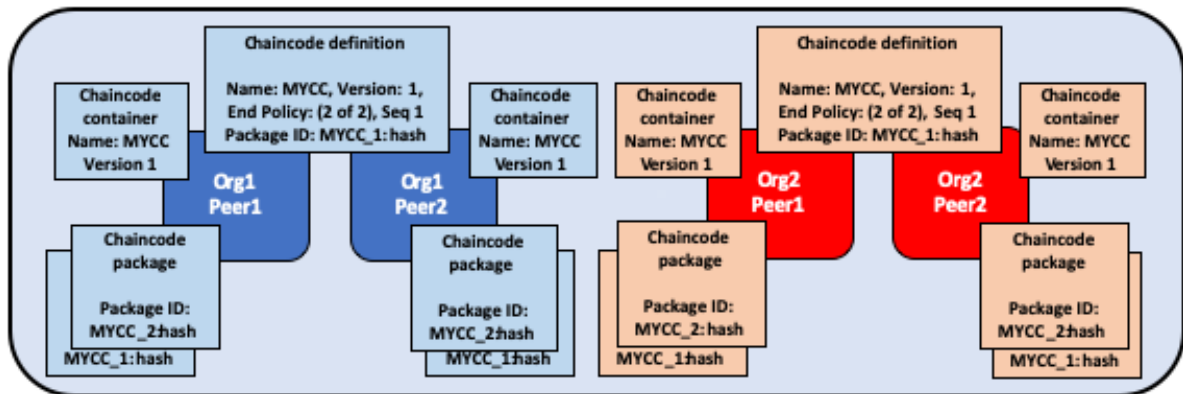
You can upgrade a chaincode using the same Fabric lifecycle process as you used to install and start the chaincode. You can upgrade the chaincode binaries, or only update the chaincode policies. Follow these steps to upgrade a chaincode:

1. **Repackage the chaincode:** You only need to complete this step if you are upgrading the chaincode binaries.



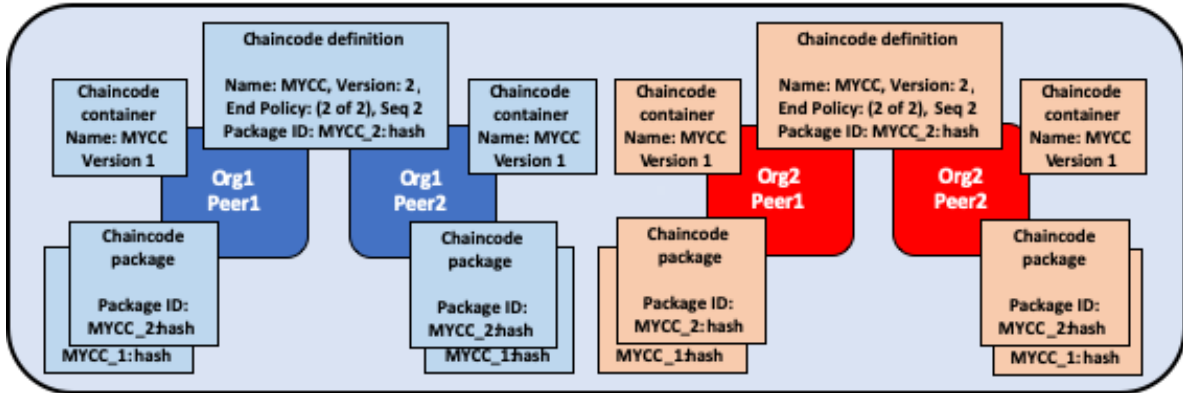
Org1 and Org2 upgrade the chaincode binaries and repackage the chaincode. Both organizations use a different package label.

2. **Install the new chaincode package on your peers:** Once again, you only need to complete this step if you are upgrading the chaincode binaries. Installing the new chaincode package will generate a package ID, which you will need to pass to the new chaincode definition. You also need to change the chaincode version, which is used by the lifecycle process to track if the chaincode binaries have been upgraded.



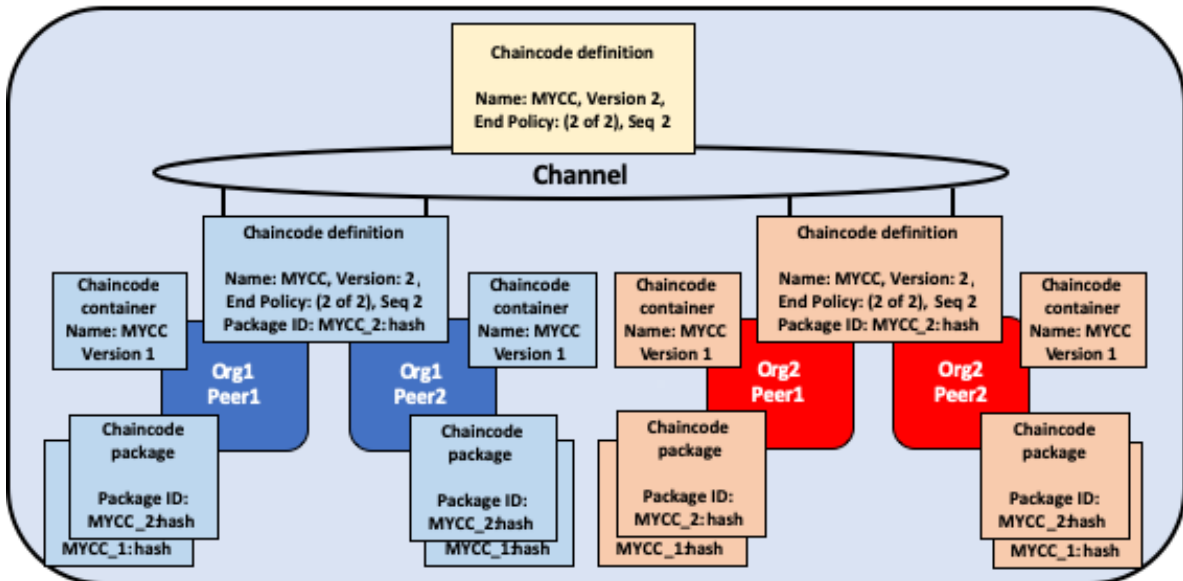
Org1 and Org2 install the new package on their peers. The installation creates a new packageID.

3. **Approve a new chaincode definition:** If you are upgrading the chaincode binaries, you need to update the chaincode version and the package ID in the chaincode definition. You can also update your chaincode endorsement policy without having to repackage your chaincode binaries. Channel members simply need to approve a definition with the new policy. The new definition needs to increment the **sequence** variable in the definition by one.



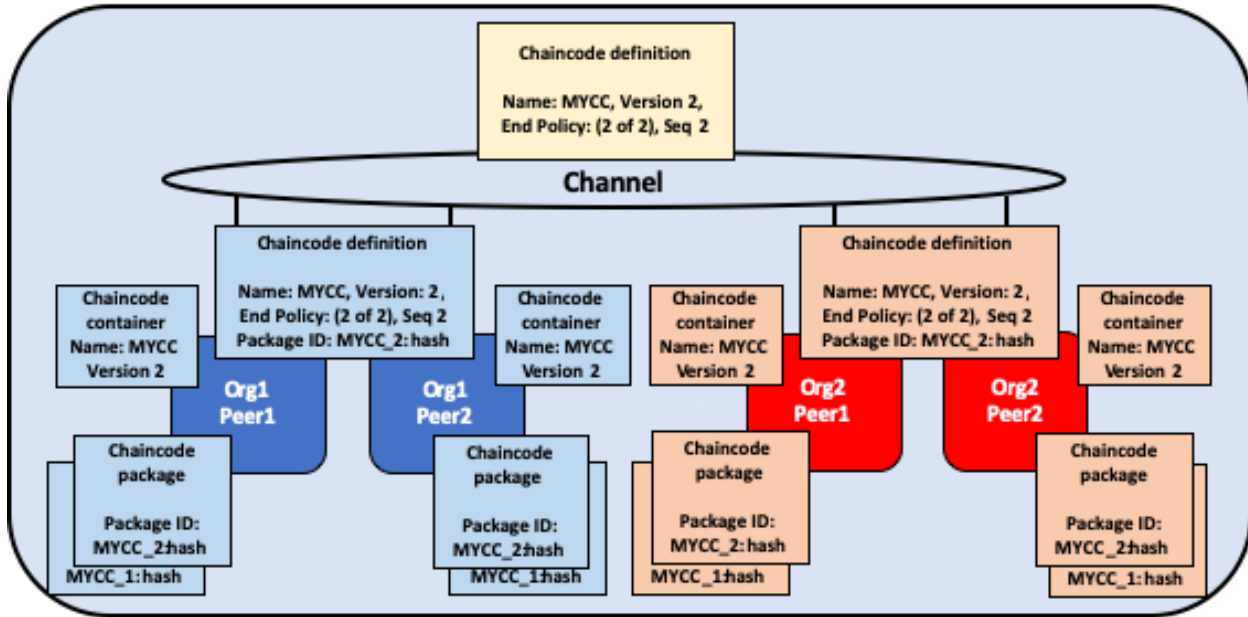
Organization administrators from Org1 and Org2 approve the new chaincode definition for their respective organizations. The new definition references the new packageID and changes the chaincode version. Since this is the first update of the chaincode, the sequence is incremented from one to two.

4. **Commit the definition to the channel:** When a sufficient number of channel members have approved the new chaincode definition, one organization can commit the new definition to upgrade the chaincode definition to the channel. There is no separate upgrade command as part of the lifecycle process.



An organization administrator from Org1 or Org2 commits the new chaincode definition to the channel.

After you commit the chaincode definition, a new chaincode container will launch with the code from the upgraded chaincode binaries. If you requested the execution of the `Init` function in the chaincode definition, you need to initialize the upgraded chaincode by invoking the `Init` function again after the new definition is successfully committed. If you updated the chaincode definition without changing the chaincode version, the chaincode container will remain the same and you do not need to invoke `Init` function.



Once the new definition has been committed to the channel, each peer will automatically start the new chaincode container.

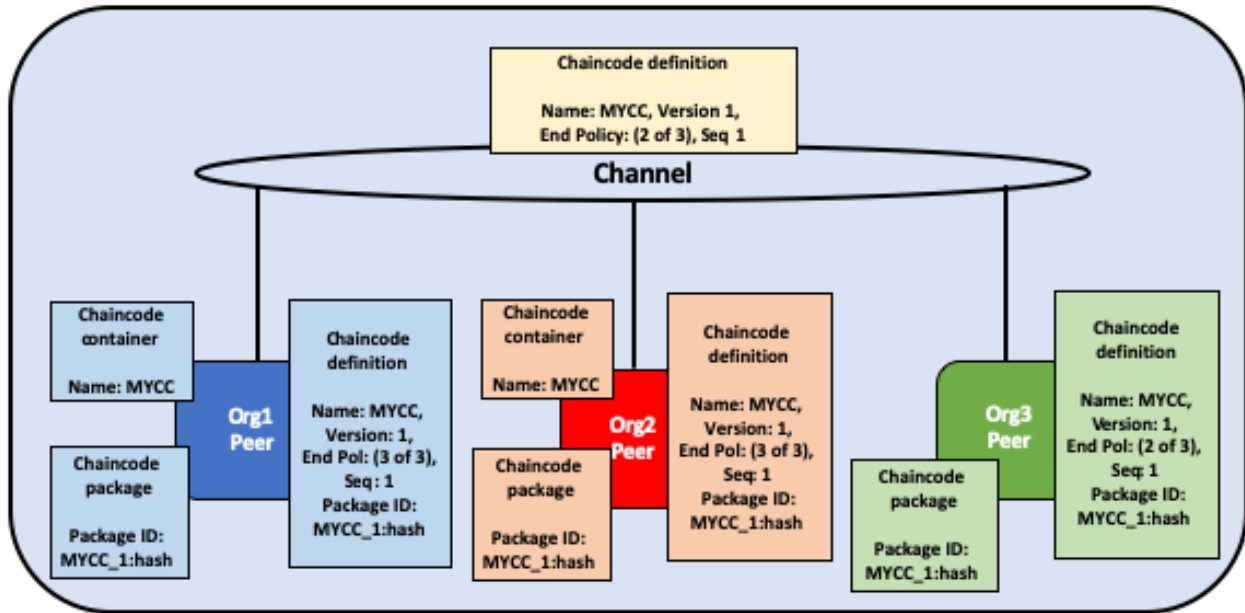
The Fabric chaincode lifecycle uses the **sequence** in the chaincode definition to keep track of upgrades. All channel members need to increment the sequence number by one and approve a new definition to upgrade the chaincode. The version parameter is used to track the chaincode binaries, and needs to be changed only when you upgrade the chaincode binaries.

4.11.5 Deployment scenarios

The following examples illustrate how you can use the Fabric chaincode lifecycle to manage channels and chaincode.

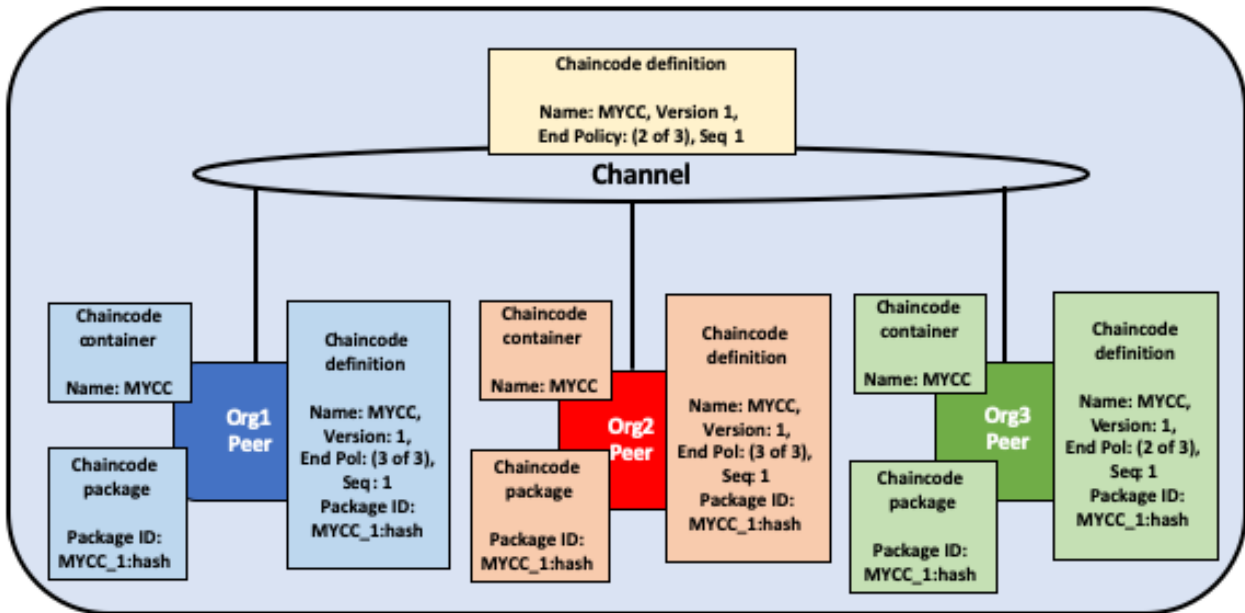
Joining a channel

A new organization can join a channel with a chaincode already defined, and start using the chaincode after installing the chaincode package and approving the chaincode definition that has already been committed to the channel.



Org3 joins the channel and approves the same chaincode definition that was previously committed to the channel by Org1 and Org2.

After approving the chaincode definition, the new organization can start using the chaincode after the package has been installed on their peers. The definition does not need to be committed again. If the endorsement policy is set the default policy that requires endorsements from a majority of channel members, then the endorsement policy will be updated automatically to include the new organization.

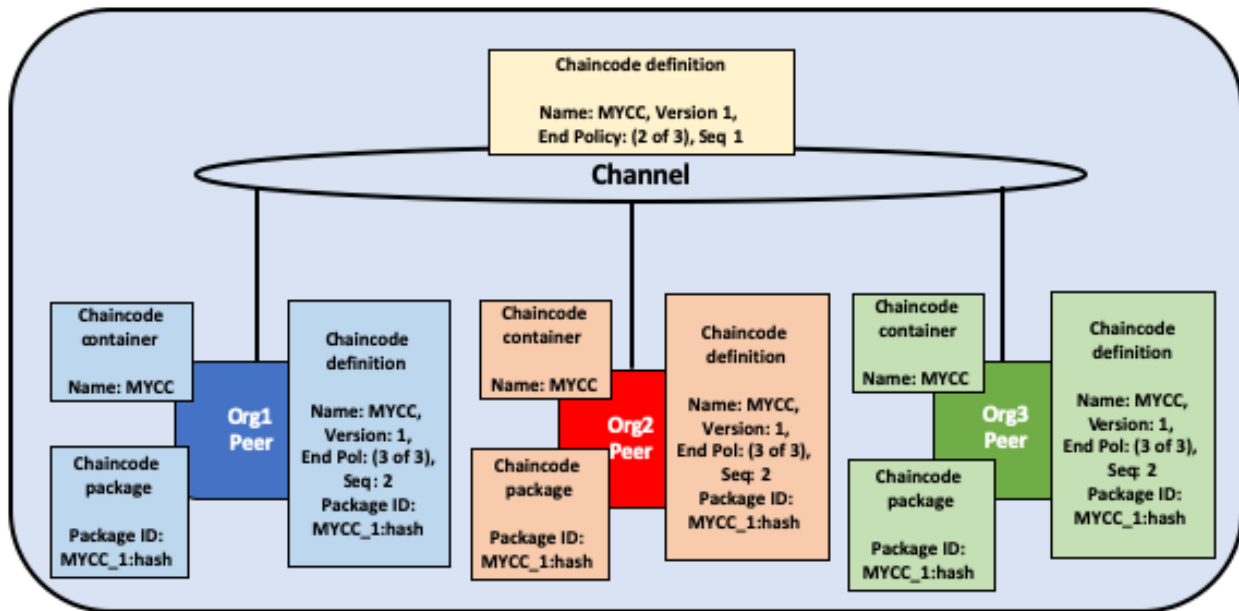


The chaincode container will start after the first invoke of the chaincode on the Org3 peer.

Updating an endorsement policy

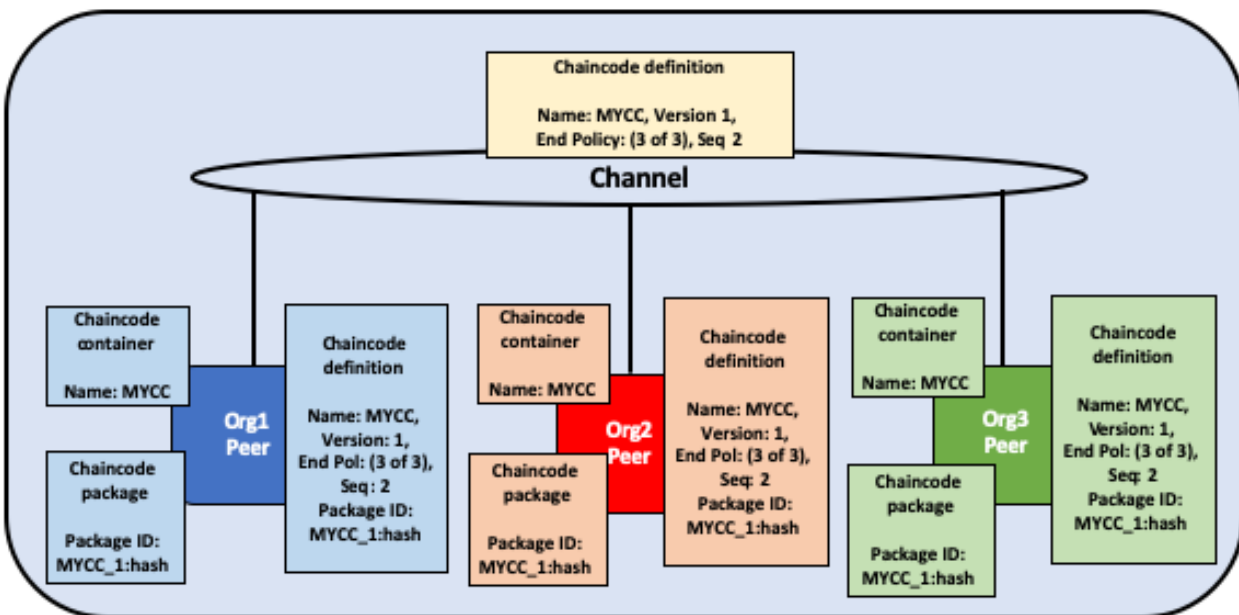
You can use the chaincode definition to update an endorsement policy without having to repack or re-install the chaincode. Channel members can approve a chaincode definition with a new endorsement policy and commit it to the

channel.



Org1, Org2, and Org3 approve a new endorsement policy requiring that all three organizations endorse a transaction. They increment the definition sequence from one to two, but do not need to update the chaincode version.

The new endorsement policy will take effect after the new definition is committed to the channel. Channel members do not have to restart the chaincode container by invoking the `chaincode` or executing the `Init` function in order to update the endorsement policy.

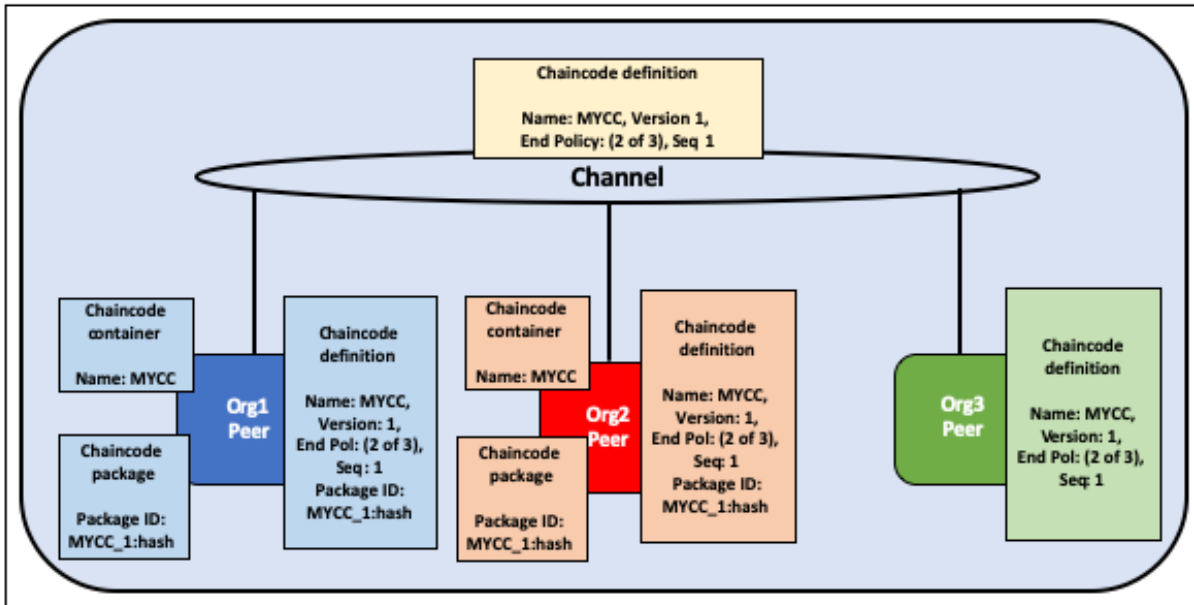


One organization commits the new chaincode definition to the channel to update the endorsement policy.

Approving a definition without installing the chaincode

You can approve a chaincode definition without installing the chaincode package. This allows you to endorse a chaincode definition before it is committed to the channel, even if you do not want to use the chaincode to endorse

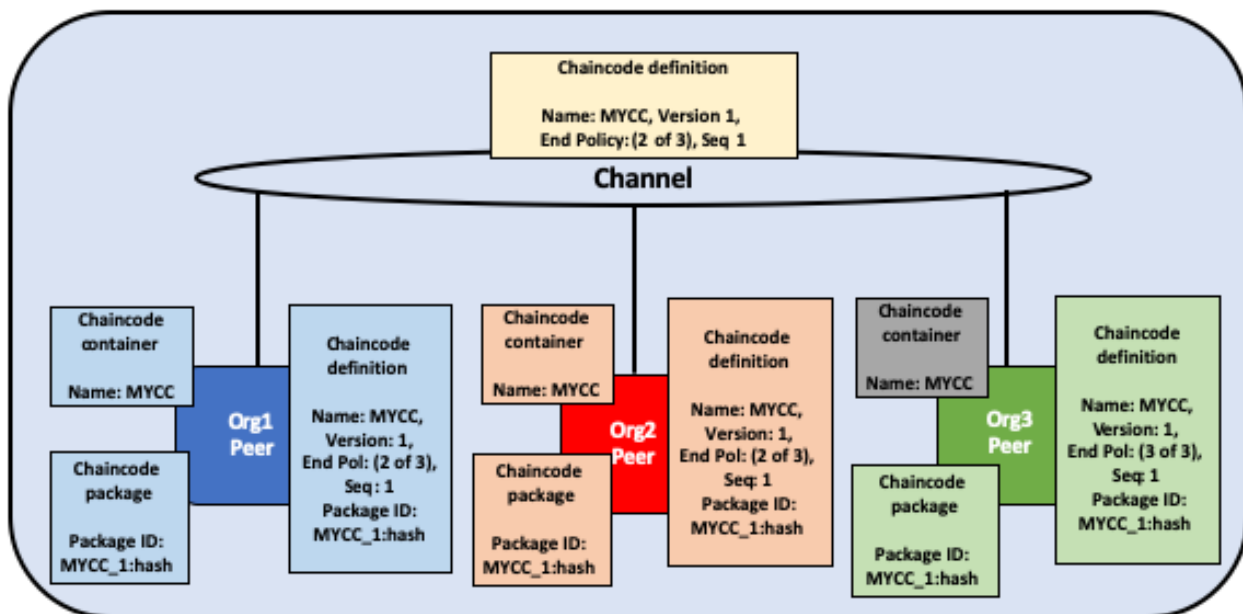
transactions or query the ledger. You need to approve the same parameters as other members of the channel, but not need to include the packageID as part of the chaincode definition.



Org3 does not install the chaincode package. As a result, they do not need to provide a packageID as part of chaincode definition. However, Org3 can still endorse the definition of MYCC that has been committed to the channel.

One organization disagrees on the chaincode definition

An organization that does not approve a chaincode definition that has been committed to the channel cannot use the chaincode. Organizations that have either not approved a chaincode definition, or approved a different chaincode definition will not be able to execute the chaincode on their peers.



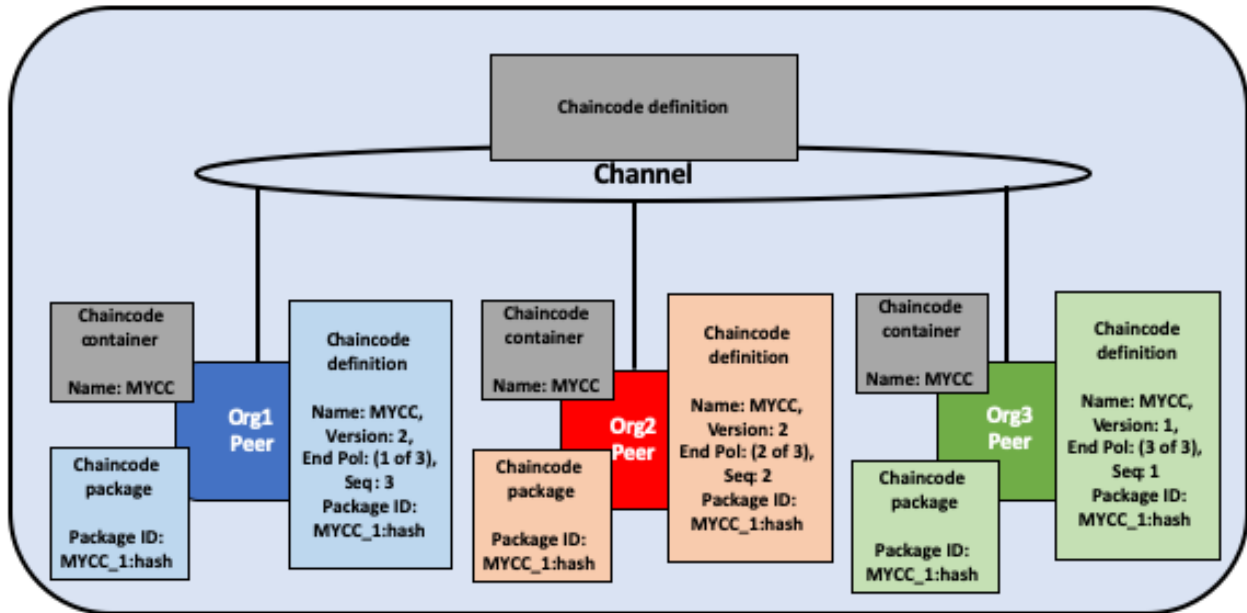
Org3 approves a chaincode definition with a different endorsement policy than Org1 and Org2. As a result, Org3 cannot use the MYCC chaincode on the channel. However, Org1 or Org2 can still get enough endorsements to commit

the definition to the channel and use the chaincode. Transactions from the chaincode will still be added to the ledger and stored on the Org3 peer. However, the Org3 will not be able to endorse transactions.

An organization can approve a new chaincode definition with any sequence number or version. This allows you to approve the definition that has been committed to the channel and start using the chaincode. You can also approve a new chaincode definition in order to correct any mistakes made in the process of approving or packaging a chaincode.

The channel does not agree on a chaincode definition

If the organizations on a channel do not agree on a chaincode definition, the definition cannot be committed to the channel. None of the channel members will be able to use the chaincode.

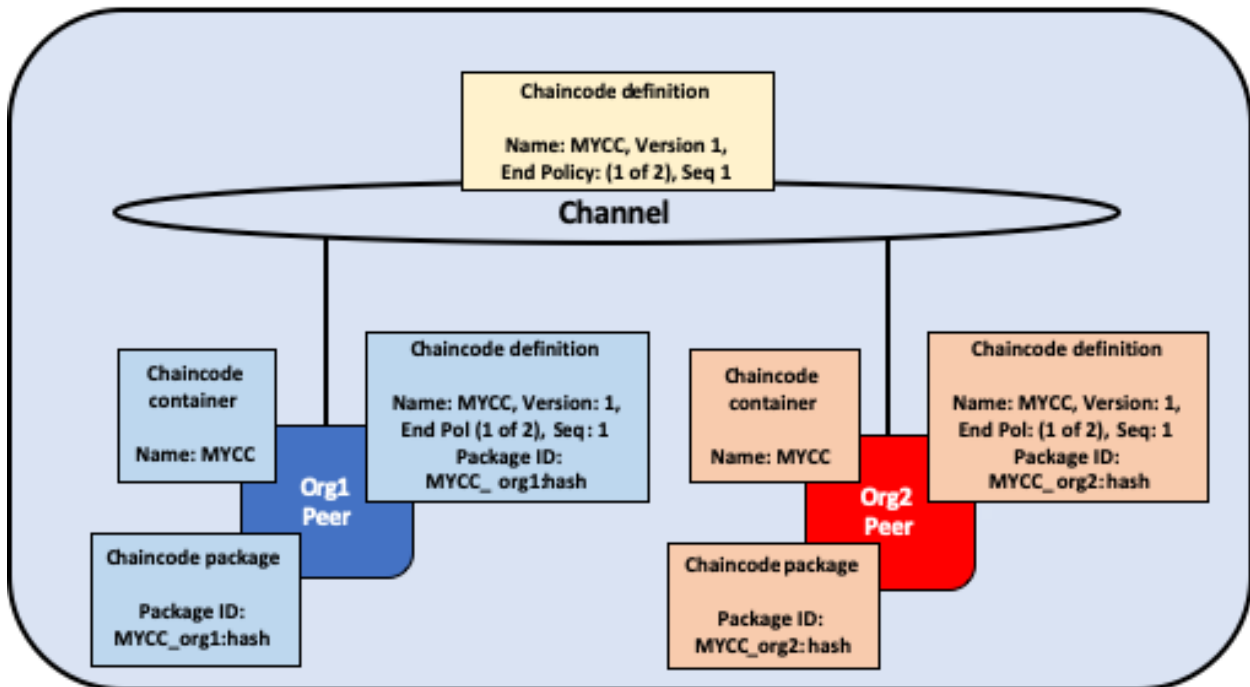


Org1, Org2, and Org3 all approve different chaincode definitions. As a result, no member of the channel can get enough endorsements to commit a chaincode definition to the channel. No channel member will be able to use the chaincode.

Organizations install different chaincode packages

Each organization can use a different packageID when they approve a chaincode definition. This allows channel members to install different chaincode binaries that use the same endorsement policy and read and write to data in the same chaincode namespace.

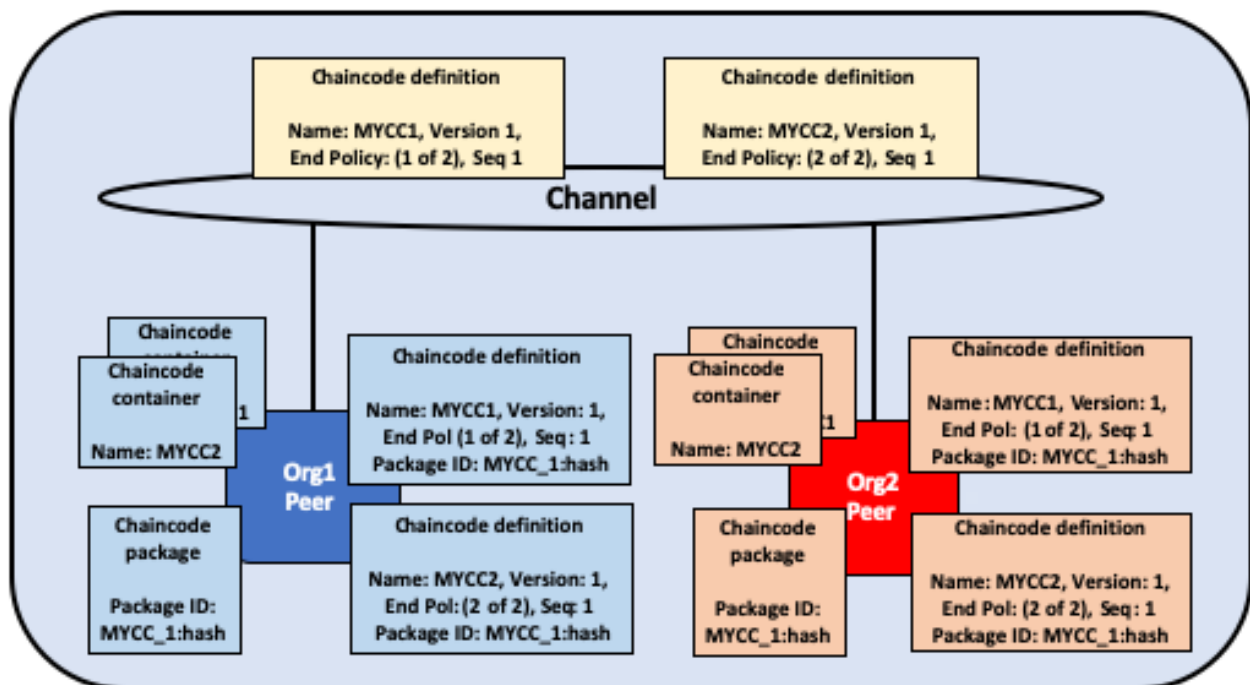
Organizations can use this capability to install smart contracts that contain business logic that is specific to their organization. Each organization's smart contract could contain additional validation that the organization requires before their peers endorse a transaction. Each organization can also write code that helps integrate the smart contract with data from their existing systems.



Org1 and Org2 each install versions of the MYCC chaincode containing business logic that is specific to their organization.

Creating multiple chaincodes using one package

You can use one chaincode package to create multiple chaincode instances on a channel by approving and committing multiple chaincode definitions. Each definition needs to specify a different chaincode name. This allows you to run multiple instances of a smart contract on a channel, but have the contract be subject to different endorsement policies.



Org1 and Org2 use the MYCC_1 chaincode package to approve and commit two different chaincode definitions. As a result, both peers have two chaincode containers running on their peers. MYCC1 has an endorsement policy of 1 out of 2, while MYCC2 has an endorsement policy of 2 out of 2.

4.11.6 Migrate to the new Fabric lifecycle

For information about migrating to the new lifecycle, check out [Considerations for getting to v2.0](#).

If you need to update your channel configurations to enable the new lifecycle, check out [Enabling the new chaincode lifecycle](#).

4.11.7 More information

You can watch video below to learn more about the motivation of the new Fabric chaincode lifecycle and how it is implemented.

4.12 Private data

4.12.1 What is private data?

In cases where a group of organizations on a channel need to keep data private from other organizations on that channel, they have the option to create a new channel comprising just the organizations who need access to the data. However, creating separate channels in each of these cases creates additional administrative overhead (maintaining chaincode versions, policies, MSPs, etc), and doesn't allow for use cases in which you want all channel participants to see a transaction while keeping a portion of the data private.

That's why Fabric offers the ability to create **private data collections**, which allow a defined subset of organizations on a channel the ability to endorse, commit, or query private data without having to create a separate channel.

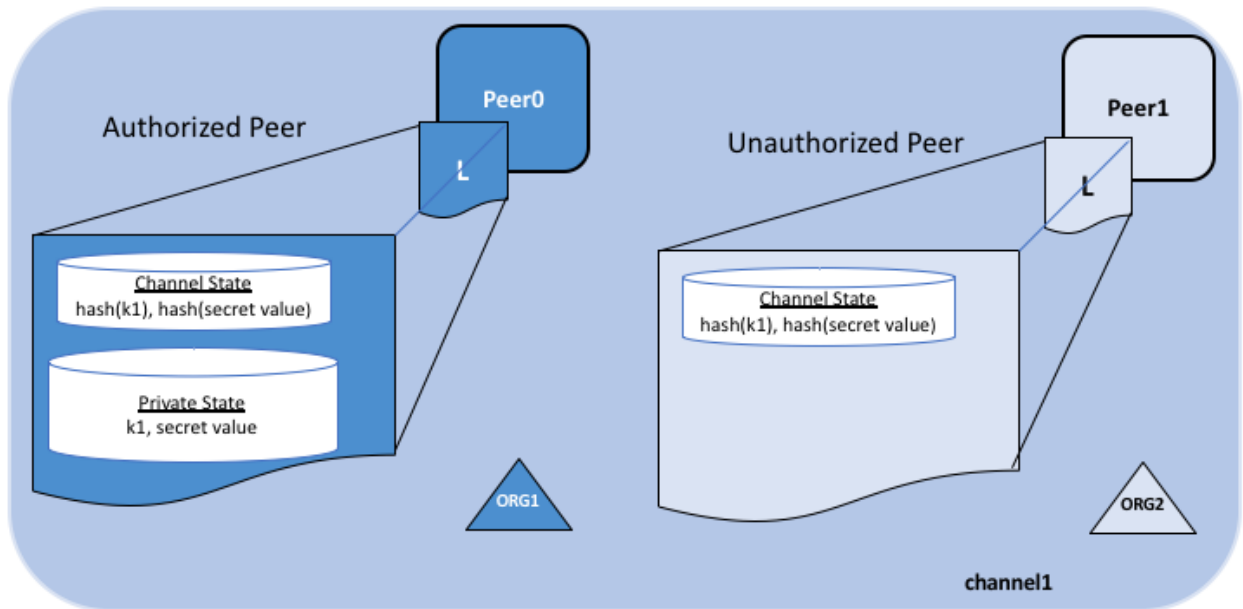
Private data collections can be defined explicitly within a chaincode definition. Additionally, every chaincode has an implicit private data namespace reserved for organization-specific private data. These implicit organization-specific private data collections can be used to store an individual organization's private data, which is useful if you would like to store private data related to a single organization, such as details about an asset owned by an organization or an organization's approval for a step in a multi-party business process implemented in chaincode.

4.12.2 What is a private data collection?

A collection is the combination of two elements:

1. **The actual private data**, sent peer-to-peer [via gossip protocol](#) to only the organization(s) authorized to see it. This data is stored in a private state database on the peers of authorized organizations, which can be accessed from chaincode on these authorized peers. The ordering service is not involved here and does not see the private data. Note that because gossip distributes the private data peer-to-peer across authorized organizations, it is required to set up anchor peers on the channel, and configure `CORE_PEER_GOSSIP_EXTERNALENDPOINT` on each peer, in order to bootstrap cross-organization communication.
2. **A hash of that data**, which is endorsed, ordered, and written to the ledgers of every peer on the channel. The hash serves as evidence of the transaction and is used for state validation and can be used for audit purposes.

The following diagram illustrates the ledger contents of a peer authorized to have private data and one which is not.



Collection members may decide to share the private data with other parties if they get into a dispute or if they want to transfer the asset to a third party. The third party can then compute the hash of the private data and see if it matches the state on the channel ledger, proving that the state existed between the collection members at a certain point in time.

In some cases, you may decide to have a set of collections each comprised of a single organization. For example an organization may record private data in their own collection, which could later be shared with other channel members and referenced in chaincode transactions. We'll see examples of this in the sharing private data topic below.

When to use a collection within a channel vs. a separate channel

- Use **channels** when entire transactions (and ledgers) must be kept confidential within a set of organizations that are members of the channel.
- Use **collections** when transactions (and ledgers) must be shared among a set of organizations, but when only a subset of those organizations should have access to some (or all) of the data within a transaction. Additionally, since private data is disseminated peer-to-peer rather than via blocks, use private data collections when transaction data must be kept confidential from ordering service nodes.

4.12.3 A use case to explain collections

Consider a group of five organizations on a channel who trade produce:

- A **Farmer** selling his goods abroad
- A **Distributor** moving goods abroad
- A **Shipper** moving goods between parties
- A **Wholesaler** purchasing goods from distributors
- A **Retailer** purchasing goods from shippers and wholesalers

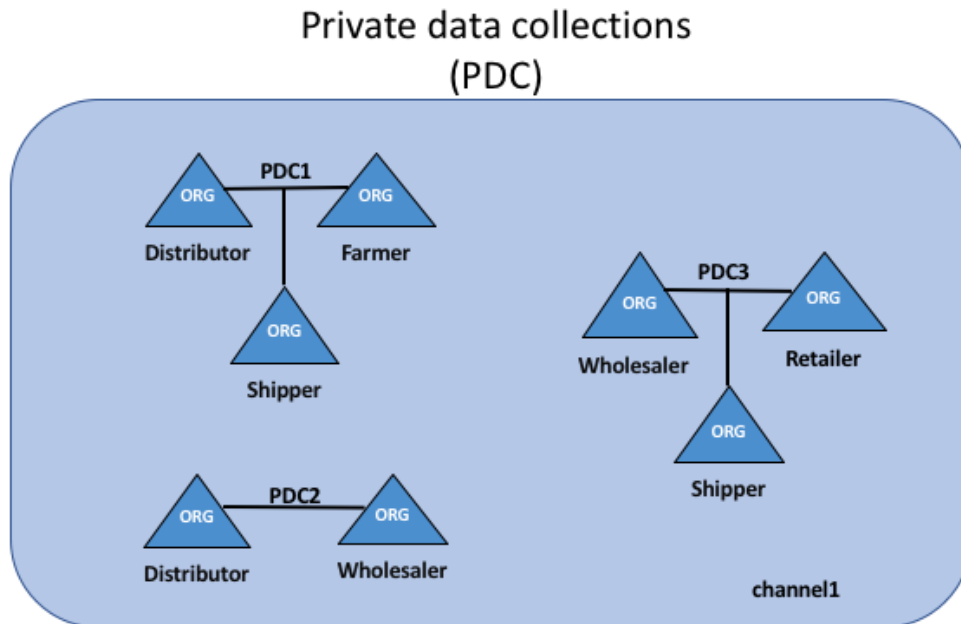
The **Distributor** might want to make private transactions with the **Farmer** and **Shipper** to keep the terms of the trades confidential from the **Wholesaler** and the **Retailer** (so as not to expose the markup they're charging).

The **Distributor** may also want to have a separate private data relationship with the **Wholesaler** because it charges them a lower price than it does the **Retailer**.

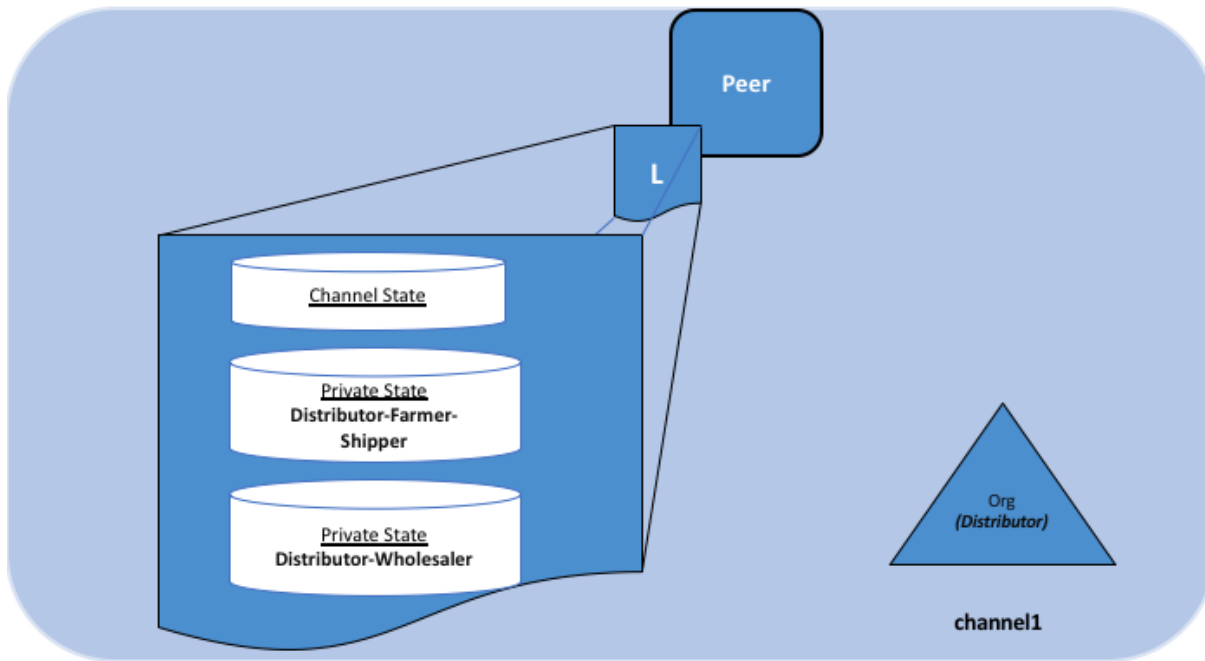
The **Wholesaler** may also want to have a private data relationship with the **Retailer** and the **Shipper**.

Rather than defining many small channels for each of these relationships, multiple private data collections (**PDC**) can be defined to share private data between:

1. PDC1: **Distributor**, **Farmer** and **Shipper**
2. PDC2: **Distributor** and **Wholesaler**
3. PDC3: **Wholesaler**, **Retailer** and **Shipper**



Using this example, peers owned by the **Distributor** will have multiple private databases inside their ledger which includes the private data from the **Distributor**, **Farmer** and **Shipper** relationship and the **Distributor** and **Wholesaler** relationship.



4.12.4 Transaction flow with private data

When private data collections are referenced in chaincode, the transaction flow is slightly different in order to protect the confidentiality of the private data as transactions are proposed, endorsed, and committed to the ledger.

For details on transaction flows that don't use private data refer to our documentation on [transaction flow](#).

1. The client application submits a proposal request to invoke a chaincode function (reading or writing private data) to endorsing peers which are part of authorized organizations of the collection. The private data, or data used to generate private data in chaincode, is sent in a `transient` field of the proposal.
2. The endorsing peers simulate the transaction and store the private data in a `transient data store` (a temporary storage local to the peer). They distribute the private data, based on the collection policy, to authorized peers via `gossip`.
3. The endorsing peer sends the proposal response back to the client. The proposal response includes the endorsed read/write set, which includes public data, as well as a hash of any private data keys and values. *No private data is sent back to the client*. For more information on how endorsement works with private data, click [here](#).
4. The client application submits the transaction (which includes the proposal response with the private data hashes) to the ordering service. The transactions with the private data hashes get included in blocks as normal. The block with the private data hashes is distributed to all the peers. In this way, all peers on the channel can validate transactions with the hashes of the private data in a consistent way, without knowing the actual private data.
5. At block commit time, authorized peers use the collection policy to determine if they are authorized to have access to the private data. If they do, they will first check their local `transient data store` to determine if they have already received the private data at chaincode endorsement time. If not, they will attempt to pull the private data from another authorized peer. Then they will validate the private data against the hashes in the public block and commit the transaction and the block. Upon validation/commit, the private data is moved to their copy of the private state database and private writeset storage. The private data is then deleted from the `transient data store`.

4.12.5 Sharing private data

In many scenarios private data keys/values in one collection may need to be shared with other channel members or with other private data collections, for example when you need to transact on private data with a channel member or group of channel members who were not included in the original private data collection. The receiving parties will typically want to verify the private data against the on-chain hashes as part of the transaction.

There are several aspects of private data collections that enable the sharing and verification of private data:

- First, you don't necessarily have to be a member of a collection to write to a key in a collection, as long as the endorsement policy is satisfied. Endorsement policy can be defined at the chaincode level, key level (using state-based endorsement), or collection level (starting in Fabric v2.0).
- Second, starting in v1.4.2 there is a chaincode API `GetPrivateDataHash()` that allows chaincode on non-member peers to read the hash value of a private key. This is an important feature as you will see later, because it allows chaincode to verify private data against the on-chain hashes that were created from private data in previous transactions.

This ability to share and verify private data should be considered when designing applications and the associated private data collections. While you can certainly create sets of multilateral private data collections to share data among various combinations of channel members, this approach may result in a large number of collections that need to be defined. Alternatively, consider using a smaller number of private data collections (e.g. one collection per organization, or one collection per pair of organizations), and then sharing private data with other channel members, or with other collections as the need arises. Starting in Fabric v2.0, implicit organization-specific collections are available for any chaincode to utilize, so that you don't even have to define these per-organization collections when deploying chaincode.

Private data sharing patterns

When modeling private data collections per organization, multiple patterns become available for sharing or transferring private data without the overhead of defining many multilateral collections. Here are some of the sharing patterns that could be leveraged in chaincode applications:

- **Use a corresponding public key for tracking public state** - You can optionally have a matching public key for tracking public state (e.g. asset properties, current ownership. etc), and for every organization that should have access to the asset's corresponding private data, you can create a private key/value in each organization's private data collection.
- **Chaincode access control** - You can implement access control in your chaincode, to specify which clients can query private data in a collection. For example, store an access control list for a private data collection key or range of keys, then in the chaincode get the client submitter's credentials (using `GetCreator()` chaincode API or CID library API `GetID()` or `GetMSPID()`), and verify they have access before returning the private data. Similarly you could require a client to pass a passphrase into chaincode, which must match a passphrase stored at the key level, in order to access the private data. Note, this pattern can also be used to restrict client access to public state data.
- **Sharing private data out of band** - As an off-chain option, you could share private data out of band with other organizations, and they can hash the key/value to verify it matches the on-chain hash by using `GetPrivateDataHash()` chaincode API. For example, an organization that wishes to purchase an asset from you may want to verify an asset's properties and that you are the legitimate owner by checking the on-chain hash, prior to agreeing to the purchase.
- **Sharing private data with other collections** - You could 'share' the private data on-chain with chaincode that creates a matching key/value in the other organization's private data collection. You'd pass the private data key/value to chaincode via transient field, and the chaincode could confirm a hash of the passed private data matches the on-chain hash from your collection using `GetPrivateDataHash()`, and then write the private data to the other organization's private data collection.

- **Transferring private data to other collections** - You could ‘transfer’ the private data with chaincode that deletes the private data key in your collection, and creates it in another organization’s collection. Again, use the transient field to pass the private data upon chaincode invoke, and in the chaincode use `GetPrivateDataHash()` to confirm that the data exists in your private data collection, before deleting the key from your collection and creating the key in another organization’s collection. To ensure that a transaction always deletes from one collection and adds to another collection, you may want to require endorsements from additional parties, such as a regulator or auditor.
- **Using private data for transaction approval** - If you want to get a counterparty’s approval for a transaction before it is completed (e.g. an on-chain record that they agree to purchase an asset for a certain price), the chaincode can require them to ‘pre-approve’ the transaction, by either writing a private key to their private data collection or your collection, which the chaincode will then check using `GetPrivateDataHash()`. In fact, this is exactly the same mechanism that the built-in lifecycle system chaincode uses to ensure organizations agree to a chaincode definition before it is committed to a channel. Starting with Fabric v2.0, this pattern becomes more powerful with collection-level endorsement policies, to ensure that the chaincode is executed and endorsed on the collection owner’s own trusted peer. Alternatively, a mutually agreed key with a key-level endorsement policy could be used, that is then updated with the pre-approval terms and endorsed on peers from the required organizations.
- **Keeping transactors private** - Variations of the prior pattern can also eliminate leaking the transactors for a given transaction. For example a buyer indicates agreement to buy on their own collection, then in a subsequent transaction seller references the buyer’s private data in their own private data collection. The proof of transaction with hashed references is recorded on-chain, only the buyer and seller know that they are the transactors, but they can reveal the pre-images if a need-to-know arises, such as in a subsequent transaction with another party who could verify the hashes.

Coupled with the patterns above, it is worth noting that transactions with private data can be bound to the same conditions as regular channel state data, specifically:

- **Key level transaction access control** - You can include ownership credentials in a private data value, so that subsequent transactions can verify that the submitter has ownership privilege to share or transfer the data. In this case the chaincode would get the submitter’s credentials (e.g. using `GetCreator()` chaincode API or CID library API `GetID()` or `GetMSPID()`), combine it with other private data that gets passed to the chaincode, hash it, and use `GetPrivateDataHash()` to verify that it matches the on-chain hash before proceeding with the transaction.
- **Key level endorsement policies** - And also as with normal channel state data, you can use state-based endorsement to specify which organizations must endorse transactions that share or transfer private data, using `SetPrivateDataValidationParameter()` chaincode API, for example to specify that only an owner’s organization peer, custodian’s organization peer, or other third party must endorse such transactions.

Example scenario: Asset transfer using private data collections

The private data sharing patterns mentioned above can be combined to enable powerful chaincode-based applications. For example, consider how an asset transfer scenario could be implemented using per-organization private data collections:

- An asset may be tracked by a UUID key in public chaincode state. Only the asset’s ownership is recorded, nothing else is known about the asset.
- The chaincode will require that any transfer request must originate from the owning client, and the key is bound by state-based endorsement requiring that a peer from the owner’s organization and a regulator’s organization must endorse any transfer requests.
- The asset owner’s private data collection contains the private details about the asset, keyed by a hash of the UUID. Other organizations and the ordering service will only see a hash of the asset details.
- Let’s assume the regulator is a member of each collection as well, and therefore persists the private data, although this need not be the case.

A transaction to trade the asset would unfold as follows:

1. Off-chain, the owner and a potential buyer strike a deal to trade the asset for a certain price.
2. The seller provides proof of their ownership, by either passing the private details out of band, or by providing the buyer with credentials to query the private data on their node or the regulator's node.
3. Buyer verifies a hash of the private details matches the on-chain public hash.
4. The buyer invokes chaincode to record their bid details in their own private data collection. The chaincode is invoked on buyer's peer, and potentially on regulator's peer if required by the collection endorsement policy.
5. The current owner (seller) invokes chaincode to sell and transfer the asset, passing in the private details and bid information. The chaincode is invoked on peers of the seller, buyer, and regulator, in order to meet the endorsement policy of the public key, as well as the endorsement policies of the buyer and seller private data collections.
6. The chaincode verifies that the submitting client is the owner, verifies the private details against the hash in the seller's collection, and verifies the bid details against the hash in the buyer's collection. The chaincode then writes the proposed updates for the public key (setting ownership to the buyer, and setting endorsement policy to be the buying organization and regulator), writes the private details to the buyer's private data collection, and potentially deletes the private details from seller's collection. Prior to final endorsement, the endorsing peers ensure private data is disseminated to any other authorized peers of the seller and regulator.
7. The seller submits the transaction with the public data and private data hashes for ordering, and it is distributed to all channel peers in a block.
8. Each peer's block validation logic will consistently verify the endorsement policy was met (buyer, seller, regulator all endorsed), and verify that public and private state that was read in the chaincode has not been modified by any other transaction since chaincode execution.
9. All peers commit the transaction as valid since it passed validation checks. Buyer peers and regulator peers retrieve the private data from other authorized peers if they did not receive it at endorsement time, and persist the private data in their private data state database (assuming the private data matched the hashes from the transaction).
10. With the transaction completed, the asset has been transferred, and other channel members interested in the asset may query the history of the public key to understand its provenance, but will not have access to any private details unless an owner shares it on a need-to-know basis.

The basic asset transfer scenario could be extended for other considerations, for example the transfer chaincode could verify that a payment record is available to satisfy payment versus delivery requirements, or verify that a bank has submitted a letter of credit, prior to the execution of the transfer chaincode. And instead of transactors directly hosting peers, they could transact through custodian organizations who are running peers.

4.12.6 Purging private data

For very sensitive data, even the parties sharing the private data might want — or might be required by government regulations — to periodically “purge” the data on their peers, leaving behind a hash of the data on the blockchain to serve as immutable evidence of the private data.

In some of these cases, the private data only needs to exist on the peer's private database until it can be replicated into a database external to the peer's blockchain. The data might also only need to exist on the peers until a chaincode business process is done with it (trade settled, contract fulfilled, etc).

To support these use cases, private data can be purged if it has not been modified for a configurable number of blocks. Purged private data cannot be queried from chaincode, and is not available to other requesting peers.

4.12.7 How a private data collection is defined

For more details on collection definitions, and other low level information about private data and collections, refer to the [private data reference topic](#).

4.13 Channel capabilities

Audience: Channel administrators, node administrators

Note: this is an advanced Fabric concept that is not necessary for new users or application developers to understand. However, as channels and networks mature, understanding and managing capabilities becomes vital. Furthermore, it is important to recognize that updating capabilities is a different, though often related, process to upgrading nodes. We'll describe this in detail in this topic.

Because Fabric is a distributed system that will usually involve multiple organizations, it is possible (and typical) that different versions of Fabric code will exist on different nodes within the network as well as on the channels in that network. Fabric allows this — it is not necessary for every peer and ordering node to be at the same version level. In fact, supporting different version levels is what enables rolling upgrades of Fabric nodes.

What **is** important is that networks and channels process things in the same way, creating deterministic results for things like channel configuration updates and chaincode invocations. Without deterministic results, one peer on a channel might invalidate a transaction while another peer validates it.

To that end, Fabric defines levels of what are called “capabilities”. These capabilities, which are defined in the configuration of each channel, ensure determinism by defining a level at which behaviors produce consistent results. As you'll see, these capabilities have versions which are closely related to node binary versions. Capabilities enable nodes running at different version levels to behave in a compatible and consistent way given the channel configuration at a specific block height. You will also see that capabilities exist in many parts of the configuration tree, defined along the lines of administration for particular tasks.

As you'll see, sometimes it is necessary to update your channel to a new capability level to enable a new feature.

4.13.1 Node versions and capability versions

If you're familiar with Hyperledger Fabric, you're aware that it follows a typical versioning pattern: v1.1, v1.2.1, v2.0, etc. These versions refer to releases and their related binary versions. Capabilities follow the same versioning convention. There are v1.1 capabilities and v1.2 capabilities and 2.0 capabilities and so on. But it's important to note a few distinctions.

- **There is not necessarily a new capability level with each release.** The need to establish a new capability is determined on a case by case basis and relies chiefly on the backwards compatibility of new features and older binary versions. Allowing channels to be created [without the use of a system channel](#), a new feature in v2.3, did not change the way either transactions or ordering service functions were handled and thus did not require the establishment of any new capabilities. [Private Data](#), on the other hand, could not be handled by peers before v1.2, requiring the establishment of a v1.2 capability level. Because not every release contains a new feature (or a bug fix) that changes the way transactions are processed, certain releases will not require any new capabilities (for example, v1.4) while others will only have new capabilities at particular levels (such as v1.2 and v1.3). We'll discuss the “levels” of capabilities and where they reside in the configuration tree later.
- **Nodes must be at least at the level of certain capabilities in a channel.** When a peer joins a channel, it reads all of the blocks in the ledger sequentially, starting with the genesis block of the channel and continuing through the transaction blocks and any subsequent configuration blocks. If a node, for example a peer, attempts to read a block containing an update to a capability it doesn't understand (for example, a v1.4.x peer trying to read a block containing a v2.0 application capability), **the peer will crash**. This crashing behavior is intentional, as a v1.4.x peer should not attempt validate or commit any transactions past this point. Before joining a channel,

make sure the node is at least the Fabric version (binary) level of the capabilities specified in the channel config relevant to the node. We'll discuss which capabilities are relevant to which nodes later. However, because no user wants their nodes to crash, it is strongly recommended to update all nodes to the required level (preferably, to the latest release) before attempting to update capabilities. This is in line with the default Fabric recommendation to **always** be at the latest binary and capability levels.

If users are unable to upgrade their binaries, then capabilities must be left at their lower levels. Lower level binaries and capabilities will still work together as they're meant to. However, keep in mind that it is a best practice to always update to new binaries even if a user chooses not to update their capabilities. Because capabilities themselves also include bug-fixes, it is always recommended to update capabilities once the network binaries support them.

4.13.2 Capability configuration groupings

As we discussed earlier, there is not a single capability level encompassing an entire channel. Rather, there are three capabilities, each representing an area of administration.

- **Orderer:** These capabilities govern tasks and processing exclusive to the ordering service. Because these capabilities do not involve processes that affect transactions or the peers, updating them falls solely to the ordering service admins (peers do not need to understand orderer capabilities and will therefore not crash no matter what the orderer capability is updated to). Note that these capabilities did not change between v1.1 and v1.4.2. However, as we'll see in the **channel** section, this does not mean that v1.1 ordering nodes will work on all channels with capability levels below v1.4.2.
- **Application:** These capabilities govern tasks and processing exclusive to the peers. Because ordering service admins have no role in deciding the nature of transactions between peer organizations, changing this capability level falls exclusively to peer organizations. For example, Private Data can only be enabled on a channel with the v1.2 (or higher) application group capability enabled. In the case of Private Data, this is the only capability that must be enabled, as nothing about the way Private Data works requires a change to channel administration or the way the ordering service processes transactions.
- **Channel:** This grouping encompasses tasks that are **jointly administered** by the peer organizations and the ordering service. For example, this is the capability that defines the level at which channel configuration updates, which are initiated by peer organizations and orchestrated by the ordering service, are processed. On a practical level, **this grouping defines the minimum level for all of the binaries in a channel, as both ordering nodes and peers must be at least at the binary level corresponding to this capability in order to process the capability.**

While it is possible to create an ordering service using a legacy process in which a "system channel" administered by the ordering service is created before any application channels are created, the recommended path is to create a channel without using a system channel.

If you do use the [legacy system channel process](#), the **orderer** and **channel** capabilities of a channel are inherited by default from the ordering system channel, where modifying them are the exclusive purview of ordering service admins. As a result, peer organizations should inspect the genesis block of a channel prior to joining their peers to that channel. Although the channel capability is administered by the orderers in the orderer system channel (just as the consortium membership is), it is typical and expected that the ordering admins will coordinate with the consortium admins to ensure that the channel capability is only upgraded when the consortium is ready for it. Note: because the ordering system channel does not define an **application** capability, this capability must be specified in the channel profile when creating the genesis block for the channel.

If you use the recommended process to [Create a channel](#), all these capability levels are specified in the application channel genesis block that is created.

Take caution when modifying an application capability. Because the ordering service does not validate that the capability level exists, it will allow a channel to be modified to contain, for example, a v1.8 application capability even if no such capability exists. Any peer attempting to read a configuration block with this capability would, as we have

shown, crash, and even if it was possible to modify the channel once again to a valid capability level, it would not matter, as no peer would be able to get past the block with the invalid v1.8 capability.

For a full look at the current valid orderer, application, and channel capabilities check out a [sample configtx.yaml file](#), which lists them in the “Capabilities” section.

For more specific information about capabilities and where they reside in the channel configuration, check out [defining capability requirements](#).

4.14 Security Model

Hyperledger Fabric is a permissioned blockchain where each component and actor has an identity, and policies define access control and governance. This topic provides an overview of the Fabric security model and includes links to additional information.

4.14.1 Identities

The different actors in a blockchain network include peers, orderers, client applications, administrators and more. Each of these actors — active elements inside or outside a network able to consume services — has a digital identity encapsulated in an X.509 digital certificate issued by a Certificate Authority (CA). These identities matter because they determine the exact permissions over resources and access to information that actors have in a blockchain network.

For more information see the [Identity topic](#).

4.14.2 Membership Service Providers

For an identity to be verifiable, it must come from a trusted authority. A membership service provider (MSP) is that trusted authority in Fabric. More specifically, an MSP is a component that defines the rules that govern the valid identities for an organization. A Hyperledger Fabric channel defines a set of organization MSPs as members. The default MSP implementation in Fabric uses X.509 certificates issued by a Certificate Authority (CA) as identities, adopting a traditional Public Key Infrastructure (PKI) hierarchical model. Identities can be associated with roles within a MSP such as ‘client’ and ‘admin’ by utilizing Node OU roles. Node OU roles can be used in policy definitions in order to restrict access to Fabric resources to certain MSPs and roles.

For more information see the [Membership Service Providers \(MSPs\) topic](#).

4.14.3 Policies

In Hyperledger Fabric, policies are the mechanism for infrastructure management. Fabric policies represent how members come to agreement on accepting or rejecting changes to the network, a channel, or a smart contract. Policies are agreed to by the channel members when the channel is originally configured, but they can also be modified as the channel evolves. For example, they describe the criteria for adding or removing members from a channel, change how blocks are formed, or specify the number of organizations required to endorse a smart contract. All of these actions are described by a policy which defines who can perform the action. Simply put, everything you want to do on a Fabric network is controlled by a policy. Once they are written, policies evaluate the collection of signatures attached to transactions and proposals and validate if the signatures fulfill the governance agreed to by the network.

Policies can be used in Channel Policies, Channel Modification Policies, Access Control Lists, Chaincode Lifecycle Policies, and Chaincode Endorsement Policies.

For more information see the [Policies topic](#).

Channel Policies

Policies in the channel configuration define various usage and administrative policies on a channel. For example, the policy for adding a peer organization to a channel is defined within the administrative domain of the peer organizations (known as the Application group). Similarly, adding ordering nodes in the consenter set of the channel is controlled by a policy inside the Orderer group. Actions that cross both the peer and orderer organizational domains are contained in the Channel group.

For more information see the [Channel Policies](#) topic.

Channel Modification Policies

Modification policies specify the group of identities required to sign (approve) any channel configuration update. It is the policy that defines how a channel policy is updated. Thus, each channel configuration element includes a reference to a policy which governs its modification.

For more information see the [Modification Policies](#) topic.

Access Control Lists

Access Control Lists (ACLs) provide the ability to configure access to channel resources by associating those resources with existing policies.

For more information see the [Access Control Lists \(ACLs\)](#) topic.

Chaincode Lifecycle Policy

The number of organizations that need to approve a chaincode definition before it can be successfully committed to a channel is governed by the channel's LifecycleEndorsement policy.

For more information see the [Chaincode Lifecycle](#) topic.

Chaincode Endorsement Policies

Every smart contract inside a chaincode package has an endorsement policy that specifies how many peers belonging to different channel members need to execute and validate a transaction against a given smart contract in order for the transaction to be considered valid. Hence, the endorsement policies define the organizations (through their peers) who must “endorse” (i.e., sign) the execution of a proposal.

For more information see the [Endorsement policies](#) topic.

4.14.4 Peers

Peers are a fundamental element of the network because they host ledgers and smart contracts. Peers have an identity of their own, and are managed by an administrator of an organization.

For more information see the [Peers and Identity](#) topic and [Peer Deployment and Administration](#) topic.

4.14.5 Ordering service nodes

Ordering service nodes order transactions into blocks and then distribute blocks to connected peers for validation and commit. Ordering service nodes have an identity of their own, and are managed by an administrator of an organization.

For more information see the [Ordering Nodes and Identity](#) topic and [Ordering Node Deployment and Administration](#) topic.

4.14.6 Transport Layer Security (TLS)

Fabric supports secure communication between nodes using Transport Layer Security (TLS). TLS communication can use both one-way (server only) and two-way (server and client) authentication.

For more information see the [Transport Layer Security \(TLS\)](#) topic.

4.14.7 Peer and Ordering service node operations service

The peer and the orderer host an HTTP server that offers a RESTful “operations” API. This API is unrelated to the Fabric network services and is intended to be used by operators, not administrators or “users” of the network.

As the operations service is focused on operations and intentionally unrelated to the Fabric network, it does not use the Membership Services Provider for access control. Instead, the operations service relies entirely on mutual TLS with client certificate authentication.

For more information see the [Operations Service](#) topic.

4.14.8 Hardware Security Modules

The cryptographic operations performed by Fabric nodes can be delegated to a Hardware Security Module (HSM). An HSM protects your private keys and handles cryptographic operations, allowing your peers to endorse transactions and orderer nodes to sign blocks without exposing their private keys.

Fabric currently leverages the PKCS11 standard to communicate with an HSM.

For more information see the [Hardware Security Module \(HSM\)](#) topic.

4.14.9 Fabric Applications

A Fabric application can interact with a blockchain network by submitting transactions to a ledger or querying ledger content. An application interacts with a blockchain network using one of the Fabric SDKs.

The Fabric v2.x SDKs only support transaction and query functions and event listening. Support for administrative functions for channels and nodes has been removed from the SDKs in favor of the CLI tools.

Applications typically reside in a managed tier of an organization’s infrastructure. The organization may create client identities for the organization at large, or client identities for individual end users of the application. Client identities only have permission to submit transactions and query the ledger, they do not have administrative or operational permissions on channels or nodes.

In some use cases the application tier may persist user credentials including the private key and sign transactions. In other use cases end users of the application may want to keep their private key secret. To support these use cases, the Node.js SDK supports offline signing of transactions. In both cases, a Hardware Security Module can be used to store private keys meaning that the client application does not have access to them.

Regardless of application design, the SDKs do not have any privileged access to peer or orderer services other than that provided by the client identity. From a security perspective, the SDKs are merely a set of language specific convenience functions for interacting with the gRPC services exposed by the Fabric peers and orderers. All security enforcement is carried out by Fabric nodes as highlighted earlier in this topic, not the client SDK.

For more information see the [Applications topic](#) and [Offline Signing tutorial](#).

4.15 Use Cases

The Hyperledger Requirements WG is documenting a number of blockchain use cases and maintaining an inventory [here](#).

5.1 Prerequisites

Before you begin, you should confirm that you have installed all the prerequisites below on the platform where you will be running Hyperledger Fabric.

Note: These prerequisites are recommended for Fabric users. If you are a Fabric developer you should refer to the instructions for *Setting up the development environment*.

5.1.1 Install Git

Download the latest version of [git](#) if it is not already installed, or if you have problems running the curl commands.

5.1.2 Install cURL

Download the latest version of the [cURL](#) tool if it is not already installed or if you get errors running the curl commands from the documentation.

Note: If you're on Windows please see the specific note on *Windows extras* below.

5.1.3 Docker and Docker Compose

You will need the following installed on the platform on which you will be operating, or developing on (or for), Hyperledger Fabric:

- MacOSX, *nix, or Windows 10: [Docker](#) Docker version 17.06.2-ce or greater is required.
- Older versions of Windows: [Docker Toolbox](#) - again, Docker version Docker 17.06.2-ce or greater is required.

The Fabric sample test network has been successfully verified with Docker Desktop version 2.5.0.1. Higher versions may not work at this time.

Note: The following applies to linux systems running systemd.

Make sure the docker daemon is running.

```
sudo systemctl start docker
```

Optional: If you want the docker daemon to start when the system starts, use the following:

```
sudo systemctl enable docker
```

Add your user to the docker group.

```
sudo usermod -a -G docker <username>
```

Note: Installing Docker for Mac or Windows, or Docker Toolbox will also install Docker Compose. If you already had Docker installed, you should check that you have Docker Compose version 1.14.0 or greater installed. If not, we recommend that you install a more recent version of Docker.

You can check the version of Docker Compose you have installed with the following command from a terminal prompt:

```
docker-compose --version
```

5.1.4 Windows extras

On Windows 10 you should use the native Docker distribution and you may use the Windows PowerShell. However, for the `binaries` command to succeed you will still need to have the `uname` command available. You can get it as part of Git but beware that only the 64bit version is supported.

Before running any `git clone` commands, run the following commands:

```
git config --global core.autocrlf false
git config --global core.longpaths true
```

You can check the setting of these parameters with the following commands:

```
git config --get core.autocrlf
git config --get core.longpaths
```

These need to be `false` and `true` respectively.

The `curl` command that comes with Git and Docker Toolbox is old and does not handle properly the redirect used in [Getting Started](#). Make sure you have and use a newer version which can be downloaded from the [cURL downloads page](#)

Note: If you have questions not addressed by this documentation, or run into issues with any of the tutorials, please visit the [Still Have Questions?](#) page for some tips on where to find additional help.

5.2 Install Samples, Binaries, and Docker Images

While we work on developing real installers for the Hyperledger Fabric binaries, we provide a script that will download and install samples and binaries to your system. We think that you'll find the sample applications installed useful to learn more about the capabilities and operations of Hyperledger Fabric.

Note: If you are running on **Windows** you will want to make use of the Docker Quickstart Terminal for the upcoming terminal commands. Please visit the [Prerequisites](#) if you haven't previously installed it.

If you are using Docker Toolbox or macOS, you will need to use a location under `/Users` (macOS) when installing and running the samples.

If you are using Docker for Mac, you will need to use a location under `/Users`, `/Volumes`, `/private`, or `/tmp`. To use a different location, please consult the Docker documentation for [file sharing](#).

If you are using Docker for Windows, please consult the Docker documentation for [shared drives](#) and use a location under one of the shared drives.

Determine a location on your machine where you want to place the *fabric-samples* repository and enter that directory in a terminal window. The command that follows will perform the following steps:

1. If needed, clone the [hyperledger/fabric-samples](#) repository
2. Checkout the appropriate version tag
3. Install the Hyperledger Fabric platform-specific binaries and config files for the version specified into the `/bin` and `/config` directories of *fabric-samples*
4. Download the Hyperledger Fabric docker images for the version specified

Once you are ready, and in the directory into which you will install the Fabric Samples and binaries, go ahead and execute the command to pull down the binaries and images.

Note: If you want the latest production release, omit all version identifiers.

```
curl -sSL https://bit.ly/2ysbOFE | bash -s
```

Note: If you want a specific release, pass a version identifier for Fabric and Fabric-CA docker images. The command below demonstrates how to download the latest production releases - **Fabric v2.3.3** and **Fabric CA v1.5.2**

```
curl -sSL https://bit.ly/2ysbOFE | bash -s -- <fabric_version> <fabric-ca_version>
curl -sSL https://bit.ly/2ysbOFE | bash -s -- 2.3.3 1.5.2
```

Note: If you get an error running the above curl command, you may have too old a version of curl that does not handle redirects or an unsupported environment.

Please visit the [Prerequisites](#) page for additional information on where to find the latest version of curl and get the right environment. Alternately, you can substitute the un-shortened URL: <https://raw.githubusercontent.com/hyperledger/fabric/master/scripts/bootstrap.sh>

Note: For additional use pattern you can use the `-h` flag to view the help and available commands for the Fabric-Samples bootstrap script. For example: `curl -sSL https://bit.ly/2ysb0FE | bash -s -- -h`

The command above downloads and executes a bash script that will download and extract all of the platform-specific binaries you will need to set up your network and place them into the cloned repo you created above. It retrieves the following platform-specific binaries:

- `configtxgen`,
- `configtxlator`,
- `cryptogen`,
- `discover`,
- `idemixgen`
- `orderer`,
- `osnadmin`,
- `peer`,
- `fabric-ca-client`,
- `fabric-ca-server`

and places them in the `bin` sub-directory of the current working directory.

You may want to add that to your `PATH` environment variable so that these can be picked up without fully qualifying the path to each binary. e.g.:

```
export PATH=<path to download location>/bin:$PATH
```

Finally, the script will download the Hyperledger Fabric docker images from [Docker Hub](#) into your local Docker registry and tag them as 'latest'.

The script lists out the Docker images installed upon conclusion.

Look at the names for each image; these are the components that will ultimately comprise our Hyperledger Fabric network. You will also notice that you have two instances of the same image ID - one tagged as "amd64-1.x.x" and one tagged as "latest". Prior to 1.2.0, the image being downloaded was determined by `uname -m` and showed as "x86_64-1.x.x".

Note: On different architectures, the `x86_64/amd64` would be replaced with the string identifying your architecture.

Note: If you have questions not addressed by this documentation, or run into issues with any of the tutorials, please visit the [Still Have Questions?](#) page for some tips on where to find additional help.

5.3 Using the Fabric test network

After you have downloaded the Hyperledger Fabric Docker images and samples, you can deploy a test network by using scripts that are provided in the `fabric-samples` repository. The test network is provided for learning about Fabric by running nodes on your local machine. Developers can use the network to test their smart contracts and applications. The network is meant to be used only as a tool for education and testing and not as a model for how to

set up a network. In general, modifications to the scripts are discouraged and could break the network. It is based on a limited configuration that should not be used as a template for deploying a production network:

- It includes two peer organizations and an ordering organization.
- For simplicity, a single node Raft ordering service is configured.
- To reduce complexity, a TLS Certificate Authority (CA) is not deployed. All certificates are issued by the root CAs.
- The sample network deploys a Fabric network with Docker Compose. Because the nodes are isolated within a Docker Compose network, the test network is not configured to connect to other running Fabric nodes.

To learn how to use Fabric in production, see [Deploying a production network](#).

Note: These instructions have been verified to work against the latest stable Fabric Docker images and the pre-compiled setup utilities within the supplied tar file. If you run these commands with images or tools from the current master branch, it is possible that you will encounter errors.

5.3.1 Before you begin

Before you can run the test network, you need to clone the `fabric-samples` repository and download the Fabric images. Make sure that you have installed the [Prerequisites](#) and [Installed the Samples, Binaries and Docker Images](#).

Note: The test network has been successfully verified with Docker Desktop version 2.5.0.1 and is the recommended version at this time. Higher versions may not work.

5.3.2 Bring up the test network

You can find the scripts to bring up the network in the `test-network` directory of the `fabric-samples` repository. Navigate to the test network directory by using the following command:

```
cd fabric-samples/test-network
```

In this directory, you can find an annotated script, `network.sh`, that stands up a Fabric network using the Docker images on your local machine. You can run `./network.sh -h` to print the script help text:

```
Usage:
network.sh <Mode> [Flags]
Modes:
  up - Bring up Fabric orderer and peer nodes. No channel is created
  up createChannel - Bring up fabric network with one channel
  createChannel - Create and join a channel after the network is created
  deployCC - Deploy a chaincode to a channel (defaults to asset-transfer-basic)
  down - Bring down the network

Flags:
Used with network.sh up, network.sh createChannel:
-c <use CAs> - Use Certificate Authorities to generate network crypto material
-c <channel name> - Name of channel to create (defaults to "mychannel")
-s <dbtype> - Peer state database to deploy: goleveldb (default) or couchdb
-r <max retry> - CLI times out after certain number of attempts (defaults to 5)
-d <delay> - CLI delays for a certain number of seconds (defaults to 3)
-i <imagetag> - Docker image tag of Fabric to deploy (defaults to "latest")
-cai <ca_imagetag> - Docker image tag of Fabric CA to deploy (defaults to "latest")
↪")
-verbose - Verbose mode
```

(continues on next page)

(continued from previous page)

```

Used with network.sh deployCC
-c <channel name> - Name of channel to deploy chaincode to
-ccn <name> - Chaincode name.
-ccl <language> - Programming language of the chaincode to deploy: go (default), ↪
↪ java, javascript, typescript
-ccv <version> - Chaincode version. 1.0 (default), v2, version3.x, etc
-ccs <sequence> - Chaincode definition sequence. Must be an integer, 1 (default),
↪ 2, 3, etc
-ccp <path> - File path to the chaincode.
-cccp <policy> - (Optional) Chaincode endorsement policy using signature policy ↪
↪ syntax. The default policy requires an endorsement from Org1 and Org2
-cccg <collection-config> - (Optional) File path to private data collections ↪
↪ configuration file
-cci <fcn name> - (Optional) Name of chaincode initialization function. When a ↪
↪ function is provided, the execution of init will be requested and the function will ↪
↪ be invoked.

-h - Print this message

Possible Mode and flag combinations
up -ca -r -d -s -i -cai -verbose
up createChannel -ca -c -r -d -s -i -cai -verbose
createChannel -c -r -d -verbose
deployCC -ccn -ccl -ccv -ccs -ccp -cci -r -d -verbose

Examples:
network.sh up createChannel -ca -c mychannel -s couchdb -i 2.0.0
network.sh createChannel -c channelName
network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-javascript/ -
↪ ccl javascript
network.sh deployCC -ccn mychaincode -ccp ./user/mychaincode -ccv 1 -ccl javascript

```

From inside the `test-network` directory, run the following command to remove any containers or artifacts from any previous runs:

```
./network.sh down
```

You can then bring up the network by issuing the following command. You will experience problems if you try to run the script from another directory:

```
./network.sh up
```

This command creates a Fabric network that consists of two peer nodes, one ordering node. No channel is created when you run `./network.sh up`, though we will get there in a *future step*. If the command completes successfully, you will see the logs of the nodes being created:

```

Creating network "net_test" with the default driver
Creating volume "net_orderer.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating peer0.org2.example.com ... done
Creating orderer.example.com ... done
Creating peer0.org1.example.com ... done
Creating cli ... done
CONTAINER ID    IMAGE                                COMMAND                                CREATED
↪ STATUS        PORTS                                NAMES

```

(continues on next page)

(continued from previous page)

```

1667543b5634    hyperledger/fabric-tools:latest    "/bin/bash"    1 second ago    cli
↳ Up Less than a second
b6b117c81c7f    hyperledger/fabric-peer:latest    "peer node start"    2 seconds ago    peer0.
↳ Up 1 second    0.0.0.0:7051->7051/tcp
↳ org1.example.com
703ead770e05    hyperledger/fabric-orderer:latest    "orderer"    2 seconds ago    orderer.
↳ Up Less than a second    0.0.0.0:7050->7050/tcp, 0.0.0.0:7053->7053/tcp
↳ example.com
718d43f5f312    hyperledger/fabric-peer:latest    "peer node start"    2 seconds ago    peer0.
↳ Up 1 second    7051/tcp, 0.0.0.0:9051->9051/tcp
↳ org2.example.com

```

If you don't get this result, jump down to [Troubleshooting](#) for help on what might have gone wrong. By default, the network uses the cryptogen tool to bring up the network. However, you can also [bring up the network with Certificate Authorities](#).

The components of the test network

After your test network is deployed, you can take some time to examine its components. Run the following command to list all of Docker containers that are running on your machine. You should see the three nodes that were created by the `network.sh` script:

```
docker ps -a
```

Each node and user that interacts with a Fabric network needs to belong to an organization in order to participate in the network. The test network includes two peer organizations, Org1 and Org2. It also includes a single orderer organization that maintains the ordering service of the network.

Peers are the fundamental components of any Fabric network. Peers store the blockchain ledger and validate transactions before they are committed to the ledger. Peers run the smart contracts that contain the business logic that is used to manage the assets on the blockchain ledger.

Every peer in the network needs to belong to an organization. In the test network, each organization operates one peer each, `peer0.org1.example.com` and `peer0.org2.example.com`.

Every Fabric network also includes an **ordering service**. While peers validate transactions and add blocks of transactions to the blockchain ledger, they do not decide on the order of transactions or include them into new blocks. On a distributed network, peers may be running far away from each other and not have a common view of when a transaction was created. Coming to consensus on the order of transactions is a costly process that would create overhead for the peers.

An ordering service allows peers to focus on validating transactions and committing them to the ledger. After ordering nodes receive endorsed transactions from clients, they come to consensus on the order of transactions and then add them to blocks. The blocks are then distributed to peer nodes, which add the blocks to the blockchain ledger.

The sample network uses a single node Raft ordering service that is operated by the orderer organization. You can see the ordering node running on your machine as `orderer.example.com`. While the test network only uses a single node ordering service, a production network would have multiple ordering nodes, operated by one or multiple orderer organizations. The different ordering nodes would use the Raft consensus algorithm to come to agreement on the order of transactions across the network.

5.3.3 Creating a channel

Now that we have peer and orderer nodes running on our machine, we can use the script to create a Fabric channel for transactions between Org1 and Org2. Channels are a private layer of communication between specific network

members. Channels can be used only by organizations that are invited to the channel, and are invisible to other members of the network. Each channel has a separate blockchain ledger. Organizations that have been invited “join” their peers to the channel to store the channel ledger and validate the transactions on the channel.

You can use the `network.sh` script to create a channel between Org1 and Org2 and join their peers to the channel. Run the following command to create a channel with the default name of `mychannel`:

```
./network.sh createChannel
```

If the command was successful, you can see the following message printed in your logs:

```
===== Channel successfully joined =====
```

You can also use the `channel` flag to create a channel with custom name. As an example, the following command would create a channel named `channel1`:

```
./network.sh createChannel -c channel1
```

The `channel` flag also allows you to create multiple channels by specifying different channel names. After you create `mychannel` or `channel1`, you can use the command below to create a second channel named `channel2`:

```
./network.sh createChannel -c channel2
```

If you want to bring up the network and create a channel in a single step, you can use the `up` and `createChannel` modes together:

```
./network.sh up createChannel
```

5.3.4 Starting a chaincode on the channel

After you have created a channel, you can start using [smart contracts](#) to interact with the channel ledger. Smart contracts contain the business logic that governs assets on the blockchain ledger. Applications run by members of the network can invoke smart contracts to create assets on the ledger, as well as change and transfer those assets. Applications also query smart contracts to read data on the ledger.

To ensure that transactions are valid, transactions created using smart contracts typically need to be signed by multiple organizations to be committed to the channel ledger. Multiple signatures are integral to the trust model of Fabric. Requiring multiple endorsements for a transaction prevents one organization on a channel from tampering with the ledger on their peer or using business logic that was not agreed to. To sign a transaction, each organization needs to invoke and execute the smart contract on their peer, which then signs the output of the transaction. If the output is consistent and has been signed by enough organizations, the transaction can be committed to the ledger. The policy that specifies the set organizations on the channel that need to execute the smart contract is referred to as the endorsement policy, which is set for each chaincode as part of the chaincode definition.

In Fabric, smart contracts are deployed on the network in packages referred to as chaincode. A Chaincode is installed on the peers of an organization and then deployed to a channel, where it can then be used to endorse transactions and interact with the blockchain ledger. Before a chaincode can be deployed to a channel, the members of the channel need to agree on a chaincode definition that establishes chaincode governance. When the required number of organizations agree, the chaincode definition can be committed to the channel, and the chaincode is ready to be used.

After you have used the `network.sh` to create a channel, you can start a chaincode on the channel using the following command:

```
./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-go -ccl go
```

The `deployCC` subcommand will install the **asset-transfer (basic)** chaincode on `peer0.org1.example.com` and `peer0.org2.example.com` and then deploy the chaincode on the channel specified using the `channel` flag (or `mychannel` if no channel is specified). If you are deploying a chaincode for the first time, the script will install the chaincode dependencies. You can use the `language` flag, `-l`, to install the Go, typescript or javascript versions of the chaincode. You can find the `asset-transfer (basic)` chaincode in the `asset-transfer-basic` folder of the `fabric-samples` directory. This folder contains sample chaincode that are provided as examples and used by tutorials to highlight Fabric features.

5.3.5 Interacting with the network

After you bring up the test network, you can use the `peer` CLI to interact with your network. The `peer` CLI allows you to invoke deployed smart contracts, update channels, or install and deploy new smart contracts from the CLI.

Make sure that you are operating from the `test-network` directory. If you followed the instructions to [install the Samples, Binaries and Docker Images](#), You can find the `peer` binaries in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

```
export PATH=${PWD}/../bin:$PATH
```

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

You can now set the environment variables that allow you to operate the `peer` CLI as `Org1`:

```
# Environment variables for Org1

export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

The `CORE_PEER_TLS_ROOTCERT_FILE` and `CORE_PEER_MSPCONFIGPATH` environment variables point to the `Org1` crypto material in the `organizations` folder.

If you used `./network.sh deployCC -ccl go` to install and start the `asset-transfer (basic)` chaincode, you can invoke the `InitLedger` function of the (Go) chaincode to put an initial list of assets on the ledger (if using typescript or javascript `./network.sh deployCC -ccl javascript` for example, you will invoke the `InitLedger` function of the respective chaincodes).

Run the following command to initialize the ledger with assets:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic_
--peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
"function": "InitLedger", "Args": []}'
```

If successful, you should see similar output to below:

```
-> INFO 001 Chaincode invoke successful. result: status:200
```

You can now query the ledger from your CLI. Run the following command to get the list of assets that were added to your channel ledger:

```
peer chaincode query -C mychannel -n basic -c '{"Args":["GetAllAssets"]}'
```

If successful, you should see the following output:

```
[
  {"ID": "asset1", "color": "blue", "size": 5, "owner": "Tomoko", "appraisedValue": 300},
  {"ID": "asset2", "color": "red", "size": 5, "owner": "Brad", "appraisedValue": 400},
  {"ID": "asset3", "color": "green", "size": 10, "owner": "Jin Soo", "appraisedValue": 500},
  {"ID": "asset4", "color": "yellow", "size": 10, "owner": "Max", "appraisedValue": 600},
  {"ID": "asset5", "color": "black", "size": 15, "owner": "Adriana", "appraisedValue": 700},
  {"ID": "asset6", "color": "white", "size": 15, "owner": "Michel", "appraisedValue": 800}
]
```

Chaincodes are invoked when a network member wants to transfer or change an asset on the ledger. Use the following command to change the owner of an asset on the ledger by invoking the asset-transfer (basic) chaincode:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"function":"TransferAsset","Args":["asset6","Christopher"]}'
```

If the command is successful, you should see the following response:

```
2019-12-04 17:38:21.048 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Chaincode invoke successful. result: status:200
```

Because the endorsement policy for the asset-transfer (basic) chaincode requires the transaction to be signed by Org1 and Org2, the chaincode invoke command needs to target both peer0.org1.example.com and peer0.org2.example.com using the `--peerAddresses` flag. Because TLS is enabled for the network, the command also needs to reference the TLS certificate for each peer using the `--tlsRootCertFiles` flag.

After we invoke the chaincode, we can use another query to see how the invoke changed the assets on the blockchain ledger. Since we already queried the Org1 peer, we can take this opportunity to query the chaincode running on the Org2 peer. Set the following environment variables to operate as Org2:

```
# Environment variables for Org2

export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
```

(continues on next page)

(continued from previous page)

```
export CORE_PEER_ADDRESS=localhost:9051
```

You can now query the asset-transfer (basic) chaincode running on `peer0.org2.example.com`:

```
peer chaincode query -C mychannel -n basic -c '{"Args":["ReadAsset","asset6"]}'
```

The result will show that "asset6" was transferred to Christopher:

```
{"ID":"asset6","color":"white","size":15,"owner":"Christopher","appraisedValue":800}
```

5.3.6 Bring down the network

When you are finished using the test network, you can bring down the network with the following command:

```
./network.sh down
```

The command will stop and remove the node and chaincode containers, delete the organization crypto material, and remove the chaincode images from your Docker Registry. The command also removes the channel artifacts and docker volumes from previous runs, allowing you to run `./network.sh up` again if you encountered any problems.

5.3.7 Next steps

Now that you have used the test network to deploy Hyperledger Fabric on your local machine, you can use the tutorials to start developing your own solution:

- Learn how to deploy your own smart contracts to the test network using the [Deploying a smart contract to a channel](#) tutorial.
- Visit the [Writing Your First Application](#) tutorial to learn how to use the APIs provided by the Fabric SDKs to invoke smart contracts from your client applications.
- If you are ready to deploy a more complicated smart contract to the network, follow the [commercial paper tutorial](#) to explore a use case in which two organizations use a blockchain network to trade commercial paper.

You can find the complete list of Fabric tutorials on the [tutorials](#) page.

5.3.8 Bring up the network with Certificate Authorities

Hyperledger Fabric uses public key infrastructure (PKI) to verify the actions of all network participants. Every node, network administrator, and user submitting transactions needs to have a public certificate and private key to verify their identity. These identities need to have a valid root of trust, establishing that the certificates were issued by an organization that is a member of the network. The `network.sh` script creates all of the cryptographic material that is required to deploy and operate the network before it creates the peer and ordering nodes.

By default, the script uses the cryptogen tool to create the certificates and keys. The tool is provided for development and testing, and can quickly create the required crypto material for Fabric organizations with a valid root of trust. When you run `./network.sh up`, you can see the cryptogen tool creating the certificates and keys for Org1, Org2, and the Orderer Org.

```
creating Org1, Org2, and ordering service organization with crypto from 'cryptogen'
/Usr/fabric-samples/test-network/../../bin/cryptogen
```

(continues on next page)

(continued from previous page)

```
#####
#### Generate certificates using cryptogen tool #####
#####

#####
##### Create Org1 Identities #####
#####
+ cryptogen generate --config=./organizations/cryptogen/crypto-config-org1.yaml --
↳output=organizations
org1.example.com
+ res=0
+ set +x
#####
##### Create Org2 Identities #####
#####
+ cryptogen generate --config=./organizations/cryptogen/crypto-config-org2.yaml --
↳output=organizations
org2.example.com
+ res=0
+ set +x
#####
##### Create Orderer Org Identities #####
#####
+ cryptogen generate --config=./organizations/cryptogen/crypto-config-orderer.yaml --
↳output=organizations
+ res=0
+ set +x
```

However, the test network script also provides the option to bring up the network using Certificate Authorities (CAs). In a production network, each organization operates a CA (or multiple intermediate CAs) that creates the identities that belong to their organization. All of the identities created by a CA run by the organization share the same root of trust. Although it takes more time than using cryptogen, bringing up the test network using CAs provides an introduction to how a network is deployed in production. Deploying CAs also allows you to enroll client identities with the Fabric SDKs and create a certificate and private key for your applications.

If you would like to bring up a network using Fabric CAs, first run the following command to bring down any running networks:

```
./network.sh down
```

You can then bring up the network with the CA flag:

```
./network.sh up -ca
```

After you issue the command, you can see the script bringing up three CAs, one for each organization in the network.

```
#####
#### Generate certificates using Fabric CA's #####
#####
Creating network "net_default" with the default driver
Creating ca_org2 ... done
Creating ca_org1 ... done
Creating ca_orderer ... done
```

It is worth taking time to examine the logs generated by the `./network.sh` script after the CAs have been deployed. The test network uses the Fabric CA client to register node and user identities with the CA of each organization. The script then uses the enroll command to generate an MSP folder for each identity. The MSP folder contains the

certificate and private key for each identity, and establishes the identity's role and membership in the organization that operated the CA. You can use the following command to examine the MSP folder of the Org1 admin user:

```
tree organizations/peerOrganizations/org1.example.com/users/Admin@org1.example.com/
```

The command will reveal the MSP folder structure and configuration file:

```
organizations/peerOrganizations/org1.example.com/users/Admin@org1.example.com/
├── msp
│   ├── IssuerPublicKey
│   ├── IssuerRevocationPublicKey
│   ├── cacerts
│   │   └── localhost-7054-ca-org1.pem
│   ├── config.yaml
│   ├── keystore
│   │   └── 58e81e6f1ee8930df46841bf88c22a08ae53c1332319854608539ee78ed2fd65_sk
│   ├── signcerts
│   │   └── cert.pem
│   └── user
```

You can find the certificate of the admin user in the `signcerts` folder and the private key in the `keystore` folder. To learn more about MSPs, see the [Membership Service Provider](#) concept topic.

Both cryptogen and the Fabric CAs generate the cryptographic material for each organization in the `organizations` folder. You can find the commands that are used to set up the network in the `registerEnroll.sh` script in the `organizations/fabric-ca` directory. To learn more about how you would use the Fabric CA to deploy a Fabric network, visit the [Fabric CA operations guide](#). You can learn more about how Fabric uses PKI by visiting the [identity](#) and [membership](#) concept topics.

5.3.9 What's happening behind the scenes?

If you are interested in learning more about the sample network, you can investigate the files and scripts in the `test-network` directory. The steps below provide a guided tour of what happens when you issue the command of `./network.sh up`.

- `./network.sh` creates the certificates and keys for two peer organizations and the orderer organization. By default, the script uses the cryptogen tool using the configuration files located in the `organizations/cryptogen` folder. If you use the `-ca` flag to create Certificate Authorities, the script uses Fabric CA server configuration files and `registerEnroll.sh` script located in the `organizations/fabric-ca` folder. Both cryptogen and the Fabric CAs create the crypto material and MSP folders for all three organizations in the `organizations` folder.
- Once the organization crypto material has been generated, the `network.sh` can bring up the nodes of the network. The script uses the `docker-compose-test-net.yaml` file in the `docker` folder to create the peer and orderer nodes. The `docker` folder also contains the `docker-compose-e2e.yaml` file that brings up the nodes of the network alongside three Fabric CAs. This file is meant to be used to run end-to-end tests by the Fabric SDK. Refer to the [Node SDK](#) repo for details on running these tests.
- If you use the `createChannel` subcommand, `./network.sh` runs the `createChannel.sh` script in the `scripts` folder to create a channel using the supplied channel name. The script uses the `configtxgen` tool to create the channel genesis block based on the `TwoOrgsApplicationGenesis` channel profile in the `configtx/configtx.yaml` file. After creating the channel, the script uses the peer cli to join `peer0.org1.example.com` and `peer0.org2.example.com` to the channel, and make both of the peers anchor peers.
- If you issue the `deployCC` command, `./network.sh` runs the `deployCC.sh` script to install the **asset-transfer (basic)** chaincode on both peers and then define then chaincode on the channel. Once the chaincode

definition is committed to the channel, the peer cli initializes the chaincode using the `Init` and invokes the chaincode to put initial data on the ledger.

5.3.10 Troubleshooting

If you have any problems with the tutorial, review the following:

- You should always start your network fresh. You can use the following command to remove the artifacts, crypto material, containers, volumes, and chaincode images from previous runs:

```
./network.sh down
```

You **will** see errors if you do not remove old containers, images, and volumes.

- If you see Docker errors, first check your Docker version ([Prerequisites](#)), and then try restarting your Docker process. Problems with Docker are oftentimes not immediately recognizable. For example, you may see errors that are the result of your node not being able to access the crypto material mounted within a container.

If problems persist, you can remove your images and start from scratch:

```
docker rm -f $(docker ps -aq)
docker rmi -f $(docker images -q)
```

- If you are running Docker Desktop on macOS and experience the following error during chaincode installation:

```
Error: chaincode install failed with status: 500 - failed to invoke backing_
↳ implementation of 'InstallChaincode': could not build chaincode: docker build_
↳ failed: docker image inspection failed: Get "http://unix.sock/images/dev-peer0.
↳ org1.example.com-basic_1.0-
↳ 4ec191e793b27e953ff2ede5a8bcc63152cecb1e4c3f301a26e22692c61967ad-
↳ 42f57faac8360472e47cbbbf3940e81bba83439702d085878d148089a1b213ca/json": dial_
↳ unix /host/var/run/docker.sock: connect: no such file or directory
Chaincode installation on peer0.org1 has failed
Deploying chaincode failed
```

This problem is caused by a newer version of Docker Desktop for macOS. To resolve this issue, in the Docker Desktop preferences, uncheck the box `Use gRPC FUSE for file sharing` to use the legacy `osxfs` file sharing instead and click **Apply & Restart**.

- If you see errors on your `create`, `approve`, `commit`, `invoke` or `query` commands, make sure you have properly updated the channel name and chaincode name. There are placeholder values in the supplied sample commands.
- If you see the error below:

```
Error: Error endorsing chaincode: rpc error: code = 2 desc = Error installing_
↳ chaincode code mycc:1.0(chaincode /var/hyperledger/production/chaincodes/mycc.1.
↳ 0 exits)
```

You likely have chaincode images (e.g. `dev-peer1.org2.example.com-asset-transfer-1.0` or `dev-peer0.org1.example.com-asset-transfer-1.0`) from prior runs. Remove them and try again.

```
docker rmi -f $(docker images | grep dev-peer[0-9] | awk '{print $3}')
```

- If you see the below error:

```
[configtx/tool/localconfig] Load -> CRIT 002 Error reading configuration:
↳Unsupported Config Type ""
panic: Error reading configuration: Unsupported Config Type ""
```

Then you did not set the `FABRIC_CFG_PATH` environment variable properly. The `configtxgen` tool needs this variable in order to locate the `configtx.yaml`. Go back and execute an `export FABRIC_CFG_PATH=$PWD/configtx/configtx.yaml`, then recreate your channel artifacts.

- If you see an error stating that you still have “active endpoints”, then prune your Docker networks. This will wipe your previous networks and start you with a fresh environment:

```
docker network prune
```

You will see the following message:

```
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N]
```

Select `y`.

- If you try to create a channel with the command `./network.sh createChannel`, and it fails with the following error:

```
[comm.tls] ClientHandshake -> ERRO 003 Client TLS handshake failed after 1.
↳908956ms with error: EOF remoteaddress=127.0.0.1:7051
Error: error getting endorser client for channel: endorser client failed to
↳connect to localhost:7051: failed to create new connection: context deadline
↳exceeded
After 5 attempts, peer0.org1 has failed to join channel 'mychannel'
```

You need to uninstall Docker Desktop and reinstall the recommended version 2.5.0.1. Then, reclone the `fabric-samples` repository before reattempting the commands.

- If you see an error similar to the following:

```
/bin/bash: ./scripts/createChannel.sh: /bin/bash^M: bad interpreter: No such file
↳or directory
```

Ensure that the file in question (**`createChannel.sh`** in this example) is encoded in the Unix format. This was most likely caused by not setting `core.autocrlf` to `false` in your Git configuration (see [Windows extras](#)). There are several ways of fixing this. If you have access to the vim editor for instance, open the file:

```
vim ./fabric-samples/test-network/scripts/createChannel.sh
```

Then change its format by executing the following vim command:

```
:set ff=unix
```

If you continue to see errors, share your logs on the **fabric-questions** channel on [Hyperledger Rocket Chat](#) or on [StackOverflow](#).

Before we begin, if you haven’t already done so, you may wish to check that you have all the *Prerequisites* installed on the platform(s) on which you’ll be developing blockchain applications and/or operating Hyperledger Fabric.

Once you have the prerequisites installed, you are ready to download and install HyperLedger Fabric. While we work on developing real installers for the Fabric binaries, we provide a script that will *Install Samples, Binaries, and Docker Images* to your system. The script also will download the Docker images to your local registry.

After you have downloaded the Fabric Samples and Docker images to your local machine, you can get started working with Fabric with the *Using the Fabric test network* tutorial.

5.4 Hyperledger Fabric smart contract (chaincode) APIs

Hyperledger Fabric offers a number of APIs to support developing smart contracts (chaincode) in various programming languages. Smart contract APIs are available for Go, Node.js, and Java:

- [Go contract-api](#).
- [Node.js contract API](#) and [Node.js contract API documentation](#).
- [Java contract API](#) and [Java contract API documentation](#).

5.5 Hyperledger Fabric application SDKs

Hyperledger Fabric offers a number of SDKs to support developing applications in various programming languages. SDKs are available for Node.js and Java:

- [Node.js SDK](#) and [Node.js SDK documentation](#).
- [Java SDK](#) and [Java SDK documentation](#).
- [Go SDK](#) and [Go SDK documentation](#).

Prerequisites for developing with the SDKs can be found in the [Node.js SDK README](#) , [Java SDK README](#), and [Go SDK README](#).

In addition, there is one other application SDK that has not yet been officially released for Python, but is still available for downloading and testing:

- [Python SDK](#).

Currently, Node.js, Java and Go support the new application programming model delivered in Hyperledger Fabric v1.4.

5.6 Hyperledger Fabric CA

Hyperledger Fabric provides an optional [certificate authority service](#) that you may choose to use to generate the certificates and key material to configure and manage identity in your blockchain network. However, any CA that can generate ECDSA certificates may be used.

6.1 The scenario

Audience: Architects, Application and smart contract developers, Business professionals

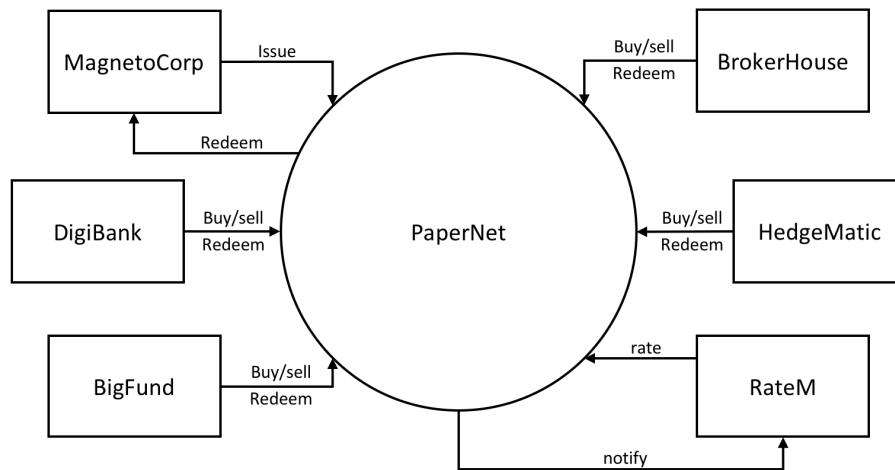
In this topic, we're going to describe a business scenario involving six organizations who use PaperNet, a commercial paper network built on Hyperledger Fabric, to issue, buy and redeem commercial paper. We're going to use the scenario to outline requirements for the development of commercial paper applications and smart contracts used by the participant organizations.

6.1.1 What is commercial paper?

Commercial paper is a commonly used type of unsecured, short-term debt instrument issued by corporations, typically used for the financing of payroll, accounts payable and inventories, and meeting other short-term liabilities. Maturities on commercial paper typically last several days, and rarely range longer than 270 days. The face value of the commercial paper is the value the issuing corporation would be paying the redeemer of the paper upon maturity. While buying the paper, the lender buys it for a price lesser than the face value. The difference between the face value and the price the lender bought the paper for is the profit made by the lender.

6.1.2 PaperNet network

PaperNet is a commercial paper network that allows suitably authorized participants to issue, trade, redeem and rate commercial paper.



The PaperNet commercial paper network. Six organizations currently use PaperNet network to issue, buy, sell, redeem and rate commercial paper. MagentoCorp issues and redeems commercial paper. DigiBank, BigFund, BrokerHouse and HedgeMatic all trade commercial paper with each other. RateM provides various measures of risk for commercial paper.

Let's see how MagnetoCorp uses PaperNet and commercial paper to help its business.

6.1.3 Introducing the actors

MagnetoCorp is a well-respected company that makes self-driving electric vehicles. In early April 2020, MagnetoCorp won a large order to manufacture 10,000 Model D cars for Daintree, a new entrant in the personal transport market. Although the order represents a significant win for MagnetoCorp, Daintree will not have to pay for the vehicles until they start to be delivered on November 1, six months after the deal was formally agreed between MagnetoCorp and Daintree.

To manufacture the vehicles, MagnetoCorp will need to hire 1000 workers for at least 6 months. This puts a short term strain on its finances – it will require an extra 5M USD each month to pay these new employees. **Commercial paper** is designed to help MagnetoCorp overcome its short term financing needs – to meet payroll every month based on the expectation that it will be cash rich when Daintree starts to pay for its new Model D cars.

At the end of May, MagnetoCorp needs 5M USD to meet payroll for the extra workers it hired on May 1. To do this, it issues a commercial paper with a face value of 5M USD with a maturity date 6 months in the future – when it expects to see cash flow from Daintree. DigiBank thinks that MagnetoCorp is creditworthy, and therefore doesn't require much of a premium above the central bank base rate of 2%, which would value 4.95M USD today at 5M USD in 6 months time. It therefore purchases the MagnetoCorp 6 month commercial paper for 4.94M USD – a slight discount compared to the 4.95M USD it is worth. DigiBank fully expects that it will be able to redeem 5M USD from MagnetoCorp in 6 months time, making it a profit of 10K USD for bearing the increased risk associated with this commercial paper. This extra 10K means it receives a 2.4% return on investment – significantly better than the risk free return of 2%.

At the end of June, when MagnetoCorp issues a new commercial paper for 5M USD to meet June's payroll, it is purchased by BigFund for 4.94M USD. That's because the commercial conditions are roughly the same in June as they are in May, resulting in BigFund valuing MagnetoCorp commercial paper at the same price that DigiBank did in May.

Each subsequent month, MagnetoCorp can issue new commercial paper to meet its payroll obligations, and these may be purchased by DigiBank, or any other participant in the PaperNet commercial paper network – BigFund, HedgeMatic or BrokerHouse. These organizations may pay more or less for the commercial paper depending on two factors – the

central bank base rate, and the risk associated with MagnetoCorp. This latter figure depends on a variety of factors such as the production of Model D cars, and the creditworthiness of MagnetoCorp as assessed by RateM, a ratings agency.

The organizations in PaperNet have different roles, MagnetoCorp issues paper, DigiBank, BigFund, HedgeMatic and BrokerHouse trade paper and RateM rates paper. Organizations of the same role, such as DigiBank, Bigfund, HedgeMatic and BrokerHouse are competitors. Organizations of different roles are not necessarily competitors, yet might still have opposing business interest, for example MagentoCorp will desire a high rating for its papers to sell them at a high price, while DigiBank would benefit from a low rating, such that it can buy them at a low price. As can be seen, even a seemingly simple network such as PaperNet can have complex trust relationships. A blockchain can help establish trust among organizations that are competitors or have opposing business interests that might lead to disputes. Fabric in particular has the means to capture even fine-grained trust relationships.

Let's pause the MagnetoCorp story for a moment, and develop the client applications and smart contracts that PaperNet uses to issue, buy, sell and redeem commercial paper as well as capture the trust relationships between the organizations. We'll come back to the role of the rating agency, RateM, a little later.

6.2 Analysis

Audience: Architects, Application and smart contract developers, Business professionals

Let's analyze commercial paper in a little more detail. PaperNet participants such as MagnetoCorp and DigiBank use commercial paper transactions to achieve their business objectives – let's examine the structure of a commercial paper and the transactions that affect it over time. We will also consider which organizations in PaperNet need to sign off on a transaction based on the trust relationships among the organizations in the network. Later we'll focus on how money flows between buyers and sellers; for now, let's focus on the first paper issued by MagnetoCorp.

6.2.1 Commercial paper lifecycle

A paper 00001 is issued by MagnetoCorp on May 31. Spend a few moments looking at the first **state** of this paper, with its different properties and values:

```
Issuer = MagnetoCorp
Paper = 00001
Owner = MagnetoCorp
Issue date = 31 May 2020
Maturity = 30 November 2020
Face value = 5M USD
Current state = issued
```

This paper state is a result of the **issue** transaction and it brings MagnetoCorp's first commercial paper into existence! Notice how this paper has a 5M USD face value for redemption later in the year. See how the `Issuer` and `Owner` are the same when paper 00001 is issued. Notice that this paper could be uniquely identified as `MagnetoCorp00001` – a composition of the `Issuer` and `Paper` properties. Finally, see how the property `Current state = issued` quickly identifies the stage of MagnetoCorp paper 00001 in its lifecycle.

Shortly after issuance, the paper is bought by DigiBank. Spend a few moments looking at how the same commercial paper has changed as a result of this **buy** transaction:

```
Issuer = MagnetoCorp
Paper = 00001
Owner = DigiBank
Issue date = 31 May 2020
Maturity date = 30 November 2020
```

(continues on next page)

(continued from previous page)

```
Face value = 5M USD
Current state = trading
```

The most significant change is that of `Owner` – see how the paper initially owned by `MagnetoCorp` is now owned by `DigiBank`. We could imagine how the paper might be subsequently sold to `BrokerHouse` or `HedgeMatic`, and the corresponding change to `Owner`. Note how `Current state` allow us to easily identify that the paper is now trading.

After 6 months, if `DigiBank` still holds the commercial paper, it can redeem it with `MagnetoCorp`:

```
Issuer = MagnetoCorp
Paper = 00001
Owner = MagnetoCorp
Issue date = 31 May 2020
Maturity date = 30 November 2020
Face value = 5M USD
Current state = redeemed
```

This final **redeem** transaction has ended the commercial paper’s lifecycle – it can be considered closed. It is often mandatory to keep a record of redeemed commercial papers, and the `redeemed` state allows us to quickly identify these. The value of `Owner` of a paper can be used to perform access control on the **redeem** transaction, by comparing the `Owner` against the identity of the transaction creator. Fabric supports this through the `getCreator()` [chaincode API](#). If Go is used as a chaincode language, the [client identity chaincode library](#) can be used to retrieve additional attributes of the transaction creator.

6.2.2 Transactions

We’ve seen that paper 00001’s lifecycle is relatively straightforward – it moves between `issued`, `trading` and `redeemed` as a result of an **issue**, **buy**, or **redeem** transaction.

These three transactions are initiated by `MagnetoCorp` and `DigiBank` (twice), and drive the state changes of paper 00001. Let’s have a look at the transactions that affect this paper in a little more detail:

Issue

Examine the first transaction initiated by `MagnetoCorp`:

```
Txn = issue
Issuer = MagnetoCorp
Paper = 00001
Issue time = 31 May 2020 09:00:00 EST
Maturity date = 30 November 2020
Face value = 5M USD
```

See how the **issue** transaction has a structure with properties and values. This transaction structure is different to, but closely matches, the structure of paper 00001. That’s because they are different things – paper 00001 reflects a state of `PaperNet` that is a result of the **issue** transaction. It’s the logic behind the **issue** transaction (which we cannot see) that takes these properties and creates this paper. Because the transaction **creates** the paper, it means there’s a very close relationship between these structures.

The only organization that is involved in the **issue** transaction is `MagnetoCorp`. Naturally, `MagnetoCorp` needs to sign off on the transaction. In general, the issuer of a paper is required to sign off on a transaction that issues a new paper.

Buy

Next, examine the **buy** transaction which transfers ownership of paper 00001 from MagnetoCorp to DigiBank:

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = MagnetoCorp
New owner = DigiBank
Purchase time = 31 May 2020 10:00:00 EST
Price = 4.94M USD
```

See how the **buy** transaction has fewer properties that end up in this paper. That's because this transaction only **modifies** this paper. It's only `New owner = DigiBank` that changes as a result of this transaction; everything else is the same. That's OK – the most important thing about the **buy** transaction is the change of ownership, and indeed in this transaction, there's an acknowledgement of the current owner of the paper, MagnetoCorp.

You might ask why the `Purchase time` and `Price` properties are not captured in paper 00001? This comes back to the difference between the transaction and the paper. The 4.94 M USD price tag is actually a property of the transaction, rather than a property of this paper. Spend a little time thinking about this difference; it is not as obvious as it seems. We're going to see later that the ledger will record both pieces of information – the history of all transactions that affect this paper, as well its latest state. Being clear on this separation of information is really important.

It's also worth remembering that paper 00001 may be bought and sold many times. Although we're skipping ahead a little in our scenario, let's examine what transactions we **might** see if paper 00001 changes ownership.

If we have a purchase by BigFund:

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = DigiBank
New owner = BigFund
Purchase time = 2 June 2020 12:20:00 EST
Price = 4.93M USD
```

Followed by a subsequent purchase by HedgeMatic:

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = BigFund
New owner = HedgeMatic
Purchase time = 3 June 2020 15:59:00 EST
Price = 4.90M USD
```

See how the paper owners changes, and how in our example, the price changes. Can you think of a reason why the price of MagnetoCorp commercial paper might be falling?

Intuitively, a **buy** transaction demands that both the selling as well as the buying organization need to sign off on such a transaction such that there is proof of the mutual agreement among the two parties that are part of the deal.

Redeem

The **redeem** transaction for paper 00001 represents the end of its lifecycle. In our relatively simple example, HedgeMatic initiates the transaction which transfers the commercial paper back to MagnetoCorp:

```
Txn = redeem
Issuer = MagnetoCorp
Paper = 00001
Current owner = HedgeMatic
Redeem time = 30 Nov 2020 12:00:00 EST
```

Again, notice how the **redeem** transaction has very few properties; all of the changes to paper 00001 can be calculated data by the redeem transaction logic: the `Issuer` will become the new owner, and the `Current state` will change to `redeemed`. The `Current owner` property is specified in our example, so that it can be checked against the current holder of the paper.

From a trust perspective, the same reasoning of the **buy** transaction also applies to the **redeem** instruction: both organizations involved in the transaction are required to sign off on it.

6.2.3 The Ledger

In this topic, we've seen how transactions and the resultant paper states are the two most important concepts in PaperNet. Indeed, we'll see these two fundamental elements in any Hyperledger Fabric distributed **ledger** – a world state, that contains the current value of all objects, and a blockchain that records the history of all transactions that resulted in the current world state.

The required sign-offs on transactions are enforced through rules, which are evaluated before appending a transaction to the ledger. Only if the required signatures are present, Fabric will accept a transaction as valid.

You're now in a great place to translate these ideas into a smart contract. Don't worry if your programming is a little rusty, we'll provide tips and pointers to understand the program code. Mastering the commercial paper smart contract is the first big step towards designing your own application. Or, if you're a business analyst who's comfortable with a little programming, don't be afraid to dig a little deeper!

6.3 Process and Data Design

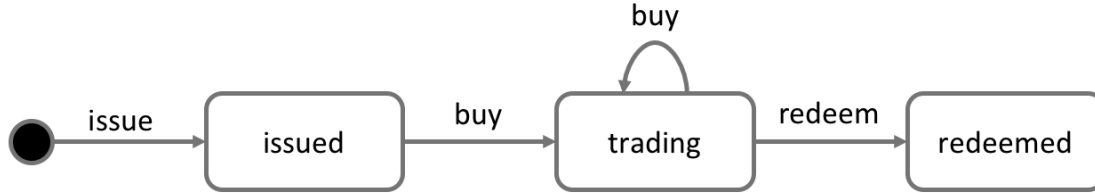
Audience: Architects, Application and smart contract developers, Business professionals

This topic shows you how to design the commercial paper processes and their related data structures in PaperNet. Our **analysis** highlighted that modelling PaperNet using states and transactions provided a precise way to understand what's happening. We're now going to elaborate on these two strongly related concepts to help us subsequently design the smart contracts and applications of PaperNet.

6.3.1 Lifecycle

As we've seen, there are two important concepts that concern us when dealing with commercial paper; **states** and **transactions**. Indeed, this is true for *all* blockchain use cases; there are conceptual objects of value, modeled as states, whose lifecycle transitions are described by transactions. An effective analysis of states and transactions is an essential starting point for a successful implementation.

We can represent the life cycle of a commercial paper using a state transition diagram:



The state transition diagram for commercial paper. Commercial papers transition between **issued**, **trading** and **redeemed** states by means of the **issue**, **buy** and **redeem** transactions.

See how the state diagram describes how commercial papers change over time, and how specific transactions govern the life cycle transitions. In Hyperledger Fabric, smart contracts implement transaction logic that transition commercial papers between their different states. Commercial paper states are actually held in the ledger world state; so let's take a closer look at them.

6.3.2 Ledger state

Recall the structure of a commercial paper:

```

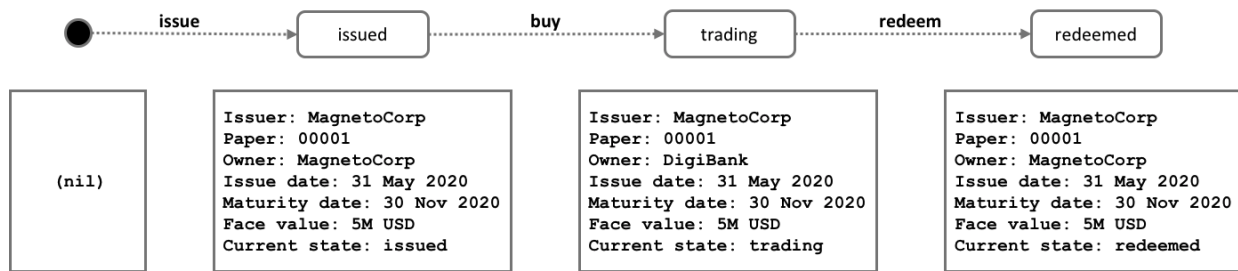
Issuer: MagnetoCorp
Paper: 00001
Owner: DigiBank
Issue date: 31 May 2020
Maturity date: 30 Nov 2020
Face value: 5M USD
Current state: trading
  
```

A commercial paper can be represented as a set of properties, each with a value. Typically, some combination of these properties will provide a unique key for each paper.

See how a commercial paper `Paper` property has value `00001`, and the `Face value` property has value `5M USD`. Most importantly, the `Current state` property indicates whether the commercial paper is `issued`, `trading` or `redeemed`. In combination, the full set of properties make up the **state** of a commercial paper. Moreover, the entire collection of these individual commercial paper states constitutes the ledger **world state**.

All ledger state share this form; each has a set of properties, each with a different value. This *multi-property* aspect of states is a powerful feature – it allows us to think of a Fabric state as a vector rather than a simple scalar. We then represent facts about whole objects as individual states, which subsequently undergo transitions controlled by transaction logic. A Fabric state is implemented as a key/value pair, in which the value encodes the object properties in a format that captures the object's multiple properties, typically JSON. The **ledger database** can support advanced query operations against these properties, which is very helpful for sophisticated object retrieval.

See how MagnetoCorp's paper `00001` is represented as a state vector that transitions according to different transaction stimuli:



A commercial paper state is brought into existence and transitions as a result of different transactions. Hyperledger Fabric states have multiple properties, making them vectors rather than scalars.

Notice how each individual paper starts with the empty state, which is technically a `nil` state for the paper, as it doesn't exist! See how paper `00001` is brought into existence by the **issue** transaction, and how it is subsequently updated as a result of the **buy** and **redeem** transactions.

Notice how each state is self-describing; each property has a name and a value. Although all our commercial papers currently have the same properties, this need not be the case for all time, as Hyperledger Fabric supports different states having different properties. This allows the same ledger world state to contain different forms of the same asset as well as different types of asset. It also makes it possible to update a state's structure; imagine a new regulation that requires an additional data field. Flexible state properties support the fundamental requirement of data evolution over time.

6.3.3 State keys

In most practical applications, a state will have a combination of properties that uniquely identify it in a given context – it's **key**. The key for a PaperNet commercial paper is formed by a concatenation of the `Issuer` and `paper` properties; so for MagnetoCorp's first paper, it's `MagnetoCorp00001`.

A state key allows us to uniquely identify a paper; it is created as a result of the **issue** transaction and subsequently updated by **buy** and **redeem**. Hyperledger Fabric requires each state in a ledger to have a unique key.

When a unique key is not available from the available set of properties, an application-determined unique key is specified as an input to the transaction that creates the state. This unique key is usually with some form of **UUID**, which although less readable, is a standard practice. What's important is that every individual state object in a ledger must have a unique key.

Note: You should avoid using `U+0000` (nil byte) in keys.

6.3.4 Multiple states

As we've seen, commercial papers in PaperNet are stored as state vectors in a ledger. It's a reasonable requirement to be able to query different commercial papers from the ledger; for example: find all the papers issued by MagnetoCorp, or: find all the papers issued by MagnetoCorp in the `redeemed` state.

To make these kinds of search tasks possible, it's helpful to group all related papers together in a logical list. The PaperNet design incorporates the idea of a commercial paper list – a logical container which is updated whenever commercial papers are issued or otherwise changed.

Logical representation

It's helpful to think of all PaperNet commercial papers being in a single list of commercial papers:

commercial paper: MagnetoCorp paper 00004

Issuer : MagnetoCorp	Paper: 00004	Owner: DigiBank	Issue date: 31 August 2020	Maturity date: 31 March 2021	Face value: 5m USD	Current state: issued
-------------------------	-----------------	--------------------	-------------------------------	---------------------------------	-----------------------	--------------------------

commercial paper list: org.papernet.paper

add	Issuer : MagnetoCorp	Paper: 00001	Owner: DigiBank	Issue date: 31 May 2020	Maturity date: 31 December 2020	Face value: 5m USD	Current state: trading
	Issuer : MagnetoCorp	Paper: 00002	Owner: BigFund	Issue date: 30 June 2020	Maturity date: 31 January 2021	Face value: 5m USD	Current state: trading
	Issuer : MagnetoCorp	Paper: 00003	Owner: BrokerHouse	Issue date: 31 July 2020	Maturity date: 28 February 2021	Face value: 5m USD	Current state: trading

MagnetoCorp's newly created commercial paper 00004 is added to the list of existing commercial papers.

New papers can be added to the list as a result of an **issue** transaction, and papers already in the list can be updated with **buy** or **redeem** transactions. See how the list has a descriptive name: `org.papernet.papers`; it's a really good idea to use this kind of [DNS name](#) because well-chosen names will make your blockchain designs intuitive to other people. This idea applies equally well to smart contract [names](#).

Physical representation

While it's correct to think of a single list of papers in PaperNet – `org.papernet.papers` – lists are best implemented as a set of individual Fabric states, whose composite key associates the state with its list. In this way, each state's composite key is both unique and supports effective list query.

key	value
org.papernet.paperMagnetoCorp00001	Issuer : MagnetoCorp, Paper: 00001, Owner: DigiBank, Issue date: 31 May 2020, Maturity date: 31 December 2020, Face value: 5m USD, Current state: trading
org.papernet.paperMagnetoCorp00002	Issuer : MagnetoCorp, Paper: 00002, Owner: BigFund, Issue date: 30 June 2020, Maturity date: 31 January 2021, Face value: 5m USD, Current state: trading
org.papernet.paperMagnetoCorp00003	Issuer : MagnetoCorp, Paper: 00003, Owner: BrokerHouse, Issue date: 31 July 2020, Maturity date: 28 February 2021, Face value: 5m USD, Current state: trading
org.papernet.paperMagnetoCorp00004	Issuer : MagnetoCorp, Paper: 00004, Owner: DigiBank, Issue date: 31 August 2020, Maturity date: 31 March 2021, Face value: 5m USD, Current state: issued

Representing a list of PaperNet commercial papers as a set of distinct Hyperledger Fabric states

Notice how each paper in the list is represented by a vector state, with a unique **composite** key formed by the concatenation of `org.papernet.paper`, Issuer and Paper properties. This structure is helpful for two reasons:

- It allows us to examine any state vector in the ledger to determine which list it's in, without reference to a separate list. It's analogous to looking at set of sports fans, and identifying which team they support by the colour of the shirt they are wearing. The sports fans self-declare their allegiance; we don't need a list of fans.
- Hyperledger Fabric internally uses a concurrency control mechanism to update a ledger, such that keeping papers in separate state vectors vastly reduces the opportunity for shared-state collisions. Such collisions require transaction re-submission, complicate application design, and decrease performance.

This second point is actually a key take-away for Hyperledger Fabric; the physical design of state vectors is **very important** to optimum performance and behaviour. Keep your states separate!

6.3.5 Trust relationships

We have discussed how the different roles in a network, such as issuer, trader or rating agencies as well as different business interests determine who needs to sign off on a transaction. In Fabric, these rules are captured by so-called **endorsement policies**. The rules can be set on a chaincode granularity, as well as for individual state keys.

This means that in PaperNet, we can set one rule for the whole namespace that determines which organizations can issue new papers. Later, rules can be set and updated for individual papers to capture the trust relationships of buy and redeem transactions.

In the next topic, we will show you how to combine these design concepts to implement the PaperNet commercial paper smart contract, and then an application in exploits it!

6.4 Smart Contract Processing

Audience: Architects, Application and smart contract developers

At the heart of a blockchain network is a smart contract. In PaperNet, the code in the commercial paper smart contract defines the valid states for commercial paper, and the transaction logic that transition a paper from one state to another. In this topic, we're going to show you how to implement a real world smart contract that governs the process of issuing, buying and redeeming commercial paper.

We're going to cover:

- *What is a smart contract and why it's important*
- *How to define a smart contract*
- *How to define a transaction*
- *How to implement a transaction*
- *How to represent a business object in a smart contract*
- *How to store and retrieve an object in the ledger*

If you'd like, you can [download the sample](#) and even [run it locally](#). It is written in JavaScript and Java, but the logic is quite language independent, so you'll easily be able to see what's going on! (The sample will become available for Go as well.)

6.4.1 Smart Contract

A smart contract defines the different states of a business object and governs the processes that move the object between these different states. Smart contracts are important because they allow architects and smart contract developers to define the key business processes and data that are shared across the different organizations collaborating in a blockchain network.

In the PaperNet network, the smart contract is shared by the different network participants, such as MagnetoCorp and DigiBank. The same version of the smart contract must be used by all applications connected to the network so that they jointly implement the same shared business processes and data.

6.4.2 Implementation Languages

There are two runtimes that are supported, the Java Virtual Machine and Node.js. This gives the opportunity to use one of JavaScript, TypeScript, Java or any other language that can run on one of these supported runtimes.

In Java and TypeScript, annotations or decorators are used to provide information about the smart contract and its structure. This allows for a richer development experience — for example, author information or return types can be enforced. Within JavaScript, conventions must be followed, therefore, there are limitations around what can be determined automatically.

Examples here are given in both JavaScript and Java.

6.4.3 Contract class

A copy of the PaperNet commercial paper smart contract is contained in a single file. View it with your browser, or open it in your favorite editor if you've downloaded it.

- `papercontract.js` - [JavaScript version](#)
- `CommercialPaperContract.java` - [Java version](#)

You may notice from the file path that this is MagnetoCorp's copy of the smart contract. MagnetoCorp and DigiBank must agree on the version of the smart contract that they are going to use. For now, it doesn't matter which organization's copy you use, they are all the same.

Spend a few moments looking at the overall structure of the smart contract; notice that it's quite short! Towards the top of the file, you'll see that there's a definition for the commercial paper smart contract:

JavaScript

```
class CommercialPaperContract extends Contract {...}
```

Java

```
@Contract(...)
@Default
public class CommercialPaperContract implements ContractInterface {...}
```

The `CommercialPaperContract` class contains the transaction definitions for commercial paper – **issue**, **buy** and **redeem**. It's these transactions that bring commercial papers into existence and move them through their lifecycle. We'll examine these *transactions* soon, but for now notice for JavaScript, that the `CommercialPaperContract` extends the Hyperledger Fabric `Contract` class.

With Java, the class must be decorated with the `@Contract(...)` annotation. This provides the opportunity to supply additional information about the contract, such as license and author. The `@Default()` annotation indicates that this contract class is the default contract class. Being able to mark a contract class as the default contract class is useful in some smart contracts which have multiple contract classes.

If you are using a TypeScript implementation, there are similar `@Contract(...)` annotations that fulfill the same purpose as in Java.

For more information on the available annotations, consult the available API documentation:

- [API documentation for Java smart contracts](#)
- [API documentation for Node.js smart contracts](#)

The Fabric contract class is also available for smart contracts written in Go. While we do not discuss the Go contract API in this topic, it uses similar concepts as the API for Java and JavaScript:

- [API documentation for Go smart contracts](#)

These classes, annotations, and the Context class, were brought into scope earlier:

JavaScript

```
const { Contract, Context } = require('fabric-contract-api');
```

Java

```
import org.hyperledger.fabric.contract.Context;
import org.hyperledger.fabric.contract.ContractInterface;
import org.hyperledger.fabric.contract.annotation.Contract;
import org.hyperledger.fabric.contract.annotation.Contract;
import org.hyperledger.fabric.contract.annotation.Default;
import org.hyperledger.fabric.contract.annotation.Info;
import org.hyperledger.fabric.contract.annotation.License;
import org.hyperledger.fabric.contract.annotation.Transaction;
```

Our commercial paper contract will use built-in features of these classes, such as automatic method invocation, a per-transaction context, transaction handlers, and class-shared state.

Notice also how the JavaScript class constructor uses its superclass to initialize itself with an explicit contract name:

```
constructor() {
    super('org.papernet.commercialpaper');
}
```

With the Java class, the constructor is blank as the explicit contract name can be specified in the @Contract() annotation. If it's absent, then the name of the class is used.

Most importantly, org.papernet.commercialpaper is very descriptive – this smart contract is the agreed definition of commercial paper for all PaperNet organizations.

Usually there will only be one smart contract per file – contracts tend to have different lifecycles, which makes it sensible to separate them. However, in some cases, multiple smart contracts might provide syntactic help for applications, e.g. EuroBond, DollarBond, YenBond, but essentially provide the same function. In such cases, smart contracts and transactions can be disambiguated.

6.4.4 Transaction definition

Within the class, locate the **issue** method.

JavaScript

```
async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {...}
↪ }
```

Java

```
@Transaction
public CommercialPaper issue(CommercialPaperContext ctx,
                             String issuer,
                             String paperNumber,
                             String issueDateTime,
                             String maturityDateTime,
                             int faceValue) {...}
```

The Java annotation @Transaction is used to mark this method as a transaction definition; TypeScript has an equivalent annotation.

This function is given control whenever this contract is called to `issue` a commercial paper. Recall how commercial paper 00001 was created with the following transaction:

```
Txn = issue
Issuer = MagnetoCorp
Paper = 00001
Issue time = 31 May 2020 09:00:00 EST
Maturity date = 30 November 2020
Face value = 5M USD
```

We’ve changed the variable names for programming style, but see how these properties map almost directly to the `issue` method variables.

The `issue` method is automatically given control by the contract whenever an application makes a request to issue a commercial paper. The transaction property values are made available to the method via the corresponding variables. See how an application submits a transaction using the Hyperledger Fabric SDK in the [application](#) topic, using a sample application program.

You might have noticed an extra variable in the `issue` definition – `ctx`. It’s called the **transaction context**, and it’s always first. By default, it maintains both per-contract and per-transaction information relevant to *transaction logic*. For example, it would contain MagnetoCorp’s specified transaction identifier, a MagnetoCorp issuing user’s digital certificate, as well as access to the ledger API.

See how the smart contract extends the default transaction context by implementing its own `createContext()` method rather than accepting the default implementation:

JavaScript

```
createContext() {
  return new CommercialPaperContext()
}
```

Java

```
@Override
public Context createContext(ChaincodeStub stub) {
  return new CommercialPaperContext(stub);
}
```

This extended context adds a custom property `paperList` to the defaults:

JavaScript

```
class CommercialPaperContext extends Context {
  constructor() {
    super();
    // All papers are held in a list of papers
    this.paperList = new PaperList(this);
  }
}
```

Java

```
class CommercialPaperContext extends Context {
  public CommercialPaperContext(ChaincodeStub stub) {
    super(stub);
    this.paperList = new PaperList(this);
  }
}
```

(continues on next page)

(continued from previous page)

```
    public PaperList paperList;  
}
```

We'll soon see how `ctx.paperList` can be subsequently used to help store and retrieve all PaperNet commercial papers.

To solidify your understanding of the structure of a smart contract transaction, locate the **buy** and **redeem** transaction definitions, and see if you can see how they map to their corresponding commercial paper transactions.

The **buy** transaction:

```
Txn = buy  
Issuer = MagnetoCorp  
Paper = 00001  
Current owner = MagnetoCorp  
New owner = DigiBank  
Purchase time = 31 May 2020 10:00:00 EST  
Price = 4.94M USD
```

JavaScript

```
async buy(ctx, issuer, paperNumber, currentOwner, newOwner, price, purchaseTime) {...}
```

Java

```
@Transaction  
public CommercialPaper buy(CommercialPaperContext ctx,  
                           String issuer,  
                           String paperNumber,  
                           String currentOwner,  
                           String newOwner,  
                           int price,  
                           String purchaseDateTime) {...}
```

The **redeem** transaction:

```
Txn = redeem  
Issuer = MagnetoCorp  
Paper = 00001  
Redeemer = DigiBank  
Redeem time = 31 Dec 2020 12:00:00 EST
```

JavaScript

```
async redeem(ctx, issuer, paperNumber, redeemingOwner, redeemDateTime) {...}
```

Java

```
@Transaction  
public CommercialPaper redeem(CommercialPaperContext ctx,  
                              String issuer,  
                              String paperNumber,  
                              String redeemingOwner,  
                              String redeemDateTime) {...}
```

In both cases, observe the 1:1 correspondence between the commercial paper transaction and the smart contract method definition.

All of the JavaScript functions use the `async` and `await` keywords which allow JavaScript functions to be treated as if they were synchronous function calls.

6.4.5 Transaction logic

Now that you've seen how contracts are structured and transactions are defined, let's focus on the logic within the smart contract.

Recall the first **issue** transaction:

```
Txn = issue
Issuer = MagnetoCorp
Paper = 00001
Issue time = 31 May 2020 09:00:00 EST
Maturity date = 30 November 2020
Face value = 5M USD
```

It results in the **issue** method being passed control:

JavaScript

```
async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {

    // create an instance of the paper
    let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime,
    ↪maturityDateTime, faceValue);

    // Smart contract, rather than paper, moves paper into ISSUED state
    paper.setIssued();

    // Newly issued paper is owned by the issuer
    paper.setOwner(issuer);

    // Add the paper to the list of all similar commercial papers in the ledger world,
    ↪state
    await ctx.paperList.addPaper(paper);

    // Must return a serialized paper to caller of smart contract
    return paper.toBuffer();
}
```

Java

```
@Transaction
public CommercialPaper issue(CommercialPaperContext ctx,
                             String issuer,
                             String paperNumber,
                             String issueDateTime,
                             String maturityDateTime,
                             int faceValue) {

    System.out.println(ctx);

    // create an instance of the paper
    CommercialPaper paper = CommercialPaper.createInstance(issuer, paperNumber,
    ↪issueDateTime, maturityDateTime,
        faceValue, issuer, "");
```

(continues on next page)

(continued from previous page)

```

    // Smart contract, rather than paper, moves paper into ISSUED state
    paper.setIssued();

    // Newly issued paper is owned by the issuer
    paper.setOwner(issuer);

    System.out.println(paper);
    // Add the paper to the list of all similar commercial papers in the ledger
    // world state
    ctx.paperList.addPaper(paper);

    // Must return a serialized paper to caller of smart contract
    return paper;
}

```

The logic is simple: take the transaction input variables, create a new commercial paper `paper`, add it to the list of all commercial papers using `paperList`, and return the new commercial paper (serialized as a buffer) as the transaction response.

See how `paperList` is retrieved from the transaction context to provide access to the list of commercial papers. `issue()`, `buy()` and `redeem()` continually re-access `ctx.paperList` to keep the list of commercial papers up-to-date.

The logic for the **buy** transaction is a little more elaborate:

JavaScript

```

async buy(ctx, issuer, paperNumber, currentOwner, newOwner, price, purchaseDateTime) {

    // Retrieve the current paper using key fields provided
    let paperKey = CommercialPaper.makeKey([issuer, paperNumber]);
    let paper = await ctx.paperList.getPaper(paperKey);

    // Validate current owner
    if (paper.getOwner() !== currentOwner) {
        throw new Error('Paper ' + issuer + paperNumber + ' is not owned by ' +
currentOwner);
    }

    // First buy moves state from ISSUED to TRADING
    if (paper.isIssued()) {
        paper.setTrading();
    }

    // Check paper is not already REDEEMED
    if (paper.isTrading()) {
        paper.setOwner(newOwner);
    } else {
        throw new Error('Paper ' + issuer + paperNumber + ' is not trading. Current
state = ' + paper.getCurrentState());
    }

    // Update the paper
    await ctx.paperList.updatePaper(paper);
    return paper.toBuffer();
}

```

Java

```
@Transaction
public CommercialPaper buy(CommercialPaperContext ctx,
                           String issuer,
                           String paperNumber,
                           String currentOwner,
                           String newOwner,
                           int price,
                           String purchaseDateTime) {

    // Retrieve the current paper using key fields provided
    String paperKey = State.makeKey(new String[] { paperNumber });
    CommercialPaper paper = ctx.paperList.getPaper(paperKey);

    // Validate current owner
    if (!paper.getOwner().equals(currentOwner)) {
        throw new RuntimeException("Paper " + issuer + paperNumber + " is not owned_
↪by " + currentOwner);
    }

    // First buy moves state from ISSUED to TRADING
    if (paper.isIssued()) {
        paper.setTrading();
    }

    // Check paper is not already REDEEMED
    if (paper.isTrading()) {
        paper.setOwner(newOwner);
    } else {
        throw new RuntimeException(
            "Paper " + issuer + paperNumber + " is not trading. Current state = "
↪+ paper.getState());
    }

    // Update the paper
    ctx.paperList.updatePaper(paper);
    return paper;
}
```

See how the transaction checks `currentOwner` and that paper is TRADING before changing the owner with `paper.setOwner(newOwner)`. The basic flow is simple though – check some pre-conditions, set the new owner, update the commercial paper on the ledger, and return the updated commercial paper (serialized as a buffer) as the transaction response.

Why don't you see if you can understand the logic for the **redeem** transaction?

6.4.6 Representing an object

We've seen how to define and implement the **issue**, **buy** and **redeem** transactions using the `CommercialPaper` and `PaperList` classes. Let's end this topic by seeing how these classes work.

Locate the `CommercialPaper` class:

JavaScript In the `paper.js` file:

```
class CommercialPaper extends State {...}
```

Java In the `CommercialPaper.java` file:

```
@DataType()  
public class CommercialPaper extends State {...}
```

This class contains the in-memory representation of a commercial paper state. See how the `createInstance` method initializes a new commercial paper with the provided parameters:

JavaScript

```
static createInstance(issuer, paperNumber, issueDateTime, maturityDateTime, ↵  
↵faceValue) {  
    return new CommercialPaper({ issuer, paperNumber, issueDateTime, maturityDateTime, ↵  
↵faceValue });  
}
```

Java

```
public static CommercialPaper createInstance(String issuer, String paperNumber, ↵  
↵String issueDateTime,  
    String maturityDateTime, int faceValue, String owner, String state) {  
    return new CommercialPaper().setIssuer(issuer).setPaperNumber(paperNumber).  
↵setMaturityDateTime(maturityDateTime)  
        .setFaceValue(faceValue).setKey().setIssueDateTime(issueDateTime).  
↵setOwner(owner).setState(state);  
}
```

Recall how this class was used by the **issue** transaction:

JavaScript

```
let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime, ↵  
↵maturityDateTime, faceValue);
```

Java

```
CommercialPaper paper = CommercialPaper.createInstance(issuer, paperNumber, ↵  
↵issueDateTime, maturityDateTime,  
    faceValue, issuer, "");
```

See how every time the issue transaction is called, a new in-memory instance of a commercial paper is created containing the transaction data.

A few important points to note:

- This is an in-memory representation; we'll see *later* how it appears on the ledger.
- The `CommercialPaper` class extends the `State` class. `State` is an application-defined class which creates a common abstraction for a state. All states have a business object class which they represent, a composite key, can be serialized and de-serialized, and so on. `State` helps our code be more legible when we are storing more than one business object type on the ledger. Examine the `State` class in the `state.js` file.
- A paper computes its own key when it is created – this key will be used when the ledger is accessed. The key is formed from a combination of `issuer` and `paperNumber`.

```
constructor(obj) {  
    super(CommercialPaper.getClass(), [obj.issuer, obj.paperNumber]);  
    Object.assign(this, obj);  
}
```

- A paper is moved to the `ISSUED` state by the transaction, not by the paper class. That's because it's the smart contract that governs the lifecycle state of the paper. For example, an `import` transaction might create a new set of papers immediately in the `TRADING` state.

The rest of the `CommercialPaper` class contains simple helper methods:

```
getOwner() {
    return this.owner;
}
```

Recall how methods like this were used by the smart contract to move the commercial paper through its lifecycle. For example, in the **redeem** transaction we saw:

```
if (paper.getOwner() === redeemingOwner) {
    paper.setOwner(paper.getIssuer());
    paper.setRedeemed();
}
```

6.4.7 Access the ledger

Now locate the `PaperList` class in the `paperlist.js` file:

```
class PaperList extends StateList {
```

This utility class is used to manage all PaperNet commercial papers in Hyperledger Fabric state database. The `PaperList` data structures are described in more detail in the [architecture](#) topic.

Like the `CommercialPaper` class, this class extends an application-defined `StateList` class which creates a common abstraction for a list of states – in this case, all the commercial papers in PaperNet.

The `addPaper()` method is a simple veneer over the `StateList.addState()` method:

```
async addPaper(paper) {
    return this.addState(paper);
}
```

You can see in the `StateList.js` file how the `StateList` class uses the Fabric API `putState()` to write the commercial paper as state data in the ledger:

```
async addState(state) {
    let key = this.ctx.stub.createCompositeKey(this.name, state.getSplitKey());
    let data = State.serialize(state);
    await this.ctx.stub.putState(key, data);
}
```

Every piece of state data in a ledger requires these two fundamental elements:

- **Key:** key is formed with `createCompositeKey()` using a fixed name and the key of state. The name was assigned when the `PaperList` object was constructed, and `state.getSplitKey()` determines each state's unique key.
- **Data:** data is simply the serialized form of the commercial paper state, created using the `State.serialize()` utility method. The `State` class serializes and deserializes data using JSON, and the `State`'s business object class as required, in our case `CommercialPaper`, again set when the `PaperList` object was constructed.

Notice how a `StateList` doesn't store anything about an individual state or the total list of states – it delegates all of that to the Fabric state database. This is an important design pattern – it reduces the opportunity for [ledger MVCC collisions](#) in Hyperledger Fabric.

The `StateList` `getState()` and `updateState()` methods work in similar ways:

```
async getState(key) {  
  let ledgerKey = this.ctx.stub.createCompositeKey(this.name, State.splitKey(key));  
  let data = await this.ctx.stub.getState(ledgerKey);  
  let state = State.deserialize(data, this.supportedClasses);  
  return state;  
}
```

```
async updateState(state) {  
  let key = this.ctx.stub.createCompositeKey(this.name, state.getSplitKey());  
  let data = State.serialize(state);  
  await this.ctx.stub.putState(key, data);  
}
```

See how they use the Fabric APIs `putState()`, `getState()` and `createCompositeKey()` to access the ledger. We'll expand this smart contract later to list all commercial papers in `paperNet` – what might the method look like to implement this ledger retrieval?

That's it! In this topic you've understood how to implement the smart contract for `PaperNet`. You can move to the next sub topic to see how an application calls the smart contract using the Fabric SDK.

6.5 Application

Audience: Architects, Application and smart contract developers

An application can interact with a blockchain network by submitting transactions to a ledger or querying ledger content. This topic covers the mechanics of how an application does this; in our scenario, organizations access `PaperNet` using applications which invoke **issue**, **buy** and **redeem** transactions defined in a commercial paper smart contract. Even though `MagnetoCorp`'s application to issue a commercial paper is basic, it covers all the major points of understanding.

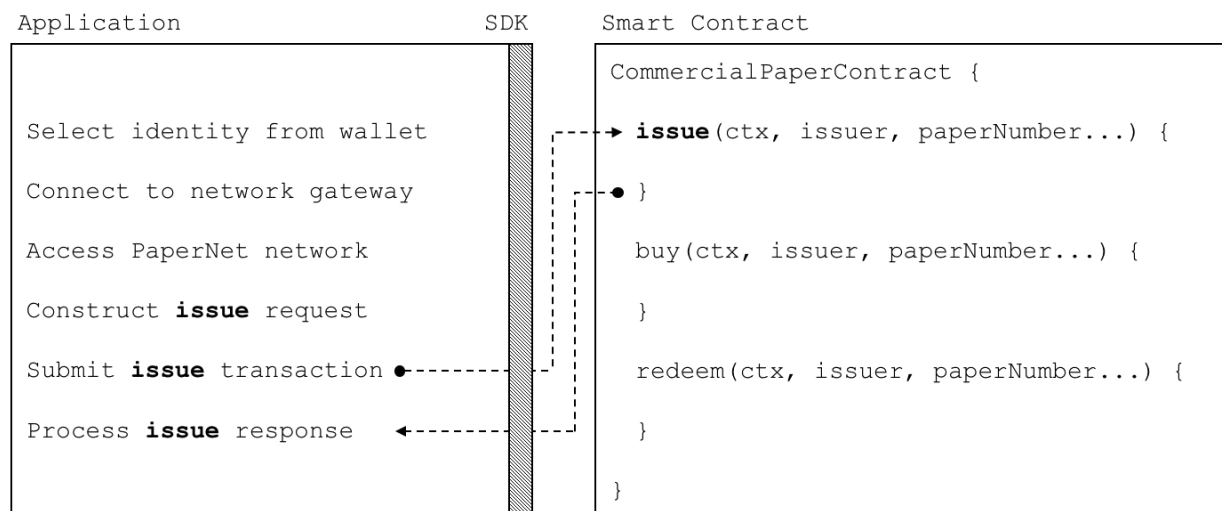
In this topic, we're going to cover:

- *The application flow to invoke a smart contract*
- *How an application uses a wallet and identity*
- *How an application connects using a gateway*
- *How to access a particular network*
- *How to construct a transaction request*
- *How to submit a transaction*
- *How to process a transaction response*

To help your understanding, we'll make reference to the commercial paper sample application provided with Hyperledger Fabric. You can [download it](#) and [run it locally](#). It is written in both JavaScript and Java, but the logic is quite language independent, so you'll easily be able to see what's going on! (The sample will become available for Go as well.)

6.5.1 Basic Flow

An application interacts with a blockchain network using the Fabric SDK. Here's a simplified diagram of how an application invokes a commercial paper smart contract:



A PaperNet application invokes the commercial paper smart contract to submit an issue transaction request.

An application has to follow six basic steps to submit a transaction:

- Select an identity from a wallet
- Connect to a gateway
- Access the desired network
- Construct a transaction request for a smart contract
- Submit the transaction to the network
- Process the response

You're going to see how a typical application performs these six steps using the Fabric SDK. You'll find the application code in the `issue.js` file. [View it](#) in your browser, or open it in your favourite editor if you've downloaded it. Spend a few moments looking at the overall structure of the application; even with comments and spacing, it's only 100 lines of code!

6.5.2 Wallet

Towards the top of `issue.js`, you'll see two Fabric classes are brought into scope:

```
const { Wallets, Gateway } = require('fabric-network');
```

You can read about the `fabric-network` classes in the [node SDK documentation](#), but for now, let's see how they are used to connect MagnetoCorp's application to PaperNet. The application uses the Fabric **Wallet** class as follows:

```
const wallet = await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');
```

See how `wallet` locates a `wallet` in the local filesystem. The identity retrieved from the wallet is clearly for a user called `Isabella`, who is using the `issue` application. The wallet holds a set of identities – X.509 digital certificates –

which can be used to access PaperNet or any other Fabric network. If you run the tutorial, and look in this directory, you'll see the identity credentials for Isabella.

Think of a [wallet](#) holding the digital equivalents of your government ID, driving license or ATM card. The X.509 digital certificates within it will associate the holder with a organization, thereby entitling them to rights in a network channel. For example, Isabella might be an administrator in MagnetoCorp, and this could give her more privileges than a different user – Balaji from DigiBank. Moreover, a smart contract can retrieve this identity during smart contract processing using the [transaction context](#).

Note also that wallets don't hold any form of cash or tokens – they hold identities.

6.5.3 Gateway

The second key class is a Fabric **Gateway**. Most importantly, a [gateway](#) identifies one or more peers that provide access to a network – in our case, PaperNet. See how `issue.js` connects to its gateway:

```
await gateway.connect(connectionProfile, connectionOptions);
```

`gateway.connect()` has two important parameters:

- **connectionProfile**: the file system location of a [connection profile](#) that identifies a set of peers as a gateway to PaperNet
- **connectionOptions**: a set of options used to control how `issue.js` interacts with PaperNet

See how the client application uses a gateway to insulate itself from the network topology, which might change. The gateway takes care of sending the transaction proposal to the right peer nodes in the network using the [connection profile](#) and [connection options](#).

Spend a few moments examining the [connection profile](#) `./gateway/connectionProfile.yaml`. It uses [YAML](#), making it easy to read.

It was loaded and converted into a JSON object:

```
let connectionProfile = yaml.safeLoad(file.readFileSync('./gateway/connectionProfile.  
↪yaml', 'utf8'));
```

Right now, we're only interested in the `channels:` and `peers:` sections of the profile: (We've modified the details slightly to better explain what's happening.)

```
channels:  
  papernet:  
    peers:  
      peer1.magnetocorp.com:  
        endorsingPeer: true  
        eventSource: true  
  
      peer2.digibank.com:  
        endorsingPeer: true  
        eventSource: true  
  
peers:  
  peer1.magnetocorp.com:  
    url: grpc://localhost:7051  
    grpcOptions:  
      ssl-target-name-override: peer1.magnetocorp.com  
      request-timeout: 120  
    tlsCACerts:
```

(continues on next page)

(continued from previous page)

```

    path: certificates/magnetocorp/magnetocorp.com-cert.pem

peer2.digibank.com:
  url: grpcs://localhost:8051
  grpcOptions:
    ssl-target-name-override: peer1.digibank.com
  tlsCACerts:
    path: certificates/digibank/digibank.com-cert.pem

```

See how `channel`: identifies the PaperNet: network channel, and two of its peers. MagnetoCorp has `peer1.magnetocorp.com` and DigiBank has `peer2.digibank.com`, and both have the role of endorsing peers. Link to these peers via the `peers`: key, which contains details about how to connect to them, including their respective network addresses.

The connection profile contains a lot of information – not just peers – but network channels, network orderers, organizations, and CAs, so don't worry if you don't understand all of it!

Let's now turn our attention to the `connectionOptions` object:

```

let connectionOptions = {
  identity: userName,
  wallet: wallet,
  discovery: { enabled:true, asLocalhost: true }
};

```

See how it specifies that `identity`, `userName`, and `wallet`, should be used to connect to a gateway. These were assigned values earlier in the code.

There are other [connection options](#) which an application could use to instruct the SDK to act intelligently on its behalf. For example:

```

let connectionOptions = {
  identity: userName,
  wallet: wallet,
  eventHandlerOptions: {
    commitTimeout: 100,
    strategy: EventStrategies.MSPID_SCOPE_ANYFORTX
  },
}

```

Here, `commitTimeout` tells the SDK to wait 100 seconds to hear whether a transaction has been committed. And `strategy: EventStrategies.MSPID_SCOPE_ANYFORTX` specifies that the SDK can notify an application after a single MagnetoCorp peer has confirmed the transaction, in contrast to `strategy: EventStrategies.NETWORK_SCOPE_ALLFORTX` which requires that all peers from MagnetoCorp and DigiBank to confirm the transaction.

If you'd like to, [read more](#) about how connection options allow applications to specify goal-oriented behaviour without having to worry about how it is achieved.

6.5.4 Network channel

The peers defined in the gateway `connectionProfile.yaml` provide `issue.js` with access to PaperNet. Because these peers can be joined to multiple network channels, the gateway actually provides the application with access to multiple network channels!

See how the application selects a particular channel:

```
const network = await gateway.getNetwork('PaperNet');
```

From this point onwards, `network` will provide access to PaperNet. Moreover, if the application wanted to access another network, BondNet, at the same time, it is easy:

```
const network2 = await gateway.getNetwork('BondNet');
```

Now our application has access to a second network, BondNet, simultaneously with PaperNet!

We can see here a powerful feature of Hyperledger Fabric – applications can participate in a **network of networks**, by connecting to multiple gateway peers, each of which is joined to multiple network channels. Applications will have different rights in different channels according to their wallet identity provided in `gateway.connect()`.

6.5.5 Construct request

The application is now ready to **issue** a commercial paper. To do this, it's going to use `CommercialPaperContract` and again, it's fairly straightforward to access this smart contract:

```
const contract = await network.getContract('papercontract', 'org.papernet.  
↪commercialpaper');
```

Note how the application provides a name – `papercontract` – and an explicit contract name: `org.papernet.commercialpaper`! We see how a **contract name** picks out one contract from the `papercontract.js` chaincode file that contains many contracts. In PaperNet, `papercontract.js` was installed and deployed to the channel with the name `papercontract`, and if you're interested, read [how](#) to deploy a chaincode containing multiple smart contracts.

If our application simultaneously required access to another contract in PaperNet or BondNet this would be easy:

```
const euroContract = await network.getContract('EuroCommercialPaperContract');  
  
const bondContract = await network2.getContract('BondContract');
```

In these examples, note how we didn't use a qualifying contract name – we have only one smart contract per file, and `getContract()` will use the first contract it finds.

Recall the transaction MagnetoCorp uses to issue its first commercial paper:

```
Txn = issue  
Issuer = MagnetoCorp  
Paper = 00001  
Issue time = 31 May 2020 09:00:00 EST  
Maturity date = 30 November 2020  
Face value = 5M USD
```

Let's now submit this transaction to PaperNet!

6.5.6 Submit transaction

Submitting a transaction is a single method call to the SDK:

```
const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001  
↪', '2020-05-31', '2020-11-30', '5000000');
```

See how the `submitTransaction()` parameters match those of the transaction request. It's these values that will be passed to the `issue()` method in the smart contract, and used to create a new commercial paper. Recall its signature:

```
async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {...  
  ↪ }
```

It might appear that a smart contract receives control shortly after the application issues `submitTransaction()`, but that's not the case. Under the covers, the SDK uses the `connectionOptions` and `connectionProfile` details to send the transaction proposal to the right peers in the network, where it can get the required endorsements. But the application doesn't need to worry about any of this – it just issues `submitTransaction` and the SDK takes care of it all!

Note that the `submitTransaction` API includes a process for listening for transaction commits. Listening for commits is required because without it, you will not know whether your transaction has successfully been ordered, validated, and committed to the ledger.

Let's now turn our attention to how the application handles the response!

6.5.7 Process response

Recall from `papercontract.js` how the **issue** transaction returns a commercial paper response:

```
return paper.toBuffer();
```

You'll notice a slight quirk – the new `paper` needs to be converted to a buffer before it is returned to the application. Notice how `issue.js` uses the class method `CommercialPaper.fromBuffer()` to rehydrate the response buffer as a commercial paper:

```
let paper = CommercialPaper.fromBuffer(issueResponse);
```

This allows `paper` to be used in a natural way in a descriptive completion message:

```
console.log(`${paper.issuer} commercial paper : ${paper.paperNumber} successfully_  
  ↪ issued for value ${paper.faceValue}`);
```

See how the same `paper` class has been used in both the application and smart contract – if you structure your code like this, it'll really help readability and reuse.

As with the transaction proposal, it might appear that the application receives control soon after the smart contract completes, but that's not the case. Under the covers, the SDK manages the entire consensus process, and notifies the application when it is complete according to the `strategy` `connectionOption`. If you're interested in what the SDK does under the covers, read the detailed [transaction flow](#).

That's it! In this topic you've understood how to call a smart contract from a sample application by examining how MagnetoCorp's application issues a new commercial paper in PaperNet. Now examine the key ledger and smart contract data structures are designed by in the [architecture topic](#) behind them.

6.6 Application design elements

This section elaborates the key features for client application and smart contract development found in Hyperledger Fabric. A solid understanding of the features will help you design and implement efficient and effective solutions.

6.6.1 Contract names

Audience: Architects, application and smart contract developers, administrators

A chaincode is a generic container for deploying code to a Hyperledger Fabric blockchain network. One or more related smart contracts are defined within a chaincode. Every smart contract has a name that uniquely identifies it within a chaincode. Applications access a particular smart contract within a chaincode using its contract name.

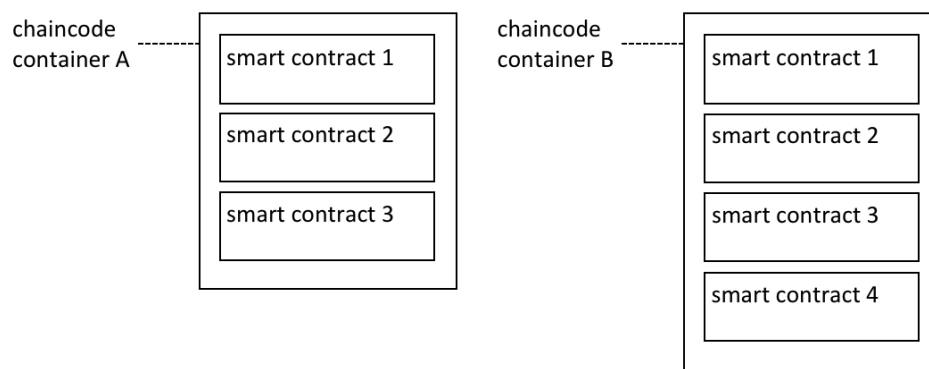
In this topic, we're going to cover:

- *How a chaincode contains multiple smart contracts*
- *How to assign a smart contract name*
- *How to use a smart contract from an application*
- *The default smart contract*

Chaincode

In the [Developing Applications](#) topic, we can see how the Fabric SDKs provide high level programming abstractions which help application and smart contract developers to focus on their business problem, rather than the low level details of how to interact with a Fabric network.

Smart contracts are one example of a high level programming abstraction, and it is possible to define smart contracts within in a chaincode container. When a chaincode is installed on your peer and deployed to a channel, all the smart contracts within it are made available to your applications.



Multiple smart contracts can be defined within a chaincode. Each is uniquely identified by their name within a chaincode.

In the diagram [above](#), chaincode A has three smart contracts defined within it, whereas chaincode B has four smart contracts. See how the chaincode name is used to fully qualify a particular smart contract.

The ledger structure is defined by a set of deployed smart contracts. That's because the ledger contains facts about the business objects of interest to the network (such as commercial paper within PaperNet), and these business objects are moved through their lifecycle (e.g. issue, buy, redeem) by the transaction functions defined within a smart contract.

In most cases, a chaincode will only have one smart contract defined within it. However, it can make sense to keep related smart contracts together in a single chaincode. For example, commercial papers denominated in different currencies might have contracts `EuroPaperContract`, `DollarPaperContract`, `YenPaperContract` which might need to be kept synchronized with each other in the channel to which they are deployed.

Name

Each smart contract within a chaincode is uniquely identified by its contract name. A smart contract can explicitly assign this name when the class is constructed, or let the `Contract` class implicitly assign a default name.

Examine the `papercontract.js` chaincode file:

```
class CommercialPaperContract extends Contract {
  constructor() {
    // Unique name when multiple contracts per chaincode file
    super('org.papernet.commercialpaper');
  }
}
```

See how the `CommercialPaperContract` constructor specifies the contract name as `org.papernet.commercialpaper`. The result is that within the `papercontract` chaincode, this smart contract is now associated with the contract name `org.papernet.commercialpaper`.

If an explicit contract name is not specified, then a default name is assigned – the name of the class. In our example, the default contract name would be `CommercialPaperContract`.

Choose your names carefully. It's not just that each smart contract must have a unique name; a well-chosen name is illuminating. Specifically, using an explicit DNS-style naming convention is recommended to help organize clear and meaningful names; `org.papernet.commercialpaper` conveys that the PaperNet network has defined a standard commercial paper smart contract.

Contract names are also helpful to disambiguate different smart contract transaction functions with the same name in a given chaincode. This happens when smart contracts are closely related; their transaction names will tend to be the same. We can see that a transaction is uniquely defined within a channel by the combination of its chaincode and smart contract name.

Contract names must be unique within a chaincode file. Some code editors will detect multiple definitions of the same class name before deployment. Regardless the chaincode will return an error if multiple classes with the same contract name are explicitly or implicitly specified.

Application

Once a chaincode has been installed on a peer and deployed to a channel, the smart contracts in it are accessible to an application:

```
const network = await gateway.getNetwork('papernet');

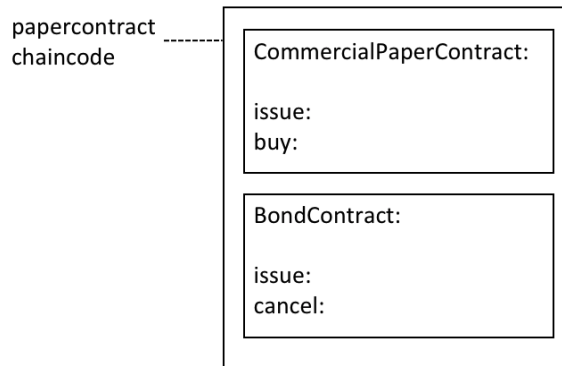
const contract = await network.getContract('papercontract', 'org.papernet.
↪commercialpaper');

const issueResponse = await contract.submitTransaction('issue', 'MagnettoCorp', '00001
↪', '2020-05-31', '2020-11-30', '5000000');
```

See how the application accesses the smart contract with the `network.getContract()` method. The `papercontract` chaincode name `org.papernet.commercialpaper` returns a contract reference which can be used to submit transactions to issue commercial paper with the `contract.submitTransaction()` API.

Default contract

The first smart contract defined in a chaincode is called the *default* smart contract. A default is helpful because a chaincode will usually have one smart contract defined within it; a default allows the application to access those transactions directly – without specifying a contract name.



A default smart contract is the first contract defined in a chaincode.

In this diagram, `CommercialPaperContract` is the default smart contract. Even though we have two smart contracts, the default smart contract makes our *previous* example easier to write:

```
const network = await gateway.getNetwork(`papernet`);

const contract = await network.getContract('papercontract');

const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001', '2020-05-31', '2020-11-30', '5000000');
```

This works because the default smart contract in `papercontract` is `CommercialPaperContract` and it has an `issue` transaction. Note that the `issue` transaction in `BondContract` can only be invoked by explicitly addressing it. Likewise, even though the `cancel` transaction is unique, because `BondContract` is *not* the default smart contract, it must also be explicitly addressed.

In most cases, a chaincode will only contain a single smart contract, so careful naming of the chaincode can reduce the need for developers to care about chaincode as a concept. In the example code *above* it feels like `papercontract` is a smart contract.

In summary, contract names are a straightforward mechanism to identify individual smart contracts within a given chaincode. Contract names make it easy for applications to find a particular smart contract and use it to access the ledger.

6.6.2 Chaincode namespace

Audience: Architects, application and smart contract developers, administrators

A chaincode namespace allows it to keep its world state separate from other chaincodes. Specifically, smart contracts in the same chaincode share direct access to the same world state, whereas smart contracts in different chaincodes cannot directly access each other's world state. If a smart contract needs to access another chaincode world state, it can do this by performing a chaincode-to-chaincode invocation. Finally, a blockchain can contain transactions which relate to different world states.

In this topic, we're going to cover:

- *The importance of namespaces*
- *What is a chaincode namespace*
- *Channels and namespaces*
- *How to use chaincode namespaces*

- *How to access world states across smart contracts*
- *Design considerations for chaincode namespaces*

Motivation

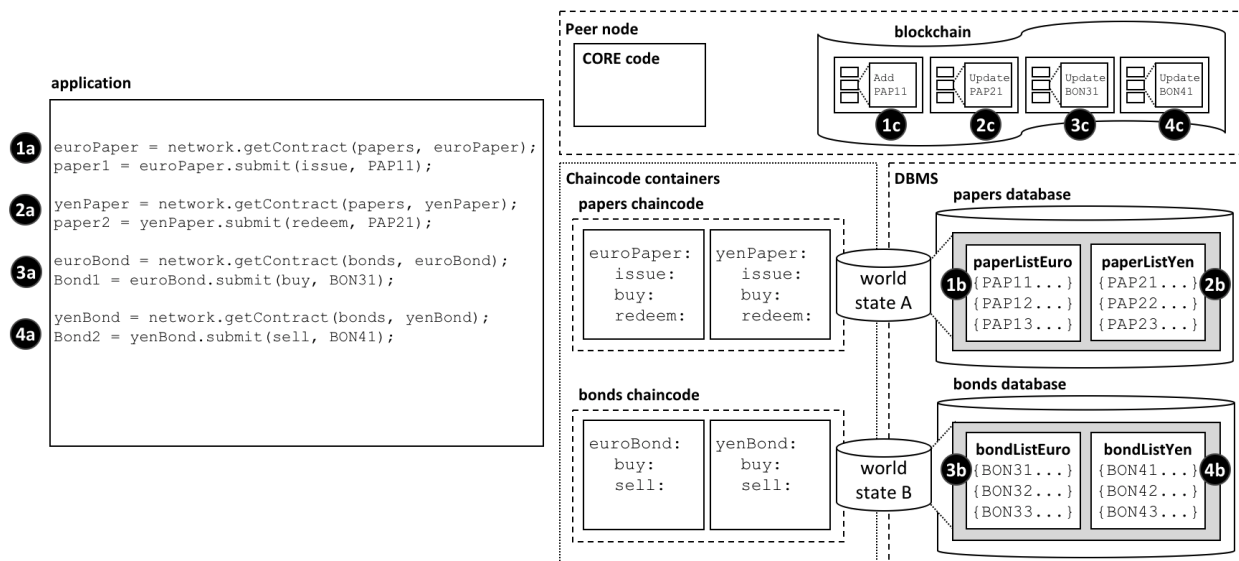
A namespace is a common concept. We understand that *Park Street, New York* and *Park Street, Seattle* are different streets even though they have the same name. The city forms a **namespace** for Park Street, simultaneously providing freedom and clarity.

It's the same in a computer system. Namespaces allow different users to program and operate different parts of a shared system, without getting in each other's way. Many programming languages have namespaces so that programs can freely assign unique identifiers, such as variable names, without worrying about other programs doing the same. We'll see that Hyperledger Fabric uses namespaces to help smart contracts keep their ledger world state separate from other smart contracts.

Scenario

Let's examine how the ledger world state organizes facts about business objects that are important to the organizations in a channel using the diagram below. Whether these objects are commercial papers, bonds, or vehicle registrations, and wherever they are in their lifecycle, they are maintained as states within the ledger world state database. A smart contract manages these business objects by interacting with the ledger (world state and blockchain), and in most cases this will involve it querying or updating the ledger world state.

It's vitally important to understand that the ledger world state is partitioned according to the chaincode of the smart contract that accesses it, and this partitioning, or *namespacing* is an important design consideration for architects, administrators and programmers.



The ledger world state is separated into different namespaces according to the chaincode that accesses it. Within a given channel, smart contracts in the same chaincode share the same world state, and smart contracts in different chaincodes cannot directly access each other's world state. Likewise, a blockchain can contain transactions that relate to different chaincode world states.

In our example, we can see four smart contracts defined in two different chaincodes, each of which is in their own chaincode container. The `euroPaper` and `yenPaper` smart contracts are defined in the `papers` chaincode. The situation is similar for the `euroBond` and `yenBond` smart contracts – they are defined in the `bonds` chaincode. This design helps application programmers understand whether they are working with commercial papers or bonds priced

in Euros or Yen, and because the rules for each financial product don't really change for different currencies, it makes sense to manage their deployment in the same chaincode.

The *diagram* also shows the consequences of this deployment choice. The database management system (DBMS) creates different world state databases for the `papers` and `bonds` chaincodes and the smart contracts contained within them. World state A and world state B are each held within distinct databases; the data are isolated from each other such that a single world state query (for example) cannot access both world states. The world state is said to be *namespaced* according to its chaincode.

See how world state A contains two lists of commercial papers `paperListEuro` and `paperListYen`. The states `PAP11` and `PAP21` are instances of each paper managed by the `euroPaper` and `yenPaper` smart contracts respectively. Because they share the same chaincode namespace, their keys (`PAPxyz`) must be unique within the namespace of the `papers` chaincode, a little like a street name is unique within a town. Notice how it would be possible to write a smart contract in the `papers` chaincode that performed an aggregate calculation over all the commercial papers – whether priced in Euros or Yen – because they share the same namespace. The situation is similar for bonds – they are held within world state B which maps to a separate `bonds` database, and their keys must be unique.

Just as importantly, namespaces mean that `euroPaper` and `yenPaper` cannot directly access world state B, and that `euroBond` and `yenBond` cannot directly access world state A. This isolation is helpful, as commercial papers and bonds are very distinct financial instruments; they have different attributes and are subject to different rules. It also means that `papers` and `bonds` could have the same keys, because they are in different namespaces. This is helpful; it provides a significant degree of freedom for naming. Use this freedom to name different business objects meaningfully.

Most importantly, we can see that a blockchain is associated with the peer operating in a particular channel, and that it contains transactions that affect both world state A and world state B. That's because the blockchain is the most fundamental data structure contained in a peer. The set of world states can always be recreated from this blockchain, because they are the cumulative results of the blockchain's transactions. A world state helps simplify smart contracts and improve their efficiency, as they usually only require the current value of a state. Keeping world states separate via namespaces helps smart contracts isolate their logic from other smart contracts, rather than having to worry about transactions that correspond to different world states. For example, a `bonds` contract does not need to worry about `paper` transactions, because it cannot see their resultant world state.

It's also worth noticing that the peer, chaincode containers and DBMS all are logically different processes. The peer and all its chaincode containers are always in physically separate operating system processes, but the DBMS can be configured to be embedded or separate, depending on its *type*. For LevelDB, the DBMS is wholly contained within the peer, but for CouchDB, it is a separate operating system process.

It's important to remember that namespace choices in this example are the result of a business requirement to share commercial papers in different currencies but isolate them separate from bonds. Think about how the namespace structure would be modified to meet a business requirement to keep every financial asset class separate, or share all commercial papers and bonds?

Channels

If a peer is joined to multiple channels, then a new blockchain is created and managed for each channel. Moreover, every time a chaincode is deployed to a new channel, a new world state database is created for it. It means that the channel also forms a kind of namespace alongside that of the chaincode for the world state.

However, the same peer and chaincode container processes can be simultaneously joined to multiple channels – unlike blockchains, and world state databases, these processes do not increase with the number of channels joined.

For example, if you deployed the `papers` and `bonds` chaincode to a new channel, there would a totally separate blockchain created, and two new world state databases created. However, the peer and chaincode containers would not increase; each would just be connected to multiple channels.

Usage

Let's use our commercial paper *example* to show how an application uses a smart contract with namespaces. It's worth noting that an application communicates with the peer, and the peer routes the request to the appropriate chaincode container which then accesses the DBMS. This routing is done by the peer **core** component shown in the diagram.

Here's the code for an application that uses both commercial papers and bonds, priced in Euros and Yen. The code is fairly self-explanatory:

```
const euroPaper = network.getContract(papers, euroPaper);
paper1 = euroPaper.submit(issue, PAP11);

const yenPaper = network.getContract(papers, yenPaper);
paper2 = yenPaper.submit(redeem, PAP21);

const euroBond = network.getContract(bonds, euroBond);
bond1 = euroBond.submit(buy, BON31);

const yenBond = network.getContract(bonds, yenBond);
bond2 = yenBond.submit(sell, BON41);
```

See how the application:

- Accesses the `euroPaper` and `yenPaper` contracts using the `getContract()` API specifying the papers chaincode. See interaction points **1a** and **2a**.
- Accesses the `euroBond` and `yenBond` contracts using the `getContract()` API specifying the bonds chaincode. See interaction points **3a** and **4a**.
- Submits an `issue` transaction to the network for commercial paper `PAP11` using the `euroPaper` contract. See interaction point **1a**. This results in the creation of a commercial paper represented by state `PAP11` in world state `A`; interaction point **1b**. This operation is captured as a transaction in the blockchain at interaction point **1c**.
- Submits a `redeem` transaction to the network for commercial paper `PAP21` using the `yenPaper` contract. See interaction point **2a**. This results in the redemption of a commercial paper represented by state `PAP21` in world state `A`; interaction point **2b**. This operation is captured as a transaction in the blockchain at interaction point **2c**.
- Submits a `buy` transaction to the network for bond `BON31` using the `euroBond` contract. See interaction point **3a**. This results in the update of a bond represented by state `BON31` in world state `B`; interaction point **3b**. This operation is captured as a transaction in the blockchain at interaction point **3c**.
- Submits a `sell` transaction to the network for bond `BON41` using the `yenBond` contract. See interaction point **4a**. This results in the update of a bond represented by state `BON41` in world state `B`; interaction point **4b**. This operation is captured as a transaction in the blockchain at interaction point **4c**.

See how smart contracts interact with the world state:

- `euroPaper` and `yenPaper` contracts can directly access world state `A`, but cannot directly access world state `B`. World state `A` is physically held in the `papers` database in the database management system (DBMS) corresponding to the papers chaincode.
- `euroBond` and `yenBond` contracts can directly access world state `B`, but cannot directly access world state `A`. World state `B` is physically held in the `bonds` database in the database management system (DBMS) corresponding to the bonds chaincode.

See how the blockchain captures transactions for all world states:

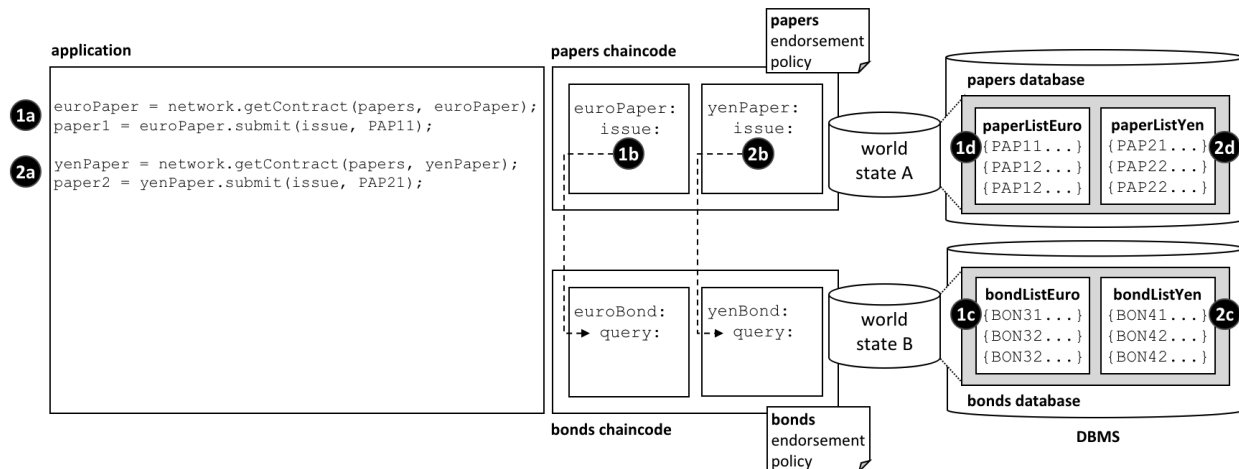
- Interactions **1c** and **2c** correspond to transactions create and update commercial papers `PAP11` and `PAP21` respectively. These are both contained within world state `A`.

- Interactions **3c** and **4c** correspond to transactions both update bonds BON31 and BON41. These are both contained within world state B.
- If world state A or world state B were destroyed for any reason, they could be recreated by replaying all the transactions in the blockchain.

Cross chaincode access

As we saw in our example *scenario*, euroPaper and yenPaper cannot directly access world state B. That's because we have designed our chaincodes and smart contracts so that these chaincodes and world states are kept separately from each other. However, let's imagine that euroPaper needs to access world state B.

Why might this happen? Imagine that when a commercial paper was issued, the smart contract wanted to price the paper according to the current return on bonds with a similar maturity date. In this case it will be necessary for the euroPaper contract to be able to query the price of bonds in world state B. Look at the following diagram to see how we might structure this interaction.



How chaincodes and smart contracts can indirectly access another world state – via its chaincode.

Notice how:

- the application submits an `issue` transaction in the `euroPaper` smart contract to issue `PAP11`. See interaction **1a**.
- the `issue` transaction in the `euroPaper` smart contract calls the `query` transaction in the `euroBond` smart contract. See interaction point **1b**.
- the `query` in `euroBond` can retrieve information from world state B. See interaction point **1c**.
- when control returns to the `issue` transaction, it can use the information in the response to price the paper and update world state A with information. See interaction point **1d**.
- the flow of control for issuing commercial paper priced in Yen is the same. See interaction points **2a**, **2b**, **2c** and **2d**.

Control is passed between chaincode using the `invokeChaincode()` [API](#).

This API passes control from one chaincode to another chaincode.

Although we have only discussed query transactions in the example, it is possible to invoke a smart contract which will update the called chaincode's world state. See the *considerations* below.

Considerations

- In general, each chaincode will have a single smart contract in it.
- Multiple smart contracts should only be deployed in the same chaincode if they are very closely related. Usually, this is only necessary if they share the same world state.
- Chaincode namespaces provide isolation between different world states. In general it makes sense to isolate unrelated data from each other. Note that you cannot choose the chaincode namespace; it is assigned by Hyperledger Fabric, and maps directly to the name of the chaincode.
- For chaincode to chaincode interactions using the `invokeChaincode()` API, both chaincodes must be installed on the same peer.
 - For interactions that only require the called chaincode's world state to be queried, the invocation can be in a different channel to the caller's chaincode.
 - For interactions that require the called chaincode's world state to be updated, the invocation must be in the same channel as the caller's chaincode.

6.6.3 Transaction context

Audience: Architects, application and smart contract developers

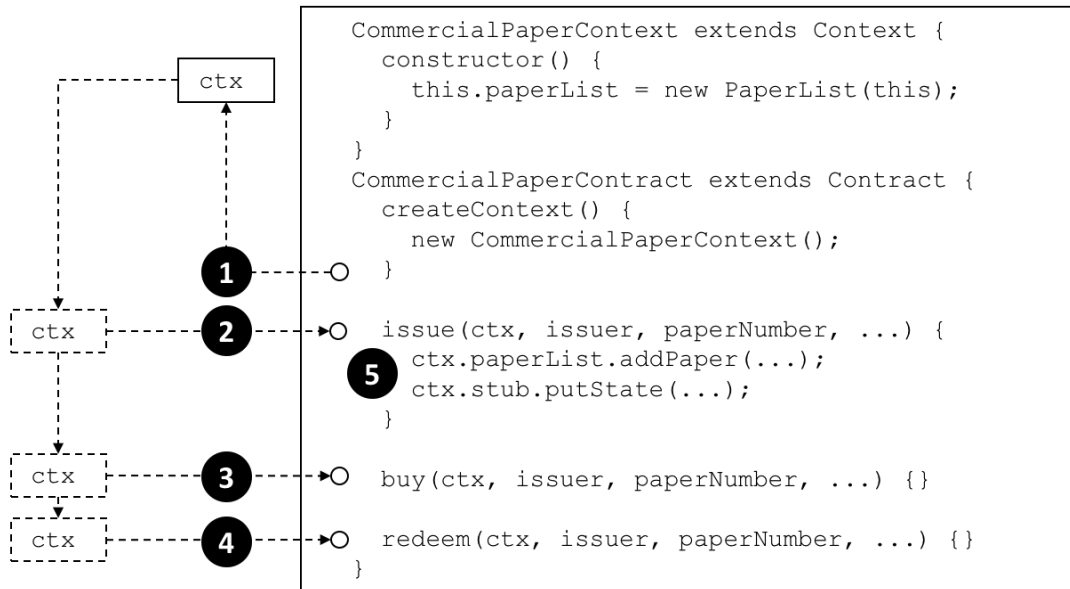
A transaction context performs two functions. Firstly, it allows a developer to define and maintain user variables across transaction invocations within a smart contract. Secondly, it provides access to a wide range of Fabric APIs that allow smart contract developers to perform operations relating to detailed transaction processing. These range from querying or updating the ledger, both the immutable blockchain and the modifiable world state, to retrieving the transaction-submitting application's digital identity.

A transaction context is created when a smart contract is deployed to a channel and made available to every subsequent transaction invocation. A transaction context helps smart contract developers write programs that are powerful, efficient and easy to reason about.

- *Why a transaction context is important*
- *How to use a transaction context*
- *What's in a transaction context*
- *Using a context stub*
- *Using a context `clientIdentity`*

Scenario

In the commercial paper sample, `papercontract` initially defines the name of the list of commercial papers for which it's responsible. Each transaction subsequently refers to this list; the issue transaction adds new papers to it, the buy transaction changes its owner, and the redeem transaction marks it as complete. This is a common pattern; when writing a smart contract it's often helpful to initialize and recall particular variables in sequential transactions.



A smart contract transaction context allows smart contracts to define and maintain user variables across transaction invocations. Refer to the text for a detailed explanation.

Programming

When a smart contract is constructed, a developer can optionally override the built-in `Context` class `createContext` method to create a custom context:

```
createContext() {
  new CommercialPaperContext();
}
```

In our example, the `CommercialPaperContext` is specialized for `CommercialPaperContract`. See how the custom context, addressed through `this`, adds the specific variable `PaperList` to itself:

```
CommercialPaperContext extends Context {
  constructor () {
    this.paperList = new PaperList(this);
  }
}
```

When the `createContext()` method returns at point (1) in the diagram [above](#), a custom context `ctx` has been created which contains `paperList` as one of its variables.

Subsequently, whenever a smart contract transaction such as `issue`, `buy` or `redeem` is called, this context will be passed to it. See how at points (2), (3) and (4) the same commercial paper context is passed into the transaction method using the `ctx` variable.

See how the context is then used at point (5):

```
ctx.paperList.addPaper(...);
ctx.stub.putState(...);
```

Notice how `paperList` created in `CommercialPaperContext` is available to the `issue` transaction. See how `paperList` is similarly used by the `redeem` and `buy` transactions; `ctx` makes the smart contracts efficient and easy to reason about.

You can also see that there's another element in the context – `ctx.stub` – which was not explicitly added by `CommercialPaperContext`. That's because `stub` and other variables are part of the built-in context. Let's now examine the structure of this built-in context, these implicit variables and how to use them.

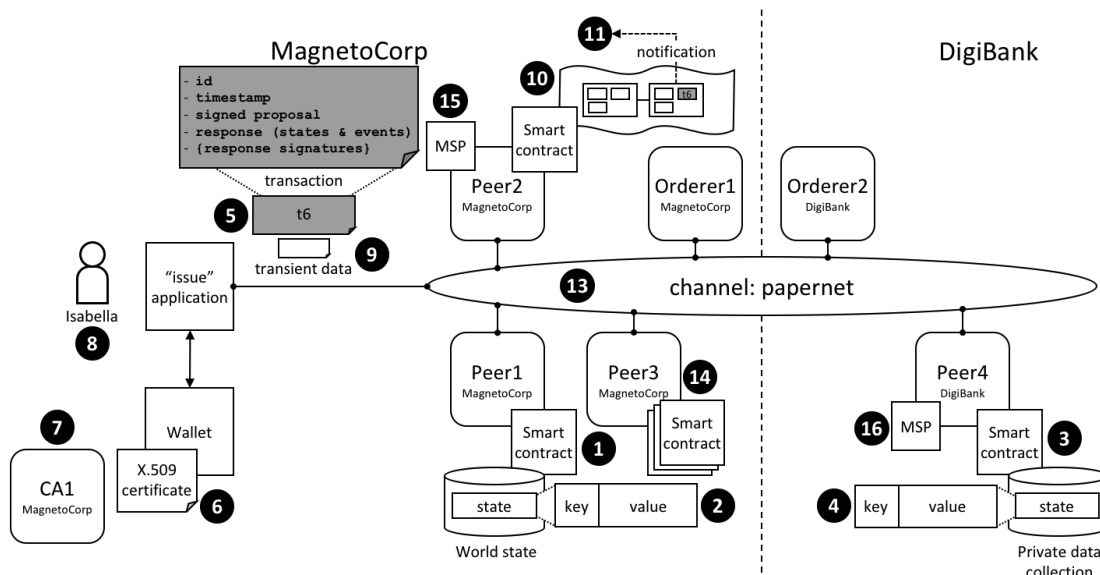
Structure

As we've seen from the [example](#), a transaction context can contain any number of user variables such as `paperList`.

The transaction context also contains two built-in elements that provide access to a wide range of Fabric functionality ranging from the client application that submitted the transaction to ledger access.

- `ctx.stub` is used to access APIs that provide a broad range of transaction processing operations from `putState()` and `getState()` to access the ledger, to `getTxID()` to retrieve the current transaction ID.
- `ctx.clientIdentity` is used to get information about the identity of the user who submitted the transaction.

We'll use the following diagram to show you what a smart contract can do using the `stub` and `clientIdentity` using the APIs available to it:



A smart contract can access a range of functionality in a smart contract via the transaction context `stub` and `clientIdentity`. Refer to the text for a detailed explanation.

Stub

The APIs in the stub fall into the following categories:

- **World state data APIs.** See interaction point (1). These APIs enable smart contracts to get, put and delete state corresponding to individual objects from the world state, using their key:
 - `getState()`
 - `putState()`
 - `deleteState()`

These basic APIs are complemented by query APIs which enable contracts to retrieve a set of states, rather than an individual state. See interaction point (2). The set is either defined by a range of key values, using full or

partial keys, or a query according to values in the underlying world state `database`. For large queries, the result sets can be paginated to reduce storage requirements:

- `getStateByRange()`
- `getStateByRangeWithPagination()`
- `getStateByPartialCompositeKey()`
- `getStateByPartialCompositeKeyWithPagination()`
- `getQueryResult()`
- `getQueryResultWithPagination()`

- **Private data APIs.** See interaction point (3). These APIs enable smart contracts to interact with a private data collection. They are analogous to the APIs for world state interactions, but for private data. There are APIs to get, put and delete a private data state by its key:

- `getPrivateData()`
- `putPrivateData()`
- `deletePrivateData()`

This set is complemented by set of APIs to query private data (4). These APIs allow smart contracts to retrieve a set of states from a private data collection, according to a range of key values, either full or partial keys, or a query according to values in the underlying world state `database`. There are currently no pagination APIs for private data collections.

- `getPrivateDataByRange()`
- `getPrivateDataByPartialCompositeKey()`
- `getPrivateDataQueryResult()`

- **Transaction APIs.** See interaction point (5). These APIs are used by a smart contract to retrieve details about the current transaction proposal being processed by the smart contract. This includes the transaction identifier and the time when the transaction proposal was created.

- `getTxID()` returns the identifier of the current transaction proposal (5).
- `getTxTimestamp()` returns the timestamp when the current transaction proposal was created by the application (5).
- `getCreator()` returns the raw identity (X.509 or otherwise) of the creator of transaction proposal. If this is an X.509 certificate then it is often more appropriate to use `ctx.ClientIdentity`.
- `getSignedProposal()` returns a signed copy of the current transaction proposal being processed by the smart contract.
- `getBinding()` is used to prevent transactions being maliciously or accidentally replayed using a nonce. (For practical purposes, a nonce is a random number generated by the client application and incorporated in a cryptographic hash.) For example, this API could be used by a smart contract at (1) to detect a replay of the transaction (5).
- `getTransient()` allows a smart contract to access the transient data an application passes to a smart contract. See interaction points (9) and (10). Transient data is private to the application-smart contract interaction. It is not recorded on the ledger and is often used in conjunction with private data collections (3).

- **Key APIs** are used by smart contracts to manipulate state key in the world state or a private data collection. See interaction points 2 and 4.

The simplest of these APIs allows smart contracts to form and split composite keys from their individual components. Slightly more advanced are the `ValidationParameter()` APIs which get and set the state based

endorsement policies for world state (2) and private data (4). Finally, `getHistoryForKey()` retrieves the history for a state by returning the set of stored values, including the transaction identifiers that performed the state update, allowing the transactions to be read from the blockchain (10).

- `createCompositeKey()`
- `splitCompositeKey()`
- `setStateValidationParameter()`
- `getStateValidationParameter()`
- `getPrivateDataValidationParameter()`
- `setPrivateDataValidationParameter()`
- `getHistoryForKey()`

- **Event APIs** are used to set an event during the processing of a smart contract.

- `setEvent()`

Smart contracts use this API to add an event to a transaction response. Note that only a single event can be created in a transaction, and must originate from the outer-most contract when contracts invoke each other via `invokeChaincode`. See interaction point (5). These events are ultimately recorded on the blockchain and sent to listening applications at interaction point (11).

- **Utility APIs** are a collection of useful APIs that don't easily fit in a pre-defined category, so we've grouped them together! They include retrieving the current channel name and passing control to a different chaincode on the same peer.

- `getChannelID()`

See interaction point (13). A smart contract running on any peer can use this API to determine on which channel the application invoked the smart contract.

- `invokeChaincode()`

See interaction point (14). Peer3 owned by MagnetoCorp has multiple smart contracts installed on it. These smart contracts are able to call each other using this API. The smart contracts must be collocated; it is not possible to call a smart contract on a different peer.

Some of these utility APIs are only used if you're using low-level chaincode, rather than smart contracts. These APIs are primarily for the detailed manipulation of chaincode input; the smart contract `Contract` class does all of this parameter marshalling automatically for developers.

- `getFunctionAndParameters()`
- `getStringArgs()`
- `getArgs()`

ClientIdentity

In most cases, the application submitting a transaction will be using an X.509 certificate. In the *example*, an X.509 certificate (6) issued by CA1 (7) is being used by Isabella (8) in her application to sign the proposal in transaction `t6` (5).

`ClientIdentity` takes the information returned by `getCreator()` and puts a set of X.509 utility APIs on top of it to make it easier to use for this common use case.

- `getX509Certificate()` returns the full X.509 certificate of the transaction submitter, including all its attributes and their values. See interaction point (6).

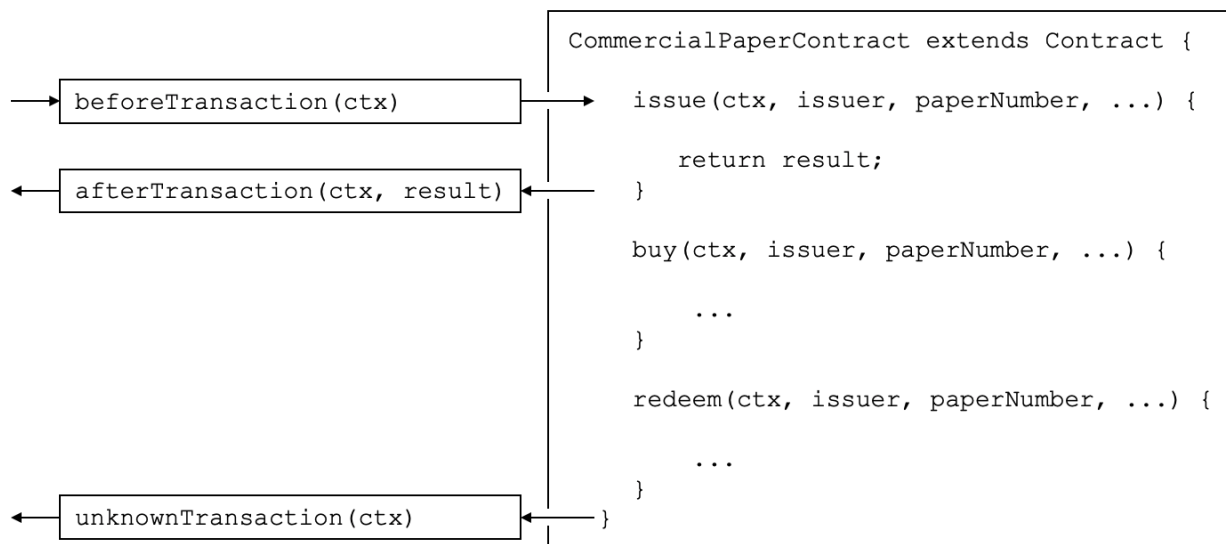
- `getAttributeValue()` returns the value of a particular X.509 attribute, for example, the organizational unit OU, or distinguished name DN. See interaction point (6).
- `assertAttributeValue()` returns TRUE if the specified attribute of the X.509 attribute has a specified value. See interaction point (6).
- `getID()` returns the unique identity of the transaction submitter, according to their distinguished name and the issuing CA's distinguished name. The format is `x509:::{subject DN}:::{issuer DN}`. See interaction point (6).
- `getMSPID()` returns the channel MSP of the transaction submitter. This allows a smart contract to make processing decisions based on the submitter's organizational identity. See interaction point (15) or (16).

6.6.4 Transaction handlers

Audience: Architects, Application and smart contract developers

Transaction handlers allow smart contract developers to define common processing at key points during the interaction between an application and a smart contract. Transaction handlers are optional but, if defined, they will receive control before or after every transaction in a smart contract is invoked. There is also a specific handler which receives control when a request is made to invoke a transaction not defined in a smart contract.

Here's an example of transaction handlers for the [commercial paper smart contract sample](#):



*Before, After and Unknown transaction handlers. In this example, `beforeTransaction()` is called before the **issue**, **buy** and **redeem** transactions. `afterTransaction()` is called after the **issue**, **buy** and **redeem** transactions. `unknownTransaction()` is only called if a request is made to invoke a transaction not defined in the smart contract. (The diagram is simplified by not repeating `beforeTransaction` and `afterTransaction` boxes for each transaction.)*

Types of handler

There are three types of transaction handlers which cover different aspects of the interaction between an application and a smart contract:

- **Before handler:** is called before every smart contract transaction is invoked. The handler will usually modify the transaction context to be used by the transaction. The handler has access to the full range of Fabric APIs; for example, it can issue `getState()` and `putState()`.
- **After handler:** is called after every smart contract transaction is invoked. The handler will usually perform post-processing common to all transactions, and also has full access to the Fabric APIs.
- **Unknown handler:** is called if an attempt is made to invoke a transaction that is not defined in a smart contract. Typically, the handler will record the failure for subsequent processing by an administrator. The handler has full access to the Fabric APIs.

Defining a transaction handler is optional; a smart contract will perform correctly without handlers being defined. A smart contract can define at most one handler of each type.

Defining a handler

Transaction handlers are added to the smart contract as methods with well defined names. Here's an example which adds a handler of each type:

```
CommercialPaperContract extends Contract {

    ...

    async beforeTransaction(ctx) {
        // Write the transaction ID as an informational to the console
        console.info(ctx.stub.getTxID());
    };

    async afterTransaction(ctx, result) {
        // This handler interacts with the ledger
        ctx.stub.cpList.putState(...);
    };

    async unknownTransaction(ctx) {
        // This handler throws an exception
        throw new Error('Unknown transaction function');
    };

}
```

The form of a transaction handler definition is the similar for all handler types, but notice how the `afterTransaction(ctx, result)` also receives any result returned by the transaction. The [API documentation](#) shows you the exact form of these handlers.

Handler processing

Once a handler has been added to the smart contract, it will be invoked during transaction processing. During processing, the handler receives `ctx`, the [transaction context](#), performs some processing, and returns control as it completes. Processing continues as follows:

- **Before handler:** If the handler completes successfully, the transaction is called with the updated context. If the handler throws an exception, then the transaction is not called and the smart contract fails with the exception error message.
- **After handler:** If the handler completes successfully, then the smart contract completes as determined by the invoked transaction. If the handler throws an exception, then the transaction fails with the exception error message.

- **Unknown handler:** The handler should complete by throwing an exception with the required error message. If an **Unknown handler** is not specified, or an exception is not thrown by it, there is sensible default processing; the smart contract will fail with an **unknown transaction** error message.

If the handler requires access to the function and parameters, then it is easy to do this:

```
async beforeTransaction(ctx) {  
    // Retrieve details of the transaction  
    let txnDetails = ctx.stub.getFunctionAndParameters();  
  
    console.info(`Calling function: ${txnDetails.fcn} `);  
    console.info(util.format(`Function arguments : %j ${ctx.stub.getArgs()} `));  
}
```

See how this handler uses the utility API `getFunctionAndParameters` via the [transaction context](#).

Multiple handlers

It is only possible to define at most one handler of each type for a smart contract. If a smart contract needs to invoke multiple functions during before, after or unknown handling, it should coordinate this from within the appropriate function.

6.6.5 Endorsement policies

Audience: Architects, Application and smart contract developers

Endorsement policies define the smallest set of organizations that are required to endorse a transaction in order for it to be valid. To endorse, an organization's endorsing peer needs to run the smart contract associated with the transaction and sign its outcome. When the ordering service sends the transaction to the committing peers, they will each individually check whether the endorsements in the transaction fulfill the endorsement policy. If this is not the case, the transaction is invalidated and it will have no effect on world state.

Endorsement policies work at two different granularities: they can be set for an entire namespace, as well as for individual state keys. They are formulated using basic logic expressions such as AND and OR. For example, in PaperNet this could be used as follows: the endorsement policy for a paper that has been sold from MagnetoCorp to DigiBank could be set to `AND (MagnetoCorp.peer, DigiBank.peer)`, requiring any changes to this paper to be endorsed by both MagnetoCorp and DigiBank.

6.6.6 Connection Profile

Audience: Architects, application and smart contract developers

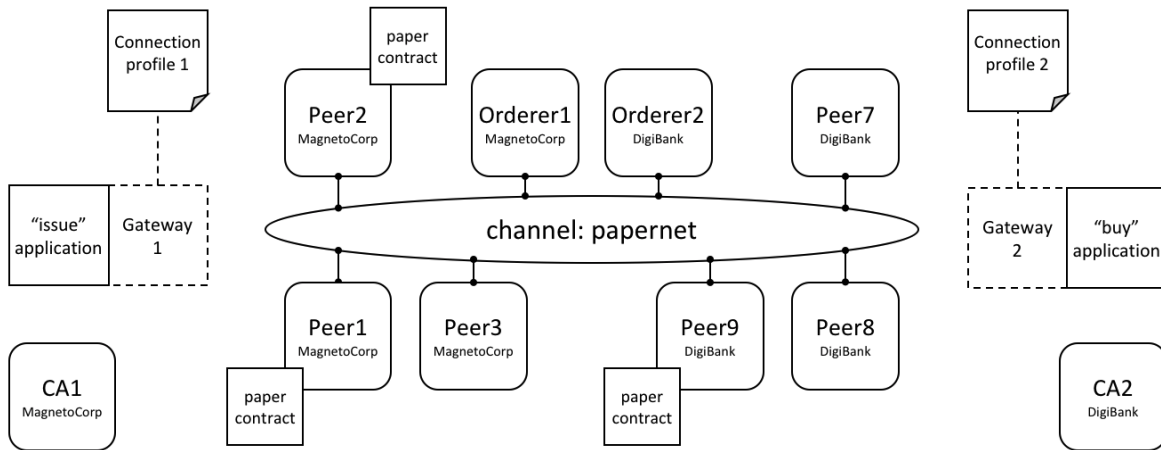
A connection profile describes a set of components, including peers, orderers and certificate authorities in a Hyperledger Fabric blockchain network. It also contains channel and organization information relating to these components. A connection profile is primarily used by an application to configure a [gateway](#) that handles all network interactions, allowing it to focus on business logic. A connection profile is normally created by an administrator who understands the network topology.

In this topic, we're going to cover:

- *Why connection profiles are important*
- *How applications use a connection profile*
- *How to define a connection profile*

Scenario

A connection profile is used to configure a gateway. Gateways are important for [many reasons](#), the primary being to simplify an application's interaction with a network channel.



Two applications, *issue* and *buy*, use gateways 1&2 configured with connection profiles 1&2. Each profile describes a different subset of MagnetoCorp and DigiBank network components. Each connection profile must contain sufficient information for a gateway to interact with the network on behalf of the *issue* and *buy* applications. See the text for a detailed explanation.

A connection profile contains a description of a network view, expressed in a technical syntax, which can either be JSON or YAML. In this topic, we use the YAML representation, as it's easier for you to read. Static gateways need more information than dynamic gateways because the latter can use [service discovery](#) to dynamically augment the information in a connection profile.

A connection profile should not be an exhaustive description of a network channel; it just needs to contain enough information sufficient for a gateway that's using it. In the network above, connection profile 1 needs to contain at least the endorsing organizations and peers for the *issue* transaction, as well as identifying the peers that will notify the gateway when the transaction has been committed to the ledger.

It's easiest to think of a connection profile as describing a *view* of the network. It could be a comprehensive view, but that's unrealistic for a few reasons:

- Peers, orderers, certificate authorities, channels, and organizations are added and removed according to demand.
- Components can start and stop, or fail unexpectedly (e.g. power outage).
- A gateway doesn't need a view of the whole network, only what's necessary to successfully handle transaction submission or event notification for example.
- Service Discovery can augment the information in a connection profile. Specifically, dynamic gateways can be configured with minimal Fabric topology information; the rest can be discovered.

A static connection profile is normally created by an administrator who understands the network topology in detail. That's because a static profile can contain quite a lot of information, and an administrator needs to capture this in the corresponding connection profile. In contrast, dynamic profiles minimize the amount of definition required and therefore can be a better choice for developers who want to get going quickly, or administrators who want to create a more responsive gateway. Connection profiles are created in either the YAML or JSON format using an editor of choice.

Usage

We'll see how to define a connection profile in a moment; let's first see how it is used by a sample MagnetoCorp issue application:

```
const yaml = require('js-yaml');
const { Gateway } = require('fabric-network');

const connectionProfile = yaml.safeLoad(fs.readFileSync('../gateway/paperNet.yaml',
  ↪ 'utf8'));

const gateway = new Gateway();

await gateway.connect(connectionProfile, connectionOptions);
```

After loading some required classes, see how the `paperNet.yaml` gateway file is loaded from the file system, converted to a JSON object using the `yaml.safeLoad()` method, and used to configure a gateway using its `connect()` method.

By configuring a gateway with this connection profile, the issue application is providing the gateway with the relevant network topology it should use to process transactions. That's because the connection profile contains sufficient information about the PaperNet channels, organizations, peers, orderers and CAs to ensure transactions can be successfully processed.

It's good practice for a connection profile to define more than one peer for any given organization – it prevents a single point of failure. This practice also applies to dynamic gateways; to provide more than one starting point for service discovery.

A DigiBank buy application would typically configure its gateway with a similar connection profile, but with some important differences. Some elements will be the same, such as the channel; some elements will overlap, such as the endorsing peers. Other elements will be completely different, such as notification peers or certificate authorities for example.

The `connectionOptions` passed to a gateway complement the connection profile. They allow an application to declare how it would like the gateway to use the connection profile. They are interpreted by the SDK to control interaction patterns with network components, for example to select which identity to connect with, or which peers to use for event notifications. Read [about](#) the list of available connection options and when to use them.

Structure

To help you understand the structure of a connection profile, we're going to step through an example for the network shown [above](#). Its connection profile is based on the PaperNet commercial paper sample, and [stored](#) in the GitHub repository. For convenience, we've reproduced it [below](#). You will find it helpful to display it in another browser window as you now read about it:

- Line 9: `name: "papernet.magnetocorp.profile.sample"`

This is the name of the connection profile. Try to use DNS style names; they are a very easy way to convey meaning.

- Line 16: `x-type: "hlfv1"`

Users can add their own `x-` properties that are “application-specific” – just like with HTTP headers. They are provided primarily for future use.

- Line 20: `description: "Sample connection profile for documentation topic"`

A short description of the connection profile. Try to make this helpful for the reader who might be seeing this for the first time!

- Line 25: `version: "1.0"`

The schema version for this connection profile. Currently only version 1.0 is supported, and it is not envisioned that this schema will change frequently.

- Line 32: `channels:`

This is the first really important line. `channels:` identifies that what follows are *all* the channels that this connection profile describes. However, it is good practice to keep different channels in different connection profiles, especially if they are used independently of each other.

- Line 36: `papernet:`

Details of `papernet`, the first channel in this connection profile, will follow.

- Line 41: `orderers:`

Details of all the orderers for `papernet` follow. You can see in line 45 that the orderer for this channel is `orderer1.magnetocorp.example.com`. This is just a logical name; later in the connection profile (lines 134 - 147), there will be details of how to connect to this orderer. Notice that `orderer2.digibank.example.com` is not in this list; it makes sense that applications use their own organization's orderers, rather than those from a different organization.

- Line 49: `peers:`

Details of all the peers for `papernet` will follow.

You can see three peers listed from MagnetoCorp: `peer1.magnetocorp.example.com`, `peer2.magnetocorp.example.com` and `peer3.magnetocorp.example.com`. It's not necessary to list all the peers in MagnetoCorp, as has been done here. You can see only one peer listed from DigiBank: `peer9.digibank.example.com`; including this peer starts to imply that the endorsement policy requires MagnetoCorp and DigiBank to endorse transactions, as we'll now confirm. It's good practice to have multiple peers to avoid single points of failure.

Underneath each peer you can see four non-exclusive roles: **endorsingPeer**, **chaincodeQuery**, **ledgerQuery** and **eventSource**. See how `peer1` and `peer2` can perform all roles as they host `papercontract`. Contrast to `peer3`, which can only be used for notifications, or ledger queries that access the blockchain component of the ledger rather than the world state, and hence do not need to have smart contracts installed. Notice how `peer9` should not be used for anything other than endorsement, because those roles are better served by MagnetoCorp peers.

Again, see how the peers are described according to their logical names and their roles. Later in the profile, we'll see the physical information for these peers.

- Line 97: `organizations:`

Details of all the organizations will follow, for all channels. Note that these organizations are for all channels, even though `papernet` is currently the only one listed. That's because organizations can be in multiple channels, and channels can have multiple organizations. Moreover, some application operations relate to organizations rather than channels. For example, an application can request notification from one or all peers within its organization, or all organizations within the network – using [connection options](#). For this, there needs to be an organization to peer mapping, and this section provides it.

- Line 101: `MagnetoCorp:`

All peers that are considered part of MagnetoCorp are listed: `peer1`, `peer2` and `peer3`. Likewise for Certificate Authorities. Again, note the logical name usages, the same as the `channels:` section; physical information will follow later in the profile.

- Line 121: `DigiBank:`

Only `peer9` is listed as part of DigiBank, and no Certificate Authorities. That's because these other peers and the DigiBank CA are not relevant for users of this connection profile.

- Line 134: `orderers:`

The physical information for `orderers` is now listed. As this connection profile only mentioned one `orderer` for `papernet`, you see `orderer1.magnetocorp.example.com` details listed. These include its IP address and port, and gRPC options that can override the defaults used when communicating with the `orderer`, if necessary. As with `peers:`, for high availability, specifying more than one `orderer` is a good idea.

- Line 152: `peers:`

The physical information for all previous `peers` is now listed. This connection profile has three `peers` for `MagnetoCorp`: `peer1`, `peer2`, and `peer3`; for `DigiBank`, a single `peer` `peer9` has its information listed. For each `peer`, as with `orderers`, their IP address and port is listed, together with gRPC options that can override the defaults used when communicating with a particular `peer`, if necessary.

- Line 194: `certificateAuthorities:`

The physical information for certificate authorities is now listed. The connection profile has a single CA listed for `MagnetoCorp`, `cal-magnetocorp`, and its physical information follows. As well as IP details, the `registrar` information allows this CA to be used for Certificate Signing Requests (CSR). These are used to request new certificates for locally generated public/private key pairs.

Now you've understood a connection profile for `MagnetoCorp`, you might like to look at a [corresponding](#) profile for `DigiBank`. Locate where the profile is the same as `MagnetoCorp`'s, see where it's similar, and finally where it's different. Think about why these differences make sense for `DigiBank` applications.

That's everything you need to know about connection profiles. In summary, a connection profile defines sufficient channels, organizations, `peers`, `orderers` and certificate authorities for an application to configure a gateway. The gateway allows the application to focus on business logic rather than the details of the network topology.

Sample

This file is reproduced inline from the GitHub commercial paper [sample](#).

```
1: ---
2: #
3: # [Required]. A connection profile contains information about a set of network
4: # components. It is typically used to configure gateway, allowing applications
5: # interact with a network channel without worrying about the underlying
6: # topology. A connection profile is normally created by an administrator who
7: # understands this topology.
8: #
9: name: "papernet.magnetocorp.profile.sample"
10: #
11: # [Optional]. Analogous to HTTP, properties with an "x-" prefix are deemed
12: # "application-specific", and ignored by the gateway. For example, property
13: # "x-type" with value "hlfv1" was originally used to identify a connection
14: # profile for Fabric 1.x rather than 0.x.
15: #
16: x-type: "hlfv1"
17: #
18: # [Required]. A short description of the connection profile
19: #
20: description: "Sample connection profile for documentation topic"
21: #
22: # [Required]. Connection profile schema version. Used by the gateway to
23: # interpret these data.
24: #
25: version: "1.0"
```

(continues on next page)

(continued from previous page)

```

26: #
27: # [Optional]. A logical description of each network channel; its peer and
28: # orderer names and their roles within the channel. The physical details of
29: # these components (e.g. peer IP addresses) will be specified later in the
30: # profile; we focus first on the logical, and then the physical.
31: #
32: channels:
33: #
34: # [Optional]. papernet is the only channel in this connection profile
35: #
36: papernet:
37: #
38: # [Optional]. Channel orderers for PaperNet. Details of how to connect to
39: # them is specified later, under the physical "orderers:" section
40: #
41: orderers:
42: #
43: # [Required]. Orderer logical name
44: #
45: - orderer1.magnetocorp.example.com
46: #
47: # [Optional]. Peers and their roles
48: #
49: peers:
50: #
51: # [Required]. Peer logical name
52: #
53: peer1.magnetocorp.example.com:
54: #
55: # [Optional]. Is this an endorsing peer? (It must have chaincode
56: # installed.) Default: true
57: #
58: endorsingPeer: true
59: #
60: # [Optional]. Is this peer used for query? (It must have chaincode
61: # installed.) Default: true
62: #
63: chaincodeQuery: true
64: #
65: # [Optional]. Is this peer used for non-chaincode queries? All peers
66: # support these types of queries, which include queryBlock(),
67: # queryTransaction(), etc. Default: true
68: #
69: ledgerQuery: true
70: #
71: # [Optional]. Is this peer used as an event hub? All peers can produce
72: # events. Default: true
73: #
74: eventSource: true
75: #
76: peer2.magnetocorp.example.com:
77: endorsingPeer: true
78: chaincodeQuery: true
79: ledgerQuery: true
80: eventSource: true
81: #
82: peer3.magnetocorp.example.com:

```

(continues on next page)

(continued from previous page)

```

83:     endorsingPeer: false
84:     chaincodeQuery: false
85:     ledgerQuery: true
86:     eventSource: true
87:     #
88:     peer9.digibank.example.com:
89:         endorsingPeer: true
90:         chaincodeQuery: false
91:         ledgerQuery: false
92:         eventSource: false
93: #
94: # [Required]. List of organizations for all channels. At least one organization
95: # is required.
96: #
97: organizations:
98:     #
99:     # [Required]. Organizational information for MagnetoCorp
100:    #
101:    MagnetoCorp:
102:        #
103:        # [Required]. The MSPID used to identify MagnetoCorp
104:        #
105:        mspid: MagnetoCorpMSP
106:        #
107:        # [Required]. The MagnetoCorp peers
108:        #
109:        peers:
110:            - peer1.magnetocorp.example.com
111:            - peer2.magnetocorp.example.com
112:            - peer3.magnetocorp.example.com
113:        #
114:        # [Optional]. Fabric-CA Certificate Authorities.
115:        #
116:        certificateAuthorities:
117:            - ca-magnetocorp
118:        #
119:        # [Optional]. Organizational information for DigiBank
120:        #
121:        DigiBank:
122:            #
123:            # [Required]. The MSPID used to identify DigiBank
124:            #
125:            mspid: DigiBankMSP
126:            #
127:            # [Required]. The DigiBank peers
128:            #
129:            peers:
130:                - peer9.digibank.example.com
131:            #
132:            # [Optional]. Orderer physical information, by orderer name
133:            #
134:            orderers:
135:                #
136:                # [Required]. Name of MagnetoCorp orderer
137:                #
138:                orderer1.magnetocorp.example.com:
139:                    #

```

(continues on next page)

(continued from previous page)

```

140:      # [Required]. This orderer's IP address
141:      #
142:      url: grpc://localhost:7050
143:      #
144:      # [Optional]. gRPC connection properties used for communication
145:      #
146:      grpcOptions:
147:          ssl-target-name-override: orderer1.magnetocorp.example.com
148:      #
149:      # [Required]. Peer physical information, by peer name. At least one peer is
150:      # required.
151:      #
152:      peers:
153:          #
154:          # [Required]. First MagetoCorp peer physical properties
155:          #
156:          peer1.magnetocorp.example.com:
157:              #
158:              # [Required]. Peer's IP address
159:              #
160:              url: grpc://localhost:7151
161:              #
162:              # [Optional]. gRPC connection properties used for communication
163:              #
164:              grpcOptions:
165:                  ssl-target-name-override: peer1.magnetocorp.example.com
166:                  request-timeout: 120001
167:              #
168:              # [Optional]. Other MagetoCorp peers
169:              #
170:          peer2.magnetocorp.example.com:
171:              url: grpc://localhost:7251
172:              grpcOptions:
173:                  ssl-target-name-override: peer2.magnetocorp.example.com
174:                  request-timeout: 120001
175:              #
176:          peer3.magnetocorp.example.com:
177:              url: grpc://localhost:7351
178:              grpcOptions:
179:                  ssl-target-name-override: peer3.magnetocorp.example.com
180:                  request-timeout: 120001
181:              #
182:              # [Required]. Digibank peer physical properties
183:              #
184:          peer9.digibank.example.com:
185:              url: grpc://localhost:7951
186:              grpcOptions:
187:                  ssl-target-name-override: peer9.digibank.example.com
188:                  request-timeout: 120001
189:              #
190:          # [Optional]. Fabric-CA Certificate Authority physical information, by name.
191:          # This information can be used to (e.g.) enroll new users. Communication is via
192:          # REST, hence options relate to HTTP rather than gRPC.
193:          #
194:          certificateAuthorities:
195:              #
196:              # [Required]. MagetoCorp CA

```

(continues on next page)

(continued from previous page)

```
197:  #
198:  cal-magnetocorp:
199:  #
200:  # [Required]. CA IP address
201:  #
202:  url: http://localhost:7054
203:  #
204:  # [Optional]. HTTP connection properties used for communication
205:  #
206:  httpOptions:
207:    verify: false
208:  #
209:  # [Optional]. Fabric-CA supports Certificate Signing Requests (CSRs). A
210:  # registrar is needed to enroll new users.
211:  #
212:  registrar:
213:    - enrollId: admin
214:      enrollSecret: adminpw
215:  #
216:  # [Optional]. The name of the CA.
217:  #
218:  caName: ca-magnetocorp
```

6.6.7 Connection Options

Audience: Architects, administrators, application and smart contract developers

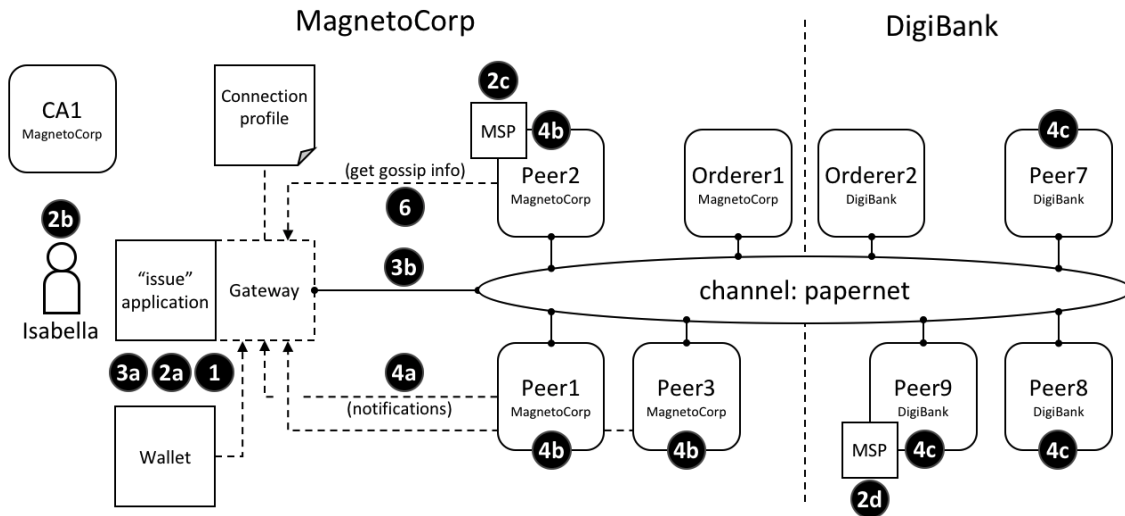
Connection options are used in conjunction with a connection profile to control *precisely* how a gateway interacts with a network. Using a gateway allows an application to focus on business logic rather than network topology.

In this topic, we're going to cover:

- *Why connection options are important*
- *How an application uses connection options*
- *What each connection option does*
- *When to use a particular connection option*

Scenario

A connection option specifies a particular aspect of a gateway's behaviour. Gateways are important for [many reasons](#), the primary being to allow an application to focus on business logic and smart contracts, while it manages interactions with the many components of a network.



The different interaction points where connection options control behaviour. These options are explained fully in the text.

One example of a connection option might be to specify that the gateway used by the `issue` application should use identity `Isabella` to submit transactions to the `papernet` network. Another might be that a gateway should wait for all three nodes from MagnetoCorp to confirm a transaction has been committed returning control. Connection options allow applications to specify the precise behaviour of a gateway's interaction with the network. Without a gateway, applications need to do a lot more work; gateways save you time, make your application more readable, and less error prone.

Usage

We'll describe the *full set* of connection options available to an application in a moment; let's first see how they are specified by the sample MagnetoCorp `issue` application:

```
const userName = 'User1@org1.example.com';
const wallet = new FileSystemWallet('../identity/user/isabella/wallet');

const connectionOptions = {
  identity: userName,
  wallet: wallet,
  eventHandlerOptions: {
    commitTimeout: 100,
    strategy: EventStrategies.MSPID_SCOPE_ANYFORTX
  }
};

await gateway.connect(connectionProfile, connectionOptions);
```

See how the `identity` and `wallet` options are simple properties of the `connectionOptions` object. They have values `userName` and `wallet` respectively, which were set earlier in the code. Contrast these options with the `eventHandlerOptions` option which is an object in its own right. It has two properties: `commitTimeout: 100` (measured in seconds) and `strategy: EventStrategies.MSPID_SCOPE_ANYFORTX`.

See how `connectionOptions` is passed to a gateway as a complement to `connectionProfile`; the network is identified by the connection profile and the options specify precisely how the gateway should interact with it. Let's now look at the available options.

Options

Here's a list of the available options and what they do.

- `wallet` identifies the wallet that will be used by the gateway on behalf of the application. See interaction **1**; the wallet is specified by the application, but it's actually the gateway that retrieves identities from it.

A wallet must be specified; the most important decision is the **type** of wallet to use, whether that's file system, in-memory, HSM or database.

- `identity` is the user identity that the application will use from `wallet`. See interaction **2a**; the user identity is specified by the application and represents the user of the application, Isabella, **2b**. The identity is actually retrieved by the gateway.

In our example, Isabella's identity will be used by different MSPs (**2c**, **2d**) to identify her as being from MagnetoCorp, and having a particular role within it. These two facts will correspondingly determine her permission over resources, such as being able to read and write the ledger, for example.

A user identity must be specified. As you can see, this identity is fundamental to the idea that Hyperledger Fabric is a *permissioned* network – all actors have an identity, including applications, peers and orderers, which determines their control over resources. You can read more about this idea in the membership services [topic](#).

- `clientTlsIdentity` is the identity that is retrieved from a wallet (**3a**) and used for secure communications (**3b**) between the gateway and different channel components, such as peers and orderers.

Note that this identity is different to the user identity. Even though `clientTlsIdentity` is important for secure communications, it is not as foundational as the user identity because its scope does not extend beyond secure network communications.

`clientTlsIdentity` is optional. You are advised to set it in production environments. You should always use a different `clientTlsIdentity` to `identity` because these identities have very different meanings and lifecycles. For example, if your `clientTlsIdentity` was compromised, then so would your `identity`; it's more secure to keep them separate.

- `eventHandlerOptions.commitTimeout` is optional. It specifies, in seconds, the maximum amount of time the gateway should wait for a transaction to be committed by any peer (**4a**) before returning control to the application. The set of peers to use for notification is determined by the `eventHandlerOptions.strategy` option. If a `commitTimeout` is not specified, the gateway will use a timeout of 300 seconds.
- `eventHandlerOptions.strategy` is optional. It identifies the set of peers that a gateway should use to listen for notification that a transaction has been committed. For example, whether to listen for a single peer, or all peers, from its organization. It can take one of the following values:
 - `EventStrategies.MSPID_SCOPE_ANYFORTX` Listen for **any** peer within the user's organization. In our example, see interaction points **4b**; any of peer 1, peer 2 or peer 3 from MagnetoCorp can notify the gateway.
 - `EventStrategies.MSPID_SCOPE_ALLFORTX` **This is the default value.** Listen for **all** peers within the user's organization. In our example peer, see interaction point **4b**. All peers from MagnetoCorp must all have notified the gateway; peer 1, peer 2 and peer 3. Peers are only counted if they are known/discovered and available; peers that are stopped or have failed are not included.
 - `EventStrategies.NETWORK_SCOPE_ANYFORTX` Listen for **any** peer within the entire network channel. In our example, see interaction points **4b** and **4c**; any of peer 1-3 from MagnetoCorp or peer 7-9 of DigiBank can notify the gateway.
 - `EventStrategies.NETWORK_SCOPE_ALLFORTX` Listen for **all** peers within the entire network channel. In our example, see interaction points **4b** and **4c**. All peers from MagnetoCorp and DigiBank must notify the gateway; peers 1-3 and peers 7-9. Peers are only counted if they are known/discovered and available; peers that are stopped or have failed are not included.

- `<PluginEventHandlerFunction>` The name of a user-defined event handler. This allows a user to define their own logic for event handling. See how to [define](#) a plugin event handler, and examine a [sample handler](#).

A user-defined event handler is only necessary if you have very specific event handling requirements; in general, one of the built-in event strategies will be sufficient. An example of a user-defined event handler might be to wait for more than half the peers in an organization to confirm a transaction has been committed.

If you do specify a user-defined event handler, it does not affect your application logic; it is quite separate from it. The handler is called by the SDK during processing; it decides when to call it, and uses its results to select which peers to use for event notification. The application receives control when the SDK has finished its processing.

If a user-defined event handler is not specified then the default values for `EventStrategies` are used.

- `discovery.enabled` is optional and has possible values `true` or `false`. The default is `true`. It determines whether the gateway uses [service discovery](#) to augment the network topology specified in the connection profile. See interaction point 6; peer's gossip information used by the gateway.

This value will be overridden by the `INITIALIZE-WITH-DISCOVERY` environment variable, which can be set to `true` or `false`.

- `discovery.asLocalhost` is optional and has possible values `true` or `false`. The default is `true`. It determines whether IP addresses found during service discovery are translated from the docker network to the local host.

Typically developers will write applications that use docker containers for their network components such as peers, orderers and CAs, but that do not run in docker containers themselves. This is why `true` is the default; in production environments, applications will likely run in docker containers in the same manner as network components and therefore address translation is not required. In this case, applications should either explicitly specify `false` or use the environment variable override.

This value will be overridden by the `DISCOVERY-AS-LOCALHOST` environment variable, which can be set to `true` or `false`.

Considerations

The following list of considerations is helpful when deciding how to choose connection options.

- `eventHandlerOptions.commitTimeout` and `eventHandlerOptions.strategy` work together. For example, `commitTimeout: 100` and `strategy: EventStrategies.MSPID_SCOPE_ANYFORTX` means that the gateway will wait for up to 100 seconds for *any* peer to confirm a transaction has been committed. In contrast, specifying `strategy: EventStrategies.NETWORK_SCOPE_ALLFORTX` means that the gateway will wait up to 100 seconds for *all* peers in *all* organizations.
- The default value of `eventHandlerOptions.strategy: EventStrategies.MSPID_SCOPE_ALLFORTX` will wait for all peers in the application's organization to commit the transaction. This is a good default because applications can be sure that all their peers have an up-to-date copy of the ledger, minimizing concurrency issues

However, as the number of peers in an organization grows, it becomes a little unnecessary to wait for all peers, in which case using a pluggable event handler can provide a more efficient strategy. For example the same set of peers could be used to submit transactions and listen for notifications, on the safe assumption that consensus will keep all ledgers synchronized.

- Service discovery requires `clientTlsIdentity` to be set. That's because the peers exchanging information with an application need to be confident that they are exchanging information with entities they trust. If `clientTlsIdentity` is not set, then `discovery` will not be obeyed, regardless of whether or not it is set.
- Although applications can set connection options when they connect to the gateway, it can be necessary for these options to be overridden by an administrator. That's because options relate to network interactions, which can vary over time. For example, an administrator trying to understand the effect of using service discovery on network performance.

A good approach is to define application overrides in a configuration file which is read by the application when it configures its connection to the gateway.

Because the `discovery` options `enabled` and `asLocalHost` are most frequently required to be overridden by administrators, the environment variables `INITIALIZE-WITH-DISCOVERY` and `DISCOVERY-AS-LOCALHOST` are provided for convenience. The administrator should set these in the production runtime environment of the application, which will most likely be a docker container.

6.6.8 Wallet

Audience: Architects, application and smart contract developers

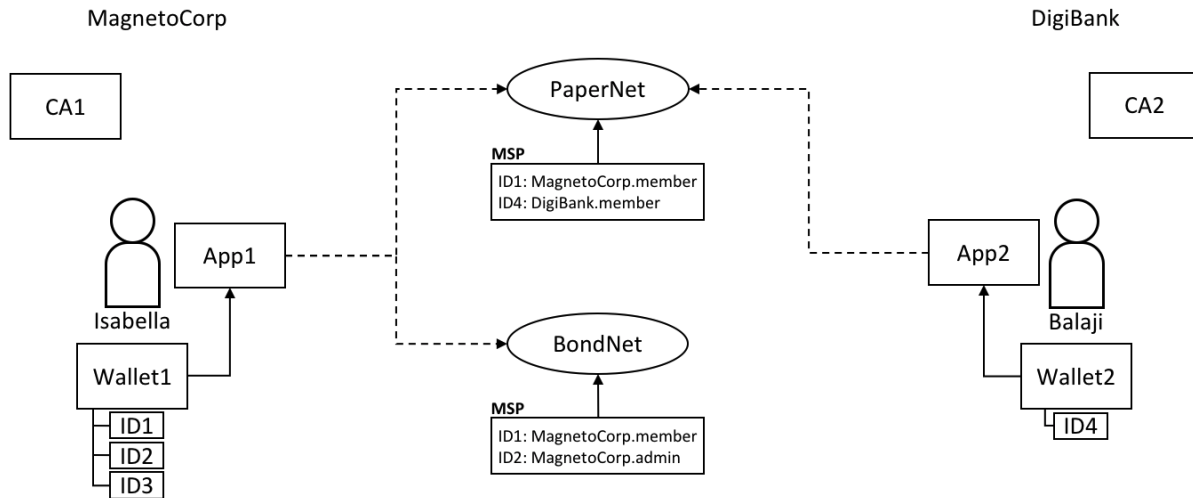
A wallet contains a set of user identities. An application run by a user selects one of these identities when it connects to a channel. Access rights to channel resources, such as the ledger, are determined using this identity in combination with an MSP.

In this topic, we're going to cover:

- *Why wallets are important*
- *How wallets are organized*
- *Different types of wallet*
- *Wallet operations*

Scenario

When an application connects to a network channel such as PaperNet, it selects a user identity to do so, for example `ID1`. The channel MSPs associate `ID1` with a role within a particular organization, and this role will ultimately determine the application's rights over channel resources. For example, `ID1` might identify a user as a member of the MagnetoCorp organization who can read and write to the ledger, whereas `ID2` might identify an administrator in MagnetoCorp who can add a new organization to a consortium.



Two users, Isabella and Balaji have wallets containing different identities they can use to connect to different network channels, PaperNet and BondNet.

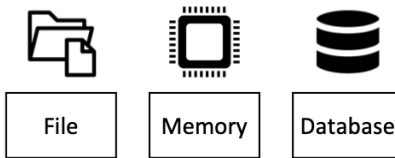
Consider the example of two users; Isabella from MagnetoCorp and Balaji from DigiBank. Isabella is going to use App 1 to invoke a smart contract in PaperNet and a different smart contract in BondNet. Similarly, Balaji is going to use App 2 to invoke smart contracts, but only in PaperNet. (It's very *easy* for applications to access multiple networks and multiple smart contracts within them.)

See how:

- MagnetoCorp uses CA1 to issue identities and DigiBank uses CA2 to issue identities. These identities are stored in user wallets.
- Balaji's wallet holds a single identity, ID4 issued by CA2. Isabella's wallet has many identities, ID1, ID2 and ID3, issued by CA1. Wallets can hold multiple identities for a single user, and each identity can be issued by a different CA.
- Both Isabella and Balaji connect to PaperNet, and its MSPs determine that Isabella is a member of the MagnetoCorp organization, and Balaji is a member of the DigiBank organization, because of the respective CAs that issued their identities. (It is *possible* for an organization to use multiple CAs, and for a single CA to support multiple organizations.)
- Isabella can use ID1 to connect to both PaperNet and BondNet. In both cases, when Isabella uses this identity, she is recognized as a member of MagnetoCorp.
- Isabella can use ID2 to connect to BondNet, in which case she is identified as an administrator of MagnetoCorp. This gives Isabella two very different privileges: ID1 identifies her as a simple member of MagnetoCorp who can read and write to the BondNet ledger, whereas ID2 identifies her as a MagnetoCorp administrator who can add a new organization to BondNet.
- Balaji cannot connect to BondNet with ID4. If he tried to connect, ID4 would not be recognized as belonging to DigiBank because CA2 is not known to BondNet's MSP.

Types

There are different types of wallets according to where they store their identities:



The three different types of wallet storage: File system, In-memory and CouchDB.

- **File system:** This is the most common place to store wallets; file systems are pervasive, easy to understand, and can be network mounted. They are a good default choice for wallets.
- **In-memory:** A wallet in application storage. Use this type of wallet when your application is running in a constrained environment without access to a file system; typically a web browser. It's worth remembering that this type of wallet is volatile; identities will be lost after the application ends normally or crashes.
- **CouchDB:** A wallet stored in CouchDB. This is the rarest form of wallet storage, but for those users who want to use the database back-up and restore mechanisms, CouchDB wallets can provide a useful option to simplify disaster recovery.

Use factory functions provided by the `Wallets` class to create wallets.

Hardware Security Module

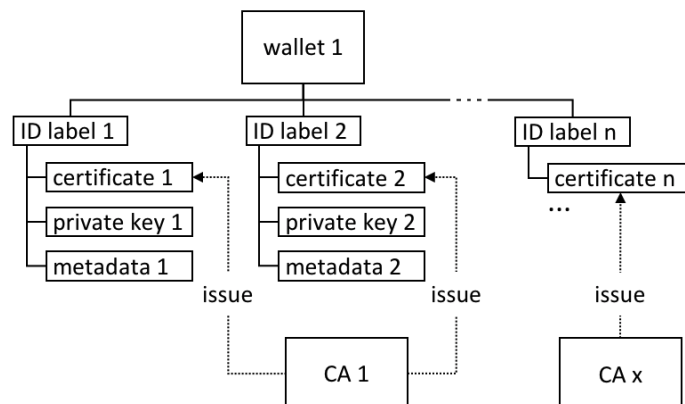
A Hardware Security Module (HSM) is an ultra-secure, tamper-proof device that stores digital identity information, particularly private keys. HSMs can be locally attached to your computer or network accessible. Most HSMs provide the ability to perform on-board encryption with private keys, such that the private keys never leave the HSM.

An HSM can be used with any of the wallet types. In this case the certificate for an identity will be stored in the wallet and the private key will be stored in the HSM.

To enable the use of HSM-managed identities, an `IdentityProvider` must be configured with the HSM connection information and registered with the wallet. For further details, refer to the [Using wallets to manage identities](#) tutorial.

Structure

A single wallet can hold multiple identities, each issued by a particular Certificate Authority. Each identity has a standard structure comprising a descriptive label, an X.509 certificate containing a public key, a private key, and some Fabric-specific metadata. Different *wallet types* map this structure appropriately to their storage mechanism.



A Fabric wallet can hold multiple identities with certificates issued by a different Certificate Authority. Identities comprise certificate, private key and Fabric metadata.

There's a couple of key class methods that make it easy to manage wallets and identities:

```
const identity: X509Identity = {
  credentials: {
    certificate: certificatePEM,
    privateKey: privateKeyPEM,
  },
  mspId: 'Org1MSP',
  type: 'X.509',
};
await wallet.put(identityLabel, identity);
```

See how an identity is created that has metadata Org1MSP, a certificate and a privateKey. See how `wallet.put()` adds this identity to the wallet with a particular `identityLabel`.

The Gateway class only requires the `mspId` and `type` metadata to be set for an identity – Org1MSP and X.509 in the above example. It *currently* uses the MSP ID value to identify particular peers from a [connection profile](#), for example when a specific notification [strategy](#) is requested. In the DigiBank gateway file `networkConnection.yaml`, see how Org1MSP notifications will be associated with `peer0.org1.example.com`:

```
organizations:
  Org1:
    mspid: Org1MSP

  peers:
    - peer0.org1.example.com
```

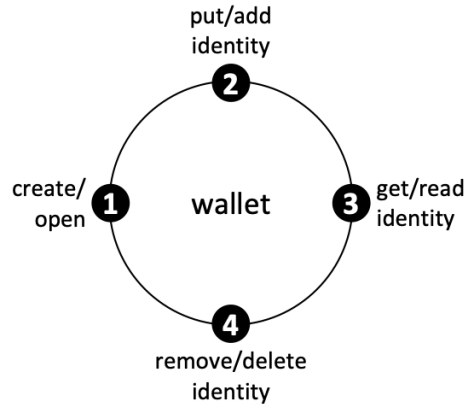
You really don't need to worry about the internal structure of the different wallet types, but if you're interested, navigate to a user identity folder in the commercial paper sample:

```
magnetocorp/identity/user/isabella/
                                wallet/
                                User1@org1.example.com.id
```

You can examine these files, but as discussed, it's easier to use the SDK to manipulate these data.

Operations

The different wallet types all implement a common [Wallet](#) interface which provides a standard set of APIs to manage identities. It means that applications can be made independent of the underlying wallet storage mechanism; for example, File system and HSM wallets are handled in a very similar way.



Wallets follow a lifecycle: they can be created or opened, and identities can be read, added and deleted.

An application can use a wallet according to a simple lifecycle. Wallets can be opened or created, and subsequently identities can be added, updated, read and deleted. Spend a little time on the different `Wallet` methods in the [JSDoc](#) to see how they work; the commercial paper tutorial provides a nice example in `addToWallet.js`:

```
const wallet = await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');

const cert = fs.readFileSync(path.join(credPath, '../User1@org1.example.com-cert.pem'), 'utf8');
const key = fs.readFileSync(path.join(credPath, '../_sk'), 'utf8');

const identityLabel = 'User1@org1.example.com';
const identity = {
  credentials: {
    certificate: cert,
    privateKey: key,
  },
  mspId: 'Org1MSP',
  type: 'X.509',
};

await wallet.put(identityLabel, identity);
```

Notice how:

- When the program is first run, a wallet is created on the local file system at `../isabella/wallet`.
- a certificate `cert` and private key are loaded from the file system.
- a new X.509 identity is created with `cert`, `key` and `Org1MSP`.
- the new identity is added to the wallet with `wallet.put()` with a label `User1@org1.example.com`.

That's everything you need to know about wallets. You've seen how they hold identities that are used by applications on behalf of users to access Fabric network resources. There are different types of wallets available depending on your application and security needs, and a simple set of APIs to help applications manage wallets and the identities within them.

6.6.9 Gateway

Audience: Architects, application and smart contract developers

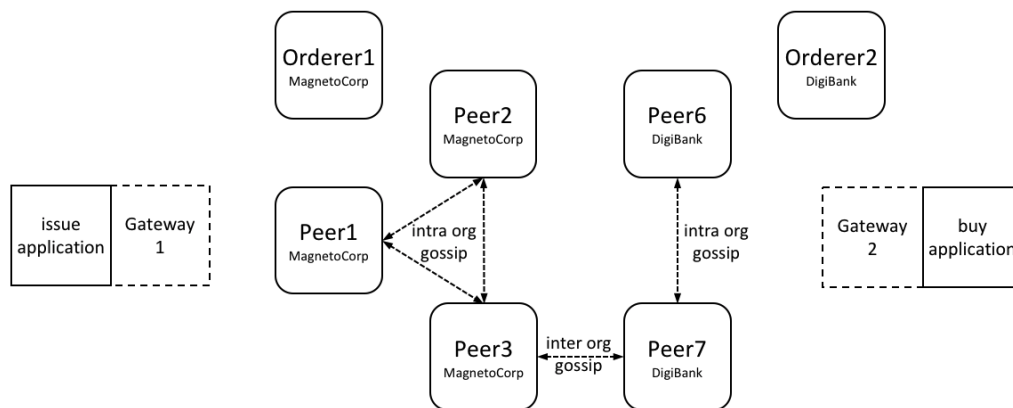
A gateway manages the network interactions on behalf of an application, allowing it to focus on business logic. Applications connect to a gateway and then all subsequent interactions are managed using that gateway's configuration.

In this topic, we're going to cover:

- *Why gateways are important*
- *How applications use a gateway*
- *How to define a static gateway*
- *How to define a dynamic gateway for service discovery*
- *Using multiple gateways*

Scenario

A Hyperledger Fabric network channel can constantly change. The peer, orderer and CA components, contributed by the different organizations in the network, will come and go. Reasons for this include increased or reduced business demand, and both planned and unplanned outages. A gateway relieves an application of this burden, allowing it to focus on the business problem it is trying to solve.



A MagnetoCorp and DigiBank applications (issue and buy) delegate their respective network interactions to their gateways. Each gateway understands the network channel topology comprising the multiple peers and orderers of two organizations MagnetoCorp and DigiBank, leaving applications to focus on business logic. Peers can talk to each other both within and across organizations using the gossip protocol.

A gateway can be used by an application in two different ways:

- **Static:** The gateway configuration is *completely* defined in a [connection profile](#). All the peers, orderers and CAs available to an application are statically defined in the connection profile used to configure the gateway. For peers, this includes their role as an endorsing peer or event notification hub, for example. You can read more about these roles in the [connection profile topic](#).

The SDK will use this static topology, in conjunction with gateway [connection options](#), to manage the transaction submission and notification processes. The connection profile must contain enough of the network topology to allow a gateway to interact with the network on behalf of the application; this includes the network channels, organizations, orderers, peers and their roles.

- **Dynamic:** The gateway configuration is minimally defined in a connection profile. Typically, one or two peers from the application's organization are specified, and they use [service discovery](#) to discover the available network topology. This includes peers, orderers, channels, deployed smart contracts and their endorsement policies. (In production environments, a gateway configuration should specify at least two peers for availability.)

The SDK will use all of the static and discovered topology information, in conjunction with gateway connection options, to manage the transaction submission and notification processes. As part of this, it will also intelligently

use the discovered topology; for example, it will *calculate* the minimum required endorsing peers using the discovered endorsement policy for the smart contract.

You might ask yourself whether a static or dynamic gateway is better? The trade-off is between predictability and responsiveness. Static networks will always behave the same way, as they perceive the network as unchanging. In this sense they are predictable – they will always use the same peers and orderers if they are available. Dynamic networks are more responsive as they understand how the network changes – they can use newly added peers and orderers, which brings extra resilience and scalability, at potentially some cost in predictability. In general it's fine to use dynamic networks, and indeed this the default mode for gateways.

Note that the *same* connection profile can be used statically or dynamically. Clearly, if a profile is going to be used statically, it needs to be comprehensive, whereas dynamic usage requires only sparse population.

Both styles of gateway are transparent to the application; the application program design does not change whether static or dynamic gateways are used. This also means that some applications may use service discovery, while others may not. In general using dynamic discovery means less definition and more intelligence by the SDK; it is the default.

Connect

When an application connects to a gateway, two options are provided. These are used in subsequent SDK processing:

```
await gateway.connect(connectionProfile, connectionOptions);
```

- **Connection profile:** `connectionProfile` is the gateway configuration that will be used for transaction processing by the SDK, whether statically or dynamically. It can be specified in YAML or JSON, though it must be converted to a JSON object when passed to the gateway:

```
let connectionProfile = yaml.safeLoad(fs.readFileSync('../gateway/paperNet.yaml',  
↪ 'utf8'));
```

Read more about [connection profiles](#) and how to configure them.

- **Connection options:** `connectionOptions` allow an application to declare rather than implement desired transaction processing behaviour. Connection options are interpreted by the SDK to control interaction patterns with network components, for example to select which identity to connect with, or which peers to use for event notifications. These options significantly reduce application complexity without compromising functionality. This is possible because the SDK has implemented much of the low level logic that would otherwise be required by applications; connection options control this logic flow.

Read about the list of available [connection options](#) and when to use them.

Static

Static gateways define a fixed view of a network. In the MagnetoCorp *scenario*, a gateway might identify a single peer from MagnetoCorp, a single peer from DigiBank, and a MagentoCorp orderer. Alternatively, a gateway might define *all* peers and orderers from MagnetCorp and DigiBank. In both cases, a gateway must define a view of the network sufficient to get commercial paper transactions endorsed and distributed.

Applications can use a gateway statically by explicitly specifying the connect option `discovery: { enabled:false }` on the `gateway.connect()` API. Alternatively, the environment variable setting `FABRIC_SDK_DISCOVERY=false` will always override the application choice.

Examine the [connection profile](#) used by the MagnetoCorp issue application. See how all the peers, orderers and even CAs are specified in this file, including their roles.

It's worth bearing in mind that a static gateway represents a view of a network at a *moment in time*. As networks change, it may be important to reflect this in a change to the gateway file. Applications will automatically pick up these changes when they re-load the gateway file.

Dynamic

Dynamic gateways define a small, fixed *starting point* for a network. In the MagnetoCorp *scenario*, a dynamic gateway might identify just a single peer from MagnetoCorp; everything else will be discovered! (To provide resiliency, it might be better to define two such bootstrap peers.)

If *service discovery* is selected by an application, the topology defined in the gateway file is augmented with that produced by this process. Service discovery starts with the gateway definition, and finds all the connected peers and orderers within the MagnetoCorp organization using the *gossip protocol*. If *anchor peers* have been defined for a channel, then service discovery will use the gossip protocol across organizations to discover components within the connected organization. This process will also discover smart contracts installed on peers and their endorsement policies defined at a channel level. As with static gateways, the discovered network must be sufficient to get commercial paper transactions endorsed and distributed.

Dynamic gateways are the default setting for Fabric applications. They can be explicitly specified using the connect option `discovery: { enabled:true }` on the `gateway.connect()` API. Alternatively, the environment variable setting `FABRIC_SDK_DISCOVERY=true` will always override the application choice.

A dynamic gateway represents an up-to-date view of a network. As networks change, service discovery will ensure that the network view is an accurate reflection of the topology visible to the application. Applications will automatically pick up these changes; they do not even need to re-load the gateway file.

Multiple gateways

Finally, it is straightforward for an application to define multiple gateways, both for the same or different networks. Moreover, applications can use the name `gateway` both statically and dynamically.

It can be helpful to have multiple gateways. Here are a few reasons:

- Handling requests on behalf of different users.
- Connecting to different networks simultaneously.
- Testing a network configuration, by simultaneously comparing its behaviour with an existing configuration.

This topic covers how to develop a client application and smart contract to solve a business problem using Hyperledger Fabric. In a real world **Commercial Paper** scenario, involving multiple organizations, you'll learn about all the concepts and tasks required to accomplish this goal. We assume that the blockchain network is already available.

The topic is designed for multiple audiences:

- Solution and application architect
- Client application developer
- Smart contract developer
- Business professional

You can choose to read the topic in order, or you can select individual sections as appropriate. Individual topic sections are marked according to reader relevance, so whether you're looking for business or technical information it'll be clear when a topic is for you.

The topic follows a typical software development lifecycle. It starts with business requirements, and then covers all the major technical activities required to develop an application and smart contract to meet these requirements.

If you'd prefer, you can try out the commercial paper scenario immediately, following an abbreviated explanation, by running the commercial paper [tutorial](#). You can return to this topic when you need fuller explanations of the concepts introduced in the tutorial.

Application developers can use the Fabric tutorials to get started building their own solutions. Start working with Fabric by deploying the [test network](#) on your local machine. You can then use the steps provided by the [Deploying a smart contract to a channel](#) tutorial to deploy and test your smart contracts. The [Writing Your First Application](#) tutorial provides an introduction to how to use the APIs provided by the Fabric SDKs to invoke smart contracts from your client applications. For an in depth overview of how Fabric applications and smart contracts work together, you can visit the [Developing Applications](#) topic.

Network operators can use the [Deploying a smart contract to a channel](#) tutorial and the [Creating a channel](#) tutorial series to learn important aspects of administering a running network. Both network operators and application developers can use the tutorials on [Private data](#) and [CouchDB](#) to explore important Fabric features. When you are ready to deploy Hyperledger Fabric in production, see the guide for [Deploying a production network](#).

There are two tutorials for updating a channel: [Updating a channel configuration](#) and [Updating the capability level of a channel](#), while [Upgrading your components](#) shows how to upgrade components like peers, ordering nodes, SDKs, and more.

Finally, we provide an introduction to how to write a basic smart contract, [Writing Your First Chaincode](#).

Note: If you have questions not addressed by this documentation, or run into issues with any of the tutorials, please visit the [Still Have Questions?](#) page for some tips on where to find additional help.

7.1 Deploying a smart contract to a channel

End users interact with the blockchain ledger by invoking smart contracts. In Hyperledger Fabric, smart contracts are deployed in packages referred to as chaincode. Organizations that want to validate transactions or query the ledger need to install a chaincode on their peers. After a chaincode has been installed on the peers joined to a channel, channel members can deploy the chaincode to the channel and use the smart contracts in the chaincode to create or update assets on the channel ledger.

A chaincode is deployed to a channel using a process known as the Fabric chaincode lifecycle. The Fabric chaincode lifecycle allows multiple organizations to agree how a chaincode will be operated before it can be used to create

transactions. For example, while an endorsement policy specifies which organizations need to execute a chaincode to validate a transaction, channel members need to use the Fabric chaincode lifecycle to agree on the chaincode endorsement policy. For a more in-depth overview about how to deploy and manage a chaincode on a channel, see [Fabric chaincode lifecycle](#).

You can use this tutorial to learn how to use the [peer lifecycle chaincode commands](#) to deploy a chaincode to a channel of the Fabric test network. Once you have an understanding of the commands, you can use the steps in this tutorial to deploy your own chaincode to the test network, or to deploy chaincode to a production network. In this tutorial, you will deploy the asset-transfer (basic) chaincode that is used by the [Writing your first application tutorial](#).

Note: These instructions use the Fabric chaincode lifecycle introduced in the v2.0 release. If you would like to use the previous lifecycle to install and instantiate a chaincode, visit the [v1.4 version of the Fabric documentation](#).

7.1.1 Start the network

We will start by deploying an instance of the Fabric test network. Before you begin, make sure that you have installed the [Prerequisites](#) and [Installed the Samples, Binaries and Docker Images](#). Use the following command to navigate to the test network directory within your local clone of the `fabric-samples` repository:

```
cd fabric-samples/test-network
```

For the sake of this tutorial, we want to operate from a known initial state. The following command will kill any active or stale docker containers and remove previously generated artifacts.

```
./network.sh down
```

You can then use the following command to start the test network:

```
./network.sh up createChannel
```

The `createChannel` command creates a channel named `mychannel` with two channel members, `Org1` and `Org2`. The command also joins a peer that belongs to each organization to the channel. If the network and the channel are created successfully, you can see the following message printed in the logs:

```
===== Channel successfully joined =====
```

We can now use the Peer CLI to deploy the asset-transfer (basic) chaincode to the channel using the following steps:

- *Step one: Package the smart contract*
- *Step two: Install the chaincode package*
- *Step three: Approve a chaincode definition*
- *Step four: Committing the chaincode definition to the channel*

7.1.2 Setup Logspout (optional)

This step is not required but is extremely useful for troubleshooting chaincode. To monitor the logs of the smart contract, an administrator can view the aggregated output from a set of Docker containers using the [logspout tool](#). The tool collects the output streams from different Docker containers into one place, making it easy to see what's happening from a single window. This can help administrators debug problems when they install smart contracts or developers when they invoke smart contracts. Because some containers are created purely for the purposes of starting a smart contract and only exist for a short time, it is helpful to collect all of the logs from your network.

A script to install and configure Logspout, `monitordocker.sh`, is already included in the `commercial-paper` sample in the Fabric samples. We will use the same script in this tutorial as well. The Logspout tool will continuously

stream logs to your terminal, so you will need to use a new terminal window. Open a new terminal and navigate to the `test-network` directory.

```
cd fabric-samples/test-network
```

You can run the `monitordocker.sh` script from any directory. For ease of use, we will copy the `monitordocker.sh` script from the `commercial-paper` sample to your working directory

```
cp ../commercial-paper/organization/digibank/configuration/cli/monitordocker.sh .
# if you're not sure where it is
find . -name monitordocker.sh
```

You can then start Logspout by running the following command:

```
./monitordocker.sh net_test
```

You should see output similar to the following:

```
Starting monitoring on all containers on the network net_basic
Unable to find image 'gliderlabs/logspout:latest' locally
latest: Pulling from gliderlabs/logspout
4fe2ade4980c: Pull complete
decca452f519: Pull complete
ad60f6b6c009: Pull complete
Digest: sha256:374e06b17b004bddc5445525796b5f7adb8234d64c5c5d663095fccafb6e4c26
Status: Downloaded newer image for gliderlabs/logspout:latest
1f99d130f15cf01706eda3e1f040496ec885036d485cb6bcc0da4a567ad84361
```

You will not see any logs at first, but this will change when we deploy our chaincode. It can be helpful to make this terminal window wide and the font small.

7.1.3 Package the smart contract

We need to package the chaincode before it can be installed on our peers. The steps are different if you want to install a smart contract written in *Go*, *JavaScript*, or *Typescript*.

Go

Before we package the chaincode, we need to install the chaincode dependencies. Navigate to the folder that contains the Go version of the `asset-transfer` (basic) chaincode.

```
cd fabric-samples/asset-transfer-basic/chaincode-go
```

The sample uses a Go module to install the chaincode dependencies. The dependencies are listed in a `go.mod` file in the `asset-transfer-basic/chaincode-go` directory. You should take a moment to examine this file.

```
$ cat go.mod
module github.com/hyperledger/fabric-samples/asset-transfer-basic/chaincode-go

go 1.14

require (
    github.com/golang/protobuf v1.3.2
    github.com/hyperledger/fabric-chaincode-go v0.0.0-20200424173110-d7076418f212
    github.com/hyperledger/fabric-contract-api-go v1.1.0
```

(continues on next page)

(continued from previous page)

```
github.com/hyperledger/fabric-protos-go v0.0.0-20200424173316-dd554ba3746e
github.com/stretchr/testify v1.5.1
)
```

The `go.mod` file imports the Fabric contract API into the smart contract package. You can open `asset-transfer-basic/chaincode-go/chaincode/smartcontract.go` in a text editor to see how the contract API is used to define the `SmartContract` type at the beginning of the smart contract:

```
// SmartContract provides functions for managing an Asset
type SmartContract struct {
    contractapi.Contract
}
```

The `SmartContract` type is then used to create the transaction context for the functions defined within the smart contract that read and write data to the blockchain ledger.

```
// CreateAsset issues a new asset to the world state with given details.
func (s *SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface, id_
→string, color string, size int, owner string, appraisedValue int) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if exists {
        return fmt.Errorf("the asset %s already exists", id)
    }

    asset := Asset{
        ID:          id,
        Color:       color,
        Size:        size,
        Owner:       owner,
        AppraisedValue: appraisedValue,
    }
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(id, assetJSON)
}
```

You can learn more about the Go contract API by visiting the [API documentation](#) and the [smart contract processing topic](#).

To install the smart contract dependencies, run the following command from the `asset-transfer-basic/chaincode-go` directory.

```
GO111MODULE=on go mod vendor
```

If the command is successful, the go packages will be installed inside a `vendor` folder.

Now that we have our dependencies, we can create the chaincode package. Navigate back to our working directory in the `test-network` folder so that we can package the chaincode together with our other network artifacts.

```
cd ../../test-network
```

You can use the `peer` CLI to create a chaincode package in the required format. The `peer` binaries are located in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

```
export PATH=${PWD}/../bin:$PATH
```

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To confirm that you are able to use the `peer` CLI, check the version of the binaries. The binaries need to be version `2.0.0` or later to run this tutorial.

```
peer version
```

You can now create the chaincode package using the `peer lifecycle chaincode package` command:

```
peer lifecycle chaincode package basic.tar.gz --path ../asset-transfer-basic/
↳chaincode-go/ --lang golang --label basic_1.0
```

This command will create a package named `basic.tar.gz` in your current directory. The `--lang` flag is used to specify the chaincode language and the `--path` flag provides the location of your smart contract code. The path must be a fully qualified path or a path relative to your present working directory. The `--label` flag is used to specify a chaincode label that will identify your chaincode after it is installed. It is recommended that your label include the chaincode name and version.

Now that we created the chaincode package, we can *install the chaincode* on the peers of the test network.

JavaScript

Before we package the chaincode, we need to install the chaincode dependencies. Navigate to the folder that contains the JavaScript version of the `asset-transfer (basic)` chaincode.

```
cd fabric-samples/asset-transfer-basic/chaincode-javascript
```

The dependencies are listed in the `package.json` file in the `asset-transfer-basic/chaincode-javascript` directory. You should take a moment to examine this file. You can find the dependencies section displayed below:

```
"dependencies": {
  "fabric-contract-api": "^2.0.0",
  "fabric-shim": "^2.0.0"
```

The `package.json` file imports the Fabric contract class into the smart contract package. You can open `lib/assetTransfer.js` in a text editor to see the contract class imported into the smart contract and used to create the `asset-transfer (basic)` class.

```
const { Contract } = require('fabric-contract-api');

class AssetTransfer extends Contract {
  ...
}
```

The `AssetTransfer` class provides the transaction context for the functions defined within the smart contract that read and write data to the blockchain ledger.

```
async CreateAsset(ctx, id, color, size, owner, appraisedValue) {
  const asset = {
    ID: id,
    Color: color,
    Size: size,
    Owner: owner,
    AppraisedValue: appraisedValue,
  };

  await ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
}
```

You can learn more about the JavaScript contract API by visiting the [API documentation](#) and the [smart contract processing topic](#).

To install the smart contract dependencies, run the following command from the `asset-transfer-basic/chaincode-javascript` directory.

```
npm install
```

If the command is successful, the JavaScript packages will be installed inside a `npm_modules` folder.

Now that we have our dependencies, we can create the chaincode package. Navigate back to our working directory in the `test-network` folder so that we can package the chaincode together with our other network artifacts.

```
cd ../../test-network
```

You can use the `peer` CLI to create a chaincode package in the required format. The `peer` binaries are located in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

```
export PATH=${PWD}/../bin:$PATH
```

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To confirm that you are able to use the `peer` CLI, check the version of the binaries. The binaries need to be version `2.0.0` or later to run this tutorial.

```
peer version
```

You can now create the chaincode package using the `peer lifecycle chaincode package` command:

```
peer lifecycle chaincode package basic.tar.gz --path ../asset-transfer-basic/
↪chaincode-javascript/ --lang node --label basic_1.0
```

This command will create a package named `basic.tar.gz` in your current directory. The `--lang` flag is used to specify the chaincode language and the `--path` flag provides the location of your smart contract code. The `--label` flag is used to specify a chaincode label that will identify your chaincode after it is installed. It is recommended that your label include the chaincode name and version.

Now that we created the chaincode package, we can *install the chaincode* on the peers of the test network.

Typescript

Before we package the chaincode, we need to install the chaincode dependencies. Navigate to the folder that contains the TypeScript version of the asset-transfer (basic) chaincode.

```
cd fabric-samples/asset-transfer-basic/chaincode-typescript
```

The dependencies are listed in the `package.json` file in the `asset-transfer-basic/chaincode-typescript` directory. You should take a moment to examine this file. You can find the dependencies section displayed below:

```
"dependencies": {
  "fabric-contract-api": "^2.0.0",
  "fabric-shim": "^2.0.0"
```

The `package.json` file imports the Fabric contract class into the smart contract package. You can open `src/assetTransfer.ts` in a text editor to see the contract class imported into the smart contract and used to create the asset-transfer (basic) class. Also notice that the `Asset` class is imported from the type definition file `asset.ts`.

```
import { Context, Contract } from 'fabric-contract-api';
import { Asset } from './asset';

export class AssetTransfer extends Contract {
  ...
}
```

The `AssetTransfer` class provides the transaction context for the functions defined within the smart contract that read and write data to the blockchain ledger.

```
// CreateAsset issues a new asset to the world state with given details.
public async CreateAsset(ctx: Context, id: string, color: string, size: number,
  owner: string, appraisedValue: number) {
  const asset = {
    ID: id,
    Color: color,
    Size: size,
    Owner: owner,
    AppraisedValue: appraisedValue,
  };

  await ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
}
```

You can learn more about the JavaScript contract API by visiting the [API documentation](#) and the [smart contract processing topic](#).

To install the smart contract dependencies, run the following command from the `asset-transfer-basic/chaincode-typescript` directory.

```
npm install
```

If the command is successful, the JavaScript packages will be installed inside a `npm_modules` folder.

Now that we have our dependencies, we can create the chaincode package. Navigate back to our working directory in the `test-network` folder so that we can package the chaincode together with our other network artifacts.

```
cd ../../test-network
```

You can use the `peer` CLI to create a chaincode package in the required format. The `peer` binaries are located in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

```
export PATH=${PWD}/../bin:$PATH
```

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To confirm that you are able to use the `peer` CLI, check the version of the binaries. The binaries need to be version `2.0.0` or later to run this tutorial.

```
peer version
```

You can now create the chaincode package using the `peer lifecycle chaincode package` command:

```
peer lifecycle chaincode package basic.tar.gz --path ../asset-transfer-basic/  
↪chaincode-typescript/ --lang node --label basic_1.0
```

This command will create a package named `basic.tar.gz` in your current directory. The `--lang` flag is used to specify the chaincode language and the `--path` flag provides the location of your smart contract code. The `--label` flag is used to specify a chaincode label that will identify your chaincode after it is installed. It is recommended that your label include the chaincode name and version.

Now that we created the chaincode package, we can *install the chaincode* on the peers of the test network.

7.1.4 Install the chaincode package

After we package the `asset-transfer (basic)` smart contract, we can install the chaincode on our peers. The chaincode needs to be installed on every peer that will endorse a transaction. Because we are going to set the endorsement policy to require endorsements from both `Org1` and `Org2`, we need to install the chaincode on the peers operated by both organizations:

- `peer0.org1.example.com`
- `peer0.org2.example.com`

Let's install the chaincode on the `Org1` peer first. Set the following environment variables to operate the `peer` CLI as the `Org1` admin user. The `CORE_PEER_ADDRESS` will be set to point to the `Org1` peer, `peer0.org1.example.com`.

```
export CORE_PEER_TLS_ENABLED=true  
export CORE_PEER_LOCALMSPID="Org1MSP"  
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.  
↪example.com/peers/peer0.org1.example.com/tls/ca.crt  
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.  
↪com/users/Admin@org1.example.com/msp  
export CORE_PEER_ADDRESS=localhost:7051
```

Issue the `peer lifecycle chaincode install` command to install the chaincode on the peer:

```
peer lifecycle chaincode install basic.tar.gz
```

If the command is successful, the peer will generate and return the package identifier. This package ID will be used to approve the chaincode in the next step. You should see output similar to the following:


```
2020-07-16 10:09:57.534 CDT [cli.lifecycle.chaincode] submitInstallProposal -> INFO_
↪001 Installed remotely: response:<status:200 payload:"\nJbasic_1.
↪0:e2db7f693d4aa6156e652741d5606e9c5f0de9ebb88c5721cb8248c3aead8123\022\tbasic_1.0" >
2020-07-16 10:09:57.534 CDT [cli.lifecycle.chaincode] submitInstallProposal -> INFO_
↪002 Chaincode code package identifier: basic_1.
↪0:e2db7f693d4aa6156e652741d5606e9c5f0de9ebb88c5721cb8248c3aead8123
```

We can now install the chaincode on the Org2 peer. Set the following environment variables to operate as the Org2 admin and target the Org2 peer, `peer0.org2.example.com`.

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

Issue the following command to install the chaincode:

```
peer lifecycle chaincode install basic.tar.gz
```

The chaincode is built by the peer when the chaincode is installed. The install command will return any build errors from the chaincode if there is a problem with the smart contract code.

7.1.5 Approve a chaincode definition

After you install the chaincode package, you need to approve a chaincode definition for your organization. The definition includes the important parameters of chaincode governance such as the name, version, and the chaincode endorsement policy.

The set of channel members who need to approve a chaincode before it can be deployed is governed by the `Application/Channel/lifecycleEndorsement` policy. By default, this policy requires that a majority of channel members need to approve a chaincode before it can be used on a channel. Because we have only two organizations on the channel, and a majority of 2 is 2, we need approve a chaincode definition of asset-transfer (basic) as Org1 and Org2.

If an organization has installed the chaincode on their peer, they need to include the packageID in the chaincode definition approved by their organization. The package ID is used to associate the chaincode installed on a peer with an approved chaincode definition, and allows an organization to use the chaincode to endorse transactions. You can find the package ID of a chaincode by using the `peer lifecycle chaincode queryinstalled` command to query your peer.

```
peer lifecycle chaincode queryinstalled
```

The package ID is the combination of the chaincode label and a hash of the chaincode binaries. Every peer will generate the same package ID. You should see output similar to the following:

```
Installed chaincodes on peer:
Package ID: basic_1.
↪0:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3, Label: basic_1.0
```

We are going to use the package ID when we approve the chaincode, so let's go ahead and save it as an environment variable. Paste the package ID returned by `peer lifecycle chaincode queryinstalled` into the command below. **Note:** The package ID will not be the same for all users, so you need to complete this step using the package ID returned from your command window in the previous step.

```
export CC_PACKAGE_ID=basic_1.
↪0:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3
```

Because the environment variables have been set to operate the peer CLI as the Org2 admin, we can approve the chaincode definition of asset-transfer (basic) as Org2. Chaincode is approved at the organization level, so the command only needs to target one peer. The approval is distributed to the other peers within the organization using gossip. Approve the chaincode definition using the [peer lifecycle chaincode approveformyorg](#) command:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic --
↪version 1.0 --package-id $CC_PACKAGE_ID --sequence 1 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

The command above uses the `--package-id` flag to include the package identifier in the chaincode definition. The `--sequence` parameter is an integer that keeps track of the number of times a chaincode has been defined or updated. Because the chaincode is being deployed to the channel for the first time, the sequence number is 1. When the asset-transfer (basic) chaincode is upgraded, the sequence number will be incremented to 2. If you are using the low level APIs provided by the Fabric Chaincode Shim API, you could pass the `--init-required` flag to the command above to request the execution of the Init function to initialize the chaincode. The first invoke of the chaincode would need to target the Init function and include the `--isInit` flag before you could use the other functions in the chaincode to interact with the ledger.

We could have provided a `--signature-policy` or `--channel-config-policy` argument to the `approveformyorg` command to specify a chaincode endorsement policy. The endorsement policy specifies how many peers belonging to different channel members need to validate a transaction against a given chaincode. Because we did not set a policy, the definition of asset-transfer (basic) will use the default endorsement policy, which requires that a transaction be endorsed by a majority of channel members present when the transaction is submitted. This implies that if new organizations are added or removed from the channel, the endorsement policy is updated automatically to require more or fewer endorsements. In this tutorial, the default policy will require a majority of 2 out of 2 and transactions will need to be endorsed by a peer from Org1 and Org2. If you want to specify a custom endorsement policy, you can use the [Endorsement Policies](#) operations guide to learn about the policy syntax.

You need to approve a chaincode definition with an identity that has an admin role. As a result, the `CORE_PEER MSPCONFIGPATH` variable needs to point to the MSP folder that contains an admin identity. You cannot approve a chaincode definition with a client user. The approval needs to be submitted to the ordering service, which will validate the admin signature and then distribute the approval to your peers.

We still need to approve the chaincode definition as Org1. Set the following environment variables to operate as the Org1 admin:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_ADDRESS=localhost:7051
```

You can now approve the chaincode definition as Org1.

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic --
↪version 1.0 --package-id $CC_PACKAGE_ID --sequence 1 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

We now have the majority we need to deploy the asset-transfer (basic) the chaincode to the channel. While only a

majority of organizations need to approve a chaincode definition (with the default policies), all organizations need to approve a chaincode definition to start the chaincode on their peers. If you commit the definition before a channel member has approved the chaincode, the organization will not be able to endorse transactions. As a result, it is recommended that all channel members approve a chaincode before committing the chaincode definition.

7.1.6 Committing the chaincode definition to the channel

After a sufficient number of organizations have approved a chaincode definition, one organization can commit the chaincode definition to the channel. If a majority of channel members have approved the definition, the commit transaction will be successful and the parameters agreed to in the chaincode definition will be implemented on the channel.

You can use the `peer lifecycle chaincode checkcommitreadiness` command to check whether channel members have approved the same chaincode definition. The flags used for the `checkcommitreadiness` command are identical to the flags used to approve a chaincode for your organization. However, you do not need to include the `--package-id` flag.

```
peer lifecycle chaincode checkcommitreadiness --channelID mychannel --name basic --
↪version 1.0 --sequence 1 --tls --cafile ${PWD}/organizations/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -
↪-output json
```

The command will produce a JSON map that displays if a channel member has approved the parameters that were specified in the `checkcommitreadiness` command:

```
{
  "Approvals": {
    "Org1MSP": true,
    "Org2MSP": true
  }
}
```

Since both organizations that are members of the channel have approved the same parameters, the chaincode definition is ready to be committed to the channel. You can use the `peer lifecycle chaincode commit` command to commit the chaincode definition to the channel. The commit command also needs to be submitted by an organization admin.

```
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride_
↪orderer.example.com --channelID mychannel --name basic --version 1.0 --sequence 1 --
↪tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.
↪example.com/msp/tlscacerts/tlsca.example.com-cert.pem --peerAddresses_
↪localhost:7051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 -
↪-tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org2.example.com/peers/
↪peer0.org2.example.com/tls/ca.crt
```

The transaction above uses the `--peerAddresses` flag to target `peer0.org1.example.com` from Org1 and `peer0.org2.example.com` from Org2. The commit transaction is submitted to the peers joined to the channel to query the chaincode definition that was approved by the organization that operates the peer. The command needs to target the peers from a sufficient number of organizations to satisfy the policy for deploying a chaincode. Because the approval is distributed within each organization, you can target any peer that belongs to a channel member.

The chaincode definition endorsements by channel members are submitted to the ordering service to be added to a block and distributed to the channel. The peers on the channel then validate whether a sufficient number of organizations have approved the chaincode definition. The `peer lifecycle chaincode commit` command will wait for the validations from the peer before returning a response.

You can use the `peer lifecycle chaincode querycommitted` command to confirm that the chaincode definition has been committed to the channel.

```
peer lifecycle chaincode querycommitted --channelID mychannel --name basic --cafile $
↪{PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/
↪msp/tlscacerts/tlsca.example.com-cert.pem
```

If the chaincode was successful committed to the channel, the `querycommitted` command will return the sequence and version of the chaincode definition:

```
Committed chaincode definition for chaincode 'basic' on channel 'mychannel':
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc,
↪Approvals: [Org1MSP: true, Org2MSP: true]
```

7.1.7 Invoking the chaincode

After the chaincode definition has been committed to a channel, the chaincode will start on the peers joined to the channel where the chaincode was installed. The asset-transfer (basic) chaincode is now ready to be invoked by client applications. Use the following command create an initial set of assets on the ledger. Note that the invoke command needs target a sufficient number of peers to meet chaincode endorsement policy.

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic
↪--peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"function": "InitLedger", "Args": []}'
```

If the command is successful, you should be able to a response similar to the following:

```
2020-02-12 18:22:20.576 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001
↪Chaincode invoke successful. result: status:200
```

We can use a query function to read the set of cars that were created by the chaincode:

```
peer chaincode query -C mychannel -n basic -c '{"Args":["GetAllAssets"]}'
```

The response to the query should be the following list of assets:

```
[{"Key": "asset1", "Record": {"ID": "asset1", "color": "blue", "size": 5, "owner": "Tomoko",
↪"appraisedValue": 300}},
{"Key": "asset2", "Record": {"ID": "asset2", "color": "red", "size": 5, "owner": "Brad",
↪"appraisedValue": 400}},
{"Key": "asset3", "Record": {"ID": "asset3", "color": "green", "size": 10, "owner": "Jin Soo",
↪"appraisedValue": 500}},
{"Key": "asset4", "Record": {"ID": "asset4", "color": "yellow", "size": 10, "owner": "Max",
↪"appraisedValue": 600}},
{"Key": "asset5", "Record": {"ID": "asset5", "color": "black", "size": 15, "owner": "Adriana",
↪"appraisedValue": 700}},
{"Key": "asset6", "Record": {"ID": "asset6", "color": "white", "size": 15, "owner": "Michel",
↪"appraisedValue": 800}}]
```

7.1.8 Upgrading a smart contract

You can use the same Fabric chaincode lifecycle process to upgrade a chaincode that has already been deployed to a channel. Channel members can upgrade a chaincode by installing a new chaincode package and then approving a chaincode definition with the new package ID, a new chaincode version, and with the sequence number incremented by one. The new chaincode can be used after the chaincode definition is committed to the channel. This process allows channel members to coordinate on when a chaincode is upgraded, and ensure that a sufficient number of channel members are ready to use the new chaincode before it is deployed to the channel.

Channel members can also use the upgrade process to change the chaincode endorsement policy. By approving a chaincode definition with a new endorsement policy and committing the chaincode definition to the channel, channel members can change the endorsement policy governing a chaincode without installing a new chaincode package.

To provide a scenario for upgrading the asset-transfer (basic) chaincode that we just deployed, let's assume that Org1 and Org2 would like to install a version of the chaincode that is written in another language. They will use the Fabric chaincode lifecycle to update the chaincode version and ensure that both organizations have installed the new chaincode before it becomes active on the channel.

We are going to assume that Org1 and Org2 initially installed the GO version of the asset-transfer (basic) chaincode, but would be more comfortable working with a chaincode written in JavaScript. The first step is to package the JavaScript version of the asset-transfer (basic) chaincode. If you used the JavaScript instructions to package your chaincode when you went through the tutorial, you can install new chaincode binaries by following the steps for packaging a chaincode written in *Go* or *TypeScript*.

Issue the following commands from the `test-network` directory to install the chaincode dependences.

```
cd ../asset-transfer-basic/chaincode-javascript
npm install
cd ../../test-network
```

You can then issue the following commands to package the JavaScript chaincode from the `test-network` directory. We will set the environment variables needed to use the `peer` CLI again in case you closed your terminal.

```
export PATH=${PWD}/../bin:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
peer lifecycle chaincode package basic_2.tar.gz --path ../asset-transfer-basic/
↪chaincode-javascript/ --lang node --label basic_2.0
```

Run the following commands to operate the `peer` CLI as the Org1 admin:

```
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

We can now use the following command to install the new chaincode package on the Org1 peer.

```
peer lifecycle chaincode install basic_2.tar.gz
```

The new chaincode package will create a new package ID. We can find the new package ID by querying our peer.

```
peer lifecycle chaincode queryinstalled
```

The `queryinstalled` command will return a list of the chaincode that have been installed on your peer similar to this output.

```
Installed chaincodes on peer:
Package ID: basic_1.
↪0:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3, Label: basic_1.0
Package ID: basic_2.
↪0:1d559f9fb3dd879601ee17047658c7e0c84eab732dca7c841102f20e42a9e7d4, Label: basic_2.0
```

You can use the package label to find the package ID of the new chaincode and save it as a new environment variable. This output is for example only – your package ID will be different, so **DO NOT COPY AND PASTE!**

```
export NEW_CC_PACKAGE_ID=basic_2.
↪0:1d559f9fb3dd879601ee17047658c7e0c84eab732dca7c841102f20e42a9e7d4
```

Org1 can now approve a new chaincode definition:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic --
↪version 2.0 --package-id $NEW_CC_PACKAGE_ID --sequence 2 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

The new chaincode definition uses the package ID of the JavaScript chaincode package and updates the chaincode version. Because the sequence parameter is used by the Fabric chaincode lifecycle to keep track of chaincode upgrades, Org1 also needs to increment the sequence number from 1 to 2. You can use the `peer lifecycle chaincode querycommitted` command to find the sequence of the chaincode that was last committed to the channel.

We now need to install the chaincode package and approve the chaincode definition as Org2 in order to upgrade the chaincode. Run the following commands to operate the peer CLI as the Org2 admin:

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

We can now use the following command to install the new chaincode package on the Org2 peer.

```
peer lifecycle chaincode install basic_2.tar.gz
```

You can now approve the new chaincode definition for Org2.

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic --
↪version 2.0 --package-id $NEW_CC_PACKAGE_ID --sequence 2 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

Use the `peer lifecycle chaincode checkcommitreadiness` command to check if the chaincode definition with sequence 2 is ready to be committed to the channel:

```
peer lifecycle chaincode checkcommitreadiness --channelID mychannel --name basic --
↪version 2.0 --sequence 2 --tls --cafile ${PWD}/organizations/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -
↪-output json
```

(continues on next page)

(continued from previous page)

The chaincode is ready to be upgraded if the command returns the following JSON:

```
{
  "Approvals": {
    "Org1MSP": true,
    "Org2MSP": true
  }
}
```

The chaincode will be upgraded on the channel after the new chaincode definition is committed. Until then, the previous chaincode will continue to run on the peers of both organizations. Org2 can use the following command to upgrade the chaincode:

```
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride_
↪orderer.example.com --channelID mychannel --name basic --version 2.0 --sequence 2 --
↪tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.
↪example.com/msp/tlscacerts/tlsca.example.com-cert.pem --peerAddresses_
↪localhost:7051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 -
↪--tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org2.example.com/peers/
↪peer0.org2.example.com/tls/ca.crt
```

A successful commit transaction will start the new chaincode right away. If the chaincode definition changed the endorsement policy, the new policy would be put in effect.

You can use the `docker ps` command to verify that the new chaincode has started on your peers:

```
$ docker ps
CONTAINER ID          IMAGE
↪
↪
↪
↪COMMAND
↪CREATED
↪STATUS
↪PORTS
↪NAMES
7bf2f1bf792b         dev-peer0.org1.example.com-basic_2.0-
↪572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c-
↪69c9e3e44ed18cafd1e58de37a70e2ec54cd49c7da0cd461fbd5e333de32879b   "docker-
↪entrypoint.s..." 2 minutes ago      Up 2 minutes
↪dev-peer0.org1.example.com-basic_2.0-
↪572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c
985e0967c27a         dev-peer0.org2.example.com-basic_2.0-
↪572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c-
↪158e9c6a4cb51dea043461fc4d3580e7df4c74a52b41e69a25705ce85405d760   "docker-
↪entrypoint.s..." 2 minutes ago      Up 2 minutes
↪dev-peer0.org2.example.com-basic_2.0-
↪572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c
31fdd19c3be7         hyperledger/fabric-peer:latest
↪
↪
↪"peer node start"      About an hour ago   Up About an hour
↪0.0.0.0:7051->7051/tcp  peer0.org1.example.com
1b17ff866fe0         hyperledger/fabric-peer:latest
↪
↪
↪"peer node start"      About an hour ago   Up About an hour
↪7051/tcp, 0.0.0.0:9051->9051/tcp  peer0.org2.example.com
4cf170c7ae9b         hyperledger/fabric-orderer:latest
```

If you used the `--init-required` flag, you need to invoke the `Init` function before you can use the upgraded chaincode. Because we did not request the execution of `Init`, we can test our new JavaScript chaincode by creating a

new car:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic_
↪--peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"function": "CreateAsset", "Args": ["asset8", "blue", "16", "Kelley", "750"]}'
```

You can query all the cars on the ledger again to see the new car:

```
peer chaincode query -C mychannel -n basic -c '{"Args":["GetAllAssets"]}'
```

You should see the following result from the JavaScript chaincode:

```
[{"Key": "asset1", "Record": {"ID": "asset1", "color": "blue", "size": 5, "owner": "Tomoko",
↪ "appraisedValue": 300}},
{"Key": "asset2", "Record": {"ID": "asset2", "color": "red", "size": 5, "owner": "Brad",
↪ "appraisedValue": 400}},
{"Key": "asset3", "Record": {"ID": "asset3", "color": "green", "size": 10, "owner": "Jin Soo",
↪ "appraisedValue": 500}},
{"Key": "asset4", "Record": {"ID": "asset4", "color": "yellow", "size": 10, "owner": "Max",
↪ "appraisedValue": 600}},
{"Key": "asset5", "Record": {"ID": "asset5", "color": "black", "size": 15, "owner": "Adriana",
↪ "appraisedValue": 700}},
{"Key": "asset6", "Record": {"ID": "asset6", "color": "white", "size": 15, "owner": "Michel",
↪ "appraisedValue": 800}},
{"Key": "asset8", "Record": {"ID": "asset8", "color": "blue", "size": 16, "owner": "Kelley",
↪ "appraisedValue": 750}}]
```

7.1.9 Clean up

When you are finished using the chaincode, you can also use the following commands to remove the Logspout tool.

```
docker stop logspout
docker rm logspout
```

You can then bring down the test network by issuing the following command from the `test-network` directory:

```
./network.sh down
```

7.1.10 Next steps

After you write your smart contract and deploy it to a channel, you can use the APIs provided by the Fabric SDKs to invoke the smart contracts from a client application. This allows end users to interact with the assets on the blockchain ledger. To get started with the Fabric SDKs, see the [Writing Your first application tutorial](#).

7.1.11 troubleshooting

Chaincode not agreed to by this org

Problem: When I try to commit a new chaincode definition to the channel, the `peer lifecycle chaincode commit` command fails with the following error:

```
Error: failed to create signed transaction: proposal response was not successful,
↳ error code 500, msg failed to invoke backing implementation of
↳ 'CommitChaincodeDefinition': chaincode definition not agreed to by this org
↳ (Org1MSP)
```

Solution: You can try to resolve this error by using the `peer lifecycle chaincode checkcommitreadiness` command to check which channel members have approved the chaincode definition that you are trying to commit. If any organization used a different value for any parameter of the chaincode definition, the commit transaction will fail. The `peer lifecycle chaincode checkcommitreadiness` will reveal which organizations did not approve the chaincode definition you are trying to commit:

```
{
  "approvals": {
    "Org1MSP": false,
    "Org2MSP": true
  }
}
```

Invoke failure

Problem: The `peer lifecycle chaincode commit` transaction is successful, but when I try to invoke the chaincode for the first time, it fails with the following error:

```
Error: endorsement failure during invoke. response: status:500 message:"make sure the
↳ chaincode asset-transfer (basic) has been successfully defined on channel mychannel
↳ and try again: chaincode definition for 'asset-transfer (basic)' exists, but
↳ chaincode is not installed"
```

Solution: You may not have set the correct `--package-id` when you approved your chaincode definition. As a result, the chaincode definition that was committed to the channel was not associated with the chaincode package you installed and the chaincode was not started on your peers. If you are running a docker based network, you can use the `docker ps` command to check if your chaincode is running:

```
docker ps
CONTAINER ID          IMAGE                                COMMAND                  CREATED
↳ STATUS              PORTS                               NAMES
7felae0a69fa         hyperledger/fabric-orderer:latest  "orderer"               5 minutes
↳ ago                Up 4 minutes                       0.0.0.0:7050->7050/tcp   orderer.example.com
2b9c684bd07e         hyperledger/fabric-peer:latest     "peer node start"       5 minutes
↳ ago                Up 4 minutes                       0.0.0.0:7051->7051/tcp   peer0.org1.example.
↳ com
39a3e41b2573         hyperledger/fabric-peer:latest     "peer node start"       5 minutes
↳ ago                Up 4 minutes                       7051/tcp, 0.0.0.0:9051->9051/tcp peer0.org2.example.
↳ com
```

If you do not see any chaincode containers listed, use the `peer lifecycle chaincode approveformyorg` command approve a chaincode definition with the correct package ID.

7.1.12 Endorsement policy failure

Problem: When I try to commit the chaincode definition to the channel, the transaction fails with the following error:

```
2020-04-07 20:08:23.306 EDT [chaincodeCmd] ClientWait -> INFO 001 txid_
→[5f569e50ae58efa6261c4ad93180d49ac85ec29a07b58f576405b826a8213aeb] committed with_
→status (ENDORSEMENT_POLICY_FAILURE) at localhost:7051
Error: transaction invalidated with status (ENDORSEMENT_POLICY_FAILURE)
```

Solution: This error is a result of the commit transaction not gathering enough endorsements to meet the Lifecycle endorsement policy. This problem could be a result of your transaction not targeting a sufficient number of peers to meet the policy. This could also be the result of some of the peer organizations not including the `Endorsement : signature policy` referenced by the default `/Channel/Application/Endorsement` policy in their `configtx.yaml` file:

```
Readers:
  Type: Signature
  Rule: "OR('Org2MSP.admin', 'Org2MSP.peer', 'Org2MSP.client')"
Writers:
  Type: Signature
  Rule: "OR('Org2MSP.admin', 'Org2MSP.client')"
Admins:
  Type: Signature
  Rule: "OR('Org2MSP.admin')"
Endorsement:
  Type: Signature
  Rule: "OR('Org2MSP.peer')"
```

When you [enable the Fabric chaincode lifecycle](#), you also need to use the new Fabric 2.0 channel policies in addition to upgrading your channel to the V2_0 capability. Your channel needs to include the new `/Channel/Application/LifecycleEndorsement` and `/Channel/Application/Endorsement` policies:

```
Policies:
  Readers:
    Type: ImplicitMeta
    Rule: "ANY Readers"
  Writers:
    Type: ImplicitMeta
    Rule: "ANY Writers"
  Admins:
    Type: ImplicitMeta
    Rule: "MAJORITY Admins"
  LifecycleEndorsement:
    Type: ImplicitMeta
    Rule: "MAJORITY Endorsement"
  Endorsement:
    Type: ImplicitMeta
    Rule: "MAJORITY Endorsement"
```

If you do not include the new channel policies in the channel configuration, you will get the following error when you approve a chaincode definition for your organization:

```
Error: proposal failed with status: 500 - failed to invoke backing implementation of
→'ApproveChaincodeDefinitionForMyOrg': could not set defaults for chaincode_
→definition in channel mychannel: policy '/Channel/Application/Endorsement' must be_
→defined for channel 'mychannel' before chaincode operations can be attempted
```

7.2 Writing Your First Application

Note: If you're not yet familiar with the fundamental architecture of a Fabric network, you may want to visit the [Key Concepts](#) section prior to continuing.

It is also worth noting that this tutorial serves as an introduction to Fabric applications and uses simple smart contracts and applications. For a more in-depth look at Fabric applications and smart contracts, check out our [Developing Applications](#) section or the [Commercial paper tutorial](#).

This tutorial provides an introduction to how Fabric applications interact with deployed blockchain networks. The tutorial uses sample programs built using the Fabric SDKs – described in detail in the [Application](#) topic – to invoke a smart contract which queries and updates the ledger with the smart contract API – described in detail in [Smart Contract Processing](#). We will also use our sample programs and a deployed Certificate Authority to generate the X.509 certificates that an application needs to interact with a permissioned blockchain.

About Asset Transfer

This Asset Transfer (basic) sample demonstrates how to initialize a ledger with assets, query those assets, create a new asset, query a single asset based on an asset ID, update an existing asset, and transfer an asset to a new owner. It involves the following two components:

1. Sample application: which makes calls to the blockchain network, invoking transactions implemented in the chaincode (smart contract). The application is located in the following `fabric-samples` directory:

```
asset-transfer-basic/application-javascript
```

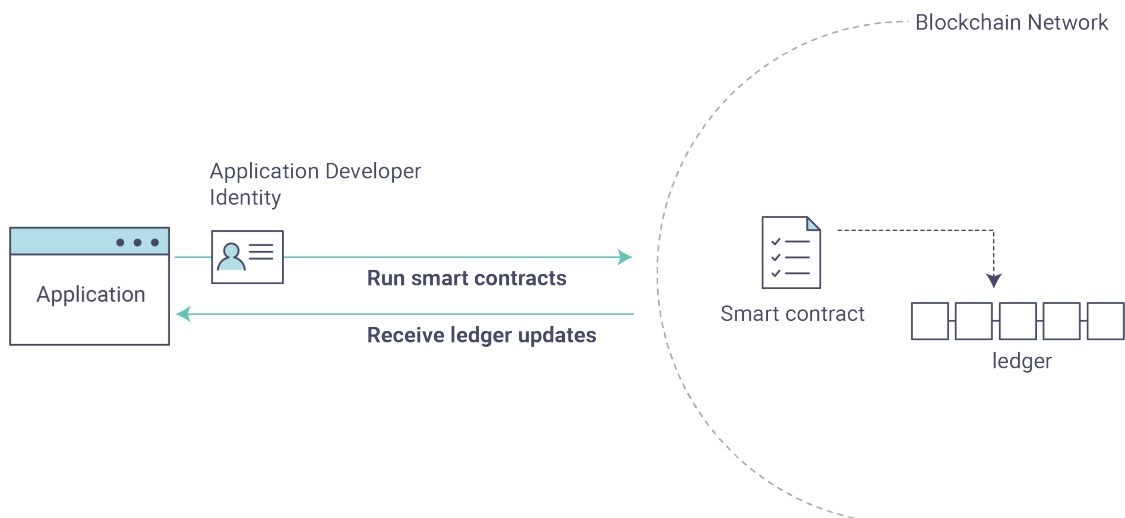
2. Smart contract itself, implementing the transactions that involve interactions with the ledger. The smart contract (chaincode) is located in the following `fabric-samples` directory:

```
asset-transfer-basic/chaincode-(javascript, java, go, typescript)
```

Please note that for the purposes of this tutorial, the terms chaincode and smart contract are used interchangeably. For this example, we will be using the javascript chaincode.

We'll go through three principle steps:

1. **Setting up a development environment.** Our application needs a network to interact with, so we'll deploy a basic network for our smart contracts and application.



2. Explore a sample smart contract. We'll inspect the sample `assetTransfer` (javascript) smart contract to learn about the transactions within it, and how they are used by an application to query and update the ledger.

3. Interact with the smart contract with a sample application. Our application will use the `assetTransfer` smart contract to create, query, and update assets on the ledger. We'll get into the code of the app and the transactions they create, including initializing the ledger with assets, querying an asset, querying a range of assets, creating a new asset, and transferring an asset to a new owner.

After completing this tutorial you should have a basic understanding of how Fabric applications and smart contracts work together to manage data on the distributed ledger of a blockchain network.

7.2.1 Before you begin

In addition to the standard *Prerequisites* for Fabric, this tutorial leverages the Hyperledger Fabric SDK for Node.js. See the Node.js SDK [README](#) for a up to date list of prerequisites.

- If you are using macOS, complete the following steps:
 1. Install [Homebrew](#).
 2. Check the Node SDK [prerequisites](#) to find out what level of Node to install.
 3. Run `brew install node` to download the latest version of node or choose a specific version, for example: `brew install node@10` according to what is supported in the prerequisites.
 4. Run `npm install`.
- If you are on Windows, you can install the [windows-build-tools](#) with npm which installs all required compilers and tooling by running the following command:

```
npm install --global windows-build-tools
```

- If you are on Linux, you need to install [Python v2.7](#), [make](#), and a C/C++ compiler toolchain such as [GCC](#). You can run the following command to install the other tools:

```
sudo apt install build-essential
```

7.2.2 Set up the blockchain network

If you've already run through *Using the Fabric test network* tutorial and have a network up and running, this tutorial will bring down your running network before bringing up a new one.

Launch the network

Note: This tutorial demonstrates the JavaScript versions of the Asset Transfer smart contract and application, but the `fabric-samples` repository also contains Go, Java and TypeScript versions of this sample smart contract. To try the Go, Java or TypeScript versions, change the `javascript` argument for `./network.sh deployCC -ccl` `javascript` below to either `go`, `java` or `typescript` and follow the instructions written to the terminal. You may use any chaincode language sample with the javascript application sample (e.g javascript application calling go chaincode functions or javascript application calling typescript chaincode functions, etc.)

Navigate to the `test-network` subdirectory within your local clone of the `fabric-samples` repository.

```
cd fabric-samples/test-network
```

If you already have a test network running, bring it down to ensure the environment is clean.

```
./network.sh down
```

Launch the Fabric test network using the `network.sh` shell script.

```
./network.sh up createChannel -c mychannel -ca
```

This command will deploy the Fabric test network with two peers, an ordering service, and three certificate authorities (Orderer, Org1, Org2). Instead of using the `cryptogen` tool, we bring up the test network using Certificate Authorities, hence the `-ca` flag. Additionally, the org admin user registration is bootstrapped when the Certificate Authority is started. In a later step, we will show how the sample application completes the admin enrollment.

Next, let's deploy the chaincode by calling the `./network.sh` script with the chaincode name and language options.

```
./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-javascript/ -
↪ccl javascript
```

Note: Behind the scenes, this script uses the chaincode lifecycle to package, install, query installed chaincode, approve chaincode for both Org1 and Org2, and finally commit the chaincode.

If the chaincode is successfully deployed, the end of the output in your terminal should look similar to below:

```
Committed chaincode definition for chaincode 'basic' on channel 'mychannel':
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc,
↪Approvals: [Org1MSP: true, Org2MSP: true]
===== Query chaincode definition successful on peer0.org2 on channel
↪'mychannel' =====
===== Chaincode initialization is not required =====
```

Sample application

Next, let's prepare the sample Asset Transfer Javascript application that will be used to interact with the deployed chaincode.

- [JavaScript application](#)

Note that the sample application is also available in Go and Java at the links below:

- [Go application](#)
- [Java application](#)

Open a new terminal, and navigate to the `application-javascript` folder.

```
cd asset-transfer-basic/application-javascript
```

This directory contains sample programs that were developed using the Fabric SDK for Node.js. Run the following command to install the application dependencies. It may take up to a minute to complete:

```
npm install
```

This process is installing the key application dependencies defined in the application's `package.json`. The most important of which is the `fabric-network` Node.js module; it enables an application to use identities, wallets, and gateways to connect to channels, submit transactions, and wait for notifications. This tutorial also uses the `fabric-ca-client` module to enroll users with their respective certificate authorities, generating a valid identity which is then used by the `fabric-network` module to interact with the blockchain network.

Once `npm install` completes, everything is in place to run the application. Let's take a look at the sample JavaScript application files we will be using in this tutorial. Run the following command to list the files in this directory:

```
ls
```

You should see the following:

```
app.js          node_modules    package.json    package-lock.json
```

Note: The first part of the following section involves communication with the Certificate Authority. You may find it useful to stream the CA logs when running the upcoming programs by opening a new terminal shell and running `docker logs -f ca_org1`.

When we started the Fabric test network back in the first step, an admin user — literally called `admin` — was created as the **registrar** for the Certificate Authority (CA). Our first step is to generate the private key, public key, and X.509 certificate for `admin` by having the application call the `enrollAdmin`. This process uses a **Certificate Signing Request (CSR)** — the private and public key are first generated locally and the public key is then sent to the CA which returns an encoded certificate for use by the application. These credentials are then stored in the wallet, allowing us to act as an administrator for the CA.

Let's run the application and then step through each of the interactions with the smart contract functions. From the `asset-transfer-basic/application-javascript` directory, run the following command:

```
node app.js
```

7.2.3 First, the application enrolls the admin user

Note: It is important to note that enrolling the admin and registering the app user are interactions that take place between the application and the Certificate Authority, not between the application and the chaincode. If you examine the chaincode in `asset-transfer-basic/chaincode-javascript/lib` you will find that the chaincode does not contain any functionality that supports enrolling the admin or registering the user.

In the sample application code below, you will see that after getting reference to the common connection profile path, making sure the connection profile exists, and specifying where to create the wallet, `enrollAdmin()` is executed and the admin credentials are generated from the Certificate Authority.

```
async function main() {
  try {
    // build an in memory object with the network configuration (also known as a
    ↳connection profile)
    const ccp = buildCCP();

    // build an instance of the fabric ca services client based on
    // the information in the network configuration
    const caClient = buildCAClient(FabricCAServices, ccp);
```

(continues on next page)

(continued from previous page)

```
// setup the wallet to hold the credentials of the application user
const wallet = await buildWallet(Wallets, walletPath);

// in a real application this would be done on an administrative flow, and only
↪once
await enrollAdmin(caClient, wallet);
```

This command stores the CA administrator's credentials in the `wallet` directory. You can find administrator's certificate and private key in the `wallet/admin.id` file.

Note: If you decide to start over by taking down the network and bringing it back up again, you will have to delete the `wallet` folder and its identities prior to re-running the javascript application or you will get an error. This happens because the Certificate Authority and its database are taken down when the test-network is taken down but the original wallet still remains in the `application-javascript` directory so it must be deleted. When you re-run the sample javascript application, a new wallet and credentials will be generated.

If you scroll back up to the beginning of the output in your terminal, it should be similar to below:

```
Wallet path: /Users/<your_username>/fabric-samples/asset-transfer-basic/application-
↪javascript/wallet
Successfully enrolled admin user and imported it into the wallet
```

Because the admin registration step is bootstrapped when the Certificate Authority is started, we only need to enroll the admin.

Note: Since the Fabric CA interactions are common across the samples, `enrollAdmin()` and the other CA related functions are included in the `fabric-samples/test-application/javascript/CAUtil.js` common utility.

As for the app user, we need the application to register and enroll the user in the next step.

7.2.4 Second, the application registers and enrolls an application user

Now that we have the administrator's credentials in a wallet, the application uses the `admin` user to register and enroll an app user which will be used to interact with the blockchain network. The section of the application code is shown below.

```
// in a real application this would be done only when a new user was required to be
↪added
// and would be part of an administrative flow
await registerAndEnrollUser(caClient, wallet, mspOrg1, org1UserId, 'org1.department1
↪');
```

Similar to the admin enrollment, this function uses a CSR to register and enroll `appUser` and store its credentials alongside those of `admin` in the wallet. We now have identities for two separate users — `admin` and `appUser` — that can be used by our application.

Scrolling further down in your terminal output, you should see confirmation of the app user registration similar to this:

```
Successfully registered and enrolled user appUser and imported it into the wallet
```

7.2.5 Third, the sample application prepares a connection to the channel and smart contract

In the prior steps, the application generated the admin and app user credentials and placed them in the wallet. If the credentials exist and have the correct permissions attributes associated with them, the sample application user will be able to call chaincode functions after getting reference to the channel name and contract name.

Note: Our connection configuration specifies only the peer from your own Org. We tell node client sdk to use the service discovery (running on the peer), which fetches other peers that are currently online, metadata like relevant endorsement policies and any static information it would have otherwise needed to communicate with the rest of the nodes. The `asLocalhost` set to `true` tells it to connect as localhost, since our client is running on same network as the other fabric nodes. In deployments where you are not running the client on the same network as the other fabric nodes, the `asLocalhost` option would be set to `false`.

You will notice that in the following lines of application code, the application is getting reference to the Contract using the contract name and channel name via Gateway:

```
// Create a new gateway instance for interacting with the fabric network.
// In a real application this would be done as the backend server session is setup for
// a user that has been verified.
const gateway = new Gateway();

try {
  // setup the gateway instance
  // The user will now be able to create connections to the fabric network and be_
  ↪able to
  // submit transactions and query. All transactions submitted by this gateway will be
  // signed by this user using the credentials stored in the wallet.
  await gateway.connect(ccp, {
    wallet,
    identity: userId,
    discovery: {enabled: true, asLocalhost: true} // using asLocalhost as this_
  ↪gateway is using a fabric network deployed locally
  });

  // Build a network instance based on the channel where the smart contract is_
  ↪deployed
  const network = await gateway.getNetwork(channelName);

  // Get the contract from the network.
  const contract = network.getContract(chaincodeName);
```

When a chaincode package includes multiple smart contracts, on the `getContract()` API you can specify both the name of the chaincode package and a specific smart contract to target. For example:

```
const contract = await network.getContract('chaincodeName', 'smartContractName');
```

7.2.6 Fourth, the application initializes the ledger with some sample data

Now that we are at the point where we are actually having the sample application submit transactions, let's go through them in sequence. The application code snippets and invoked chaincode snippets are provided for each called function, as well as the terminal output.

The `submitTransaction()` function is used to invoke the chaincode `InitLedger` function to populate the ledger with some sample data. Under the covers, the `submitTransaction()` function will use service discovery to find a set of required endorsing peers for the chaincode, invoke the chaincode on the required number of peers, gather the chaincode endorsed results from those peers, and finally submit the transaction to the ordering service.

Sample application 'InitLedger' call

```
// Initialize a set of asset data on the channel using the chaincode 'InitLedger'
↳function.
// This type of transaction would only be run once by an application the first time
↳it was started after it
// deployed the first time. Any updates to the chaincode deployed later would likely
↳not need to run
// an "init" type function.
console.log('\n--> Submit Transaction: InitLedger, function creates the initial set
↳of assets on the ledger');
await contract.submitTransaction('InitLedger');
console.log('*** Result: committed');
```

Chaincode 'InitLedger' function

```
async InitLedger(ctx) {
  const assets = [
    {
      ID: 'asset1',
      Color: 'blue',
      Size: 5,
      Owner: 'Tomoko',
      AppraisedValue: 300,
    },
    {
      ID: 'asset2',
      Color: 'red',
      Size: 5,
      Owner: 'Brad',
      AppraisedValue: 400,
    },
    {
      ID: 'asset3',
      Color: 'green',
      Size: 10,
      Owner: 'Jin Soo',
      AppraisedValue: 500,
    },
    {
      ID: 'asset4',
      Color: 'yellow',
      Size: 10,
      Owner: 'Max',
      AppraisedValue: 600,
    },
    {
      ID: 'asset5',
      Color: 'black',
      Size: 15,
      Owner: 'Adriana',
      AppraisedValue: 700,
    },
  ],
```

(continues on next page)

(continued from previous page)

```

    {
      ID: 'asset6',
      Color: 'white',
      Size: 15,
      Owner: 'Michel',
      AppraisedValue: 800,
    },
  ];

  for (const asset of assets) {
    asset.docType = 'asset';
    await ctx.stub.putState(asset.ID, Buffer.from(JSON.stringify(asset)));
    console.info(`Asset ${asset.ID} initialized`);
  }
}

```

The terminal output entry should look similar to below:

```

Submit Transaction: InitLedger, function creates the initial set of assets on the
↳ ledger

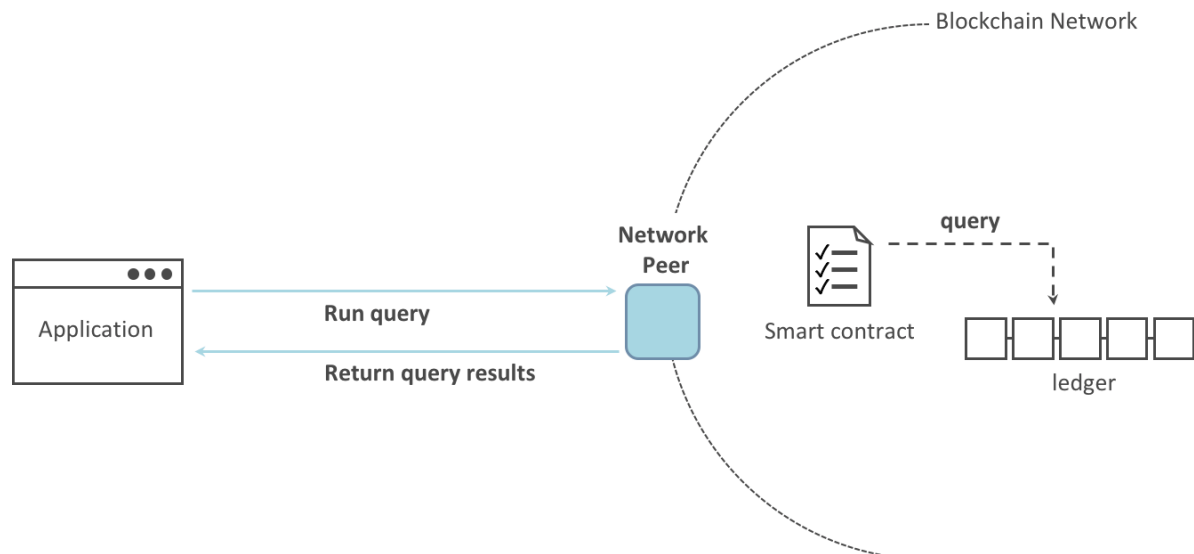
```

7.2.7 Fifth, the application invokes each of the chaincode functions

First, a word about querying the ledger.

Each peer in a blockchain network hosts a copy of the [ledger](#). An application program can view the most recent data from the ledger using read-only invocations of a smart contract running on your peers called a query.

Here is a simplified representation of how a query works:



The most common queries involve the current values of data in the ledger – its [world state](#). The world state is represented as a set of key-value pairs, and applications can query data for a single key or multiple keys. Moreover, you can use complex queries to read the data on the ledger when you use CouchDB as your state database and model

your data in JSON. This can be very helpful when looking for all assets that match certain keywords with particular values; all assets with a particular owner, for example.

Below, the sample application is just getting all the assets that we populated in the prior step when we initialized the ledger with data. The `evaluateTransaction()` function is used when you'd like to query a single peer, without submitting a transaction to the ordering service.

Sample application 'GetAllAssets' call

```
// Let's try a query type operation (function).
// This will be sent to just one peer and the results will be shown.
console.log('\n--> Evaluate Transaction: GetAllAssets, function returns all the
↳current assets on the ledger');
let result = await contract.evaluateTransaction('GetAllAssets');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);
```

Chaincode 'GetAllAssets' function

```
// GetAllAssets returns all assets found in the world state.
async GetAllAssets(ctx) {
  const allResults = [];
  // range query with empty string for startKey and endKey does an open-ended
  ↳query of all assets in the chaincode namespace.
  const iterator = await ctx.stub.getStateByRange('', '');
  let result = await iterator.next();
  while (!result.done) {
    const strValue = Buffer.from(result.value.value.toString()).toString('utf8');
    let record;
    try {
      record = JSON.parse(strValue);
    } catch (err) {
      console.log(err);
      record = strValue;
    }
    allResults.push({ Key: result.value.key, Record: record });
    result = await iterator.next();
  }
  return JSON.stringify(allResults);
}
```

The terminal output should look like this:

```
Evaluate Transaction: GetAllAssets, function returns all the current assets on the
↳ledger
Result: [
  {
    "Key": "asset1",
    "Record": {
      "ID": "asset1",
      "Color": "blue",
      "Size": 5,
      "Owner": "Tomoko",
      "AppraisedValue": 300,
      "docType": "asset"
    }
  },
  {
    "Key": "asset2",
```

(continues on next page)

(continued from previous page)

```
"Record": {
  "ID": "asset2",
  "Color": "red",
  "Size": 5,
  "Owner": "Brad",
  "AppraisedValue": 400,
  "docType": "asset"
},
{
  "Key": "asset3",
  "Record": {
    "ID": "asset3",
    "Color": "green",
    "Size": 10,
    "Owner": "Jin Soo",
    "AppraisedValue": 500,
    "docType": "asset"
  }
},
{
  "Key": "asset4",
  "Record": {
    "ID": "asset4",
    "Color": "yellow",
    "Size": 10,
    "Owner": "Max",
    "AppraisedValue": 600,
    "docType": "asset"
  }
},
{
  "Key": "asset5",
  "Record": {
    "ID": "asset5",
    "Color": "black",
    "Size": 15,
    "Owner": "Adriana",
    "AppraisedValue": 700,
    "docType": "asset"
  }
},
{
  "Key": "asset6",
  "Record": {
    "ID": "asset6",
    "Color": "white",
    "Size": 15,
    "Owner": "Michel",
    "AppraisedValue": 800,
    "docType": "asset"
  }
}
]
```

Next, the sample application submits a transaction to create 'asset13'.

Sample application 'CreateAsset' call

```
// Now let's try to submit a transaction.
// This will be sent to both peers and if both peers endorse the transaction, the
↳endorsed proposal will be sent
// to the orderer to be committed by each of the peer's to the channel ledger.
console.log('\n--> Submit Transaction: CreateAsset, creates new asset with ID, color,
↳owner, size, and appraisedValue arguments');
await contract.submitTransaction('CreateAsset', 'asset13', 'yellow', '5', 'Tom', '1300
↳');
console.log('*** Result: committed');
```

Chaincode 'CreateAsset' function

```
// CreateAsset issues a new asset to the world state with given details.
async CreateAsset(ctx, id, color, size, owner, appraisedValue) {
  const asset = {
    ID: id,
    Color: color,
    Size: size,
    Owner: owner,
    AppraisedValue: appraisedValue,
  };
  return ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
}
```

Terminal output:

```
Submit Transaction: CreateAsset, creates new asset with ID, color, owner, size, and
↳appraisedValue arguments
```

Note: In the application and chaincode snippets above, it is important to note that the sample application submits the 'CreateAsset' transaction with the same type and number of arguments the chaincode is expecting, and in the correct sequence. In this case, the transaction name and correctly sequenced arguments are: 'CreateAsset', 'asset13', 'yellow', '5', 'Tom', '1300' because the corresponding chaincode CreateAsset is expecting the correct sequence and type of arguments that define the asset object: sequence: ID, Color, Size, Owner, and AppraisedValue

type: ID (string), Color (string), Size (int), Owner (string), AppraisedValue (int).

The sample application then evaluates a query for 'asset13'.

Sample application 'ReadAsset' call

```
console.log('\n--> Evaluate Transaction: ReadAsset, function returns an
↳asset with a given assetID');
result = await contract.evaluateTransaction('ReadAsset', 'asset13');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);
```

Chaincode 'ReadAsset' function

```
// ReadAsset returns the asset stored in the world state with given id.
async ReadAsset(ctx, id) {
  const assetJSON = await ctx.stub.getState(id); // get the asset from
↳chaincode state
  if (!assetJSON || assetJSON.length === 0) {
    throw new Error(`The asset ${id} does not exist`);
  }
}
```

(continues on next page)

(continued from previous page)

```

    return assetJSON.toString();
}

```

Terminal output:

```

Evaluate Transaction: ReadAsset, function returns an asset with a given_
↪assetID
Result: {
  "ID": "asset13",
  "Color": "yellow",
  "Size": "5",
  "Owner": "Tom",
  "AppraisedValue": "1300"
}

```

In the next part of the sequence, the sample application evaluates to see if `asset1` exists, which will return a boolean value of `true`, because we populated the ledger with `asset1` when we initialized the ledger with assets. You may recall that the original appraised value of `asset1` was 300. The application then submits a transaction to update `asset1` with a new appraised value, and then immediately evaluates to read `asset1` from the ledger to show the new appraised value of 350.

Sample application 'AssetExists', 'UpdateAsset', and 'ReadAsset' calls

```

console.log('\n--> Evaluate Transaction: AssetExists, function returns "true" if an_
↪asset with given assetID exist');
result = await contract.evaluateTransaction('AssetExists', 'asset1');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);

console.log('\n--> Submit Transaction: UpdateAsset asset1, change the appraisedValue_
↪to 350');
await contract.submitTransaction('UpdateAsset', 'asset1', 'blue', '5', 'Tomoko', '350'
↪);
console.log(`*** Result: committed`);

console.log('\n--> Evaluate Transaction: ReadAsset, function returns "asset1"_
↪attributes');
result = await contract.evaluateTransaction('ReadAsset', 'asset1');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);

```

Chaincode 'AssetExists', 'UpdateAsset', and 'ReadAsset' functions

```

// AssetExists returns true when asset with given ID exists in world state.
async AssetExists(ctx, id) {
    const assetJSON = await ctx.stub.getState(id);
    return assetJSON && assetJSON.length > 0;
}

// UpdateAsset updates an existing asset in the world state with provided parameters.
async UpdateAsset(ctx, id, color, size, owner, appraisedValue) {
    const exists = await this.AssetExists(ctx, id);
    if (!exists) {
        throw new Error(`The asset ${id} does not exist`);
    }

    // overwriting original asset with new asset
    const updatedAsset = {
        ID: id,
        Color: color,

```

(continues on next page)

(continued from previous page)

```

        Size: size,
        Owner: owner,
        AppraisedValue: appraisedValue,
    };
    return ctx.stub.putState(id, Buffer.from(JSON.stringify(updatedAsset)));
}
// ReadAsset returns the asset stored in the world state with given id.
async ReadAsset(ctx, id) {
    const assetJSON = await ctx.stub.getState(id); // get the asset from chaincode
    ↪state
    if (!assetJSON || assetJSON.length === 0) {
        throw new Error(`The asset ${id} does not exist`);
    }
    return assetJSON.toString();
}

```

Terminal Output:

```

Evaluate Transaction: AssetExists, function returns "true" if an asset with given
↪assetID exist
Result: true

Submit Transaction: UpdateAsset asset1, change the appraisedValue to 350

Evaluate Transaction: ReadAsset, function returns "asset1" attributes
Result: {
  "ID": "asset1",
  "Color": "blue",
  "Size": "5",
  "Owner": "Tomoko",
  "AppraisedValue": "350"
}

```

In this part of the sequence, the sample application attempts to submit an 'UpdateAsset' transaction for an asset that we know does not exist (asset70). We expect that we will get an error because you cannot update an asset that does not exist, which is why it is a good idea to check if an asset exists prior to attempting an asset update or deletion.

Sample application 'UpdateAsset' call

```

try {
    // How about we try a transactions where the executing chaincode throws an error
    // Notice how the submitTransaction will throw an error containing the error thrown
    ↪by the chaincode
    console.log('\n--> Submit Transaction: UpdateAsset asset70, asset70 does not exist
    ↪and should return an error');
    await contract.submitTransaction('UpdateAsset', 'asset70', 'blue', '5', 'Tomoko',
    ↪'300');
    console.log('***** FAILED to return an error');
} catch (error) {
    console.log(`*** Successfully caught the error: \n    ${error}`);
}

```

Chaincode 'UpdateAsset' function

```

// UpdateAsset updates an existing asset in the world state with provided parameters.
async UpdateAsset(ctx, id, color, size, owner, appraisedValue) {
    const exists = await this.AssetExists(ctx, id);

```

(continues on next page)

(continued from previous page)

```

    if (!exists) {
        throw new Error(`The asset ${id} does not exist`);
    }

    // overwriting original asset with new asset
    const updatedAsset = {
        ID: id,
        Color: color,
        Size: size,
        Owner: owner,
        AppraisedValue: appraisedValue,
    };
    return ctx.stub.putState(id, Buffer.from(JSON.stringify(updatedAsset)));
}

```

Terminal output:

```

Submit Transaction: UpdateAsset asset70
2020-08-02T11:12:12.322Z - error: [Transaction]: Error: No valid responses from any
↳ peers. Errors:
  peer=peer0.org1.example.com:7051, status=500, message=error in simulation:
↳ transaction returned with failure: Error: The asset asset70 does not exist
  peer=peer0.org2.example.com:9051, status=500, message=error in simulation:
↳ transaction returned with failure: Error: The asset asset70 does not exist
Expected an error on UpdateAsset of non-existing Asset: Error: No valid responses
↳ from any peers. Errors:
  peer=peer0.org1.example.com:7051, status=500, message=error in simulation:
↳ transaction returned with failure: Error: The asset asset70 does not exist
  peer=peer0.org2.example.com:9051, status=500, message=error in simulation:
↳ transaction returned with failure: Error: The asset asset70 does not exist

```

In this final part of the sample application transaction sequence, the application submits a transaction to transfer an existing asset to a new owner and then reads the asset back from the ledger to display the new owner Tom.

Sample application 'TransferAsset', and 'ReadAsset' calls

```

console.log('\n--> Submit Transaction: TransferAsset asset1, transfer to new owner of
↳ Tom');
await contract.submitTransaction('TransferAsset', 'asset1', 'Tom');
console.log('*** Result: committed');

console.log('\n--> Evaluate Transaction: ReadAsset, function returns "asset1"
↳ attributes');
result = await contract.evaluateTransaction('ReadAsset', 'asset1');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);

```

Chaincode 'TransferAsset', and 'ReadAsset' functions

```

// TransferAsset updates the owner field of asset with given id in the world state.
async TransferAsset(ctx, id, newOwner) {
    const assetString = await this.ReadAsset(ctx, id);
    const asset = JSON.parse(assetString);
    asset.Owner = newOwner;
    return ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
}

// ReadAsset returns the asset stored in the world state with given id.
async ReadAsset(ctx, id) {

```

(continues on next page)

(continued from previous page)

```

    const assetJSON = await ctx.stub.getState(id); // get the asset from chaincode_
↪state
    if (!assetJSON || assetJSON.length === 0) {
        throw new Error(`The asset ${id} does not exist`);
    }
    return assetJSON.toString();
}

```

Terminal output:

```

Submit Transaction: TransferAsset asset1, transfer to new owner of Tom
Evaluate Transaction: ReadAsset, function returns "asset1" attributes
Result: {
  "ID": "asset1",
  "Color": "blue",
  "Size": "5",
  "Owner": "Tom",
  "AppraisedValue": "350"
}

```

7.2.8 A closer look

Let's take a closer look at how the sample javascript application uses the APIs provided by the [Fabric Node SDK](#) to interact with our Fabric network. Use an editor (e.g. atom or visual studio) to open `app.js` located in the `asset-transfer-basic/application-javascript` directory.

The application starts by bringing in scope two key classes from the `fabric-network` module; `Wallets` and `Gateway`. These classes will be used to locate the `appUser` identity in the wallet, and use it to connect to the network:

```
const { Gateway, Wallets } = require('fabric-network');
```

First, the program sets up the gateway connection with the `userId` stored in the wallet and specifies discovery options.

```

// setup the gateway instance
// The user will now be able to create connections to the fabric network and be able_
↪to
// submit transactions and query. All transactions submitted by this gateway will be
// signed by this user using the credentials stored in the wallet.
await gateway.connect(ccp, {
  wallet,
  identity: userId,
  discovery: {enabled: true, asLocalhost: true} // using asLocalhost as this gateway_
↪is using a fabric network deployed locally
});

```

Note at the top of the sample application code we require external utility files to build the `CAClient`, `registerUser`, `enrolAdmin`, `buildCCP` (common connection profile), and `buildWallet`. These utility programs are located in `AppUtil.js` in the `test-application/javascript` directory.

In `AppUtil.js`, `ccpPath` describes the path to the connection profile that our application will use to connect to our network. The connection profile was loaded from inside the `fabric-samples/test-network` directory and parsed as a JSON file:

```
const ccpPath = path.resolve(__dirname, '..', '..', 'test-network', 'organizations',
↪ 'peerOrganizations', 'org1.example.com', 'connection-org1.json');
```

If you'd like to understand more about the structure of a connection profile, and how it defines the network, check out the [connection profile](#) topic.

A network can be divided into multiple channels, and the next important line of code connects the application to a particular channel within the network, `mychannel`, where our smart contract was deployed. Note that we assigned constants near the top of the sample application to account for the channel name and the contract name:

```
const channelName = 'mychannel';
const chaincodeName = 'basic';
```

```
const network = await gateway.getNetwork(channelName);
```

Within this channel, we can access the asset-transfer ('basic') smart contract to interact with the ledger:

```
const contract = network.getContract(chaincodeName);
```

Within asset-transfer ('basic') there are many different **transactions**, and our application initially uses the `InitLedger` transaction to populate the ledger world state with data:

```
await contract.submitTransaction('InitLedger');
```

The `evaluateTransaction` method represents one of the simplest interactions with a smart contract in blockchain network. It simply picks a peer defined in the connection profile and sends the request to it, where it is evaluated. The smart contract queries the assets on the peer's copy of the ledger and returns the result to the application. This interaction does not result in an update of the ledger.

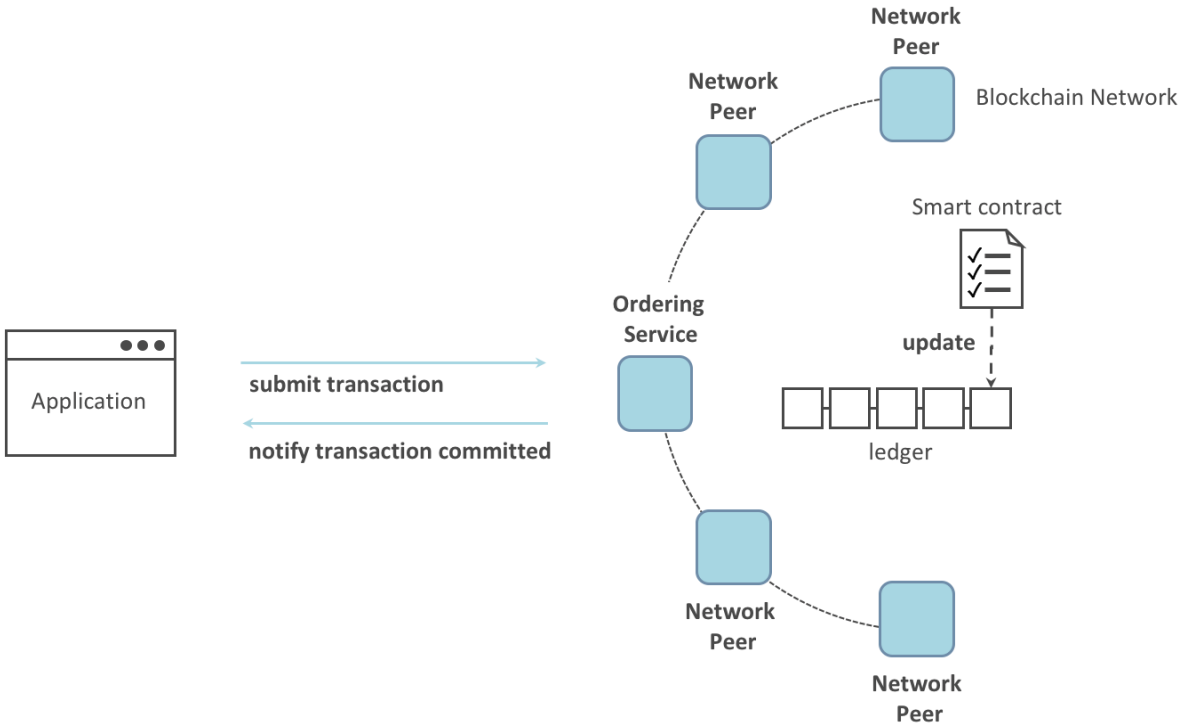
`submitTransaction` is much more sophisticated than `evaluateTransaction`. Rather than interacting with a single peer, the SDK will send the `submitTransaction` proposal to every required organization's peer in the blockchain network based on the chaincode's endorsement policy. Each of these peers will execute the requested smart contract using this proposal, to generate a transaction response which it endorses (signs) and returns to the SDK. The SDK collects all the endorsed transaction responses into a single transaction, which it then submits to the orderer. The orderer collects and sequences transactions from various application clients into a block of transactions. These blocks are distributed to every peer in the network, where every transaction is validated and committed. Finally, the SDK is notified via an event, allowing it to return control to the application.

Note: `submitTransaction` includes an event listener that checks to make sure the transaction has been validated and committed to the ledger. Applications should either utilize a commit listener, or leverage an API like `submitTransaction` that does this for you. Without doing this, your transaction may not have been successfully ordered, validated, and committed to the ledger.

`submitTransaction` does all this for the application! The process by which the application, smart contract, peers and ordering service work together to keep the ledger consistent across the network is called consensus, and it is explained in detail in this [section](#).

7.2.9 Updating the ledger

From an application perspective, updating the ledger is simple. An application submits a transaction to the blockchain network, and when it has been validated and committed, the application receives a notification that the transaction has been successful. Behind the scenes, this involves the process of consensus whereby the different components of the blockchain network work together to ensure that every proposed update to the ledger is valid and performed in an agreed and consistent order.



7.2.10 The asset-transfer ('basic') smart contract

The smart contract sample is available in the following languages:

- Golang
- Java
- JavaScript
- Typescript

7.2.11 Clean up

When you are finished using the asset-transfer sample, you can bring down the test network using `network.sh` script.

```
./network.sh down
```

This command will bring down the CAs, peers, and ordering node of the network that we created. Note that all of the data on the ledger will be lost. If you want to go through the tutorial again, you will start from a clean initial state.

7.2.12 Summary

Now that we've seen how the sample application and chaincode are written and how they interact with each other, you should have a pretty good sense of how applications interact with a blockchain network using a smart contract to query or update the ledger. You've seen the basics of the roles smart contracts, APIs, and the SDK play in queries and updates and you should have a feel for how different kinds of applications could be used to perform other business tasks and operations.

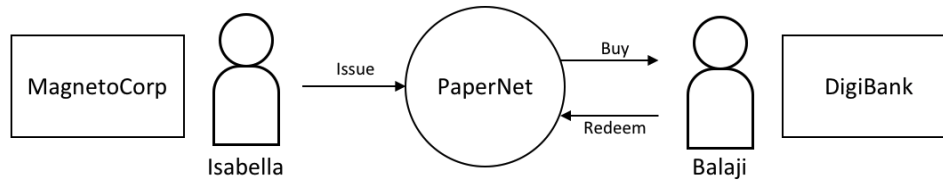
7.2.13 Additional resources

As we said in the introduction, we have a whole section on *Developing Applications* that includes in-depth information on smart contracts, process and data design, a tutorial using a more in-depth Commercial Paper [tutorial](#) and a large amount of other material relating to the development of applications.

7.3 Commercial paper tutorial

Audience: Architects, application and smart contract developers, administrators

This tutorial will show you how to install and use a commercial paper sample application and smart contract. It is a task-oriented topic, so it emphasizes procedures above concepts. When you'd like to understand the concepts in more detail, you can read the *Developing Applications* topic.



In this tutorial two organizations, MagnetoCorp and DigiBank, trade commercial paper with each other using PaperNet, a Hyperledger Fabric blockchain network.

Once you've set up the test network, you'll act as Isabella, an employee of MagnetoCorp, who will issue a commercial paper on its behalf. You'll then switch roles to take the role of Balaji, an employee of DigiBank, who will buy this commercial paper, hold it for a period of time, and then redeem it with MagnetoCorp for a small profit.

You'll act as a developer, end user, and administrator, each in different organizations, performing the following steps designed to help you understand what it's like to collaborate as two different organizations working independently, but according to mutually agreed rules in a Hyperledger Fabric network.

- *Set up machine and download samples*
- *Create the network*
- *Examine the commercial paper smart contract*
- *Deploy the smart contract to the channel* by approving the chaincode definition as MagnetoCorp and Digibank.
- Understand the structure of a MagnetoCorp *application*, including its *dependencies*
- Configure and use a *wallet and identities*
- Run a MagnetoCorp application to *issue a commercial paper*
- Understand how DigiBank uses the smart contract in their *applications*
- As Digibank, run applications that *buy* and *redeem* commercial paper

This tutorial has been tested on MacOS and Ubuntu, and should work on other Linux distributions. A Windows version is under development.

7.3.1 Prerequisites

Before you start, you must install some prerequisite technology required by the tutorial. We've kept these to a minimum so that you can get going quickly.

You **must** have the following technologies installed:

- **Node** The Node.js SDK README contains the up to date list of prerequisites.

You **will** find it helpful to install the following technologies:

- A source code editor, such as **Visual Studio Code** version 1.28, or higher. VS Code will help you develop and test your application and smart contract. Install VS Code [here](#).

Many excellent code editors are available including [Atom](#), [Sublime Text](#) and [Brackets](#).

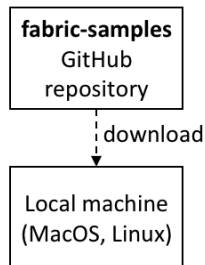
You **may** find it helpful to install the following technologies as you become more experienced with application and smart contract development. There's no requirement to install these when you first run the tutorial:

- **Node Version Manager**. NVM helps you easily switch between different versions of node – it can be really helpful if you're working on multiple projects at the same time. Install NVM [here](#).

7.3.2 Download samples

The commercial paper tutorial is one of the samples in the `fabric-samples` repository. Before you begin this tutorial, ensure that you have followed the instructions to install the Fabric [Prerequisites](#) and [Download the Samples, Binaries and Docker Images](#). When you are finished, you will have cloned the `fabric-samples` repository that contains the tutorial scripts, smart contract, and application files.

`https://github.com/hyperledger/fabric-samples`



Download the `fabric-samples` GitHub repository to your local machine.

After downloading, feel free to examine the directory structure of `fabric-samples`:

```

$ cd fabric-samples
$ ls

CODEOWNERS          SECURITY.md          first-network
CODE_OF_CONDUCT.md  chaincode           high-throughput
CONTRIBUTING.md    chaincode-docker-devmode  interest_rate_swaps
LICENSE             ci                  off_chain_data
MAINTAINERS.md      commercial-paper      test-network
README.md           fabcar
  
```

Notice the `commercial-paper` directory – that's where our sample is located!

You've now completed the first stage of the tutorial! As you proceed, you'll open multiple command windows for different users and components. For example:

- To show peer, orderer and CA log output from your network.

- To approve the chaincode as an administrator from MagnetoCorp and as an administrator from DigiBank.
- To run applications on behalf of Isabella and Balaji, who will use the smart contract to trade commercial paper with each other.

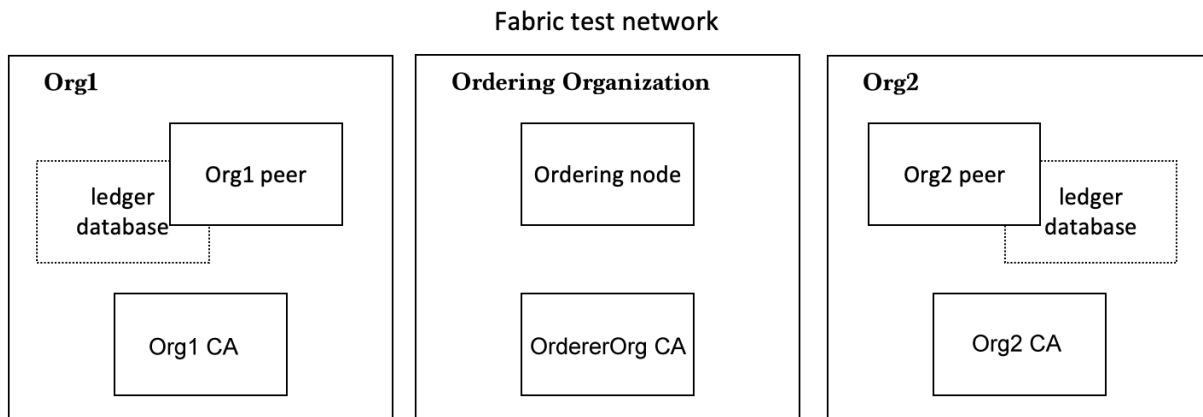
We'll make it clear when you should run a command from particular command window; for example:

```
(isabella)$ ls
```

indicates that you should run the `ls` command from Isabella's window.

7.3.3 Create the network

This tutorial will deploy a smart contract using the Fabric test network. The test network consists of two peer organizations and one ordering organization. The two peer organizations operate one peer each, while the ordering organization operates a single node Raft ordering service. We will also use the test network to create a single channel named `mychannel` that both peer organizations will be members of.



The Fabric test network is comprised of two peer organizations, Org1 and Org2, and one ordering organization. Each component runs as a Docker container.

Each organization runs their own Certificate Authority. The two peers, the [state databases](#), the ordering service node, and each organization CA each run in their own Docker container. In production environments, organizations typically use existing CAs that are shared with other systems; they're not dedicated to the Fabric network.

The two organizations of the test network allow us to interact with a blockchain ledger as two organizations that operate separate peers. In this tutorial, we will operate Org1 of the test network as DigiBank and Org2 as MagnetoCorp.

You can start the test network and create the channel with a script provided in the commercial paper directory. Change to the `commercial-paper` directory in the `fabric-samples`:

```
cd fabric-samples/commercial-paper
```

Then use the script to start the test network:

```
./network-starter.sh
```

While the script is running, you will see logs of the test network being deployed. When the script is complete, you can use the `docker ps` command to see the Fabric nodes running on your local machine:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
↪CREATED	STATUS	PORTS
↪ NAMES		
a86f50ca1907	hyperledger/fabric-peer:latest	"peer node start"
↪About a minute ago	Up About a minute	7051/tcp, 0.0.0.0:9051->9051/tcp
↪ peer0.org2.example.com		
77d0fcaee61b	hyperledger/fabric-peer:latest	"peer node start"
↪About a minute ago	Up About a minute	0.0.0.0:7051->7051/tcp
↪ peer0.org1.example.com		
7eb5f64bfe5f	hyperledger/fabric-couchdb	"tini -- /docker-ent..."
↪About a minute ago	Up About a minute	4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp
↪ couchdb0		
2438df719f57	hyperledger/fabric-couchdb	"tini -- /docker-ent..."
↪About a minute ago	Up About a minute	4369/tcp, 9100/tcp, 0.0.0.0:7984->5984/tcp
↪ couchdb1		
03373d116c5a	hyperledger/fabric-orderer:latest	"orderer"
↪About a minute ago	Up About a minute	0.0.0.0:7050->7050/tcp
↪ orderer.example.com		
6b4d87f65909	hyperledger/fabric-ca:latest	"sh -c 'fabric-ca-se..."
↪About a minute ago	Up About a minute	7054/tcp, 0.0.0.0:8054->8054/tcp
↪ ca_org2		
7b01f5454832	hyperledger/fabric-ca:latest	"sh -c 'fabric-ca-se..."
↪About a minute ago	Up About a minute	7054/tcp, 0.0.0.0:9054->9054/tcp
↪ ca_orderer		
87aef6062f23	hyperledger/fabric-ca:latest	"sh -c 'fabric-ca-se..."
↪About a minute ago	Up About a minute	0.0.0.0:7054->7054/tcp
↪ ca_org1		

See if you can map these containers to the nodes of the test network (you may need to horizontally scroll to locate the information):

- The Org1 peer, `peer0.org1.example.com`, is running in container `a86f50ca1907`
- The Org2 peer, `peer0.org2.example.com`, is running in container `77d0fcaee61b`
- The CouchDB database for the Org1 peer, `couchdb0`, is running in container `7eb5f64bfe5f`
- The CouchDB database for the Org2 peer, `couchdb1`, is running in container `2438df719f57`
- The Ordering node, `orderer.example.com`, is running in container `03373d116c5a`
- The Org1 CA, `ca_org1`, is running in container `87aef6062f23`
- The Org2 CA, `ca_org2`, is running in container `6b4d87f65909`
- The Ordering Org CA, `ca_orderer`, is running in container `7b01f5454832`

These containers all form a [Docker network](#) called `fabric_test`. You can view the network with the `docker network` command:

```
$ docker network inspect fabric_test
```

```
[
  {
    "Name": "net_test",
    "Id": "f4c9712139311004b8f7acc14e9f90170c5dcfd8cdd06303c7b074624b44dc9f",
    "Created": "2020-04-28T22:45:38.525016Z",
    "Containers": {
```

(continues on next page)

(continued from previous page)

```

    "03373d116c5abf2ca94f6f00df98bb74f89037f511d6490de4a217ed8b6fbcd0": {
      "Name": "orderer.example.com",
      "EndpointID":
↪ "0eed871a2aaf9a5dbcf7896aa3c0f53cc61f57b3417d36c56747033fd9f81972",
      "MacAddress": "02:42:c0:a8:70:05",
      "IPv4Address": "192.168.112.5/20",
      "IPv6Address": ""
    },
    "2438df719f57a597de592cfc76db30013adfdcfa0cec5b375f6b7259f67baff8": {
      "Name": "couchdb1",
      "EndpointID":
↪ "52527fb450a7c80ea509cb571d18e2196a95c630d0f41913de8ed5abbd68993d",
      "MacAddress": "02:42:c0:a8:70:06",
      "IPv4Address": "192.168.112.6/20",
      "IPv6Address": ""
    },
    "6b4d87f65909afd335d7acfe6d79308d6e4b27441b25a829379516e4c7335b88": {
      "Name": "ca_org2",
      "EndpointID":
↪ "1cc322a995880d76e1dd1f37ddf9c43f86997156124d4ecbb0eba9f833218407",
      "MacAddress": "02:42:c0:a8:70:04",
      "IPv4Address": "192.168.112.4/20",
      "IPv6Address": ""
    },
    "77d0fcaee61b8fff43d3331073ab9ce36561a90370b9ef3f77c663c8434e642": {
      "Name": "peer0.org1.example.com",
      "EndpointID":
↪ "05d0d34569eee412e28313ba7ee06875a68408257dc47e64c0f4f5ef4a9dc491",
      "MacAddress": "02:42:c0:a8:70:08",
      "IPv4Address": "192.168.112.8/20",
      "IPv6Address": ""
    },
    "7b01f5454832984fcd9650f05b4affce97319f661710705e6381dfb76cd99fdb": {
      "Name": "ca_orderer",
      "EndpointID":
↪ "057390288a424f49d6e9d6f788049b1e18aa28bccd56d860b2be8ceb8173ef74",
      "MacAddress": "02:42:c0:a8:70:02",
      "IPv4Address": "192.168.112.2/20",
      "IPv6Address": ""
    },
    "7eb5f64bfe5f20701aae8a6660815c4e3a81c3834b71f9e59a62fb99bed1afc7": {
      "Name": "couchdb0",
      "EndpointID":
↪ "bfe740be15ec9dab7baf3806964e6b1f0b67032ce1b7ae26ac7844a1b422ddc4",
      "MacAddress": "02:42:c0:a8:70:07",
      "IPv4Address": "192.168.112.7/20",
      "IPv6Address": ""
    },
    "87aef6062f2324889074cda80fec8fe014d844e10085827f380a91eea4ccdd74": {
      "Name": "ca_org1",
      "EndpointID":
↪ "a740090d33ca94dd7c6aaf14a79e1cb35109b549ee291c80195beccc901b16b7",
      "MacAddress": "02:42:c0:a8:70:03",
      "IPv4Address": "192.168.112.3/20",
      "IPv6Address": ""
    },
    "a86f50ca19079f59552e8674932edd02f7f9af93ded14db3b4c404fd6b1abe9c": {

```

(continues on next page)

(continued from previous page)

```

        "Name": "peer0.org2.example.com",
        "EndpointID":
↪ "6e56772b4783b1879a06f86901786fed1c307966b72475ce4631405ba8bca79a",
        "MacAddress": "02:42:c0:a8:70:09",
        "IPv4Address": "192.168.112.9/20",
        "IPv6Address": ""
    },
    },
    "Options": {},
    "Labels": {}
}
]

```

See how the eight containers use different IP addresses, while being part of a single Docker network. (We've abbreviated the output for clarity.)

Because we are operating the test network as DigiBank and MagnetoCorp, `peer0.org1.example.com` will belong to the DigiBank organization while `peer0.org2.example.com` will be operated by MagnetoCorp. Now that the test network is up and running, we can refer to our network as PaperNet from this point forward.

To recap: you've downloaded the Hyperledger Fabric samples repository from GitHub and you've got a Fabric network running on your local machine. Let's now start to play the role of MagnetoCorp, who wishes to issue and trade commercial paper.

7.3.4 Monitor the network as MagnetoCorp

The commercial paper tutorial allows you to act as two organizations by providing two separate folders for DigiBank and MagnetoCorp. The two folders contain the smart contracts and application files for each organization. Because the two organizations have different roles in the trading of the commercial paper, the application files are different for each organization. Open a new window in the `fabric-samples` repository and use the following command to change into the MagnetoCorp directory:

```
cd commercial-paper/organization/magnetocorp
```

The first thing we are going to do as MagnetoCorp is monitor the components of PaperNet. An administrator can view the aggregated output from a set of Docker containers using the `logspout` tool. The tool collects the different output streams into one place, making it easy to see what's happening from a single window. This can be really helpful for administrators when installing smart contracts or for developers when invoking smart contracts, for example.

In the MagnetoCorp directory, run the following command to run the `monitordocker.sh` script and start the `logspout` tool for the containers associated with PaperNet running on `net_test`:

```

(magnetocorp admin)$ ./configuration/cli/monitordocker.sh net_test
...
latest: Pulling from gliderlabs/logspout
4fe2ade4980c: Pull complete
decca452f519: Pull complete
(...)
Starting monitoring on all containers on the network net_test
b7f3586e5d0233de5a454df369b8eadab0613886fc9877529587345fc01a3582

```

Note that you can pass a port number to the above command if the default port in `monitordocker.sh` is already in use.

```
(magnetocorp admin)$ ./monitordocker.sh net_test <port_number>
```

This window will now show output from the Docker containers for the remainder of the tutorial, so go ahead and open another command window. The next thing we will do is examine the smart contract that MagnetoCorp will use to issue to the commercial paper.

7.3.5 Examine the commercial paper smart contract

issue, buy and redeem are the three functions at the heart of the commercial paper smart contract. It is used by applications to submit transactions which correspondingly issue, buy and redeem commercial paper on the ledger. Our next task is to examine this smart contract.

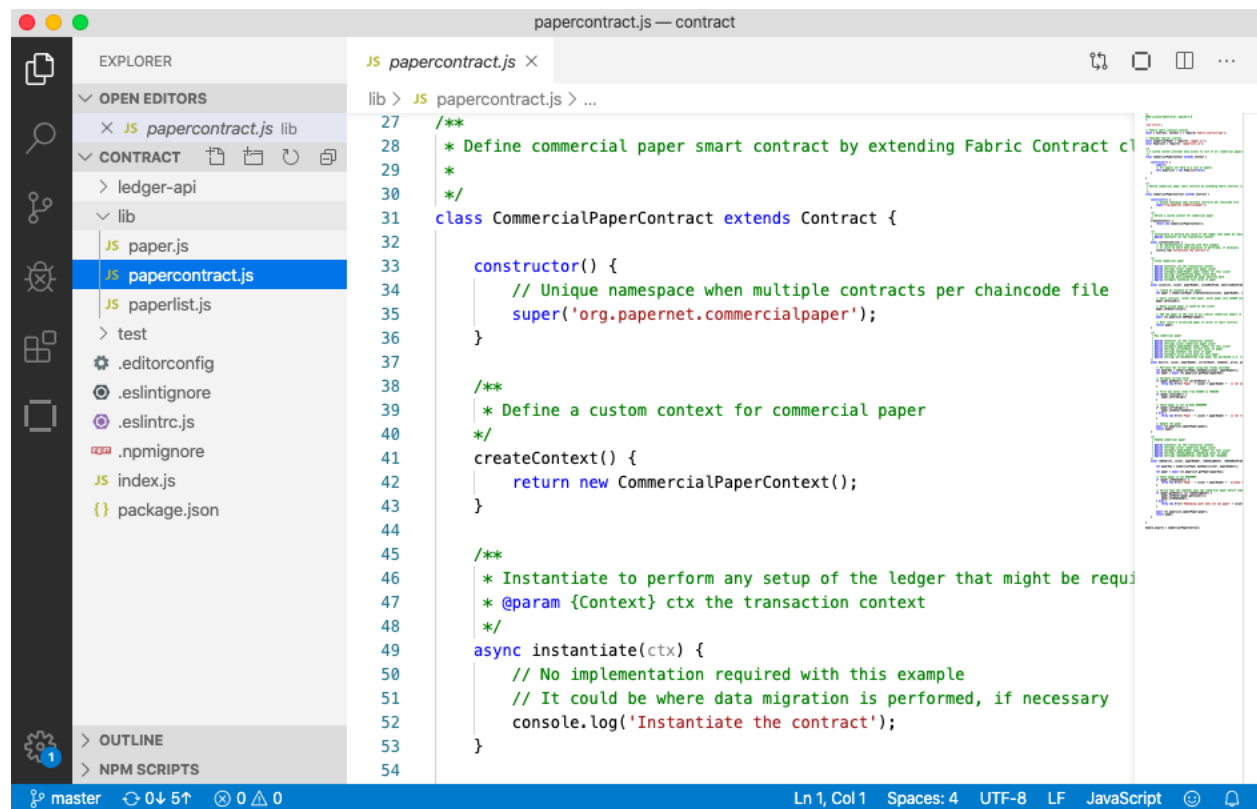
Open a new terminal in the `fabric-samples` directory and change into the `MagnetoCorp` folder to act as the MagnetoCorp developer.

```
cd commercial-paper/organization/magnetocorp
```

You can then view the smart contract in the `contract` directory using your chosen editor (VS Code in this tutorial):

```
(magnetocorp developer)$ code contract
```

In the `lib` directory of the folder, you'll see `papercontract.js` file – this contains the commercial paper smart contract!



An example code editor displaying the commercial paper smart contract in `papercontract.js`

`papercontract.js` is a JavaScript program designed to run in the Node.js environment. Note the following key program lines:

- `const { Contract, Context } = require('fabric-contract-api');`

This statement brings into scope two key Hyperledger Fabric classes that will be used extensively by the smart contract – `Contract` and `Context`. You can learn more about these classes in the [fabric-shim JS DOCS](#).

- `class CommercialPaperContract extends Contract {`

This defines the smart contract class `CommercialPaperContract` based on the built-in Fabric `Contract` class. The methods which implement the key transactions to issue, buy and redeem commercial paper are defined within this class.

- `async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime...)`
`{`

This method defines the commercial paper `issue` transaction for `PaperNet`. The parameters that are passed to this method will be used to create the new commercial paper.

Locate and examine the `buy` and `redeem` transactions within the smart contract.

- `let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime...);`

Within the `issue` transaction, this statement creates a new commercial paper in memory using the `CommercialPaper` class with the supplied transaction inputs. Examine the `buy` and `redeem` transactions to see how they similarly use this class.

- `await ctx.paperList.addPaper(paper);`

This statement adds the new commercial paper to the ledger using `ctx.paperList`, an instance of a `PaperList` class that was created when the smart contract context `CommercialPaperContext` was initialized. Again, examine the `buy` and `redeem` methods to see how they use this class.

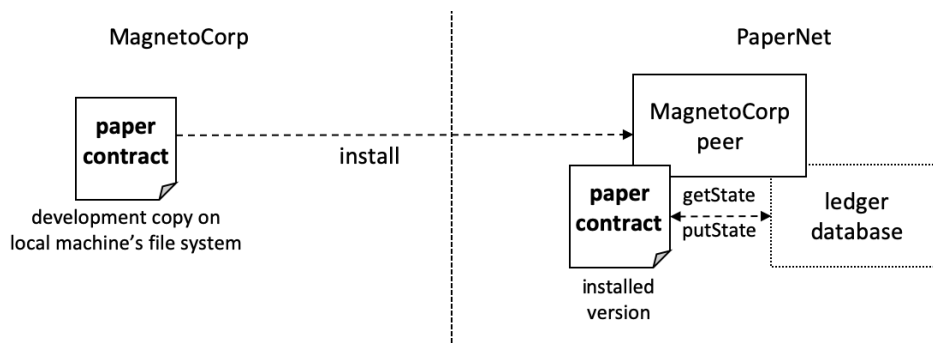
- `return paper;`

This statement returns a binary buffer as response from the `issue` transaction for processing by the caller of the smart contract.

Feel free to examine other files in the `contract` directory to understand how the smart contract works, and read in detail how `papercontract.js` is designed in the [smart contract processing](#) topic.

7.3.6 Deploy the smart contract to the channel

Before `papercontract` can be invoked by applications, it must be installed onto the appropriate peer nodes of the test network and then defined on the channel using the [Fabric chaincode lifecycle](#). The Fabric chaincode lifecycle allows multiple organizations to agree to the parameters of a chaincode before the chaincode is deployed to a channel. As a result, we need to install and approve the chaincode as administrators of both `MagnetoCorp` and `DigiBank`.



A MagnetoCorp administrator installs a copy of the `papercontract` onto a MagnetoCorp peer.

Smart contracts are the focus of application development, and are contained within a Hyperledger Fabric artifact called [chaincode](#). One or more smart contracts can be defined within a single chaincode, and installing a chaincode will allow them to be consumed by the different organizations in `PaperNet`. It means that only administrators need to worry about chaincode; everyone else can think in terms of smart contracts.

Install and approve the smart contract as MagnetoCorp

We will first install and approve the smart contract as the MagnetoCorp admin. Make sure that you are operating from the `magnetocorp` folder, or navigate back to that folder using the following command:

```
cd commercial-paper/organization/magnetocorp
```

A MagnetoCorp administrator can interact with PaperNet using the `peer` CLI. However, the administrator needs to set certain environment variables in their command window to use the correct set of `peer` binaries, send commands to the address of the MagnetoCorp peer, and sign requests with the correct cryptographic material.

You can use a script provided by the sample to set the environment variables in your command window. Run the following command in the `magnetocorp` directory:

```
source magnetocorp.sh
```

You will see the full list of environment variables printed in your window. We can now use this command window to interact with PaperNet as the MagnetoCorp administrator.

The first step is to install the `papercontract` smart contract. The smart contract can be packaged into a chaincode using the `peer lifecycle chaincode package` command. In the MagnetoCorp administrator's command window, run the following command to create the chaincode package:

```
(magnetocorp admin)$ peer lifecycle chaincode package cp.tar.gz --lang node --path ./
↳ contract --label cp_0
```

The MagnetoCorp admin can now install the chaincode on the MagnetoCorp peer using the `peer lifecycle chaincode install` command:

```
(magnetocorp admin)$ peer lifecycle chaincode install cp.tar.gz
```

When the chaincode package is installed, you will see messages similar to the following printed in your terminal:

```
2020-01-30 18:32:33.762 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO_
↳ 001 Installed remotely: response:<status:200 payload:"\nEcp_
↳ 0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c\022\004cp_0" >
2020-01-30 18:32:33.762 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO_
↳ 002 Chaincode code package identifier: cp_
↳ 0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

Because the MagnetoCorp admin has set `CORE_PEER_ADDRESS=localhost:9051` to target its commands to `peer0.org2.example.com`, the `INFO 001 Installed remotely...` indicates that `papercontract` has been successfully installed on this peer.

After we install the smart contract, we need to approve the chaincode definition for `papercontract` as MagnetoCorp. The first step is to find the `packageID` of the chaincode we installed on our peer. We can query the `packageID` using the `peer lifecycle chaincode queryinstalled` command:

```
peer lifecycle chaincode queryinstalled
```

The command will return the same package identifier as the install command. You should see output similar to the following:

```
Installed chaincodes on peer:
Package ID: cp_0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c,
↳ Label: cp_0
```

We will need the package ID in the next step, so we will save it as an environment variable. The package ID may not be the same for all users, so you need to complete this step using the package ID returned from your command window.

```
export PACKAGE_ID=cp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

The admin can now approve the chaincode definition for MagnetoCorp using the `peer lifecycle chaincode approveformyorg` command:

```
(magnetocorp admin)$ peer lifecycle chaincode approveformyorg --orderer_
↪localhost:7050 --ordererTLSHostnameOverride orderer.example.com --channelID_
↪mychannel --name papercontract -v 0 --package-id $PACKAGE_ID --sequence 1 --tls --
↪cafile $ORDERER_CA
```

One of the most important chaincode parameters that channel members need to agree to using the chaincode definition is the chaincode [endorsement policy](#). The endorsement policy describes the set of organizations that must endorse (execute and sign) a transaction before it can be determined to be valid. By approving the `papercontract` chaincode without the `--policy` flag, the MagnetoCorp admin agrees to using the channel's default Endorsement policy, which in the case of the `mychannel` test channel requires a majority of organizations on the channel to endorse a transaction. All transactions, whether valid or invalid, will be recorded on the [ledger blockchain](#), but only valid transactions will update the [world state](#).

Install and approve the smart contract as DigiBank

Based on the `mychannel LifecycleEndorsement` policy, the Fabric Chaincode lifecycle will require a majority of organizations on the channel to agree to the chaincode definition before the chaincode can be committed to the channel. This implies that we need to approve the `papernet` chaincode as both MagnetoCorp and DigiBank to get the required majority of 2 out of 2. Open a new terminal window in the `fabric-samples` and navigate to the folder that contains the DigiBank smart contract and application files:

```
(digibank admin)$ cd commercial-paper/organization/digibank/
```

Use the script in the DigiBank folder to set the environment variables that will allow you to act as the DigiBank admin:

```
source digibank.sh
```

We can now install and approve `papercontract` as the DigiBank. Run the following command to package the chaincode:

```
(digibank admin)$ peer lifecycle chaincode package cp.tar.gz --lang node --path ./
↪contract --label cp_0
```

The admin can now install the chaincode on the DigiBank peer:

```
(digibank admin)$ peer lifecycle chaincode install cp.tar.gz
```

We then need to query and save the packageID of the chaincode that was just installed:

```
(digibank admin)$ peer lifecycle chaincode queryinstalled
```

Save the package ID as an environment variable. Complete this step using the package ID returned from your console.

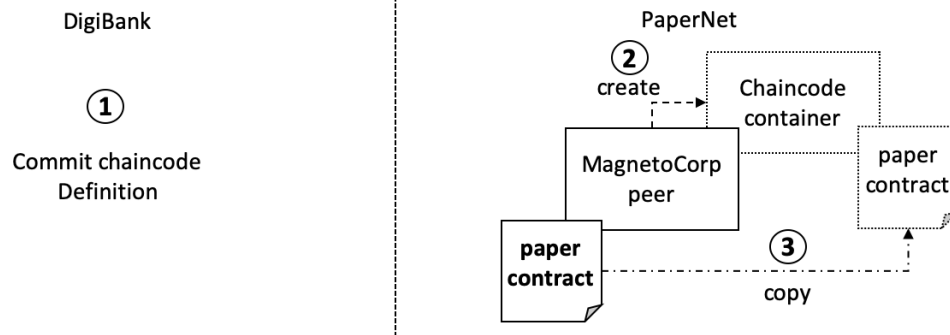
```
export PACKAGE_ID=cp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

The DigiBank admin can now approve the chaincode definition of `papercontract`:

```
(digibank admin)$ peer lifecycle chaincode approveformyorg --orderer localhost:7050 --
→ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name
→papercontract -v 0 --package-id $PACKAGE_ID --sequence 1 --tls --cafile $ORDERER_CA
```

Commit the chaincode definition to the channel

Now that DigiBank and MagnetoCorp have both approved the `papernet` chaincode, we have the majority we need (2 out of 2) to commit the chaincode definition to the channel. Once the chaincode is successfully defined on the channel, the `CommercialPaper` smart contract inside the `papercontract` chaincode can be invoked by client applications on the channel. Since either organization can commit the chaincode to the channel, we will continue operating as the DigiBank admin:



After the DigiBank administrator commits the definition of the `papercontract` chaincode to the channel, a new Docker chaincode container will be created to run `papercontract` on both PaperNet peers

The DigiBank administrator uses the `peer lifecycle chaincode commit` command to commit the chaincode definition of `papercontract` to mychannel:

```
(digibank admin)$ peer lifecycle chaincode commit -o localhost:7050 --
→ordererTLSHostnameOverride orderer.example.com --peerAddresses localhost:7051 --
→tlsRootCertFiles ${PEER0_ORG1_CA} --peerAddresses localhost:9051 --tlsRootCertFiles
→${PEER0_ORG2_CA} --channelID mychannel --name papercontract -v 0 --sequence 1 --tls
→--cafile $ORDERER_CA --waitForEvent
```

The chaincode container will start after the chaincode definition has been committed to the channel. You can use the `docker ps` command to see `papercontract` container starting on both peers.

```
(digibank admin)$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
d4ba9dc9c55f	dev-peer0.org1.example.com-cp_0-	30 seconds ago	Up 28 seconds	"docker-
a944c0f8b6d6	dev-peer0.org2.example.com-cp_0-	31 seconds ago	Up 28 seconds	"docker-

(continues on next page)

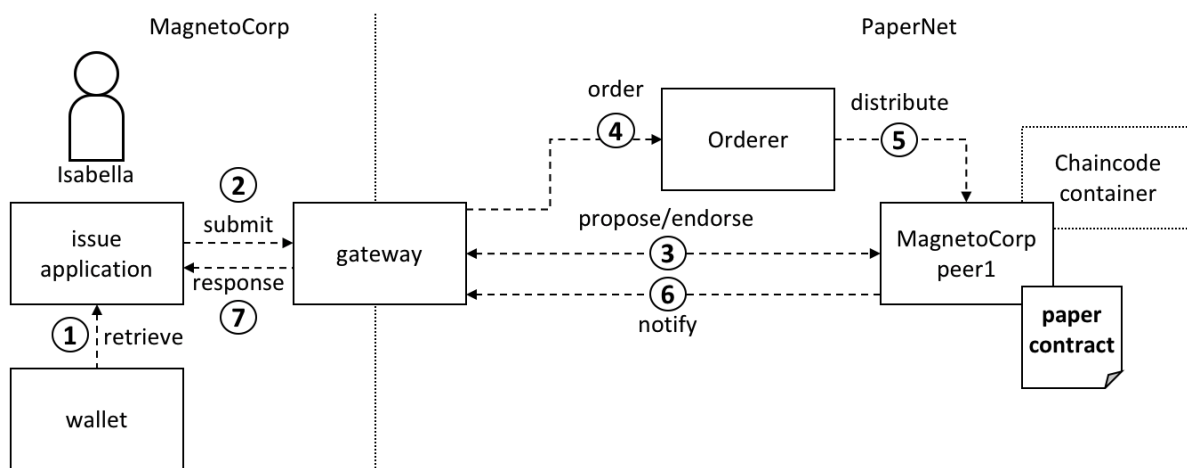
(continued from previous page)

Notice that the containers are named to indicate the peer that started it, and the fact that it's running `papercontract` version 0.

Now that we have deployed the `papercontract` chaincode to the channel, we can use the MagnetoCorp application to issue the commercial paper. Let's take a moment to examine the application structure.

7.3.7 Application structure

The smart contract contained in `papercontract` is called by MagnetoCorp's application `issue.js`. Isabella uses this application to submit a transaction to the ledger which issues commercial paper 00001. Let's quickly examine how the `issue` application works.



A gateway allows an application to focus on transaction generation, submission and response. It coordinates transaction proposal, ordering and notification processing between the different network components.

Because the `issue` application submits transactions on behalf of Isabella, it starts by retrieving Isabella's X.509 certificate from her `wallet`, which might be stored on the local file system or a Hardware Security Module `HSM`. The `issue` application is then able to utilize the gateway to submit transactions on the channel. The Hyperledger Fabric SDK provides a `gateway` abstraction so that applications can focus on application logic while delegating network interaction to the gateway. Gateways and wallets make it straightforward to write Hyperledger Fabric applications.

So let's examine the `issue` application that Isabella is going to use. Open a separate terminal window for her, and in `fabric-samples` locate the `MagnetoCorp /application` folder:

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application/
(isabella)$ ls

addToWallet.js      enrollUser.js      issue.js           package.json
```

`addToWallet.js` is the program that Isabella is going to use to load her identity into her wallet, and `issue.js` will use this identity to create commercial paper 00001 on behalf of MagnetoCorp by invoking `papercontract`.

Change to the directory that contains MagnetoCorp's copy of the application `issue.js`, and use your code editor to examine it:

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application
(isabella)$ code issue.js
```


Examine this directory; it contains the issue application and all its dependencies.

```

47
48
49 // Connect to gateway using application specified parameters
50 console.log('Connect to Fabric gateway.');
```

A code editor displaying the contents of the commercial paper application directory.

Note the following key program lines in `issue.js`:

- `const { Wallets, Gateway } = require('fabric-network');`

This statement brings two key Hyperledger Fabric SDK classes into scope – `Wallet` and `Gateway`.

- `const wallet = await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');`

This statement identifies that the application will use `isabella` wallet when it connects to the blockchain network channel. Because `Isabella's` X.509 certificate is in the local file system, the application creates a new `FileSystemWallet`. The application will select a particular identity within `isabella` wallet.

- `await gateway.connect(connectionProfile, connectionOptions);`

This line of code connects to the network using the gateway identified by `connectionProfile`, using the identity referred to in `ConnectionOptions`.

See how `../gateway/networkConnection.yaml` and `User1@org1.example.com` are used for these values respectively.

- `const network = await gateway.getNetwork('mychannel');`

This connects the application to the network channel `mychannel`, where the `papercontract` was previously deployed.

- `const contract = await network.getContract('papercontract');`

This statement gives the application access to the `papercontract` chaincode. Once an application has issued `getContract`, it can submit to any smart contract transaction implemented within the chaincode.

- `const issueResponse = await contract.submitTransaction('issue', 'Magnetocorp', '00001', ...);`

This line of code submits the a transaction to the network using the `issue` transaction defined within the smart contract. `Magnetocorp`, `00001`... are the values to be used by the `issue` transaction to create a new commercial paper.

- `let paper = CommercialPaper.fromBuffer(issueResponse);`

This statement processes the response from the `issue` transaction. The response needs to be deserialized from a buffer into `paper`, a `CommercialPaper` object which can be interpreted correctly by the application.

Feel free to examine other files in the `/application` directory to understand how `issue.js` works, and read in detail how it is implemented in the application [topic](#).

7.3.8 Application dependencies

The `issue.js` application is written in JavaScript and designed to run in the Node.js environment that acts as a client to the PaperNet network. As is common practice, Magnetocorp's application is built on many external node packages — to improve quality and speed of development. Consider how `issue.js` includes the `js-yaml` [package](#) to process the YAML gateway connection profile, or the `fabric-network` [package](#) to access the Gateway and Wallet classes:

```
const yaml = require('js-yaml');
const { Wallets, Gateway } = require('fabric-network');
```

These packages have to be downloaded from [npm](#) to the local file system using the `npm install` command. By convention, packages must be installed into an application-relative `/node_modules` directory for use at runtime.

Open the `package.json` file to see how `issue.js` identifies the packages to download and their exact versions by examining the “dependencies” section of the file.

npm versioning is very powerful; you can read more about it [here](#).

Let's install these packages with the `npm install` command – this may take up to a minute to complete:

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application/
(isabella)$ npm install

(          ) extract:lodash: sill extract ansi-styles@3.2.1
(...)
added 738 packages in 46.701s
```

See how this command has updated the directory:

```
(isabella)$ ls

enrollUser.js      node_modules      package.json
issue.js           package-lock.json
```

Examine the `node_modules` directory to see the packages that have been installed. There are lots, because `js-yaml` and `fabric-network` are themselves built on other npm packages! Helpfully, the `package-lock.json` [file](#) identifies the exact versions installed, which can prove invaluable if you want to exactly reproduce environments; to test, diagnose problems or deliver proven applications for example.

7.3.9 Wallet

Isabella is almost ready to run `issue.js` to issue MagnetoCorp commercial paper 00001; there's just one remaining task to perform! As `issue.js` acts on behalf of Isabella, and therefore MagnetoCorp, it will use identity from her `wallet` that reflects these facts. We now need to perform this one-time activity of generating the appropriate X.509 credentials to her wallet.

The MagnetoCorp Certificate Authority running on PaperNet, `ca_org2`, has an application user that was registered when the network was deployed. Isabella can use the identity name and secret to generate the X.509 cryptographic material for the `issue.js` application. The process of using a CA to generate client side cryptographic material is referred to as **enrollment**. In a real world scenario, a network operator would provide the name and secret of a client identity that was registered with the CA to an application developer. The developer would then use the credentials to enroll their application and interact with the network.

The `enrollUser.js` program uses the `fabric-ca-client` class to generate a private and public key pair, and then issues a **Certificate Signing Request** to the CA. If the identity name and secret submitted by Isabella match the credentials registered with the CA, the CA will issue and sign a certificate that encodes the public key, establishing that Isabella belongs to MagnetoCorp. When the signing request is complete, `enrollUser.js` stores the private key and signing certificate in Isabella's wallet. You can examine the `enrollUser.js` file to learn more about how the Node SDK uses the `fabric-ca-client` class to complete these tasks.

In Isabella's terminal window, run the `enrollUser.js` program to add identity information to her wallet:

```
(isabella)$ node enrollUser.js

Wallet path: /Users/nikhilgupta/fabric-samples/commercial-paper/organization/
↳ magnetocorp/identity/user/isabella/wallet
Successfully enrolled client user "isabella" and imported it into the wallet
```

We can now turn our focus to the result of this program — the contents of the wallet which will be used to submit transactions to PaperNet:

```
(isabella)$ ls ../identity/user/isabella/wallet/

isabella.id
```

Isabella can store multiple identities in her wallet, though in our example, she only uses one. The `wallet` folder contains an `isabella.id` file that provides the information that Isabella needs to connect to the network. Other identities used by Isabella would have their own file. You can open this file to see the identity information that `issue.js` will use on behalf of Isabella inside a JSON file. The output has been formatted for clarity.

```
(isabella)$ cat ../identity/user/isabella/wallet/*

{
  "credentials": {
    "certificate": "-----BEGIN CERTIFICATE-----
↳ \nMIICKTCCAdCgAwIBAgIQWkvLG+sqeO3LwwQK6avZDAKBggqhkJOPQQDAjBzMQsw\nnCQYDVQQGEwJVUzETMBEGA1UECBMKQ2l0aW4uMAsGA1U
↳ 53dbo00wSZA0BgNVHQ8BAf8EBAMCB4AwDAYDVR0TAQH/
↳ BAIwADArBgNV\nHSMEJDAigCDOCDm4irsZFU3D6Hak4+84QRglN43iwg8w1V6DRhgLyDAKBggqhkJ0\nnPQQDAgNHADBEAiBhzK
↳ mRtUdaJagIgiYpbZ\nXf0CSiTXIWOJIsswN4Jp+ZxkJfFVmXndqKqz+VM=\n-----END CERTIFICATE---
↳ --\n",
    "privateKey": "-----BEGIN PRIVATE KEY-----
↳ \nMIGHAgEAMBMGBYqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQggs55vQg2oXi8gNi8\nnNidE8Fy5zenohArDq3FGJD8cKU2hRAN
↳ 53db\n-----END PRIVATE KEY-----\n"
  },
  "mspId": "Org2MSP",
  "type": "X.509",
```

(continues on next page)

(continued from previous page)

```
"version": 1
}
```

In the file you can notice the following:

- a "privateKey": used to sign transactions on Isabella's behalf, but not distributed outside of her immediate control.
- a "certificate": which contains Isabella's public key and other X.509 attributes added by the Certificate Authority at certificate creation. This certificate is distributed to the network so that different actors at different times can cryptographically verify information created by Isabella's private key.

You can Learn more about certificates [here](#). In practice, the certificate file also contains some Fabric-specific metadata such as Isabella's organization and role – read more in the [wallet](#) topic.

7.3.10 Issue application

Isabella can now use `issue.js` to submit a transaction that will issue MagnetoCorp commercial paper 00001:

```
(isabella)$ node issue.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper issue transaction.
Process issue transaction response.{"class":"org.papernet.commercialpaper","key":"\
→ "MagnetoCorp\":"00001\","currentState":1,"issuer":"MagnetoCorp","paperNumber":
→ "00001","issueDateTime":"2020-05-31","maturityDateTime":"2020-11-30","faceValue":
→ "5000000","owner":"MagnetoCorp"}
MagnetoCorp commercial paper : 00001 successfully issued for value 5000000
Transaction complete.
Disconnect from Fabric gateway.
Issue program complete.
```

The `node` command initializes a Node.js environment, and runs `issue.js`. We can see from the program output that MagnetoCorp commercial paper 00001 was issued with a face value of 5M USD.

As you've seen, to achieve this, the application invokes the `issue` transaction defined in the `CommercialPaper` smart contract within `papercontract.js`. The smart contract interacts with the ledger via the Fabric APIs, most notably `putState()` and `getState()`, to represent the new commercial paper as a vector state within the world state. We'll see how this vector state is subsequently manipulated by the `buy` and `redeem` transactions also defined within the smart contract.

All the time, the underlying Fabric SDK handles the transaction endorsement, ordering and notification process, making the application's logic straightforward; the SDK uses a [gateway](#) to abstract away network details and [connectionOptions](#) to declare more advanced processing strategies such as transaction retry.

Let's now follow the lifecycle of MagnetoCorp 00001 by switching our emphasis to an employee of DigiBank, Balaji, who will buy the commercial paper using a DigiBank application.

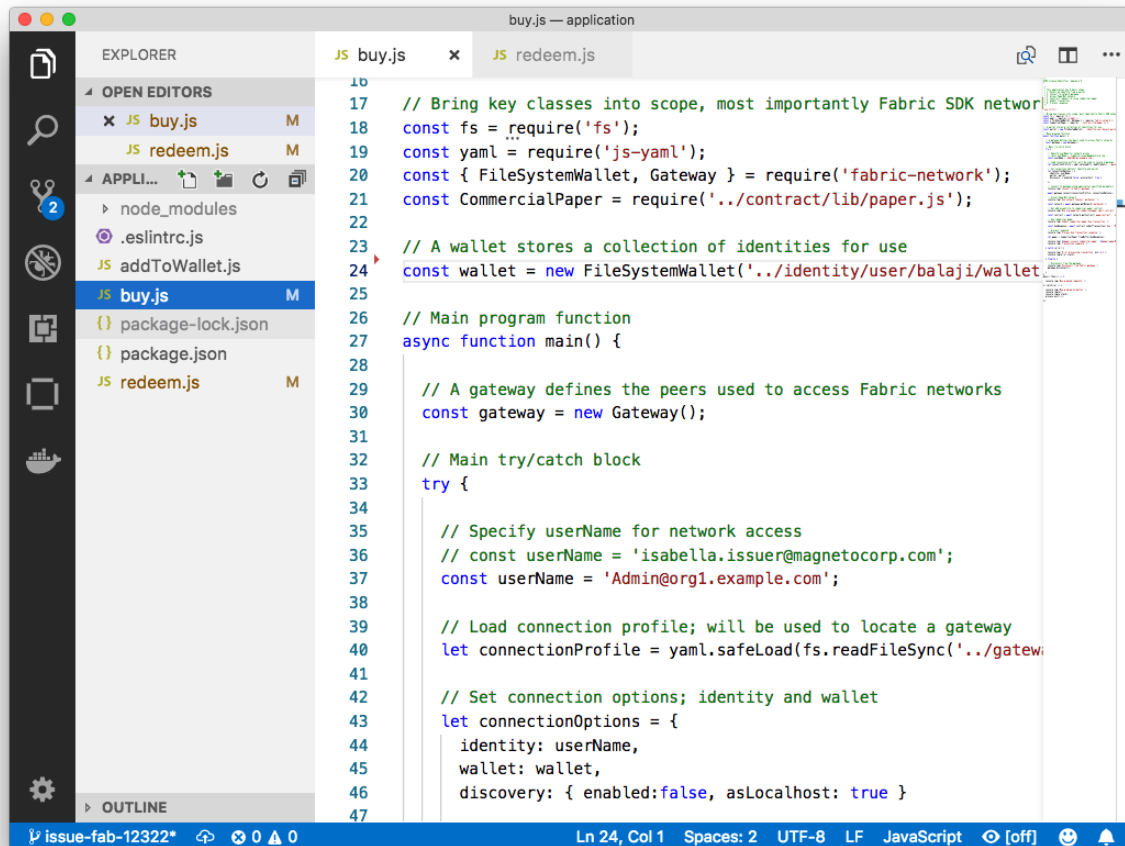
7.3.11 Digibank applications

Balaji uses DigiBank's `buy` application to submit a transaction to the ledger which transfers ownership of commercial paper 00001 from MagnetoCorp to DigiBank. The `CommercialPaper` smart contract is the same as that used by MagnetoCorp's application, however the transaction is different this time – it's `buy` rather than `issue`. Let's examine how DigiBank's application works.

Open a separate terminal window for Balaji. In `fabric-samples`, change to the DigiBank application directory that contains the application, `buy.js`, and open it with your editor:

```
(balaji)$ cd commercial-paper/organization/digibank/application/
(balaji)$ code buy.js
```

As you can see, this directory contains both the `buy` and `redeem` applications that will be used by Balaji.



DigiBank's commercial paper directory containing the `buy.js` and `redeem.js` applications.

DigiBank's `buy.js` application is very similar in structure to MagnetoCorp's `issue.js` with two important differences:

- **Identity:** the user is a DigiBank user `Balaji` rather than MagnetoCorp's `Isabella`

```
const wallet = await Wallets.newFileSystemWallet('../identity/user/balaji/wallet
↪');
```

See how the application uses the `balaji` wallet when it connects to the PaperNet network channel. `buy.js` selects a particular identity within `balaji` wallet.

- **Transaction:** the invoked transaction is `buy` rather than `issue`

```
const buyResponse = await contract.submitTransaction('buy', 'MagnetoCorp', '00001
↪', ...);
```

A buy transaction is submitted with the values `MagnetoCorp`, `00001`, ..., that are used by the `CommercialPaper` smart contract class to transfer ownership of commercial paper `00001` to `DigiBank`.

Feel free to examine other files in the `application` directory to understand how the application works, and read in detail how `buy.js` is implemented in the [application topic](#).

7.3.12 Run as DigiBank

The `DigiBank` applications which buy and redeem commercial paper have a very similar structure to `MagnetoCorp`'s issue application. Therefore, let's install their dependencies and set up `Balaji`'s wallet so that he can use these applications to buy and redeem commercial paper.

Like `MagnetoCorp`, `Digibank` must install the required application packages using the `npm install` command, and again, this make take a short time to complete.

In the `DigiBank` administrator window, install the application dependencies:

```
(digibank admin)$ cd commercial-paper/organization/digibank/application/
(digibank admin)$ npm install

(          ) extract:lodash: sill extract ansi-styles@3.2.1
(...)
added 738 packages in 46.701s
```

In `Balaji`'s command window, run the `enrollUser.js` program to generate a certificate and private key and them to his wallet:

```
(balaji)$ node enrollUser.js

Wallet path: /Users/nikhilgupta/fabric-samples/commercial-paper/organization/digibank/
→identity/user/balaji/wallet
Successfully enrolled client user "balaji" and imported it into the wallet
```

The `addToWallet.js` program has added identity information for `balaji`, to his wallet, which will be used by `buy.js` and `redeem.js` to submit transactions to `PaperNet`.

Like `Isabella`, `Balaji` can store multiple identities in his wallet, though in our example, he only uses one. His corresponding id file at `digibank/identity/user/balaji/wallet/balaji.id` is very similar `Isabella`'s — feel free to examine it.

7.3.13 Buy application

`Balaji` can now use `buy.js` to submit a transaction that will transfer ownership of `MagnetoCorp` commercial paper `00001` to `DigiBank`.

Run the `buy` application in `Balaji`'s window:

```
(balaji)$ node buy.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper buy transaction.
Process buy transaction response.
MagnetoCorp commercial paper : 00001 successfully purchased by DigiBank
Transaction complete.
```

(continues on next page)

(continued from previous page)

```
Disconnect from Fabric gateway.  
Buy program complete.
```

You can see the program output that MagnetoCorp commercial paper 00001 was successfully purchased by Balaji on behalf of DigiBank. `buy.js` invoked the `buy` transaction defined in the `CommercialPaper` smart contract which updated commercial paper 00001 within the world state using the `putState()` and `getState()` Fabric APIs. As you've seen, the application logic to buy and issue commercial paper is very similar, as is the smart contract logic.

7.3.14 Redeem application

The final transaction in the lifecycle of commercial paper 00001 is for DigiBank to redeem it with MagnetoCorp. Balaji uses `redeem.js` to submit a transaction to perform the redeem logic within the smart contract.

Run the `redeem` transaction in Balaji's window:

```
(balaji)$ node redeem.js  
  
Connect to Fabric gateway.  
Use network channel: mychannel.  
Use org.papernet.commercialpaper smart contract.  
Submit commercial paper redeem transaction.  
Process redeem transaction response.  
MagnetoCorp commercial paper : 00001 successfully redeemed with MagnetoCorp  
Transaction complete.  
Disconnect from Fabric gateway.  
Redeem program complete.
```

Again, see how the commercial paper 00001 was successfully redeemed when `redeem.js` invoked the `redeem` transaction defined in `CommercialPaper`. Again, it updated commercial paper 00001 within the world state to reflect that the ownership returned to MagnetoCorp, the issuer of the paper.

7.3.15 Clean up

When you are finished using the Commercial Paper tutorial, you can use a script to clean up your environment. Use a command window to navigate back to the root directory of the commercial paper sample:

```
cd fabric-samples/commercial-paper
```

You can then bring down the network with the following command:

```
./network-clean.sh
```

This command will bring down the peers, CouchDB containers, and ordering node of the network, in addition to the logspout tool. It will also remove the identities that we created for Isabella and Balaji. Note that all of the data on the ledger will be lost. If you want to go through the tutorial again, you will start from a clean initial state.

7.3.16 Further reading

To understand how applications and smart contracts shown in this tutorial work in more detail, you'll find it helpful to read [Developing Applications](#). This topic will give you a fuller explanation of the commercial paper scenario, the PaperNet business network, its actors, and how the applications and smart contracts they use work in detail.

Also feel free to use this sample to start creating your own applications and smart contracts!

7.4 Using Private Data in Fabric

This tutorial will demonstrate the use of Private Data Collections (PDC) to provide storage and retrieval of private data on the blockchain network for authorized peers of organizations. The collection is specified using a collection definition file containing the policies governing that collection.

The information in this tutorial assumes knowledge of private data stores and their use cases. For more information, check out [Private data](#).

Note: These instructions use the new Fabric chaincode lifecycle introduced in the Fabric v2.0 release. If you would like to use the previous lifecycle model to use private data with chaincode, visit the v1.4 version of the [Using Private Data in Fabric](#) tutorial.

The tutorial will take you through the following steps to practice defining, configuring and using private data with Fabric:

1. *Asset transfer private data sample use case*
2. *Build a collection definition JSON file*
3. *Read and Write private data using chaincode APIs*
4. *Deploy the private data smart contract to the channel*
5. *Register identities*
6. *Create an asset in private data*
7. *Query the private data as an authorized peer*
8. *Query the private data as an unauthorized peer*
9. *Transfer the Asset*
10. *Purge Private Data*
11. *Using indexes with private data*
12. *Additional resources*

This tutorial will deploy the [asset transfer private data sample](#) to the Fabric test network to demonstrate how to create, deploy, and use a collection of private data. You should have completed the task [Install Samples, Binaries, and Docker Images](#).

7.4.1 Asset transfer private data sample use case

This sample demonstrates the use of three private data collections, `assetCollection`, `Org1MSPPPrivateCollection` & `Org2MSPPPrivateCollection` to transfer an asset between `Org1` and `Org2`, using following use case:

A member of `Org1` creates a new asset, henceforth referred as owner. The public details of the asset, including the identity of the owner, are stored in the private data collection named `assetCollection`. The asset is also created with an appraised value supplied by the owner. The appraised value is used by each participant to agree to the transfer of the asset, and is only stored in owner organization's collection. In our case, the initial appraisal value agreed by the owner is stored in the `Org1MSPPPrivateCollection`.

To purchase the asset, the buyer needs to agree to the same appraised value as the asset owner. In this step, the buyer (a member of `Org2`) creates an agreement to trade and agree to an appraisal value using smart contract function `'AgreeToTransfer'`. This value is stored in `Org2MSPPPrivateCollection` collection. Now, the asset owner

can transfer the asset to the buyer using smart contract function `'TransferAsset'`. The `'TransferAsset'` function uses the hash on the channel ledger to confirm that the owner and the buyer have agreed to the same appraised value before transferring the asset.

Before we go through the transfer scenario, we will discuss how organizations can use private data collections in Fabric.

7.4.2 Build a collection definition JSON file

Before a set of organizations can transact using private data, all organizations on channel need to build a collection definition file that defines the private data collections associated with each chaincode. Data that is stored in a private data collection is only distributed to the peers of certain organizations instead of all members of the channel. The collection definition file describes all of the private data collections that organizations can read and write to from a chaincode.

Each collection is defined by the following properties:

- `name`: Name of the collection.
- `policy`: Defines the organization peers allowed to persist the collection data.
- `requiredPeerCount`: Number of peers required to disseminate the private data as a condition of the endorsement of the chaincode
- `maxPeerCount`: For data redundancy purposes, the number of other peers that the current endorsing peer will attempt to distribute the data to. If an endorsing peer goes down, these other peers are available at commit time if there are requests to pull the private data.
- `blockToLive`: For very sensitive information such as pricing or personal information, this value represents how long the data should live on the private database in terms of blocks. The data will live for this specified number of blocks on the private database and after that it will get purged, making this data obsolete from the network. To keep private data indefinitely, that is, to never purge private data, set the `blockToLive` property to 0.
- `memberOnlyRead`: a value of `true` indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed read access to private data.
- `memberOnlyWrite`: a value of `true` indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed write access to private data.
- `endorsementPolicy`: defines the endorsement policy that needs to be met in order to write to the private data collection. The collection level endorsement policy overrides to chaincode level policy. For more information on building a policy definition refer to the [Endorsement policies](#) topic.

The same collection definition file needs to be deployed by all organizations that use the chaincode, even if the organization does not belong to any collections. In addition to the collections that are explicitly defined in a collection file, each organization has access to an implicit collection on their peers that can only be read by their organization. For an example that uses implicit data collections, see the [Secured asset transfer in Fabric](#).

The asset transfer private data example contains a `collections_config.json` file that defines three private data collection definitions: `assetCollection`, `Org1MSPPprivateCollection`, and `Org2MSPPprivateCollection`.

```
// collections_config.json

[
  {
    "name": "assetCollection",
    "policy": "OR('Org1MSP.member', 'Org2MSP.member')",
    "requiredPeerCount": 1,
```

(continues on next page)

(continued from previous page)

```

    "maxPeerCount": 1,
    "blockToLive":1000000,
    "memberOnlyRead": true,
    "memberOnlyWrite": true
  },
  {
    "name": "Org1MSPPrivateCollection",
    "policy": "OR('Org1MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 1,
    "blockToLive":3,
    "memberOnlyRead": true,
    "memberOnlyWrite": false,
    "endorsementPolicy": {
      "signaturePolicy": "OR('Org1MSP.member') "
    }
  },
  {
    "name": "Org2MSPPrivateCollection",
    "policy": "OR('Org2MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 1,
    "blockToLive":3,
    "memberOnlyRead": true,
    "memberOnlyWrite": false,
    "endorsementPolicy": {
      "signaturePolicy": "OR('Org2MSP.member') "
    }
  }
]

```

The `policy` property in the `assetCollection` definition specifies that both Org1 and Org2 can store the collection on their peers. The `memberOnlyRead` and `memberOnlyWrite` parameters are used to specify that only Org1 and Org2 clients can read and write to this collection.

The `Org1MSPPrivateCollection` collection allows only peers of Org1 to have the private data in their private database, while the `Org2MSPPrivateCollection` collection can only be stored by the peers of Org2. The `endorsementPolicy` parameter is used to create a collection specific endorsement policy. Each update to `Org1MSPPrivateCollection` or `Org2MSPPrivateCollection` needs to be endorsed by the organization that stores the collection on their peers. We will see how these collections are used to transfer the asset in the course of the tutorial.

This collection definition file is deployed when the chaincode definition is committed to the channel using the `peer lifecycle chaincode commit` command. More details on this process are provided in Section 3 below.

7.4.3 Read and Write private data using chaincode APIs

The next step in understanding how to privatize data on a channel is to build the data definition in the chaincode. The asset transfer private data sample divides the private data into three separate data definitions according to how the data will be accessed.

```

// Peers in Org1 and Org2 will have this private data in a side database
type Asset struct {
    Type string `json:"objectType"` //Type is used to distinguish the various_
    ↪types of objects in state database

```

(continues on next page)

(continued from previous page)

```

    ID      string `json:"assetID"`
    Color   string `json:"color"`
    Size    int    `json:"size"`
    Owner   string `json:"owner"`
}

// AssetPrivateDetails describes details that are private to owners

// Only peers in Org1 will have this private data in a side database
type AssetPrivateDetails struct {
    ID            string `json:"assetID"`
    AppraisedValue int    `json:"appraisedValue"`
}

// Only peers in Org2 will have this private data in a side database
type AssetPrivateDetails struct {
    ID            string `json:"assetID"`
    AppraisedValue int    `json:"appraisedValue"`
}

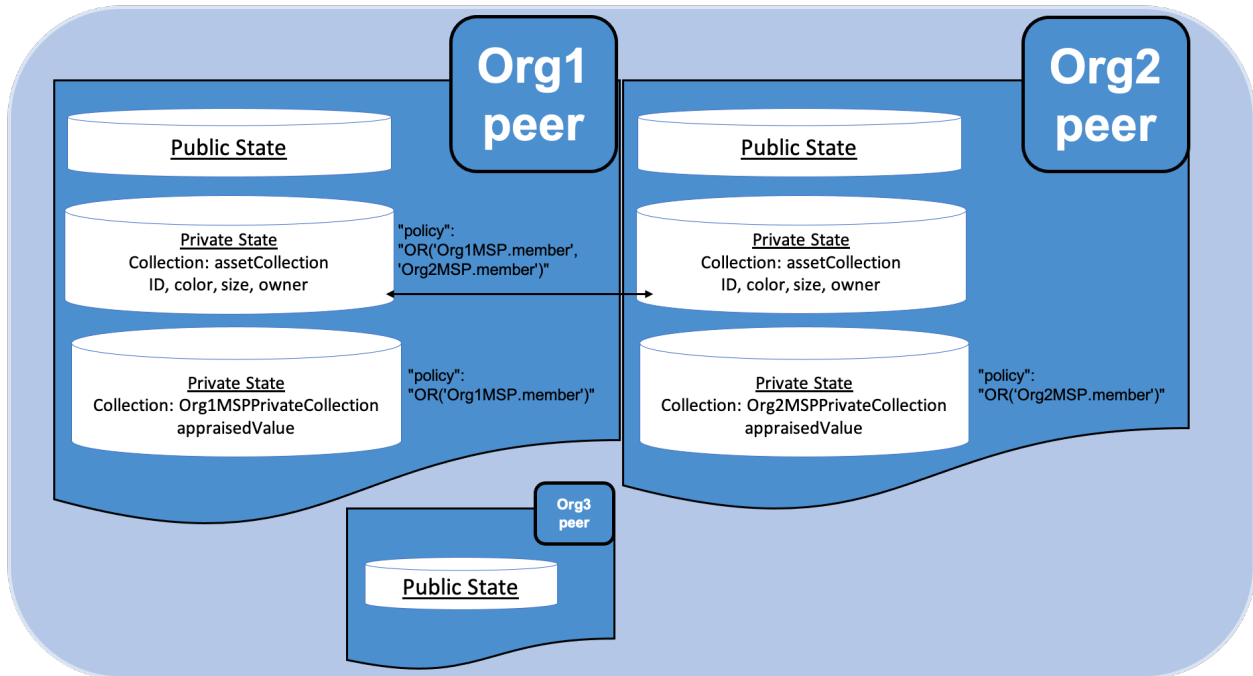
```

Specifically, access to the private data will be restricted as follows:

- objectType, color, size, and owner are stored in assetCollection and hence will be visible to members of the channel per the definition in the collection policy (Org1 and Org2).
- AppraisedValue of an asset is stored in collection Org1MSPPPrivateCollection or Org2MSPPPrivateCollection, depending on the owner of the asset. The value is only accessible to the users who belong to the organization that can store the collection.

All of the data that is created by the asset transfer private data sample smart contract is stored in PDC. The smart contract uses the Fabric chaincode API to read and write private data to private data collections using the `GetPrivateData()` and `PutPrivateData()` functions. You can find more information about those functions [here](#). This private data is stored in private state db on the peer (separate from public state db), and is disseminated between authorized peers via gossip protocol.

The following diagram illustrates the private data model used by the private data sample. Note that Org3 is only shown in the diagram to illustrate that if there were any other organizations on the channel, they would not have access to *any* of the private data collections that were defined in the configuration.



Reading collection data

The smart contract uses the chaincode API `GetPrivateData()` to query private data in the database. `GetPrivateData()` takes two arguments, the **collection name** and the data key. Recall the collection `assetCollection` allows peers of Org1 and Org2 to have the private data in a side database, and the collection `Org1MSPPriateCollection` allows only peers of Org1 to have their private data in a side database and `Org2MSPPriateCollection` allows peers of Org2 to have their private data in a side database. For implementation details refer to the following two [asset transfer private data functions](#):

- **ReadAsset** for querying the values of the `assetID`, `color`, `size` and `owner` attributes.
- **ReadAssetPrivateDetails** for querying the values of the `appraisedValue` attribute.

When we issue the database queries using the peer commands later in this tutorial, we will call these two functions.

Writing private data

The smart contract uses the chaincode API `PutPrivateData()` to store the private data into the private database. The API also requires the name of the collection. Note that the asset transfer private data sample includes three different private data collections, but it is called twice in the chaincode (in this scenario acting as Org1).

1. Write the private data `assetID`, `color`, `size` and `owner` using the collection named `assetCollection`.
2. Write the private data `appraisedValue` using the collection named `Org1MSPPriateCollection`.

If we were acting as Org2, we would replace `Org1MSPPriateCollection` with `Org2MSPPriateCollection`.

For example, in the following snippet of the `CreateAsset` function, `PutPrivateData()` is called twice, once for each set of private data.

```

// CreateAsset creates a new asset by placing the main asset details in the
↳assetCollection
// that can be read by both organizations. The appraisal value is stored in the
↳owners org specific collection.
func (s *SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface)
↳error {

    // Get new asset from transient map
    transientMap, err := ctx.GetStub().GetTransient()
    if err != nil {
        return fmt.Errorf("error getting transient: %v", err)
    }

    // Asset properties are private, therefore they get passed in transient field,
↳instead of func args
    transientAssetJSON, ok := transientMap["asset_properties"]
    if !ok {
        //log error to stdout
        return fmt.Errorf("asset not found in the transient map input")
    }

    type assetTransientInput struct {
        Type          string `json:"objectType"` //Type is used to distinguish the
↳various types of objects in state database
        ID           string `json:"assetID"`
        Color         string `json:"color"`
        Size          int    `json:"size"`
        AppraisedValue int    `json:"appraisedValue"`
    }

    var assetInput assetTransientInput
    err = json.Unmarshal(transientAssetJSON, &assetInput)
    if err != nil {
        return fmt.Errorf("failed to unmarshal JSON: %v", err)
    }

    if len(assetInput.Type) == 0 {
        return fmt.Errorf("objectType field must be a non-empty string")
    }
    if len(assetInput.ID) == 0 {
        return fmt.Errorf("assetID field must be a non-empty string")
    }
    if len(assetInput.Color) == 0 {
        return fmt.Errorf("color field must be a non-empty string")
    }
    if assetInput.Size <= 0 {
        return fmt.Errorf("size field must be a positive integer")
    }
    if assetInput.AppraisedValue <= 0 {
        return fmt.Errorf("appraisedValue field must be a positive integer")
    }

    // Check if asset already exists
    assetAsBytes, err := ctx.GetStub().GetPrivateData(assetCollection, assetInput.ID)
    if err != nil {
        return fmt.Errorf("failed to get asset: %v", err)
    } else if assetAsBytes != nil {

```

(continues on next page)

(continued from previous page)

```

    fmt.Println("Asset already exists: " + assetInput.ID)
    return fmt.Errorf("this asset already exists: " + assetInput.ID)
}

// Get ID of submitting client identity
clientID, err := submittingClientIdentity(ctx)
if err != nil {
    return err
}

// Verify that the client is submitting request to peer in their organization
// This is to ensure that a client from another org doesn't attempt to read or
// write private data from this peer.
err = verifyClientOrgMatchesPeerOrg(ctx)
if err != nil {
    return fmt.Errorf("CreateAsset cannot be performed: Error %v", err)
}

// Make submitting client the owner
asset := Asset{
    Type:  assetInput.Type,
    ID:    assetInput.ID,
    Color: assetInput.Color,
    Size:  assetInput.Size,
    Owner: clientID,
}
assetJSONAsBytes, err := json.Marshal(asset)
if err != nil {
    return fmt.Errorf("failed to marshal asset into JSON: %v", err)
}

// Save asset to private data collection
// Typical logger, logs to stdout/file in the fabric managed docker container,
↳running this chaincode
// Look for container name like dev-peer0.org1.example.com-{chaincodename_version}
↳-xyz
log.Printf("CreateAsset Put: collection %v, ID %v, owner %v", assetCollection,
↳assetInput.ID, clientID)

err = ctx.GetStub().PutPrivateData(assetCollection, assetInput.ID,
↳assetJSONAsBytes)
if err != nil {
    return fmt.Errorf("failed to put asset into private data collecton: %v", err)
}

// Save asset details to collection visible to owning organization
assetPrivateDetails := AssetPrivateDetails{
    ID:            assetInput.ID,
    AppraisedValue: assetInput.AppraisedValue,
}

assetPrivateDetailsAsBytes, err := json.Marshal(assetPrivateDetails) // marshal
↳asset details to JSON
if err != nil {
    return fmt.Errorf("failed to marshal into JSON: %v", err)
}

```

(continues on next page)

(continued from previous page)

```

// Get collection name for this organization.
orgCollection, err := getCollectionName(ctx)
if err != nil {
    return fmt.Errorf("failed to infer private collection name for the org: %v",
↳err)
}

// Put asset appraised value into owners org specific private data collection
log.Printf("Put: collection %v, ID %v", orgCollection, assetInput.ID)
err = ctx.GetStub().PutPrivateData(orgCollection, assetInput.ID,
↳assetPrivateDetailsAsBytes)
if err != nil {
    return fmt.Errorf("failed to put asset private details: %v", err)
}
return nil
}

```

To summarize, the policy definition above for our `collections_config.json` allows all peers in `Org1` and `Org2` to store and transact with the asset transfer private data `assetID`, `color`, `size`, `owner` in their private database. But only peers in `Org1` can store and transact with the `appraisedValue` key data in the `Org1` collection `Org1MSPPPrivateCollection` and only peers in `Org2` can store and transact with the `appraisedValue` key data in the `Org2` collection `Org2MSPPPrivateCollection`.

As an additional data privacy benefit, since a collection is being used, only the private data *hashes* go through orderer, not the private data itself, keeping private data confidential from orderer.

7.4.4 Start the network

Now we are ready to step through some commands which demonstrate how to use private data.

Try it yourself

Before installing, defining, and using the private data smart contract, we need to start the Fabric test network. For the sake of this tutorial, we want to operate from a known initial state. The following command will kill any active or stale Docker containers and remove previously generated artifacts. Therefore let's run the following command to clean up any previous environments:

```

cd fabric-samples/test-network
./network.sh down

```

From the `test-network` directory, you can use the following command to start up the Fabric test network with Certificate Authorities and CouchDB:

```

./network.sh up createChannel -ca -s couchdb

```

This command will deploy a Fabric network consisting of a single channel named `mychannel` with two organizations (each maintaining one peer node), certificate authorities, and an ordering service while using CouchDB as the state database. Either LevelDB or CouchDB may be used with collections. CouchDB was chosen to demonstrate how to use indexes with private data.

Note: For collections to work, it is important to have cross organizational gossip configured correctly. Refer to our documentation on [Gossip data dissemination protocol](#), paying particular attention to the section on “anchor peers”. Our tutorial does not focus on gossip given it is already configured in the test network, but when configuring a channel, the gossip anchors peers are critical to configure for collections to work properly.

7.4.5 Deploy the private data smart contract to the channel

We can now use the test network script to deploy the smart contract to the channel. Run the following command from the test network directory.

```
./network.sh deployCC -ccn private -ccp ../asset-transfer-private-data/chaincode-go/ -
↪ ccl go -ccep "OR('Org1MSP.peer','Org2MSP.peer')" -cccg ../asset-transfer-private-
↪ data/chaincode-go/collections_config.json
```

Note that we need to pass the path to the private data collection definition file to the command. As part of deploying the chaincode to the channel, both organizations on the channel must pass identical private data collection definitions as part of the *Fabric chaincode lifecycle*. We are also deploying the smart contract with a chaincode level endorsement policy of "OR('Org1MSP.peer', 'Org2MSP.peer')". This allows Org1 and Org2 to create an asset without receiving an endorsement from the other organization. You can see the steps required to deploy the chaincode printed in your logs after you issue the command above.

When both organizations approve the chaincode definition using the `peer lifecycle chaincode approveformyorg` command, the chaincode definition includes the path to the private data collection definition using the `--collections-config` flag. You can see the following `approveformyorg` command printed in your terminal:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name private_
↪ --version 1.0 --collections-config ../asset-transfer-private-data/chaincode-go/
↪ collections_config.json --signature-policy "OR('Org1MSP.member','Org2MSP.member')" -
↪ -package-id $CC_PACKAGE_ID --sequence 1 --tls --cafile $ORDERER_CA
```

After channel members agree to the private data collection as part of the chaincode definition, the data collection is committed to the channel using the `peer lifecycle chaincode commit` command. If you look for the commit command in your logs, you can see that it uses the same `--collections-config` flag to provide the path to the collection definition.

```
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride_
↪ orderer.example.com --channelID mychannel --name private --version 1.0 --sequence 1_
↪ --collections-config ../asset-transfer-private-data/chaincode-go/collections_config.
↪ json --signature-policy "OR('Org1MSP.member','Org2MSP.member')" --tls --cafile
↪ $ORDERER_CA --peerAddresses localhost:7051 --tlsRootCertFiles $ORG1_CA --
↪ peerAddresses localhost:9051 --tlsRootCertFiles $ORG2_CA
```

7.4.6 Register identities

The private data transfer smart contract supports ownership by individual identities that belong to the network. In our scenario, the owner of the asset will be a member of Org1, while the buyer will belong to Org2. To highlight the connection between the `GetClientIdentity().GetID()` API and the information within a user's certificate, we will register two new identities using the Org1 and Org2 Certificate Authorities (CA's), and then use the CA's to generate each identity's certificate and private key.

First, we need to set the following environment variables to use the Fabric CA client:

```
export PATH=${PWD}/../bin:${PWD}:${PATH}
export FABRIC_CFG_PATH=${PWD}/../config/
```

We will use the Org1 CA to create the identity asset owner. Set the Fabric CA client home to the MSP of the Org1 CA admin (this identity was generated by the test network script):

```
export FABRIC_CA_CLIENT_HOME=${PWD}/organizations/peerOrganizations/org1.example.com/
```

You can register a new owner client identity using the *fabric-ca-client* tool:

```
fabric-ca-client register --caname ca-org1 --id.name owner --id.secret ownerpw --id.  
↪type client --tls.certfiles ${PWD}/organizations/fabric-ca/org1/tls-cert.pem
```

You can now generate the identity certificates and MSP folder by providing the enroll name and secret to the enroll command:

```
fabric-ca-client enroll -u https://owner:ownerpw@localhost:7054 --caname ca-org1 -M $  
↪${PWD}/organizations/peerOrganizations/org1.example.com/users/owner@org1.example.com/  
↪msp --tls.certfiles ${PWD}/organizations/fabric-ca/org1/tls-cert.pem
```

Run the command below to copy the Node OU configuration file into the owner identity MSP folder.

```
cp ${PWD}/organizations/peerOrganizations/org1.example.com/msp/config.yaml ${PWD}/  
↪organizations/peerOrganizations/org1.example.com/users/owner@org1.example.com/msp/  
↪config.yaml
```

We can now use the Org2 CA to create the buyer identity. Set the Fabric CA client home the Org2 CA admin:

```
export FABRIC_CA_CLIENT_HOME=${PWD}/organizations/peerOrganizations/org2.example.com/
```

You can register a new owner client identity using the *fabric-ca-client* tool:

```
fabric-ca-client register --caname ca-org2 --id.name buyer --id.secret buyerpw --id.  
↪type client --tls.certfiles ${PWD}/organizations/fabric-ca/org2/tls-cert.pem
```

We can now enroll to generate the identity MSP folder:

```
fabric-ca-client enroll -u https://buyer:buyerpw@localhost:8054 --caname ca-org2 -M $  
↪${PWD}/organizations/peerOrganizations/org2.example.com/users/buyer@org2.example.com/  
↪msp --tls.certfiles ${PWD}/organizations/fabric-ca/org2/tls-cert.pem
```

Run the command below to copy the Node OU configuration file into the buyer identity MSP folder.

```
cp ${PWD}/organizations/peerOrganizations/org2.example.com/msp/config.yaml ${PWD}/  
↪organizations/peerOrganizations/org2.example.com/users/buyer@org2.example.com/msp/  
↪config.yaml
```

7.4.7 Create an asset in private data

Now that we have created the identity of the asset owner, we can invoke the private data smart contract to create a new asset. Copy and paste the following set of commands into your terminal in the *test-network* directory:

Try it yourself

```
export PATH=${PWD}/../bin:$PATH  
export FABRIC_CFG_PATH=${PWD}/../config/  
export CORE_PEER_TLS_ENABLED=true  
export CORE_PEER_LOCALMSPID="Org1MSP"  
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.  
↪example.com/peers/peer0.org1.example.com/tls/ca.crt  
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.  
↪com/users/owner@org1.example.com/msp  
export CORE_PEER_ADDRESS=localhost:7051
```


We will use the `CreateAsset` function to create an asset that is stored in private data — `assetID` `asset1` with a color `green`, size `20` and `appraisedValue` of `100`. Recall that private data **`appraisedValue`** will be stored separately from the private data **`assetID`**, **`color`**, **`size`**. For this reason, the `CreateAsset` function calls the `PutPrivateData()` API twice to persist the private data, once for each collection. Also note that the private data is passed using the `--transient` flag. Inputs passed as transient data will not be persisted in the transaction in order to keep the data private. Transient data is passed as binary data and therefore when using terminal it must be base64 encoded. We use an environment variable to capture the base64 encoded value, and use `tr` command to strip off the problematic newline characters that linux `base64` command adds.

Run the following command to create the asset:

```
export ASSET_PROPERTIES=$(echo -n "{\"objectType\":\"asset\",\"assetID\":\"asset1\", \"color\":\"green\", \"size\":20, \"appraisedValue\":100}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n private -c '{"function":"CreateAsset","Args":[]}' --transient "{\"asset_properties\": \"${ASSET_PROPERTIES}\"}"
```

You should see results similar to:

```
[chaincodeCmd] chaincodeInvokeOrQuery->INFO 001 Chaincode invoke successful. result: status:200
```

Note that command above only targets the `Org1` peer. The `CreateAsset` transaction writes to two collections, `assetCollection` and `Org1MSPPrivateCollection`. The `Org1MSPPrivateCollection` requires an endorsement from the `Org1` peer in order to write to the collection, while the `assetCollection` inherits the endorsement policy of the chaincode, `"OR('Org1MSP.peer', 'Org2MSP.peer')"`. An endorsement from the `Org1` peer can meet both endorsement policies and is able to create an asset without an endorsement from `Org2`.

7.4.8 Query the private data as an authorized peer

Our collection definition allows all peers of `Org1` and `Org2` to have the `assetID`, `color`, `size`, and owner private data in their side database, but only peers in `Org1` can have `Org1`'s opinion of their `appraisedValue` private data in their side database. As an authorized peer in `Org1`, we will query both sets of private data.

The first query command calls the `ReadAsset` function which passes `assetCollection` as an argument.

```
// ReadAsset reads the information from collection
func (s *SmartContract) ReadAsset(ctx contractapi.TransactionContextInterface, assetID string) (*Asset, error) {

    log.Printf("ReadAsset: collection %v, ID %v", assetCollection, assetID)
    assetJSON, err := ctx.GetStub().GetPrivateData(assetCollection, assetID) //get the asset from chaincode state
    if err != nil {
        return nil, fmt.Errorf("failed to read asset: %v", err)
    }

    //No Asset found, return empty response
    if assetJSON == nil {
        log.Printf("%v does not exist in collection %v", assetID, assetCollection)
        return nil, nil
    }

    var asset *Asset
```

(continues on next page)

(continued from previous page)

```

err = json.Unmarshal(assetJSON, &asset)
if err != nil {
    return nil, fmt.Errorf("failed to unmarshal JSON: %v", err)
}

return asset, nil
}

```

The second query command calls the `ReadAssetPrivateDetails` function which passes `Org1MSPPrivateDetails` as an argument.

```

// ReadAssetPrivateDetails reads the asset private details in organization specific_
↳collection
func (s *SmartContract) ReadAssetPrivateDetails(ctx contractapi.
↳TransactionContextInterface, collection string, assetID string) _
↳(*AssetPrivateDetails, error) {
    log.Printf("ReadAssetPrivateDetails: collection %v, ID %v", collection, assetID)
    assetDetailsJSON, err := ctx.GetStub().GetPrivateData(collection, assetID) // _
↳Get the asset from chaincode state
    if err != nil {
        return nil, fmt.Errorf("failed to read asset details: %v", err)
    }
    if assetDetailsJSON == nil {
        log.Printf("AssetPrivateDetails for %v does not exist in collection %v", _
↳assetID, collection)
        return nil, nil
    }

    var assetDetails *AssetPrivateDetails
    err = json.Unmarshal(assetDetailsJSON, &assetDetails)
    if err != nil {
        return nil, fmt.Errorf("failed to unmarshal JSON: %v", err)
    }

    return assetDetails, nil
}

```

Now Try it yourself

We can read the main details of the asset that was created by using the `ReadAsset` function to query the `assetCollection` collection as `Org1`:

```

peer chaincode query -C mychannel -n private -c '{"function":"ReadAsset","Args":{
↳"asset1"}}'

```

When successful, the command will return the following result:

```

{"objectType":"asset","assetID":"asset1","color":"green","size":20,"owner":
↳"x509::CN=appUser1,OU=admin,O=Hyperledger,ST=North Carolina,C=US::CN=ca.org1.
↳example.com,O=org1.example.com,L=Durham,ST=North Carolina,C=US"}

```

The “owner” of the asset is the identity that created the asset by invoking the smart contract. The private data smart contract uses the `GetClientIdentity().GetID()` API to read the name and issuer of the identity certificate. You can see the name and issuer of the identity certificate, in the owner attribute.

Query for the `appraisedValue` private data of `asset1` as a member of `Org1`.

```
peer chaincode query -C mychannel -n private -c '{"function":"ReadAssetPrivateDetails"
↪","Args":["Org1MSPPrivateCollection","asset1"]}'
```

You should see the following result:

```
{"assetID":"asset1","appraisedValue":100}
```

7.4.9 Query the private data as an unauthorized peer

Now we will operate a user from Org2. Org2 has the asset transfer private data `assetID`, `color`, `size`, `owner` in its side database as defined in the `assetCollection` policy, but does not store the `asset appraisedValue` data for Org1. We will query for both sets of private data.

Switch to a peer in Org2

Run the following commands to operate as an Org2 member and query the Org2 peer.

Try it yourself

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/buyer@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

Query private data Org2 is authorized to

Peers in Org2 should have the first set of asset transfer private data (`assetID`, `color`, `size` and `owner`) in their side database and can access it using the `ReadAsset()` function which is called with the `assetCollection` argument.

Try it yourself

```
peer chaincode query -C mychannel -n private -c '{"function":"ReadAsset","Args":["
↪"asset1"]}'
```

When successful, should see something similar to the following result:

```
{"objectType":"asset","assetID":"asset1","color":"green","size":20,
"owner":"x509::CN=appUser1,OU=admin,O=Hyperledger,ST=North Carolina,C=US::CN=ca.org1.
↪example.com,O=org1.example.com,L=Durham,ST=North Carolina,C=US" }
```

Query private data Org2 is not authorized to

Because the asset was created by Org1, the `appraisedValue` associated with `asset1` is stored in the `Org1MSPPrivateCollection` collection. The value is not stored by peers in Org2. Run the following command to demonstrate that the asset's `appraisedValue` is not stored in the `Org2MSPPrivateCollection` on the Org2 peer:

Try it yourself

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪private -c '{"function":"ReadAssetPrivateDetails","Args":["Org2MSPPrivateCollection
↪","asset1"]}]'
```

The empty response shows that the asset1 private details do not exist in buyer (Org2) private collection.

Nor can a user from Org2 read the Org1 private data collection:

```
peer chaincode query -C mychannel -n private -c '{"function":"ReadAssetPrivateDetails
↪","Args":["Org1MSPPrivateCollection","asset1"]}]'
```

By setting "memberOnlyRead": true in the collection configuration file, we specify that only clients from Org1 can read data from the collection. An Org2 client who tries to read the collection would only get the following response:

```
Error: endorsement failure during query. response: status:500 message:"failed to
read asset details: GET_STATE failed: transaction ID:_
↪d23e4bc0538c3abfb7a6bd4323fd5f52306e2723be56460fc6da0e5acaee6b23: tx
creator does not have read access permission on privatedata in chaincodeName:private_
↪collectionName: Org1MSPPrivateCollection"
```

Users from Org2 will only be able to see the public hash of the private data.

7.4.10 Transfer the Asset

Let's see what it takes to transfer asset1 to Org2. In this case, Org2 needs to agree to buy the asset from Org1, and they need to agree on the appraisedValue. You may be wondering how they can agree if Org1 keeps their opinion of the appraisedValue in their private side database. For the answer to this, let's continue.

Try it yourself

Switch back to the terminal with our peer CLI.

To transfer an asset, the buyer (recipient) needs to agree to the same appraisedValue as the asset owner, by calling chaincode function AgreeToTransfer. The agreed value will be stored in the Org2MSPDetailsCollection collection on the Org2 peer. Run the following commands to agree to the appraised value of 100 as Org2:

```
export ASSET_VALUE=$(echo -n "{\"assetID\":\"asset1\",\"appraisedValue\":100}" |_
↪base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪private -c '{"function":"AgreeToTransfer","Args":[]}' --transient '{"asset_value\
↪":"$ASSET_VALUE\"}'
```

The buyer can now query the value they agreed to in the Org2 private data collection:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪private -c '{"function":"ReadAssetPrivateDetails","Args":["Org2MSPPrivateCollection
↪","asset1"]}]'
```

The invoke will return the following value:

```
{ "assetID": "asset1", "appraisedValue": 100 }
```

Now that buyer has agreed to buy the asset for the appraised value, the owner can transfer the asset to Org2. The asset needs to be transferred by the identity that owns the asset, so lets go acting as Org1:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
  ↳ com/users/owner@org1.example.com/msp
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
  ↳ example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_ADDRESS=localhost:7051
```

The owner from Org1 can read the data added by the *AgreeToTransfer* transaction to view the buyer identity:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
  ↳ com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
  ↳ orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
  ↳ private -c '{"function": "ReadTransferAgreement", "Args": ["asset1"]}'
```

```
{ "assetID": "asset1", "buyerID":
  ↳ "eDUwOT06Q049YnV5ZXIsTlU9Y2xpZW50LE89SHlwZXJsZWRnZXIsU1Q9Tm9ydGggQ2Fyb2xpbnEsQz1VUzo6Q049Y2Eub3JnM.
  ↳ " }
```

We now have all we need to transfer the asset. The smart contract uses the `GetPrivateDataHash()` function to check that the hash of the asset appraisal value in `Org1MSPPrivateCollection` matches the hash of the appraisal value in the `Org2MSPPrivateCollection`. If the hashes are the same, it confirms that the owner and the interested buyer have agreed to the same asset value. If the conditions are met, the transfer function will get the client ID of the buyer from the transfer agreement and make the buyer the new owner of the asset. The transfer function will also delete the asset appraisal value from the collection of the former owner, as well as remove the transfer agreement from the `assetCollection`.

Run the following commands to transfer the asset. The owner needs to provide the `assetID` and the organization MSP ID of the buyer to the transfer transaction:

```
export ASSET_OWNER=$(echo -n "{\"assetID\": \"asset1\", \"buyerMSP\": \"Org2MSP\"}" |_
  ↳ base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
  ↳ com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
  ↳ orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
  ↳ private -c '{"function": "TransferAsset", "Args": []}' --transient "{\"asset_owner\": \"_
  ↳ $ASSET_OWNER\"}" --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/_
  ↳ organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/_
  ↳ ca.crt
```

You can query `asset1` to see the results of the transfer:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
  ↳ com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
  ↳ orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
  ↳ private -c '{"function": "ReadAsset", "Args": ["asset1"]}'
```

The results will show that the buyer identity now owns the asset:

```
{ "objectType": "asset", "assetID": "asset1", "color": "green", "size": 20, "owner":
  ↳ "x509::CN=appUser2, OU=client + OU=org2 + OU=department1::CN=ca.org2.example.com,_
  ↳ O=org2.example.com, L=Hursley, ST=Hampshire, C=UK" }
```

The “owner” of the asset now has the buyer identity.

You can also confirm that transfer removed the private details from the Org1 collection:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪private -c '{"function":"ReadAssetPrivateDetails","Args":["Org1MSPPrivateCollection
↪","asset1"]}'
```

Your query will return empty result, since the asset private data is removed from the Org1 private data collection.

7.4.11 Purge Private Data

For use cases where private data only needs to be persisted for a short period of time, it is possible to “purge” the data after a certain set number of blocks, leaving behind only a hash of the data that serves as immutable evidence of the transaction. An organization could decide to purge private data if the data contained sensitive information that was used by another transaction, but is not longer needed, or if the data is being replicated into an off-chain database.

The `appraisedValue` data in our example contains a private agreement that the organization may want to expire after a certain period of time. Thus, it has a limited lifespan, and can be purged after existing unchanged on the blockchain for a designated number of blocks using the `blockToLive` property in the collection definition.

The `Org2MSPPrivateCollection` definition has a `blockToLive` property value of 3, meaning this data will live on the side database for three blocks and then after that it will get purged. If we create additional blocks on the channel, the `appraisedValue` agreed to by Org2 will eventually get purged. We can create 3 new blocks to demonstrate:

Try it yourself

Run the following commands in your terminal to switch back to operating as member of Org2 and target the Org2 peer:

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/buyer@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

We can still query the `appraisedValue` in the `Org2MSPPrivateCollection`:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪private -c '{"function":"ReadAssetPrivateDetails","Args":["Org2MSPPrivateCollection
↪","asset1"]}'
```

You should see the value printed in your logs:

```
{"assetID":"asset1","appraisedValue":100}
```

Since we need to keep track of how many blocks we are adding before the private data gets purged, open a new terminal window and run the following command to view the private data logs for the Org2 peer. Note the highest block number.

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

Now return to the terminal where we are acting as a member of Org2 and run the following commands to create three new assets. Each command will create a new block.

```
export ASSET_PROPERTIES=$(echo -n "{\"objectType\":\"asset\", \"assetID\":\"asset2\", \"color\":\"blue\", \"size\":30, \"appraisedValue\":100}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n private -c '{"function": "CreateAsset", "Args": []}' --transient "{\"asset_properties\": \"${ASSET_PROPERTIES}\"}
```

```
export ASSET_PROPERTIES=$(echo -n "{\"objectType\":\"asset\", \"assetID\":\"asset3\", \"color\":\"red\", \"size\":25, \"appraisedValue\":100}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n private -c '{"function": "CreateAsset", "Args": []}' --transient "{\"asset_properties\": \"${ASSET_PROPERTIES}\"}
```

```
export ASSET_PROPERTIES=$(echo -n "{\"objectType\":\"asset\", \"assetID\":\"asset4\", \"color\":\"orange\", \"size\":15, \"appraisedValue\":100}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n private -c '{"function": "CreateAsset", "Args": []}' --transient "{\"asset_properties\": \"${ASSET_PROPERTIES}\"}
```

Return to the other terminal and run the following command to confirm that the new assets resulted in the creation of three new blocks:

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

The appraisedValue has now been purged from the Org2MSPDetailsCollection private data collection. Issue the query again from the Org2 terminal to see that the response is empty.

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n private -c '{"function": "ReadAssetPrivateDetails", "Args": ["Org2MSPPrivateCollection", "asset1"]}'
```

7.4.12 Using indexes with private data

Indexes can also be applied to private data collections, by packaging indexes in the META-INF/statedb/couchdb/collections/<collection_name>/indexes directory alongside the chaincode. An example index is available [here](#).

For deployment of chaincode to production environments, it is recommended to define any indexes alongside chaincode so that the chaincode and supporting indexes are deployed automatically as a unit, once the chaincode has been installed on a peer and instantiated on a channel. The associated indexes are automatically deployed upon chaincode instantiation on the channel when the `--collections-config` flag is specified pointing to the location of the collection JSON file.

Note: It is not possible to create an index for use with an implicit private data collection. An implicit collection is based on the organizations name and is created automatically. The format of the name is

`_implicit_org_<OrgsMSPid>` Please see [FAB-17916](#) for more information.

7.4.13 Clean up

When you are finished using the private data smart contract, you can bring down the test network using `network.sh` script.

```
./network.sh down
```

This command will bring down the CAs, peers, and ordering node of the network that we created. Note that all of the data on the ledger will be lost. If you want to go through the tutorial again, you will start from a clean initial state.

7.4.14 Additional resources

For additional private data education, a video tutorial has been created.

Note: The video uses the previous lifecycle model to install private data collections with chaincode.

7.5 Secured asset transfer in Fabric

This tutorial will demonstrate how an asset can be represented and traded between organizations in a Hyperledger Fabric blockchain channel, while keeping details of the asset and transaction private using private data. Each on-chain asset is a non-fungible token (NFT) that represents a specific asset having certain immutable metadata properties (such as size and color) with a unique owner. When the owner wants to sell the asset, both parties need to agree to the same price before the asset is transferred. The private asset transfer smart contract enforces that only the owner of the asset can transfer the asset. In the course of this tutorial, you will learn how Fabric features such as state based endorsement, private data, and access control come together to provide secured transactions that are both private and verifiable.

This tutorial will deploy the [secured asset transfer sample](#) to demonstrate how to transfer a private asset between two organizations without publicly sharing data. You should have completed the task [Install Samples, Binaries, and Docker Images](#).

7.5.1 Scenario requirements

The private asset transfer scenario is bound by the following requirements:

- An asset may be issued by the first owner's organization (in the real world issuance may be restricted to some authority that certifies an asset's properties).
- Ownership is managed at the organization level (the Fabric permissioning scheme would equally support ownership at an individual identity level within an organization).
- The asset identifier and owner is stored as public channel data for all channel members to see.
- The asset metadata properties however are private information known only to the asset owner (and prior owners).
- An interested buyer will want to verify an asset's private properties.
- An interested buyer will want to verify an asset's provenance, specifically the asset's origin and chain of custody. They will also want to verify that the asset has not changed since issuance, and that all prior transfers have been legitimate.

- To transfer an asset, a buyer and seller must first agree on the sales price.
- Only the current owner may transfer their asset to another organization.
- The actual private asset transfer must verify that the legitimate asset is being transferred, and verify that the price has been agreed to. Both buyer and seller must endorse the transfer.

7.5.2 How privacy is maintained

The smart contract uses the following techniques to ensure that the asset properties remain private:

- The asset metadata properties are stored in the current owning organization's implicit private data collection on the organization's peers only. Each organization on a Fabric channel has a private data collection that their own organization can use. This collection is *implicit* because it does not need to be explicitly defined in the chaincode.
- Although a hash of the private properties is automatically stored on-chain for all channel members to see, a random salt is included in the private properties so that other channel members cannot guess the private data pre-image through a dictionary attack.
- Smart contract requests utilize the transient field for private data so that private data does not get included in the final on-chain transaction.
- Private data queries must originate from a client whose org id matches the peer's org id, which must be the same as the asset owner's org id.

7.5.3 How the transfer is implemented

Before we start using the private asset transfer smart contract we will provide an overview of the transaction flow and how Fabric features are used to protect the asset created on the blockchain:

Creating the asset

The private asset transfer smart contract is deployed with an endorsement policy that requires an endorsement from any channel member. This allows any organization to create an asset that they own without requiring an endorsement from other channel members. The creation of the asset is the only transaction that uses the chaincode level endorsement policy. Transactions that update or transfer existing assets will be governed by state based endorsement policies or the endorsement policies of private data collections. Note that in other scenarios, you may want an issuing authority to also endorse create transactions.

The smart contract uses the following Fabric features to ensure that the asset can only be updated or transferred by the organization that owns the asset:

- When the asset is created, the smart contract gets the MSP ID of the organization that submitted the request, and stores the MSP ID as the owner in the asset key/value in the public chaincode world state. Subsequent smart contract requests to update or transfer the asset will use access control logic to verify that the requesting client is from the same organization. Note that in other scenarios, the ownership could be based on a specific client identity within an organization, rather than an organization itself.
- Also when the asset is created, the smart contract sets a state based endorsement policy for the asset key. The state based policy specifies that a peer from the organization that owns the asset must endorse a subsequent request to update or transfer the asset. This prevents any other organization from updating or transferring the asset using a smart contract that has been maliciously altered on their own peers.

Agreeing to the transfer

After an asset is created, channel members can use the smart contract to agree to transfer the asset:

- The owner of the asset can change the description in the public ownership record, for example to advertise that the asset is for sale. Smart contract access control enforces that this change needs to be submitted from a member of the asset owner organization. The state based endorsement policy enforces that this description change must be endorsed by a peer from the owner's organization.

The asset owner and the asset buyer agree to transfer the asset for a certain price:

- The price agreed to by the buyer and the seller is stored in each organization's implicit private data collection. The private data collection keeps the agreed price secret from other members of the channel. The endorsement policy of the private data collection ensures that the respective organization's peer endorsed the price agreement, and the smart contract access control logic ensures that the price agreement was submitted by a client of the associated organization.
- A hash of each price agreement is stored on the ledger. The two hashes will match only if the two organizations have agreed to the same price. This allows the organizations to verify that they have come to agreement on the transfer details before the transfer takes place. A random trade id is added to the price agreement, which serves as a *salt* to ensure that other channel members can not use the hash on the ledger to guess the price.

Transferring the asset

After the two organizations have agreed to the same price, the asset owner can use the transfer function to transfer the asset to the buyer:

- Smart contract access control ensures that the transfer must be initiated by a member of the organization that owns the asset.
- The transfer function verifies that the asset's private immutable properties passed to the transfer function matches the on chain hash of the asset data in private collection, to ensure that the asset owner is *selling* the same asset that they own.
- The transfer function uses the hash of the price agreement on the ledger to ensure that both organizations have agreed to the same price.
- If the transfer conditions are met, the transfer function adds the asset to the implicit private data collection of the buyer, and deletes the asset from the collection of the seller. The transfer also updates the owner in the public ownership record.
- Because of the endorsement policies of the seller and buyer implicit data collections, and the state based endorsement policy of the public record (requiring the seller to endorse), the transfer needs to be endorsed by peers from both buyer and seller.
- The state based endorsement policy of the public asset record is updated so that only a peer of the new owner of the asset can update or sell their new asset.
- The price agreements are also deleted from both the seller and buyer implicit private data collection, and a sales receipt is created in each private data collection.

7.5.4 Running the private asset transfer smart contract

You can use the Fabric test network to run the private asset transfer smart contract. The test network contains two peer organizations, Org1 and Org2, that operate one peer each. In this tutorial, we will deploy the smart contract to a channel of the test network joined by both organizations. We will first create an asset that is owned by Org1. After the two organizations agree on the price, we will transfer the asset from Org1 to Org2.

7.5.5 Deploy the test network

We are going to use the Fabric test network to run the secured asset transfer smart contract. Open a command terminal and navigate to test network directory in your local clone of `fabric-samples`. We will operate from the `test-network` directory for the remainder of the tutorial.

```
cd fabric-samples/test-network
```

First, bring down any running instances of the test network:

```
./network.sh down
```

You can then deploy a new instance the network with the following command:

```
./network.sh up createChannel -c mychannel
```

The script will deploy the nodes of the network and create a single channel named `mychannel` with `Org1` and `Org2` as channel members. We will use this channel to deploy the smart contract and trade our asset.

7.5.6 Deploy the smart contract

You can use the test network script to deploy the secured asset transfer smart contract to the channel. Run the following command to deploy the smart contract to `mychannel`:

```
./network.sh deployCC -ccn secured -ccp ../asset-transfer-secured-agreement/chaincode-  
go/ -ccl go -ccep "OR('Org1MSP.peer','Org2MSP.peer')"
```

Note that we are using the `-ccep` flag to deploy the smart contract with an endorsement policy of `"OR('Org1MSP.peer','Org2MSP.peer')"`. This allows either organization to create an asset without receiving an endorsement from the other organization.

Set the environment variables to operate as Org1

In the course of running this sample, you need to interact with the network as both `Org1` and `Org2`. To make the tutorial easier to use, we will use separate terminals for each organization. Open a new terminal and make sure that you are operating from the `test-network` directory. Set the following environment variables to operate the `peer` CLI as the `Org1` admin:

```
export PATH=${PWD}/../bin:${PWD}:$PATH  
export FABRIC_CFG_PATH=$PWD/../config/  
export CORE_PEER_TLS_ENABLED=true  
export CORE_PEER_LOCALMSPID="Org1MSP"  
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.  
com/users/Admin@org1.example.com/msp  
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.  
example.com/peers/peer0.org1.example.com/tls/ca.crt  
export CORE_PEER_ADDRESS=localhost:7051
```

The environment variables also specify the endpoint information of the `Org1` peer to submit requests.

Set the environment variables to operate as Org2

Now that we have one terminal that we can operate as `Org1`, open a new terminal for `Org2`. Make sure that this terminal is also operating from the `test-network` directory. Set the following environment variables to operate as the `Org2`

admin:

```
export PATH=${PWD}/../bin:${PWD}:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_ADDRESS=localhost:9051
```

You will need switch between the two terminals as you go through the tutorial.

7.5.7 Create an asset

Any channel member can use the smart contract to create an asset that is owned by their organization. The details of the asset will be stored in a private data collection, and can only accessed by the organization that owns the asset. A public record of the asset, its owner, and a public description is stored on the channel ledger. Any channel member can access the public ownership record to see who owns the asset, and can read the description to see if the asset is for sale.

Operate from the Org1 terminal

Before we create the asset, we need to specify the details of what our asset will be. Issue the following command to create a JSON that will describe the asset. The "salt" parameter is a random string that would prevent another member of the channel from guessing the asset using the hash on the ledger. If there was no salt, a user could theoretically guess asset parameters until the hash of the of the guess and the hash on the ledger matched (this is known as a dictionary attack). This string is encoded in Base64 format so that it can be passed to the creation transaction as transient data.

```
export ASSET_PROPERTIES=$(echo -n '{"object_type":"asset_properties","asset_id"
↪":"asset1","color":"blue","size":35,"salt":\
↪"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3"}' | base64 | tr -d \n)
```

We can now use the following command to create a asset that belongs to Org1:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"CreateAsset","Args":["asset1", "A new asset for Org1MSP"]}'
↪' --transient '{"asset_properties":"$ASSET_PROPERTIES"}'
```

We can can query the Org1 implicit data collection to see the asset that was created:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"GetAssetPrivateProperties","Args":["asset1"]}'
```

When successful, the command will return the following result:

```
{"object_type":"asset_properties","asset_id":"asset1","color":"blue","size":35,"salt":
↪"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3"}
```

We can also query the ledger to see the public ownership record:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function": "ReadAsset", "Args": ["asset1"]}'
```

The command will return the record that the asset1 is owned by Org1:

```
{"object_type": "asset", "asset_id": "asset1", "owner_org": "Org1MSP", "public_description":
↪ "A new asset for Org1MSP"}
```

Because the market for assets is hot, Org1 wants to flip this asset and put it up for sale. As the asset owner, Org1 can update the public description to advertise that the asset is for sale. Run the following command to change the asset description:

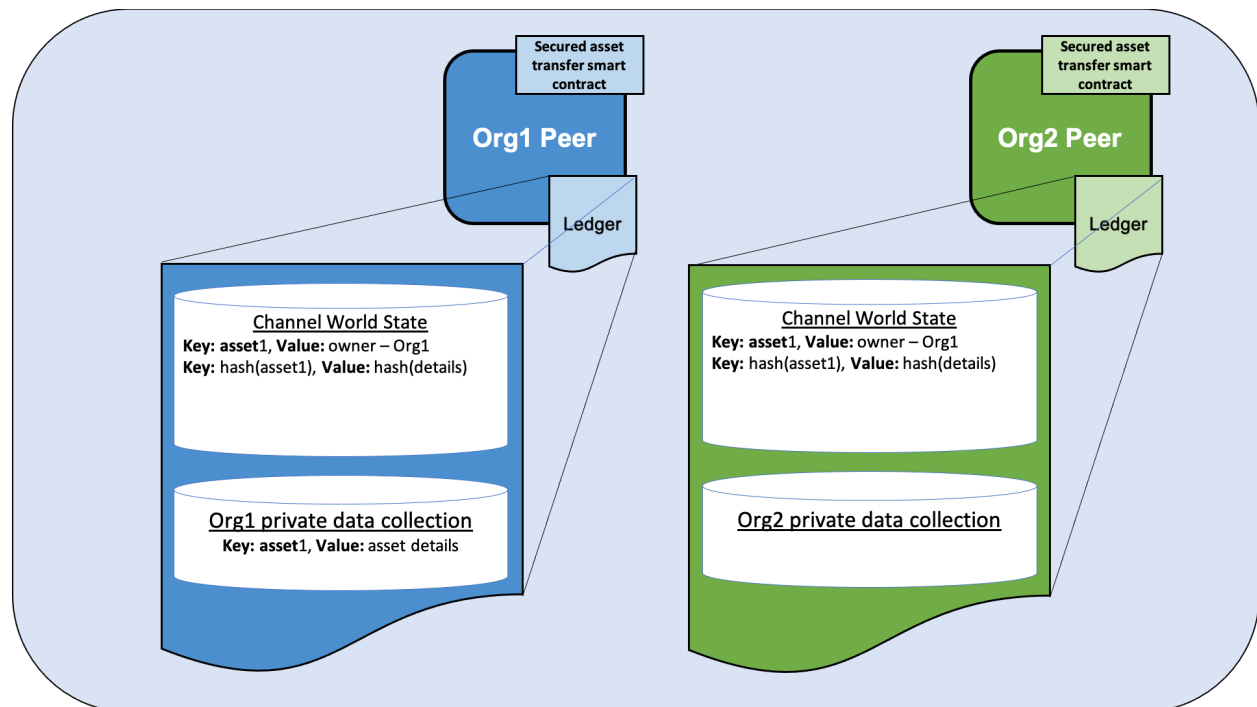
```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function": "ChangePublicDescription", "Args": ["asset1", "This asset is_
↪for sale"]}'
```

Query the ledger again to see the updated description:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function": "ReadAsset", "Args": ["asset1"]}'
```

We can now see that the asset is for sale:

```
{"object_type": "asset", "asset_id": "asset1", "owner_org": "Org1MSP", "public_description":
↪ "This asset is for sale"}
```



Figure

1: When Org1 creates an asset that they own, the asset details are stored in the Org1 implicit data collection on the

Org1 peer: The public ownership record is stored in the channel world state, and is stored on both the Org1 and Org2 peers. A hash of the asset key and a hash the asset details are also visible in the channel world state and are stored on the peers of both organizations.

Operate from the Org2 terminal

If we operate from the Org2 terminal, we can use the smart contract query the public asset data:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"ReadAsset","Args":["asset1"]}'
```

From this query, Org2 learns that asset1 is for sale:

```
{"object_type":"asset","asset_id":"asset1","owner_org":"Org1MSP","public_description":
↪"This asset is for sale"}
```

In a real chaincode you may want to query for all assets for sale, by using a JSON query, or by creating a different sale key and using a key range query to find the assets currently for sale. Any changes to the public description of the asset owned by Org1 needs to be endorsed by Org1. The endorsement policy is reinforced by an access control policy within the chaincode that any update needs to be submitted by the organization that owns the asset. Lets see what happens if Org2 tried to change the public description as a prank:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"ChangePublicDescription","Args":["asset1","the worst asset
↪"]}'
```

The smart contract does not allow Org2 to access the public description of the asset.

```
Error: endorsement failure during invoke. response: status:500 message:"a client from_
↪Org2MSP cannot update the description of a asset owned by Org1MSP"
```

7.5.8 Agree to sell the asset

To sell an asset, both the buyer and the seller must agree on an asset price. Each party stores the price that they agree to in their own private data collection. The private asset transfer smart contract enforces that both parties need to agree to the same price before the asset can be transferred.

7.5.9 Agree to sell as Org1

Operate from the Org1 terminal. Org1 will agree to set the asset price as 110 dollars. The `trade_id` is used as salt to prevent a channel member that is not a buyer or a seller from guessing the price. This value needs to be passed out of band, through email or other communication, between the buyer and the seller. The buyer and the seller can also add salt to the asset key to prevent other members of the channel from guessing which asset is for sale.

```
export ASSET_PRICE=$(echo -n "{\"asset_id\":\"asset1\",\"trade_id\":\
↪\"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\",\"price\":110}\"" | base64)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"AgreeToSell","Args":["asset1"]}' --transient '{"asset
↪price\":\"$ASSET_PRICE\"}'" (continues on next page)
```

(continued from previous page)

We can query the Org1 private data collection to read the agreed to selling price:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"GetAssetSalesPrice","Args":["asset1"]}'
```

7.5.10 Agree to buy as Org2

Operate from the Org2 terminal. Run the following command to verify the asset properties before agreeing to buy. The asset properties and salt would be passed out of band, through email or other communication, between the buyer and seller.

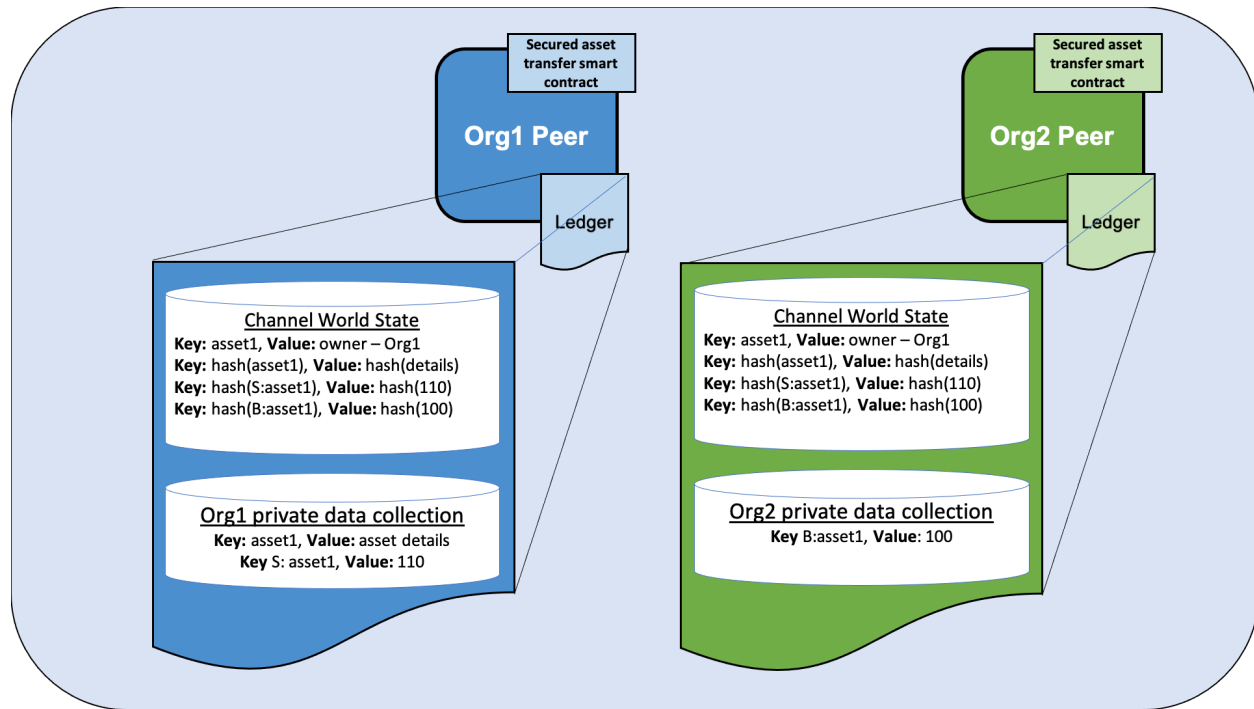
```
export ASSET_PROPERTIES=$(echo -n '{"object_type":"asset_properties","asset_id\
↪":"asset1","color":"blue","size":35,"salt\
↪":"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3\
↪"}' | base64)
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"VerifyAssetProperties","Args":["asset1"]}' --transient '{"\
↪asset_properties":"$ASSET_PROPERTIES"}'
```

Run the following command to agree to buy asset1 for 100 dollars. As of now, Org2 will agree to a different price than Org2. Don't worry, the two organizations will agree to the same price in a future step. However, we we can use this temporary disagreement as a test of what happens if the buyer and the seller agree to a different price. Org2 needs to use the same trade_id as Org1.

```
export ASSET_PRICE=$(echo -n '{"asset_id":"asset1","trade_id\
↪":"109f4b3c50d7b0df729d299bc6f8e9ef9066971f","price\
↪":"100"}' | base64)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"AgreeToBuy","Args":["asset1"]}' --transient '{"asset_
↪price":"$ASSET_PRICE"}'
```

You can read the agreed purchase price from the Org2 implicit data collection:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"GetAssetBidPrice","Args":["asset1"]}'
```



Figure

2: After Org1 and Org2 agree to transfer the asset, the price agreed to by each organization is stored in their private data collections. A composite key for the seller and the buyer is used to prevent a collision with the asset details and asset ownership record. The price that is agreed to is only stored on the peers of each organization. However, the hash of both agreements is stored in the channel world state on every peer joined to the channel.

7.5.11 Transfer the asset from Org1 to Org2

After both organizations have agreed to their price, Org1 can attempt to transfer the asset to Org2. The private asset transfer function in the smart contract uses the hash on the ledger to check that both organizations have agreed to the same price. The function will also use the hash of the private asset details to check that the asset that is transferred is the same asset that Org1 owns.

Transfer the asset as Org1

Operate from the Org1 terminal. The owner of the asset needs to initiate the transfer. Note that the command below uses the `--peerAddresses` flag to target the peers of both Org1 and Org2. Both organizations need to endorse the transfer. Also note that the asset properties and price are passed in the transfer request as transient properties. These are passed so that the current owner can be sure that the correct asset is transferred for the correct price. These properties will be checked against the on-chain hashes by both endorsers.

```

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↪secured -c '{"function":"TransferAsset","Args":["asset1","Org2MSP"]}' --transient "
↪{"asset_properties\":"$ASSET_PROPERTIES\","asset_price\":"$ASSET_PRICE\}" --
↪peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt

```

Because the two organizations have not agreed to the same price, the transfer cannot be completed:


```
Error: endorsement failure during invoke. response: status:500 message:"failed_
↳transfer verification: hash_
↳cf74b8ce092b637bd28f98f7cdd490534c102a0665e7c985d4f2ab9810e30b1c for passed price_
↳JSON {\\"asset_id\\":\\"asset1\\",\\"trade_id\\":\
↳"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\\",\\"price\\":110} does not match on-chain_
↳hash 09341dbb39e81fb50ccb3a81770254525318f777fad217ae49777487116cceb4, buyer hasn't_
↳agreed to the passed trade id and price"
```

As a result, Org1 and Org2 come to a new agreement on the price at which the asset will be purchased. Org1 drops the price of the asset to 100:

```
export ASSET_PRICE=$(echo -n {\\"asset_id\\":\\"asset1\\",\\"trade_id\\":\
↳"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\\",\\"price\\":100}" | base64)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↳com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↳orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↳secured -c '{"function":"AgreeToSell","Args":["asset1"]}' --transient {\\"asset_
↳price\\":\\"$ASSET_PRICE\\"}"
```

Now that the buyer and seller have agreed to the same price, Org1 can transfer the asset to Org2.

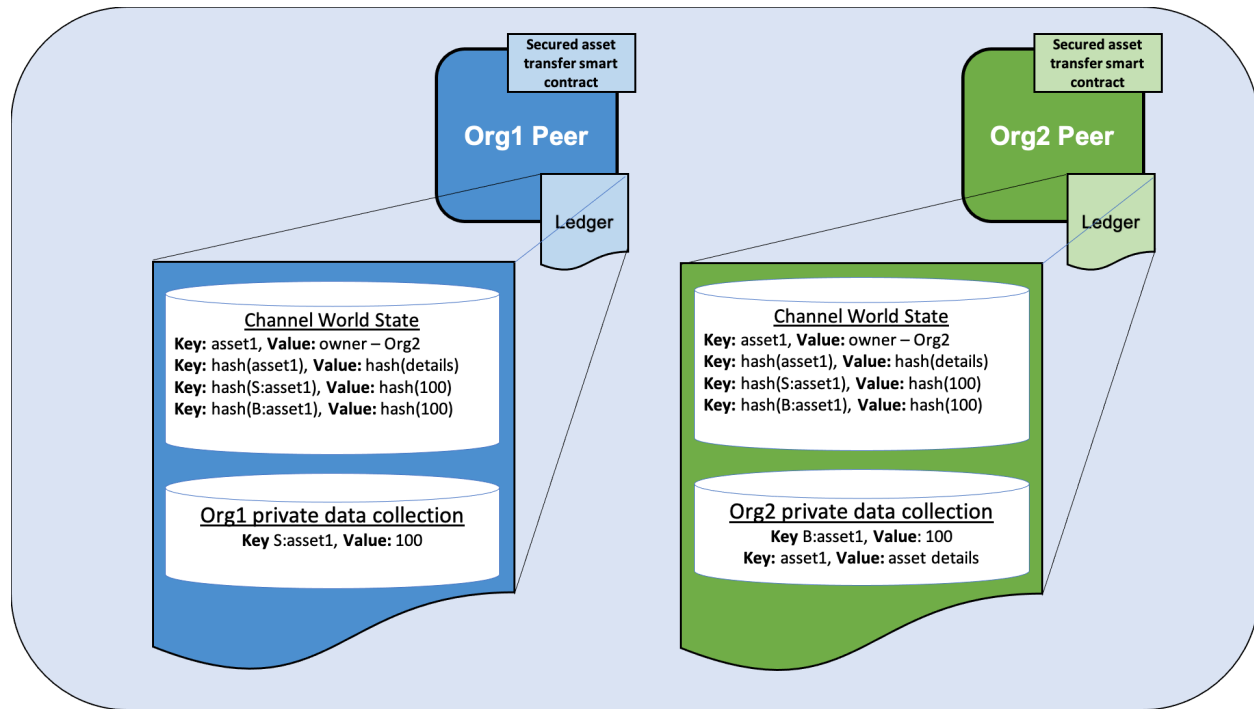
```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↳com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↳orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↳secured -c '{"function":"TransferAsset","Args":["asset1","Org2MSP"]}' --transient "
↳{\\"asset_properties\\":\\"$ASSET_PROPERTIES\\",\\"asset_price\\":\\"$ASSET_PRICE\\"}" --
↳peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↳peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↳peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↳peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

You can query the asset ownership record to verify that the transfer was successful.

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↳com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↳orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n_
↳secured -c '{"function":"ReadAsset","Args":["asset1"]}'
```

The record now lists Org2 as the asset owner:

```
{"object_type":"asset","asset_id":"asset1","owner_org":"Org2MSP","public_description":
↳"This asset is for sale"}
```



Figure

3: After the asset is transferred, the asset details are placed in the Org2 implicit data collection and deleted from the Org1 implicit data collection. As a result, the asset details are now only stored on the Org2 peer. The asset ownership record on the ledger is updated to reflect that the asset is owned by Org1.

Update the asset description as Org2

Operate from the Org2 terminal. Now that Org2 owns the asset, we can read the asset details from the Org2 implicit data collection:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n secured -c '{"function": "GetAssetPrivateProperties", "Args": ["asset1"]}'
```

Org2 can now update the asset public description:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n secured -c '{"function": "ChangePublicDescription", "Args": ["asset1", "This asset is not for sale"]}'
```

Query the ledger to verify that the asset is no longer for sale:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n secured -c '{"function": "ReadAsset", "Args": ["asset1"]}'
```

7.5.12 Clean up

When you are finished transferring assets, you can bring down the test network. The command will remove all the nodes of the test network, and delete any ledger data that you created:

```
./network.sh down
```

7.6 Using CouchDB

This tutorial will describe the steps required to use CouchDB as the state database with Hyperledger Fabric. By now, you should be familiar with Fabric concepts and have explored some of the samples and tutorials.

Note: These instructions use the new Fabric chaincode lifecycle introduced in the Fabric v2.0 release. If you would like to use the previous lifecycle model to use indexes with chaincode, visit the v1.4 version of the [Using CouchDB](#).

The tutorial will take you through the following steps:

1. *[Enable CouchDB in Hyperledger Fabric](#)*
2. *[Create an index](#)*
3. *[Add the index to your chaincode folder](#)*
4. *[Deploy the smart contract](#)*
5. *[Query the CouchDB State Database](#)*
6. *[Use best practices for queries and indexes](#)*
7. *[Query the CouchDB State Database With Pagination](#)*
8. *[Update an Index](#)*
9. *[Delete an Index](#)*

For a deeper dive into CouchDB refer to [CouchDB as the State Database](#) and for more information on the Fabric ledger refer to the [Ledger](#) topic. Follow the tutorial below for details on how to leverage CouchDB in your blockchain network.

Throughout this tutorial, we will use the [Asset transfer ledger queries sample](#) as our use case to demonstrate how to use CouchDB with Fabric, including the execution of JSON queries against the state database. You should have completed the task *[Install Samples, Binaries, and Docker Images](#)*.

7.6.1 Why CouchDB?

Fabric supports two types of peer state databases. LevelDB is the default state database embedded in the peer node. LevelDB stores chaincode data as simple key-value pairs and only supports key, key range, and composite key queries. CouchDB is an optional, alternate state database that allows you to model data on the ledger as JSON and issue rich queries against data values rather than the keys. The CouchDB support also allows you to deploy indexes with your chaincode to make queries more efficient and enable you to query large datasets.

In order to leverage the benefits of CouchDB, namely content-based JSON queries, your data must be modeled in JSON format. You must decide whether to use LevelDB or CouchDB before setting up your network. Switching a peer from using LevelDB to CouchDB is not supported due to data compatibility issues. All peers on the network must use the same database type. If you have a mix of JSON and binary data values, you can still use CouchDB, however the binary values can only be queried based on key, key range, and composite key queries.

7.6.2 Enable CouchDB in Hyperledger Fabric

CouchDB runs as a separate database process alongside the peer. There are additional considerations in terms of setup, management, and operations. A Docker image of [CouchDB](#) is available and we recommend that it be run on the same server as the peer. You will need to setup one CouchDB container per peer and update each peer container by changing the configuration found in `core.yaml` to point to the CouchDB container. The `core.yaml` file must be located in the directory specified by the environment variable `FABRIC_CFG_PATH`:

- For Docker deployments, `core.yaml` is pre-configured and located in the peer container `FABRIC_CFG_PATH` folder. However, when using Docker environments, you can pass environment variables to override the `core.yaml` properties, for example `CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS` to set the CouchDB address.
- For native binary deployments, `core.yaml` is included with the release artifact distribution.

Edit the `stateDatabase` section of `core.yaml`. Specify CouchDB as the `stateDatabase` and fill in the associated `couchDBConfig` properties. For more information, see [CouchDB configuration](#).

7.6.3 Create an index

Why are indexes important?

Indexes allow a database to be queried without having to examine every row with every query, making them run faster and more efficiently. Normally, indexes are built for frequently occurring query criteria allowing the data to be queried more efficiently. To leverage the major benefit of CouchDB – the ability to perform rich queries against JSON data – indexes are not required, but they are strongly recommended for performance. Also, if sorting is required in a query, CouchDB requires an index that includes the sorted fields.

Note: JSON queries that do not have an index may work but will throw a warning in the peer log that the index was not found. However, if a rich query includes a sort specification, then an index on that field is required; otherwise, the query will fail and an error will be thrown.

To demonstrate building an index, we will use the data from the [Asset transfer ledger queries sample](#). In this example, the Asset data structure is defined as:

```
type Asset struct {
    DocType      string `json:"docType"` //docType is used to distinguish the_
    ↪various types of objects in state database
    ID           string `json:"ID"`       //the field tags are needed to keep_
    ↪case from bouncing around
    Color        string `json:"color"`
    Size         int    `json:"size"`
    Owner        string `json:"owner"`
    AppraisedValue int   `json:"appraisedValue"`
}
```

In this structure, the attributes (`docType`, `ID`, `color`, `size`, `owner`, `appraisedValue`) define the ledger data associated with the asset. The attribute `docType` is a pattern that can be used in chaincode to differentiate different data types within the chaincode namespace that may need to be queried separately. When using CouchDB, each chaincode is represented as its own CouchDB database, that is, each chaincode has its own namespace for keys.

With respect to the Asset data structure, `docType` is used to identify that this JSON document represents an asset. Potentially there could be other JSON document types in the chaincode namespace. Any of the JSON fields can be used in CouchDB JSON queries.

When defining an index for use in chaincode queries, each one must be defined in its own text file with the extension `*.json` and the index definition must be formatted in the CouchDB index JSON format.

To define an index, three pieces of information are required:

- *fields*: these are the fields to query
- *name*: name of the index
- *type*: always “json” in this context

For example, a simple index named `foo-index` for a field named `foo`.

```
{
  "index": {
    "fields": ["foo"]
  },
  "name" : "foo-index",
  "type" : "json"
}
```

Optionally the design document attribute `ddoc` can be specified on the index definition. A [design document](#) is a CouchDB construct designed to contain indexes. Indexes can be grouped into design documents for efficiency but CouchDB recommends one index per design document.

Tip: When defining an index it is a good practice to include the `ddoc` attribute and value along with the index name. It is important to include this attribute to ensure that you can update the index later if needed. Also it gives you the ability to explicitly specify which index to use on a query.

Here is another example of an index definition from the Asset transfer ledger queries sample with the index name `indexOwner` using multiple fields `docType` and `owner` and includes the `ddoc` attribute:

```
{
  "index":{
    "fields":["docType","owner"] // Names of the fields to be queried
  },
  "ddoc":"indexOwnerDoc", // (optional) Name of the design document in which the
  ↪index will be created.
  "name":"indexOwner",
  "type":"json"
}
```

In the example above, if the design document `indexOwnerDoc` does not already exist, it is automatically created when the index is deployed. An index can be constructed with one or more attributes specified in the list of fields and any combination of attributes can be specified. An attribute can exist in multiple indexes for the same `docType`. In the following example, `index1` only includes the attribute `owner`, `index2` includes the attributes `owner` and `color`, and `index3` includes the attributes `owner`, `color`, and `size`. Also, notice each index definition has its own `ddoc` value, following the CouchDB recommended practice.

```
{
  "index":{
    "fields":["owner"] // Names of the fields to be queried
  },
  "ddoc":"index1Doc", // (optional) Name of the design document in which the index
  ↪will be created.
  "name":"index1",
  "type":"json"
}
```

(continues on next page)

(continued from previous page)

```

}

{
  "index":{
    "fields":["owner", "color"] // Names of the fields to be queried
  },
  "ddoc":"index2Doc", // (optional) Name of the design document in which the index_
↪will be created.
  "name":"index2",
  "type":"json"
}

{
  "index":{
    "fields":["owner", "color", "size"] // Names of the fields to be queried
  },
  "ddoc":"index3Doc", // (optional) Name of the design document in which the index_
↪will be created.
  "name":"index3",
  "type":"json"
}

```

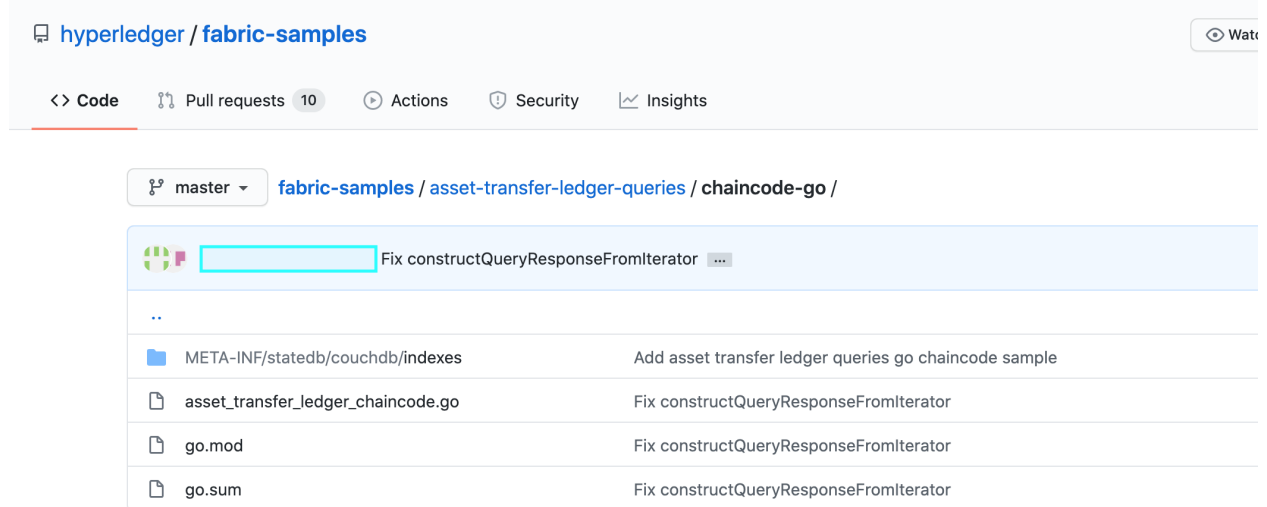
In general, you should model index fields to match the fields that will be used in query filters and sorts. For more details on building an index in JSON format refer to the [CouchDB documentation](#).

A final word on indexing, Fabric takes care of indexing the documents in the database using a pattern called `index warming`. CouchDB does not typically index new or updated documents until the next query. Fabric ensures that indexes stay ‘warm’ by requesting an index update after every block of data is committed. This ensures queries are fast because they do not have to index documents before running the query. This process keeps the index current and refreshed every time new records are added to the state database.

7.6.4 Add the index to your chaincode folder

Once you finalize an index, you need to package it with your chaincode for deployment by placing it in the appropriate metadata folder. You can package and install the chaincode using the [peer lifecycle chaincode](#) commands. The JSON index files must be located under the path `META-INF/statedb/couchdb/indexes` which is located inside the directory where the chaincode resides.

The [Asset transfer ledger queries sample](#) below illustrates how the index is packaged with the chaincode.



hyperledger / fabric-samples

<> Code Pull requests 10 Actions Security Insights

master fabric-samples / asset-transfer-ledger-queries / chaincode-go /

Fix constructQueryResponseFromIterator

..	
META-INF/statedb/couchdb/indexes	Add asset transfer ledger queries go chaincode sample
asset_transfer_ledger_chaincode.go	Fix constructQueryResponseFromIterator
go.mod	Fix constructQueryResponseFromIterator
go.sum	Fix constructQueryResponseFromIterator

This sample includes one index named `indexOwnerDoc`, to support queries by asset owner:

```
{ "index": { "fields": [ "docType", "owner" ] }, "ddoc": "indexOwnerDoc", "name": "indexOwner",
  ↪ "type": "json" }
```

Start the network

Try it yourself

We will bring up the Fabric test network and use it to deploy the asset transfer ledger queries chaincode. Use the following command to navigate to the `test-network` directory in the Fabric samples:

```
cd fabric-samples/test-network
```

For this tutorial, we want to operate from a known initial state. The following command will kill any active or stale Docker containers and remove previously generated artifacts:

```
./network.sh down
```

If you have not run through the tutorial before, you will need to vendor the chaincode dependencies before we can deploy it to the network. Run the following commands:

```
cd ../asset-transfer-ledger-queries/chaincode-go
GO111MODULE=on go mod vendor
cd ../../test-network
```

From the `test-network` directory, deploy the test network with CouchDB with the following command:

```
./network.sh up createChannel -s couchdb
```

This will create two fabric peer nodes that use CouchDB as the state database. It will also create one ordering node and a single channel named `mychannel`.

7.6.5 Deploy the smart contract

You can use the test network script to deploy the asset transfer ledger queries smart contract to the channel. Run the following command to deploy the smart contract to *mychannel*:

```
./network.sh deployCC -ccn ledger -ccp ../asset-transfer-ledger-queries/chaincode-go/
↪-ccl go -ccep "OR('Org1MSP.peer','Org2MSP.peer')"
```

Note that we are using the *-ccep* flag to deploy the smart contract with an endorsement policy of “OR(‘Org1MSP.peer’,‘Org2MSP.peer’)”. This allows either organization to create an asset without receiving an endorsement from the other organization.

Verify index was deployed

Indexes will be deployed to each peer’s CouchDB state database once the chaincode has been installed on the peer and deployed to the channel. You can verify that the CouchDB index was created successfully by examining the peer log in the Docker container.

Try it yourself

To view the logs in the peer Docker container, open a new Terminal window and run the following command to grep for message confirmation that the index was created.

```
docker logs peer0.org1.example.com 2>&1 | grep "CouchDB index"
```

You should see a result that looks like the following:

```
[couchdb] createIndex -> INFO 072 Created CouchDB index [indexOwner] in state_
↪database [mychannel_ledger] using design document [_design/indexOwnerDoc]
```

7.6.6 Query the CouchDB State Database

Now that the index has been defined in the JSON file and deployed alongside the chaincode, chaincode functions can execute JSON queries against the CouchDB state database.

Specifying an index name on a query is optional. If not specified, and an index already exists for the fields being queried, the existing index will be automatically used.

Tip: It is a good practice to explicitly include an index name on a query using the *use_index* keyword. Without it, CouchDB may pick a less optimal index. Also CouchDB may not use an index at all and you may not realize it, at the low volumes during testing. Only upon higher volumes you may realize slow performance because CouchDB is not using an index.

Build the query in chaincode

You can perform JSON queries against the data on the ledger using queries defined within your chaincode. The [Asset transfer ledger queries sample](#) includes two JSON query functions:

- **QueryAssets**

Example of an **ad hoc JSON query**. This is a query where a selector JSON query string can be passed into the function. This query would be useful to client applications that need to dynamically build

their own queries at runtime. For more information on query selectors refer to [CouchDB selector syntax](#).

- **QueryAssetsByOwner**

Example of a **parameterized query** where the query is defined in the chaincode but allows a query parameter to be passed in. In this case the function accepts a single argument, the asset owner. It then queries the state database for JSON documents matching the docType of “asset” and the owner id using the JSON query syntax.

Run the query using the peer command

In absence of a client application, we can use the peer command to test the queries defined in the chaincode. We will use the [peer chaincode query](#) command to use the Assets index `indexOwner` and query for all assets owned by “tom” using the `QueryAssets` function.

Try it yourself

Before querying the database, we should add some data. Run the following command as `Org1` to create a asset owned by “tom”:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n
ledger -c '{"Args":["CreateAsset","asset1","blue","5","tom","35"]}'
```

Next, query for all assets owned by tom:

```
// Rich Query with index name explicitly specified:
peer chaincode query -C mychannel -n ledger -c '{"Args":["QueryAssets", "{\"selector\
\": {\"docType\": \"asset\", \"owner\": \"tom\"}, \"use_index\": {\"_design/indexOwnerDoc\
\", \"indexOwner\"}]}']
```

Delving into the query command above, there are three arguments of interest:

- **QueryAssets**

Name of the function in the Assets chaincode. As you can see in the chaincode function below, `QueryAssets()` calls `getQueryResultForQueryString()`, which then passes the `queryString` to the `getQueryResult()` shim API that executes the JSON query against the state database.

```
func (t *SimpleChaincode) QueryAssets(ctx contractapi.TransactionContextInterface,
queryString string) ([]*Asset, error) {
    return getQueryResultForQueryString(ctx, queryString)
}
```

- `{"selector":{"docType":"asset","owner":"tom"}}`

This is an example of an **ad hoc selector** string which query for all documents of type `asset` where the owner attribute has a value of `tom`.

- `"use_index":["_design/indexOwnerDoc", "indexOwner"]`

Specifies both the design doc name `indexOwnerDoc` and index name `indexOwner`. In this example the selector query explicitly includes the index name, specified by using the `use_index` keyword. Recalling the index definition above *Create an index*, it contains a design doc, `"ddoc": "indexOwnerDoc"`. With CouchDB, if you plan to explicitly include the index name on the query, then the index definition must include the `ddoc` value, so it can be referenced with the `use_index` keyword.

The query runs successfully and the index is leveraged with the following results:

```
[{"docType": "asset", "ID": "asset1", "color": "blue", "size": 5, "owner": "tom",
  ↪ "appraisedValue": 35}]
```

7.6.7 Use best practices for queries and indexes

Queries that use indexes will complete faster, without having to scan the full database in CouchDB. Understanding indexes will allow you to write your queries for better performance and help your application handle larger amounts of data.

It is also important to plan the indexes you install with your chaincode. You should install only a few indexes per chaincode that support most of your queries. Adding too many indexes, or using an excessive number of fields in an index, will degrade the performance of your network. This is because the indexes are updated after each block is committed. The more indexes that need to be updated through “index warming”, the longer it will take for transactions to complete.

The examples in this section will help demonstrate how queries use indexes and what type of queries will have the best performance. Remember the following when writing your queries:

- All fields in the index must also be in the selector or sort sections of your query for the index to be used.
- More complex queries will have a lower performance and will be less likely to use an index.
- You should avoid operators that will result in a full table scan or a full index scan such as `$or`, `$in` and `$regex`.

In the previous section of this tutorial, you issued the following query against the assets chaincode:

```
// Example one: query fully supported by the index
export CHANNEL_NAME=mychannel
peer chaincode query -C $CHANNEL_NAME -n ledger -c '{"Args":["QueryAssets", "{\
  ↪ "selector": {"docType": "asset", "owner": "tom"}, "use_index": {\
  ↪ "indexOwnerDoc", "indexOwner"}"}']'
```

The asset transfer ledger queries chaincode was installed with the `indexOwnerDoc` index:

```
{"index": {"fields": ["docType", "owner"], "ddoc": "indexOwnerDoc", "name": "indexOwner",
  ↪ "type": "json"}
```

Notice that both the fields in the query, `docType` and `owner`, are included in the index, making it a fully supported query. As a result this query will be able to use the data in the index, without having to search the full database. Fully supported queries such as this one will return faster than other queries from your chaincode.

If you add extra fields to the query above, it will still use the index. However, the query will additionally have to scan the database for the extra fields, resulting in a longer response time. As an example, the query below will still use the index, but will take a longer time to return than the previous example.

```
// Example two: query fully supported by the index with additional data
peer chaincode query -C $CHANNEL_NAME -n ledger -c '{"Args":["QueryAssets", "{\
  ↪ "selector": {"docType": "asset", "owner": "tom", "color": "blue"}, "use_
  ↪ index": [{"indexOwnerDoc", "indexOwner"}]}']'
```

(continues on next page)

(continued from previous page)

A query that does not include all fields in the index will have to scan the full database instead. For example, the query below searches for the owner, without specifying the type of item owned. Since the `indexOwnerDoc` contains both the `owner` and `docType` fields, this query will not be able to use the index.

```
// Example three: query not supported by the index
peer chaincode query -C $CHANNEL_NAME -n ledger -c '{"Args":["QueryAssets", "{\
↪ "selector\":"{\\"owner\":"\\"tom\\""}, \\"use_index\":"{\\"indexOwnerDoc\\", \\"indexOwner\\"}"}\
↪ "]}'
```

In general, more complex queries will have a longer response time, and have a lower chance of being supported by an index. Operators such as `$or`, `$in`, and `$regex` will often cause the query to scan the full index or not use the index at all.

As an example, the query below contains an `$or` term that will search for every asset and every item owned by tom.

```
// Example four: query with $or supported by the index
peer chaincode query -C $CHANNEL_NAME -n ledger -c '{"Args":["QueryAssets", "{\
↪ "selector\":"{\\"$or\":"[{\\"docType\":"\\"asset\\""},{\\"owner\":"\\"tom\\""}]}, \\"use_index\
↪ ":"{\\"indexOwnerDoc\\", \\"indexOwner\\"}"}"]}'
```

This query will still use the index because it searches for fields that are included in `indexOwnerDoc`. However, the `$or` condition in the query requires a scan of all the items in the index, resulting in a longer response time.

Below is an example of a complex query that is not supported by the index.

```
// Example five: Query with $or not supported by the index
peer chaincode query -C $CHANNEL_NAME -n ledger -c '{"Args":["QueryAssets", "{\
↪ "selector\":"{\\"$or\":"[{\\"docType\":"\\"asset\\",\\"owner\":"\\"tom\\""},{\\"color\":"\\"yellow\
↪ "]}", \\"use_index\":"{\\"indexOwnerDoc\\", \\"indexOwner\\"}"}"]}'
```

The query searches for all assets owned by tom or any other items that are yellow. This query will not use the index because it will need to search the entire table to meet the `$or` condition. Depending the amount of data on your ledger, this query will take a long time to respond or may timeout.

While it is important to follow best practices with your queries, using indexes is not a solution for collecting large amounts of data. The blockchain data structure is optimized to validate and confirm transactions and is not suited for data analytics or reporting. If you want to build a dashboard as part of your application or analyze the data from your network, the best practice is to query an off chain database that replicates the data from your peers. This will allow you to understand the data on the blockchain without degrading the performance of your network or disrupting transactions.

You can use block or chaincode events from your application to write transaction data to an off-chain database or analytics engine. For each block received, the block listener application would iterate through the block transactions and build a data store using the key/value writes from each valid transaction's `rwset`. The [Peer channel-based event services](#) provide replayable events to ensure the integrity of downstream data stores. For an example of how you can use an event listener to write data to an external database, visit the [Off chain data sample](#) in the Fabric Samples.

7.6.8 Query the CouchDB State Database With Pagination

When large result sets are returned by CouchDB queries, a set of APIs is available which can be called by chaincode to paginate the list of results. Pagination provides a mechanism to partition the result set by specifying a `pagesize` and a start point – a `bookmark` which indicates where to begin the result set. The client application iteratively invokes the chaincode that executes the query until no more results are returned. For more information refer to this [topic on pagination with CouchDB](#).

We will use the [Asset transfer ledger queries sample](#) function `QueryAssetsWithPagination` to demonstrate how pagination can be implemented in chaincode and the client application.

- **QueryAssetsWithPagination –**

Example of an **ad hoc JSON query with pagination**. This is a query where a selector string can be passed into the function similar to the above example. In this case, a `pageSize` is also included with the query as well as a bookmark.

In order to demonstrate pagination, more data is required. This example assumes that you have already added asset1 from above. Run the following commands in the peer container to create four more assets owned by “tom”, to create a total of five assets owned by “tom”:

Try it yourself

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n
ledger -c '{"Args":["CreateAsset","asset2","yellow","5","tom","35"]}'
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n
ledger -c '{"Args":["CreateAsset","asset3","green","6","tom","20"]}'
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n
ledger -c '{"Args":["CreateAsset","asset4","purple","7","tom","20"]}'
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n
ledger -c '{"Args":["CreateAsset","asset5","blue","8","tom","40"]}'
```

In addition to the arguments for the query in the previous example, `QueryAssetsWithPagination` adds `pagesize` and `bookmark`. `PageSize` specifies the number of records to return per query. The `bookmark` is an “anchor” telling couchDB where to begin the page. (Each page of results returns a unique bookmark.)

- **QueryAssetsWithPagination**

As you can see in the chaincode function below, `QueryAssetsWithPagination()` calls `getQueryResultForQueryStringWithPagination()`, which then passes the queryString as well as the bookmark and `pagesize` to the `GetQueryResultWithPagination()` shim API that executes the paginated JSON query against the state database.

```
func (t *SimpleChaincode) QueryAssetsWithPagination(
    ctx contractapi.TransactionContextInterface,
    queryString,
    pageSize int,
    bookmark string) (*PaginatedQueryResult, error) {

    return getQueryResultForQueryStringWithPagination(ctx, queryString,
    int32(pageSize), bookmark)
}
```

The following example is a peer command which calls `QueryAssetsWithPagination` with a `pageSize` of 3 and no bookmark specified.

Tip: When no bookmark is specified, the query starts with the “first” page of records.

Try it yourself

```
// Rich Query with index name explicitly specified and a page size of 3:
peer chaincode query -C mychannel -n ledger -c '{"Args":["QueryAssetsWithPagination",
↪{"selector":{"docType":"asset","owner":"tom"}, "use_index":["_design/
↪indexOwnerDoc", "indexOwner"]},"3",""]}'
```

The following response is received (carriage returns added for clarity), three of the five assets are returned because the `pageSize` was set to 3:

```
{
  "records": [
    {"docType": "asset", "ID": "asset1", "color": "blue", "size": 5, "owner": "tom",
↪ "appraisedValue": 35},
    {"docType": "asset", "ID": "asset2", "color": "yellow", "size": 5, "owner": "tom",
↪ "appraisedValue": 35},
    {"docType": "asset", "ID": "asset3", "color": "green", "size": 6, "owner": "tom",
↪ "appraisedValue": 20}],
  "fetchRecordsCount": 3,
  "bookmark":
↪ "g1AAAABJeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqzJRYXp5YYg2Q5YLI5IPUgSVawJIjFXJKfm5UFANozE8s
↪ "
}
```

Note: Bookmarks are uniquely generated by CouchDB for each query and represent a placeholder in the result set. Pass the returned bookmark on the subsequent iteration of the query to retrieve the next set of results.

The following is a peer command to call `QueryAssetsWithPagination` with a `pageSize` of 3. Notice this time, the query includes the bookmark returned from the previous query.

Try it yourself

```
peer chaincode query -C $CHANNEL_NAME -n ledger -c '{"Args": [
↪ "QueryAssetsWithPagination", {"selector":{"docType":"asset","owner":"tom\
↪ "}}, {"use_index":["_design/indexOwnerDoc", "indexOwner"]},"3",
↪ "g1AAAABJeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqzJRYXp5YYg2Q5YLI5IPUgSVawJIjFXJKfm5UFANozE8s
↪ ""]}'
```

The following response is received (carriage returns added for clarity). The last two records are retrieved:

```
{
  "records": [
    {"docType": "asset", "ID": "asset4", "color": "purple", "size": 7, "owner": "tom",
↪ "appraisedValue": 20},
    {"docType": "asset", "ID": "asset5", "color": "blue", "size": 8, "owner": "tom",
↪ "appraisedValue": 40}],
  "fetchRecordsCount": 2,
  "bookmark":
↪ "g1AAAABJeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqzJRYXp5aYgmQ5YLI5IPUgSVawJIjFXJKfm5UFANqBE80
↪ "
```

(continues on next page)

(continued from previous page)

}

The returned bookmark marks the end of the result set. If we attempt to query with this bookmark, no more results will get returned.

Try it yourself

```
peer chaincode query -C $CHANNEL_NAME -n ledger -c '{"Args":[
  ↪ "QueryAssetsWithPagination", {"selector\":"docType\":"asset\","owner\":"tom\
  ↪ "}, {"use_index\":["_design/indexOwnerDoc\","indexOwner\"]},"3",
  ↪ "g1AAAABJeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqzJRYXp5aYgmQ5YLI5IPUgSVawJIjFXJKfm5UFANqBE80
  ↪ "}]}'
```

For an example of how a client application can iterate over the result sets using pagination, search for the `getQueryResultForQueryStringWithPagination` function in the [Asset transfer ledger queries sample](#).

7.6.9 Update an Index

It may be necessary to update an index over time. The same index may exist in subsequent versions of the chaincode that gets installed. In order for an index to be updated, the original index definition must have included the design document `ddoc` attribute and an index name. To update an index definition, use the same index name but alter the index definition. Simply edit the index JSON file and add or remove fields from the index. Fabric only supports the index type JSON. Changing the index type is not supported. The updated index definition gets redeployed to the peer's state database when the chaincode definition is committed to the channel. Changes to the index name or `ddoc` attributes will result in a new index being created and the original index remains unchanged in CouchDB until it is removed.

Note: If the state database has a significant volume of data, it will take some time for the index to be re-built, during which time chaincode invokes that issue queries may fail or timeout.

Iterating on your index definition

If you have access to your peer's CouchDB state database in a development environment, you can iteratively test various indexes in support of your chaincode queries. Any changes to chaincode though would require redeployment. Use the [CouchDB Fauxton interface](#) or a command line curl utility to create and update indexes.

Note: The Fauxton interface is a web UI for the creation, update, and deployment of indexes to CouchDB. If you want to try out this interface, there is an example of the format of the Fauxton version of the index in [Assets sample](#). If you have deployed the test network with CouchDB, the Fauxton interface can be loaded by opening a browser and navigating to `http://localhost:5984/_utils`.

Alternatively, if you prefer not use the Fauxton UI, the following is an example of a curl command which can be used to create the index on the database `mychannel_ledger`:

```
// Index for docType, owner.
// Example curl command line to define index in the CouchDB channel_chaincode database
curl -i -X POST -H "Content-Type: application/json" -d
  '{"index\":{"fields\":["docType\","owner\"]},
  \ "name\":"indexOwner\",'
```

(continues on next page)

(continued from previous page)

```
\ddoc\":"indexOwnerDoc\",
"type\":"json\"}" http://hostname:port/mychannel_ledger/_index
```

Note: If you are using the test network configured with CouchDB, replace `hostname:port` with `localhost:5984`.

7.6.10 Delete an Index

Index deletion is not managed by Fabric tooling. If you need to delete an index, manually issue a curl command against the database or delete it using the Fauxton interface.

The format of the curl command to delete an index would be:

```
curl -X DELETE http://localhost:5984/{database_name}/_index/{design_doc}/json/{index_
↪name} -H "accept: */*" -H "Host: localhost:5984"
```

To delete the index used in this tutorial, the curl command would be:

```
curl -X DELETE http://localhost:5984/mychannel_ledger/_index/indexOwnerDoc/json/
↪indexOwner -H "accept: */*" -H "Host: localhost:5984"
```

7.6.11 Clean up

When you are finished using the tutorial, you can bring down the test network using `network.sh` script.

```
./network.sh down
```

This command will bring down the CAs, peers, and ordering node of the network that we created. Note that all of the data on the ledger will be lost. If you want to go through the tutorial again, you will start from a clean initial state.

7.7 Creating a channel

In order to create and transfer assets on a Hyperledger Fabric network, an organization needs to join a channel. Channels are a private layer of communication between specific organizations and are invisible to other members of the network. Each channel consists of a separate ledger that can only be read and written to by channel members, who are allowed to join their peers to the channel and receive new blocks of transactions from the ordering service. While the peers, ordering nodes, and Certificate Authorities form the physical infrastructure of the network, channels are the process by which organizations connect with each other and interact.

Because of the fundamental role that channels play in the operation and governance of Fabric, we provide a series of tutorials that cover different aspects of how channels are created. Fabric v2.3 introduces the capability to create a channel without requiring a system channel, removing an extra layer of administration from the process. The **Create a channel** tutorial introduces the new flow. If you don't yet have a network and prefer to use the test network, see **Create a channel using the test network**. The legacy process for creating a channel based on a system channel continues to be supported and is described in the Fabric v2.2 [Create a channel tutorial](#). Each tutorial describes the operational steps that need to be taken by a network administrator to create a channel. For a deeper dive, the [Using configtx.yaml to build a channel configuration](#) tutorial introduces the conceptual aspects of creating a channel, followed by a separate discussion of [Channel policies](#).

7.7.1 Create a channel

To simplify the channel creation process and enhance the privacy and scalability of channels, it is now possible to create application channels (where transactions involving assets happen) without first creating a “system channel” managed by the ordering service. Use this tutorial to learn how to create new channels without a system channel by using the `configtxgen` tool to create a genesis block and the `osnadmin CLI` (which runs against a REST API exposed by each ordering service node) to join ordering nodes to a channel. This process allows ordering nodes to join (or leave) any number of channels as needed, similar to how peers can participate in multiple channels.

How this process differs from the legacy Fabric v2.2 process:

- **System channel no longer required:** Besides the creation of the system channel representing an extra step (as compared to the new process), this system channel created an extra layer of administration that, for some use cases, provided no tangible benefit.
- **Consortium no longer required:** You no longer need to define the set of organizations, known as the “consortium”, who are permitted to create channels on a particular ordering service. With this new process, all channels are application channels, so the concept of a list of organizations who can create channels no longer applies. Any set of organizations can get together and create a channel using a defined set of ordering nodes (which become the ordering service of that channel).
- **Simplified ordering node creation process:** Because a system channel genesis block no longer needs to exist before an ordering node is created, admins can now focus on the process of setting up their infrastructure and making sure their node is working correctly before joining a particular application channel.

Benefits of the new process:

- **Increased privacy:** Because all ordering nodes used to be joined to the system channel, every ordering node in a network knew about the existence of every channel on that ordering service, even if the node itself wasn’t a consenter on that channel and therefore wasn’t ordering or storing its blocks. Now, an ordering node only knows about the channels it is joined to.
- **Scalability:** When there is a large number of ordering nodes on the system channel, starting the node can take a long time as it has to wait for each orderer to replicate the system channel ledger showing it has joined. Additionally, large numbers of nodes on the system channel increases the number of heartbeat messages being sent between nodes. In large system channels with a large number of nodes, this can cause processing issues, even if no application channels have a large number of ordering nodes on it.
- **Flexibility:** In the past, the scope of the ordering service was defined by the membership of the node in a system channel, and in some of the application channels created from that particular system channel. Now, orderers can join or leave channels as needed, similar to how peers can join any channel its organization is a member of.
- **Operational benefits:**
 - Easy to list the channels that the ordering node is a consenter on.
 - Simple process to remove a channel and the blocks associated with that channel.
 - Orderer nodes can now track channels as a follower before being added to a channel’s consenter set, allowing them to detect this addition more quickly. Previously, this process could take several minutes, causing ordering node admins to restart their nodes so that the node could detect that it was joined more quickly.
 - If the MSP of a peer organization, as listed in the system channel, needs to be changed, the peer organization no longer needs to coordinate with an admin of the system channel to change the MSP.

Note: The new mode of creating channels is incompatible with creating a channel using a system channel. If the system channel exists, the `channel join` operation is not supported. Similarly, `channel join` and `channel remove` cannot be used with a Solo or Kafka ordering service. “Mixed mode” management, where the system channel is to create channels on some ordering nodes and the new process is used to create channels on other ordering nodes is not supported and highly discouraged. You must either transition to the new process or continue to use the system

channel process. For information about removing the system channel from an existing ordering service as part of transitioning to the new process, check out [Remove the system channel](#).

While creating the channel, this tutorial will take you through the following steps and concepts:

- *Prerequisites*
- *Step one: Generate the genesis block of the channel*
- *Step two: Use the `osnadmin` CLI to add the first orderer to the channel*
- *Step three: Join additional ordering nodes*
- *Next steps*

Note: If you prefer to learn how to create a channel with the test network instead, check out the [Create a channel using the test network](#) tutorial.

Folder structure

This tutorial uses the following folder structure for the generated orderer organization MSP and orderer certificates, and while it is not mandatory, it is useful when referring to the certificates referenced by the commands.



There are three sections in the folder structure above to consider:

- **Orderer organization MSP:** the `organizations/ordererOrganizations/ordererOrg1.example.com/msp` folder contains the orderer organization MSP that includes the `cacerts` and `tlscacerts` folders that you need to create and then copy in the root certificates (`ca-cert.pem`) for the organization CA and TLS CA respectively. If you are using an intermediate CA, you also need to include the corresponding `intermediatecerts` and `tlsintermediatecerts` folders.
- **Orderer local MSP:** the `organizations/ordererOrganizations/ordererOrg1.example.com/ordering-service-nodes/osn1.ordererOrg1.example.com/msp` folder, also known as the orderer local MSP, contains the enrollment certificate and private key for the ordering service `osn1` node. This folder is automatically generated when you enroll the orderer identity with a Fabric CA for the organization.
- **TLS certificates:** the `organizations/ordererOrganizations/ordererOrg1.example.com/ordering-service-nodes/osn1.ordererOrg1.example.com/tls` folder contains the TLS certificate and private key for the ordering service `osn1` node as well as the TLS CA root cert `tls-ca-cert.pem`.
- **Admin client certificates** - `admin-client/` folder contains the TLS certificates for the admin client that will be issuing the `osadmin` commands. The connection between the admin client that calls the `osnadmin` CLI and an orderer requires mutual TLS, although it is not required for the network itself. This means the admin client needs to enroll and register with a TLS CA to generate the TLS certificate (`client-tls-cert.pem`) and private key (`client-tls-key.pem`) that is provided to the `osnadmin` CLI. You will need to copy those certificates and the client TLS CA root cert (`client-tls-ca-cert.pem`) into this folder or point to where they exist on your file system. For simplicity of the tutorial, you could use the same TLS CA that the orderer organization uses. On a production network, they would likely be separate TLS CAs.

Certificate names used in this example are for illustration purposes and may not reflect the actual names of the certificates generated by the CA. When you generate the certificates you can rename them accordingly to make it easier to differentiate them.

Important: You need to create the `config.yaml` file and add it to the organization MSP and local MSP folder for each ordering node. This file enables Node OU support for the MSP, an important feature that allows the MSP's admin to be identified based on an "admin" OU in an identity's certificate. Learn more in the [Fabric CA](#) documentation.

If you are using a containerized solution for running your network (which for obvious reasons is a popular choice), **it is a best practice to mount volumes for the certificate directories external to the container where the node itself is running. This will allow the certificates to be used by an ordering node container, regardless whether the ordering node container goes down, becomes corrupted, or is restarted.**

Prerequisites

Because `osnadmin` commands are performed against an ordering node (which executes the actions), at least one orderer needs to exist so that you can join the orderer to the channel. You can attempt this tutorial with your existing ordering service or deploy a new set of ordering nodes. If you decide to use `osnadmin` commands against orderers in an existing ordering service, the system channel must first be removed from each ordering node before you can create any new channels. Choose whether you want to use your existing ordering service or deploy a new set of orderers:

- *Use existing ordering service*
- *Deploy a new set of orderers*

Use existing ordering service

Before you can take advantage of this feature on a deployed ordering service, you need to remove the system channel from each ordering node that is a consenter in your application channels. A "mixed mode" of orderers on a channel, where some nodes are part of a system channel and others are not, is not supported. The `osadmin` CLI includes both

a `channel list` and a `channel remove` command to facilitate the process of removing the system channel. If you prefer to deploy a new ordering service instead, skip ahead to *Deploy a new ordering service*.

Remove the system channel

- Before attempting these steps ensure that you have [upgraded](#) your ordering nodes to Fabric v2.3 or higher.
- Modify the `orderer.yaml` for each ordering node to support this feature and restart the node. See the [orderer sampleconfig](#) for more information about these required parameters.
 - `General.BootstrapMethod` - Set this value to `none`. Because the system channel is no longer required, the `orderer.yaml` file on each orderer needs to be configured with `BootstrapMethod: none` which means that no bootstrap block is required or used to start up the orderer.
 - `Admin.ListenAddress` - The orderer admin server address (host and port) that can be used by the `osnadmin` command to configure channels on the ordering service. This value should be a unique `host:port` combination to avoid conflicts.
 - `Admin.TLS.Enabled` - Technically this can be set to `false`, but this is not recommended. In general, you should always set this value to `true`.
 - `Admin.TLS.PrivateKey` - The path to and file name of the orderer private key issued by the TLS CA.
 - `Admin.TLS.Certificate` - The path to and file name of the orderer signed certificate issued by the TLS CA.
 - `Admin.TLS.ClientAuthRequired` - This value must be set to `true`. Note that while mutual TLS is required for all operations on the orderer Admin endpoint, the entire network is not required to use Mutual TLS.
 - `Admin.TLS.ClientRootCAs` - The path to and file name of the admin client TLS CA Root certificate. In the folder structure above, this is `admin-client/client-tls-ca-cert.pem`.
 - `ChannelParticipation.Enabled` - Set this value to `true` to enable this feature on the orderer.
- Restart each ordering node.
- Put the [system channel into maintenance mode](#) using the same process for the Kafka to Raft migration. This action stops new channel creation transactions from coming in.
- Remove the system channel from the set of orderers, one by one. During the system channel removal from an orderer, the orderer stops servicing all other application channels it is a member of until removal is complete. Therefore, care must be taken to ensure that the remaining orderers can still function and reach consensus. This operation can be staggered, such that depending on the fault tolerance setup of the respective channels, no channel down time is experienced. If an application channel can tolerate one server offline, you should still be able to submit transactions to the channel, via the other orderers that are not undergoing the removal at that time. Use the `osnadmin channel list` command to view the channels that this orderer is a member of:

```
export OSN_TLS_CA_ROOT_CERT=../config/organizations/ordererOrganizations/
↪ordererOrg1.example.com/ordering-service-nodes/osn1.ordererOrg1.example.com/tls/
↪cacerts/tls-ca-cert.pem
export ADMIN_TLS_SIGN_CERT=../config/admin-client/client-tls-cert.pem
export ADMIN_TLS_PRIVATE_KEY=../config/admin-client/client-tls-key.pem

osnadmin channel list -o [ORDERER_ADMIN_LISTENADDRESS] --ca-file $OSN_TLS_CA_ROOT_
↪CERT --client-cert $ADMIN_TLS_SIGN_CERT --client-key $ADMIN_TLS_PRIVATE_KEY
```

Where:

- ORDERER_ADMIN_LISTENADDRESS corresponds to the `Orderer.Admin.ListenAddress` defined in the `orderer.yaml` for this orderer.
- OSN_TLS_CA_ROOT_CERT with the path and file name of the orderer organization TLS CA root certificate and intermediate certificate if using an intermediate TLS CA.
- ADMIN_TLS_SIGN_CERT with the path and file name of the admin client signed certificate from the TLS CA.
- ADMIN_TLS_PRIVATE_KEY with the path and file name of the admin client private key from the TLS CA.

Note: Because the connection between the `osnadmin` CLI and the orderer requires mutual TLS, you need to pass the `--client-cert` and `--client-key` parameters on each `osadmin` command. The `--client-cert` parameter points to the admin client certificate and `--client-key` refers to the admin client private key, both issued by the admin client TLS CA.

For example:

```
osnadmin channel list -o HOST1:7081 --ca-file $TLS_CA_ROOT_CERT --client-cert
↪$OSN_TLS_SIGN_CERT --client-key $OSN_TLS_PRIVATE_KEY
```

The output of this command looks similar to:

```
{
  "systemChannel": {
    "name": "syschannel",
    "url": "/participation/v1/channels/systemchannel"
  },
  "channels": [
    {
      "name": "channel1",
      "url": "/participation/v1/channels/channel1"
    }
  ]
}
```

Now you can run `osnadmin channel remove` to remove the system channel from the node configuration:

```
osnadmin channel remove -o [ORDERER_ADMIN_LISTENADDRESS] --channelID syschannel --
↪ca-file $OSN_TLS_CA_ROOT_CERT --client-cert $ADMIN_TLS_SIGN_CERT --client-key
↪$ADMIN_TLS_PRIVATE_KEY
```

For example:

```
osnadmin channel remove -o HOST1:7081 --channelID syschannel --ca-file $OSN_TLS_
↪CA_ROOT_CERT --client-cert $ADMIN_TLS_SIGN_CERT --client-key $ADMIN_TLS_PRIVATE_
↪KEY
```

When successful you see:

```
Status: 204
```

And when you rerun the `osnadmin channel list` command, you can verify that the system channel was removed:

```
osnadmin channel list -o [ORDERER_ADMIN_LISTENADDRESS] --ca-file $OSN_TLS_CA_ROOT_
↪CERT --client-cert $ADMIN_TLS_SIGN_CERT --client-key $ADMIN_TLS_PRIVATE_KEY
```

Examine the output of the command to verify that the system channel was removed:

```
{
  "systemChannel": null,
  "channels": [
    {
      "name": "channel1",
      "url": "/participation/v1/channels/channel1"
    }
  ]
}
```

Repeat these commands for each ordering node.

Deploy a new set of orderers

Use these steps if you prefer to deploy a new set of orderers to try out this feature. Deploying the orderers is a two-step process:

- *Create the ordering organization MSP and generate ordering node certificates*
- *Configure the `orderer.yaml` file for each orderer*

Create the ordering organization MSP and generate ordering node certificates

Before you can deploy an orderer, you need to define the ordering organization MSP definition and generate the TLS and enrollment certificates for each Raft ordering node. To learn how to use a CA to create these identities, check out [Registering and enrolling identities with a CA](#). After completing that process, you should have the enrollment and TLS certificates for each node as well as the orderer organization MSP definition. To keep track of the generated certificates and MSP you can use the *folder structure* defined in this topic, although it is not mandatory.

Because this tutorial demonstrates the process for creating a channel with **three orderers** deployed for a single organization, you need to generate enrollment and TLS certificates for each node. Why three orderers? This configuration allows for a majority quorum on the Raft cluster. Namely, when there are three orderers, one at a time can go down for maintenance, while a majority (two of three) is maintained. For more information about the number of nodes you should deploy in production, check out [The Ordering Service](#). For simplicity and learning purposes, you have the ability to deploy a single node ordering service, even though such an ordering service will not be highly available and cannot therefore be considered a “production” deployment.

Configure the `orderer.yaml` file for each orderer

Follow the instructions in the [ordering service deployment guide](#) to build an ordering service with three ordering nodes. Note that when you configure the `orderer.yaml` file for each orderer, you will need to make modifications to the `ChannelParticipation` and `General.BootstrapMethod` parameters to leverage this feature.

- `General.BootstrapMethod` - Set this value to `none`. Because the system channel is no longer required, the `orderer.yaml` file on each orderer needs to be configured with `BootstrapMethod: none` which means that no bootstrap block is required or used to start up the orderer.
- `Admin.ListenAddress` - The orderer admin server address (host and port) that can be used by the `osnadmin` command to configure channels on the ordering service. This value should be a unique `host:port` combination to avoid conflicts.
- `Admin.TLS.Enabled` - Technically this can be set to `false`, but this is not recommended. In general, you should always set this value to `true`.
- `Admin.TLS.PrivateKey` - The path to and file name of the orderer private key issued by the TLS CA.

- `Admin.TLS.Certificate`: - The path to and file name of the orderer signed certificate issued by the TLS CA.
- `Admin.TLS.ClientAuthRequired`: This value must be set to `true`. Note that while mutual TLS is required for all operations on the orderer Admin endpoint, the entire network is not required to use Mutual TLS.
- `Admin.TLS.ClientRootCAs`: - The path to and file name of the admin client TLS CA Root certificate. In the folder structure above, this is `admin-client/client-tls-ca-cert.pem`.
- `ChannelParticipation.Enabled`: - Set this value to `true` to enable this feature on the orderer.

Start each orderer

1. If you have not already, set the path to the location of the Fabric binaries on your system:

```
export PATH=<path to download location>/bin:$PATH
```

2. In your terminal window set the `FABRIC_CFG_PATH` to point to the location of the `orderer.yaml` file, relative to where you are running the Fabric commands from. For example, if you download the binaries, and run the commands from the `/bin` directory and the `orderer.yaml` is under `/config`, the path would be:

```
export FABRIC_CFG_PATH=../config
```

3. You can now start the orderer by running the following command on each ordering node:

```
orderer start
```

When the orderer starts successfully, you should see something similar to the following output:

```
INFO 01d Registrar initializing without a system channel, number of application_  
↪channels: 0, with 0 consensus.Chain(s) and 0 follower.Chain(s)  
INFO 01f Starting orderer:
```

This action starts the ordering nodes without any channels. Repeat these steps for each orderer. At this point, the three nodes are not communicating with each other until we create a channel in the subsequent steps.

While the ordering node is started, there are no channels on the ordering service yet, we create a channel in the subsequent steps.

Define your peer organizations

Because the channel you are creating is meant to be used by two or more peer organizations to transact privately on the network, you need to have at least one peer organization defined to act as the channel administrator who can add other organizations. Technically, the peer nodes themselves do not yet have to be deployed, but you do need to create one or more peer organization MSP definitions and at least one peer organization needs to be provided in the `configtx.yaml` in the next step. Before proceeding to the next section, follow the steps in the [Fabric CA](#) documentation to build your peer organization MSP definition. If the peers have been deployed, you should also include their address in the `AnchorPeers` section.

Step one: Generate the genesis block of the channel

The process of creating a new channel begins by generating the genesis block for the channel that you will later submit to your orderer in the `channel join` request. Only one member needs to create the genesis block and it can be shared out of band with the other members on the channel who can inspect it to ensure they agree to the channel configuration and then used by each orderer in the ordering service.

Set up the configtxgen tool

While it is possible to build the channel creation transaction file manually, it is easier to use the `configtxgen` tool to build the block by reading a `configtx.yaml` file that defines the configuration of your channel.

The `configtxgen` tool is located in the `bin` folder of downloaded Fabric binaries.

Before using `configtxgen`, confirm you have to set the `FABRIC_CFG_PATH` environment variable to the path of the directory that contains your local copy of the `configtx.yaml` file, for example:

```
export FABRIC_CFG_PATH=../config
```

You can check that you are able to use the tool by printing the `configtxgen` help text:

```
configtxgen --help
```

The configtx.yaml file

The `configtx.yaml` file specifies the **channel configuration** of new channels in a format that is both readable and editable and that can be used by `configtxgen`. The `configtxgen` tool uses the channel profiles that are defined in `configtx.yaml` to create the channel configuration and write it to the `protobuf format` that can be consumed by Fabric.

A sample `configtx.yaml` file is located in the `config` folder of the binary alongside images that you downloaded. The file contains the following configuration sections that we need to create our new channel:

- **Organizations:** The organizations that can become members of your channel. Each organization has a reference to the cryptographic material that is used to build the `channel MSP`. Exactly which members are part of a channel configuration is defined in the **Profiles** section below.
- **Orderer:** Contains the list of ordering nodes that will form the consenter set of the channel.
- **Profiles:** Each channel profile references information from other sections of the `configtx.yaml` file to build a channel configuration. Every channel that is created must reference a profile, which contains information like the peer organizations and the channel's set of consenter. An unlimited number of profiles can be listed in the `Profiles` section. However, each must have a distinctive name (this will not become the name of the channel itself, as that is specified by a flag given when the channel is created).

Refer to the [Using configtx.yaml to build a channel configuration](#) tutorial to learn more about this file. However, the following three sections require specific configuration in order to create a channel without a system channel.

Note: Peer organizations are not required when you initially create the channel, but if you know them it is recommended to add them now to the `Profiles Organizations:` and `Applications` section to avoid having to update the channel configuration later.

Organizations:

Provide the orderer organization MSP and any peer organization MSPs that are known. Also provide the endpoint addresses of each ordering node.

- **Organizations.OrdererEndpoints:**

For example:

```
OrdererEndpoints:
  - "Host1:7080"
  - "Host2:7080"
  - "Host3:7080"
```

Orderer:

- **Orderer.OrdererType** Set this value to `etcdraft`. As mentioned before, this process does not work with Solo or Kafka ordering nodes.
- **Orderer.EtcdRaft.Consenters** Provide the list of ordering node addresses, in the form of `host:port`, that are considered active members of the consenter set. All orderers that are listed in this section will become active “consenters” on the channel when they join the channel.

For example:

```
EtcdRaft:
  Consenters:
    - Host: Host1
      Port: 7090
      ClientTLSCert: ../config/organizations/ordererOrganizations/ordererOrg1.
↪example.com/ordering-service-nodes/osn1.ordererOrg1.example.com/tls/signcerts/cert.
↪pem
      ServerTLSCert: ../config/organizations/ordererOrganizations/ordererOrg1.
↪example.com/ordering-service-nodes/osn1.ordererOrg1.example.com/tls/signcerts/cert.
↪pem
    - Host: Host2
      Port: 7091
      ClientTLSCert: ../config/organizations/ordererOrganizations/ordererOrg1.
↪example.com/ordering-service-nodes/osn2.ordererOrg1.example.com/tls/signcerts/cert.
↪pem
      ServerTLSCert: ../config/organizations/ordererOrganizations/ordererOrg1.
↪example.com/ordering-service-nodes/osn2.ordererOrg1.example.com/tls/signcerts/cert.
↪pem
    - Host: Host3
      Port: 7092
      ClientTLSCert: ../config/organizations/ordererOrganizations/ordererOrg1.
↪example.com/ordering-service-nodes/osn3.ordererOrg1.example.com/tls/signcerts/cert.
↪pem
      ServerTLSCert: ../config/organizations/ordererOrganizations/ordererOrg1.
↪example.com/ordering-service-nodes/osn3.ordererOrg1.example.com/tls/signcerts/cert.
↪pem
```

For simplicity, each orderer in this example is using the same TLS certificate for both the server and client, although this is not required.

When the channel configuration block is created, `configtxgen` reads the paths to the TLS certificates and replaces the paths with the corresponding bytes of the certificates.

Profiles:

If you are familiar with the legacy process for creating channels, you will recall that the `Profiles:` section of the `configtx.yaml` contained a consortium section under the `Orderer:` group. This consortium definition, which previously specified the peer organizations allowed to create channels on the ordering service, is no longer required. If this section exists in the profile, you cannot use this feature to create a channel. Rather, under the `Orderer:` section

you simply include MSP ID of the ordering organization or organizations in the case of a multi-organization ordering service and list the peer organizations in `Application:` section that will be members of the channel. At least one orderer organization and one peer organization must be provided.

The following snippet is an example of a channel profile that contains an orderer configuration based on the default channel, orderer, organization, and policy configurations. The `Application:` section, where the peer organizations are listed, includes the default `Application` settings as well as at least one peer organization (`SampleOrg`, in this example) and the corresponding policies for the channel.

```
Profiles:
  SampleAppChannelEtcdRaft:
    <<: *ChannelDefaults
    Orderer:
      <<: *OrdererDefaults
      OrdererType: etcdraft
      Organizations:
        - <<: *SampleOrg
          Policies:
            <<: *SampleOrgPolicies
            Admins:
              Type: Signature
              Rule: "OR('SampleOrg.member') "
      Application:
        <<: *ApplicationDefaults
        Organizations:
          - <<: *SampleOrg
            Policies:
              <<: *SampleOrgPolicies
              Admins:
                Type: Signature
                Rule: "OR('SampleOrg.member') "
```

Note: For simplicity, this snippet assumes that the peers and orderers belong to the same organization `SampleOrg`. You can refer to the [sample config](#) for the full example. For a production deployment however, it is recommended that the peer and ordering nodes belong to separate organizations which is why this tutorial uses `ordererOrg1.example.com` for the orderer organization. If you are doing the same, throughout the `configtx.yaml` you would need to change all references of the orderer organization from `SampleOrg` to `ordererOrg1.example.com`. This would include adding a new orderer organization `ordererOrg1.example.com` to the `Organizations:` section of the file.

An unlimited number of profiles can be listed in the `Profiles` section according to the needs of your use cases. However, each must have a distinctive name (this will not become the name of the channel itself, as that is specified by a flag given when the channel is created). In the next step, you specify which profile to use when you generate the channel genesis block.

Generate the genesis block

After you have completed editing the `configtx.yaml`, you can use it to create a new channel for the peer organizations. Every channel configuration starts with a genesis block. Because we previously set the environment variables for the `configtxgen` tool, you can run the following command to build the genesis block for `channel1` using the `SampleAppChannelEtcdRaft` profile:

```
configtxgen -profile SampleAppGenesisEtcdRaft -outputBlock genesis_block.pb -
↳channelID channel1
```

Where:

- `-profile` - Is the name of the profile in `configtx.yaml` that will be used to create the channel.
- `-outputBlock` - Is the location of where to store the generated configuration block file.
- `-channelID` - Is the name of the channel being created. Channel names must be all lower case, less than 250 characters long and match the regular expression `[a-z][a-z0-9.-]*`. The command uses the `-profile` flag to reference the `SampleAppGenesisEtdcRaft: profile` from `configtx.yaml`.

When the command is successful, you will see logs of `configtxgen` loading the `configtx.yaml` file and printing a channel creation transaction:

```
INFO 001 Loading configuration
INFO 002 orderer type: etcdraft
INFO 003 Loaded configuration: /fabric/config/configtx.yaml
INFO 004 Generating genesis block
INFO 005 Writing genesis block
```

Step two: Use the `osnadmin` CLI to add the first orderer to the channel

Now that the genesis block has been created, the first ordering node that receives the `osnadmin channel join` command command effectively “activates” the channel, though the channel is not fully operational until a quorum of consenters is established (if your profile listed three consenters, at least one more node, for a total of two, would have to join using the `osnadmin channel join` command).

Note: If you try to run the `osnadmin` commands (aside from the `channel list` command) against an ordering node that is a member of a system channel, you get an error which indicates that the system channel still exists. The system channel needs to be *removed* before the `osnadmin` commands can be used to create or join channels.

Each orderer needs to run the following command:

```
export OSN_TLS_CA_ROOT_CERT=../config/organizations/ordererOrganizations/ordererOrg1.
↪example.com/ordering-service-nodes/osn1.ordererOrg1.example.com/tls/cacerts/tls-ca-
↪cert.pem
export ADMIN_TLS_SIGN_CERT=../config/admin-client/client-tls-cert.pem
export ADMIN_TLS_PRIVATE_KEY=../config/admin-client/client-tls-key.pem

osnadmin channel join --channelID [CHANNEL_NAME] --config-block [CHANNEL_CONFIG_
↪BLOCK] -o [ORDERER_ADMIN_LISTENADDRESS] --ca-file $OSN_TLS_CA_ROOT_CERT --client-
↪cert $ADMIN_TLS_SIGN_CERT --client-key $ADMIN_TLS_PRIVATE_KEY
```

Replace:

- `CHANNEL_NAME` with the name you want to call this channel.
- `CHANNEL_CONFIG_BLOCK` with the path and file name of the genesis block you created in Step one if you are creating the channel. Each subsequent ordering node can join the configuration starting with the genesis block, or they can join by providing the latest config block instead.
- `ORDERER_ADMIN_LISTENADDRESS` corresponds to the `Orderer.Admin.ListenAddress` defined in the `orderer.yaml` for this orderer.
- `OSN_TLS_CA_ROOT_CERT` with the path and file name of the orderer organization TLS CA root certificate and intermediate certificate if using an intermediate TLS CA.
- `ADMIN_TLS_SIGN_CERT` with the path and file name of the admin client signed certificate from the TLS CA.
- `ADMIN_TLS_PRIVATE_KEY` with the path and file name of the admin client private key from the TLS CA.

For example:

```
osnadmin channel join --channelID channel1 --config-block genesis_block.pb -o OSN1.
↪example.com:7050 --ca-file $OSN_TLS_CA_ROOT_CERT --client-cert $ADMIN_TLS_SIGN_CERT_
↪--client-key $ADMIN_TLS_PRIVATE_KEY
```

Note: Because the connection between the `osnadmin` CLI and the orderer requires mutual TLS, you need to pass the `--client-cert` and `--client-key` parameters on each `osadmin` command. The `--client-cert` parameter points to the admin client certificate and `--client-key` refers to the admin client private key, both issued by the admin client TLS CA.

The output of this command looks similar to:

```
{
  "name": "channel1",
  "url": "participation/v1/channels/channel1",
  "consensusRelation": "consenter",
  "status": "active",
  "height": 1
}
```

When successful, the orderer is joined to the channel with ledger height 1. Because this orderer is joining from the genesis block, the status is “active”. And because the orderer is part of the channel’s consenter set, its `consensusRelation` is “consenter”. We delve more into the `consensusRelation` value in the next section.

You can repeat the same command on the other two orderers. Remember to update the orderer endpoint, `-o [ORDERER_ADMIN_LISTENADDRESS]`, on the command for the corresponding orderer. Because these two orderers join the channel from the genesis block and are part of the consenter set, their status transitions almost immediately from “onboarding” to “active” and the “consensusRelation” status is “consenter”. At this point, the channel becomes operational because a majority of consenter (as defined in the channel configuration) are “active”. You should see something similar to the following in your orderer logs:

```
INFO 087 raft.node: 1 elected leader 1 at term 2 channel=channel1 node=1
INFO 088 Raft leader changed: 0 -> 1 channel=channel1 node=1
INFO 089 Start accepting requests as Raft leader at block [0] channel=channel1 node=1
```

After the first orderer is added to the channel, subsequent nodes can join from either the genesis block or from the latest config block. When an orderer joins from a config block, its status is always “onboarding” while its ledger catches up to the config block that was specified in the join command, after which the status is automatically updated to “active”.

Use the `osnadmin channel list` command with the `--channelID` flag to view the detailed status and `consensusRelation` of any **channel** on any ordering node:

```
osnadmin channel list --channelID [CHANNEL_NAME] -o [ORDERER_ADMIN_LISTENADDRESS] --
↪ca-file $OSN_TLS_CA_ROOT_CERT --client-cert $ADMIN_TLS_SIGN_CERT --client-key
↪$ADMIN_TLS_PRIVATE_KEY
```

For example:

```
osnadmin channel list --channelID channel1 -o HOST2:7081 --ca-file $OSN_TLS_CA_ROOT_
↪CERT --client-cert $ADMIN_TLS_SIGN_CERT --client-key $ADMIN_TLS_PRIVATE_KEY
```

Replace:

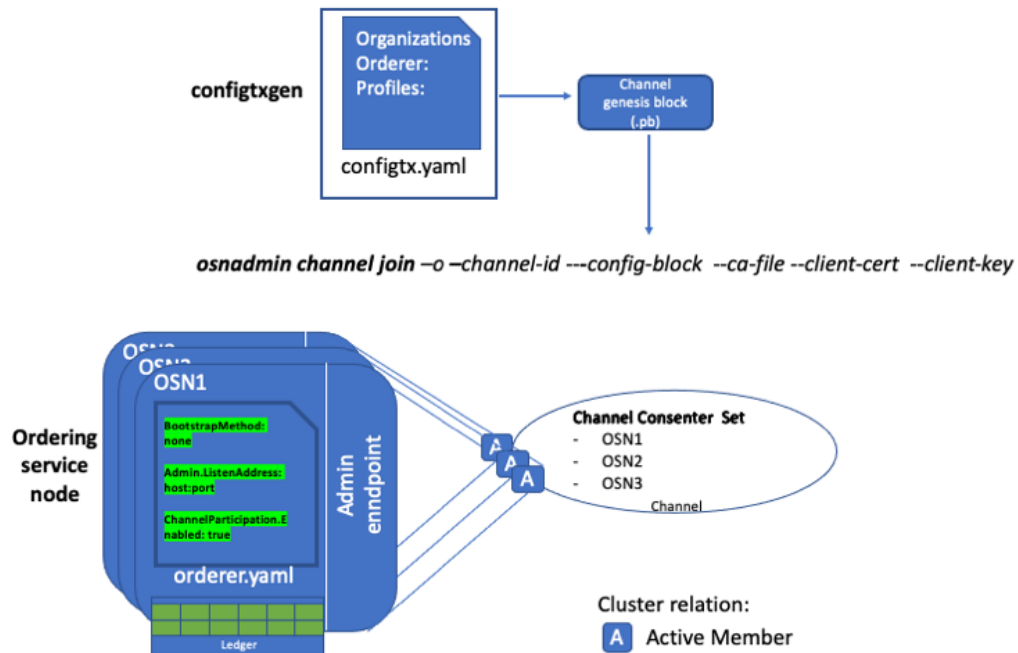
- `CHANNEL_NAME` with the name of the channel specified in the `configtxgen` command when you generated the channel.
- `ORDERER_ADMIN_LISTENADDRESS` with `Orderer.Admin.ListenAddress` defined in the `orderer.yaml` for this orderer.

- `OSN_TLS_CA_ROOT_CERT` with the path and file name of the orderer organization TLS CA root certificate and intermediate certificate if using an intermediate TLS CA.
- `ADMIN_TLS_SIGN_CERT` with the path and file name of the admin client signed certificate from the TLS CA.
- `ADMIN_TLS_PRIVATE_KEY` with the path and file name of the admin client private key from the TLS CA.

The output of this command looks similar to:

```
{
  "name": "channel1",
  "url": "participation/v1/channels/channel1",
  "consensusRelation": "consenter",
  "status": "active",
  "height": 1
}
```

The following diagram summarizes the steps you have completed:



You used the `configtxgen` command to create the channel genesis block and provided that file when you ran the `osnadmin channel join` command for each orderer by targeting the admin endpoint on each node.

Assuming you have successfully run the `osnadmin channel join` on all three ordering nodes, you now have an active channel and the ordering service is ready to order transactions into blocks. Peers can join the channel and clients can begin to transact.

If you want to join additional ordering nodes to the consenter set of the channel, follow the instructions in the next section.

Step three: Join additional ordering nodes

Over time it may be necessary to add additional ordering nodes to the consenter set of a channel. Other organizations may want to contribute their own orderers to the cluster, or it might prove advantageous under production conditions to permit multiple orderers to go down for maintenance at the same time.

Anytime a new orderer joins the cluster, you need to ensure that you do not lose quorum while the ledger on the new orderer catches up. Adding a fourth orderer to an existing three node cluster changes the majority from two to three. Therefore, during the catch-up process and until the new orderer starts executing consensus, the cluster is in a state of reduced fault tolerance. To account for this situation and avoid any downtime, a command that can ascertain the `consensusRelation` status of an orderer, which can be either “consenter” or “follower”, has been introduced. When a node joins as a follower, the channel ledger is replicated from other orderers, but as it does not take part in consensus, channel operations are not impacted. Because replicating a long chain of blocks could take a long time, joining as a follower is most useful for channels with high block heights because it allows you to confirm that an orderer is able to replicate blocks before having it count towards quorum. To join an orderer to a channel as a follower, **do not include the node in the channel configuration consenter set**.

To simplify the tutorial, we assume this additional orderer is part of the same organization as the previous three orderers and the orderer organization is already part of the channel configuration. If the organization of the new orderer is not already part of the channel configuration, a channel configuration update that adds the organization must be submitted (and signed by a set of signatures that satisfies the orderer group’s admins policy) before the new orderer can begin pulling blocks. An orderer can pull blocks from other cluster members if its owning organization is defined in the channel configuration, and its orderer signature satisfies the `/Channel/Readers` policy.

For this tutorial, the new orderer is not part of the consenter set. Run the following command to join the new orderer to the channel:

```
osnadmin channel join --channelID [CHANNEL_NAME] --config-block [CHANNEL_CONFIG_
↪BLOCK] -o [ORDERER_ADMIN_LISTENADDRESS] --ca-file $OSN_TLS_CA_ROOT_CERT --client-
↪cert $ADMIN_TLS_SIGN_CERT --client-key $ADMIN_TLS_PRIVATE_KEY
```

An orderer can join the channel by providing the genesis block, or the latest config block. But the value of `consensusRelation` will always be “follower” until this orderer is added to the channel’s consenter set, by submitting an update to the channel configuration.

If **joining from the genesis block**, the output of this command looks similar to:

```
{
  "name": "channel1",
  "url": "participation/v1/channels/channel1",
  "status": "active",
  "consensusRelation": "follower",
  "height": 1
}
```

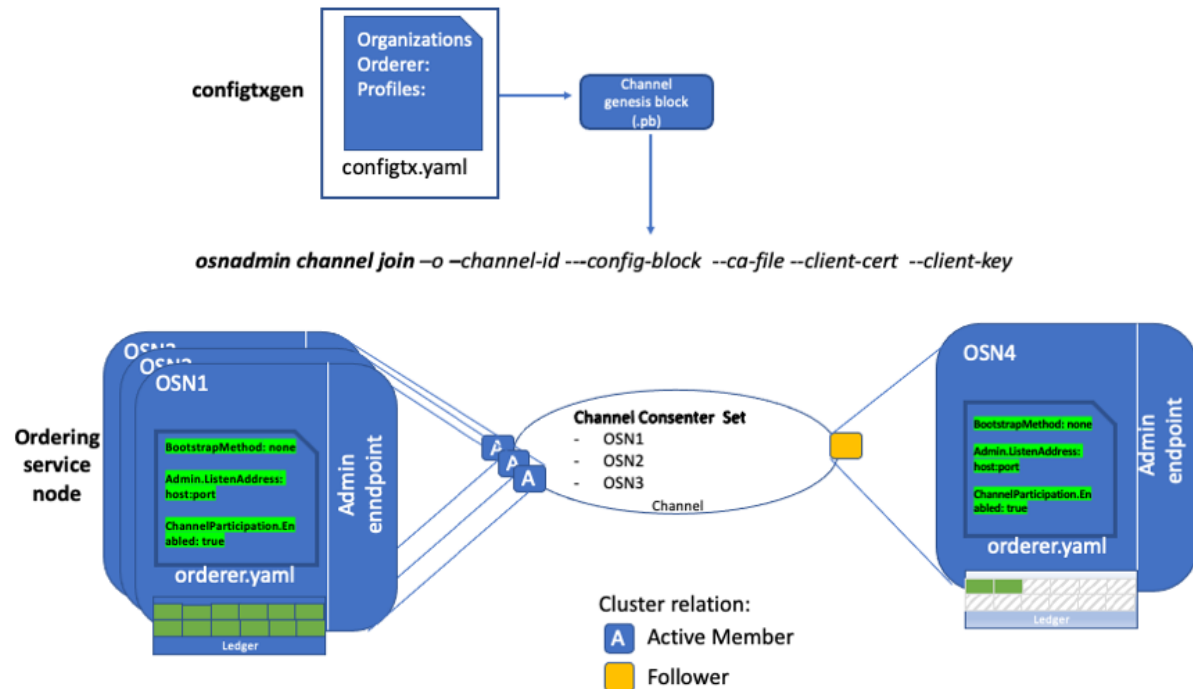
Otherwise, if **joining from the latest config block**, the status is **onboarding** until the channel ledger has caught up to the specified config block. You can issue the same `osnadmin channel list` command to confirm the status changes to **active** and the ledger height has caught up. For example, if the total number of blocks on the channel ledger at the specified config block was 1151, after the ledger catches up, the output would look similar to:

```
{
  "name": "channel1",
  "url": "participation/v1/channels/channel1",
  "status": "active",
  "consensusRelation": "follower",
  "height": 1151
}
```

However, it is important to note that the transition from **onboarding** to **active** should not be the only criteria used to decide whether it is the right time to add the new orderer to the channel’s consenter set. What is more important is the height that the new orderer reached, compared to the height of the consenter. For example, the specified config block may be at height 1151, but there could be a huge number of normal blocks that follow. After OSN4 becomes **active** (caught up to the specified config), it is better to compare the height of OSN1 with that of OSN4, and if they are close enough to each other – only then add OSN4 to the consenter set of the channel.

As a reminder, to pull the blocks to the channel ledger, even as a follower, this new orderer must belong to an organization that is part of the current channel configuration. Otherwise, it is unable to replicate any blocks to its ledger. Also, notice the `consensusRelation` status is still **follower** because the node is not part of the channel's consenter set.

The diagram shows the new ordering node that has been added as a follower:



OSN4 is added to the channel as a follower. Its status is “onboarding” while its ledger catches up to the config block provided on the `osnadmin channel join` command.

Although the orderer status transitions from **onboarding** to **active**, it is still unable to participate in the ordering service because its `consensusRelation` status is **follower**, but it will continue to pull blocks and stay current with the ledger. After the status becomes **active** and you are ready for the orderer to transition from a follower to a consenter, any orderer organization admin can add the orderer to the consenter set on the channel by submitting a [channel configuration update transaction](#), or by using the `fabric-config` library.

Note: you should never attempt any configuration changes to the Raft consenter, such as adding a consenter, unless all consenter are online and healthy, as you risk losing quorum and therefore the ability to process transactions.

Ensure that the orderer organization has write privileges in the channel configuration so the orderer can become a consenter. When the channel update transaction is complete, you can use the `osnadmin channel list` command again to confirm that the `consensusRelation` status for the orderer automatically changes to **consenter**.

```
{
  "name": "channel1",
  "url": "participation/v1/channels/channel1",
  "status": "active",
  "consensusRelation": "consenter",
  "height": 1152
}
```

Note that if the config block specified on the join command was at a height of 1151, and there were 1000 other blocks after it, then the config block that adds OSN4 as a consenter would be at 2152.

Next steps

Join peers to the channel

After the channel has been created, you can follow the normal process to join peers to the channel and configure anchor peers. If the peer organization was not originally included in the channel configuration, you need to submit a [channel configuration update transaction](#) to add the organization. If it did include the peer organization, then all peer organizations that are members of the channel can fetch the channel genesis block from the ordering service using the [peer channel fetch](#) command. The organization can then use the genesis block to join the peer to the channel using the [peer channel join](#) command. Once the peer is joined to the channel, the peer will build the blockchain ledger by retrieving the other blocks on the channel from the ordering service.

Add or remove orderers from existing channels

You can continue to use the `osnadmin channel join` and `osnadmin channel remove` commands to add and remove orderers on each channel according to your business needs. Be aware that before you remove a channel from an orderer, it is recommended that you first remove the orderer from the channel's consenter set by submitting a channel update request.

7.7.2 Create a channel using the test network

Use this tutorial along with the test network to learn how to create a channel genesis block and then create a new application channel that the test network peers can join. Rather than requiring you to set up an orderer, or remove the system channel from an existing orderer, this tutorial leverages the nodes from the Fabric sample test network. Because the test network deploys an ordering service and peers for you, this tutorial focuses solely on the process to create a channel. It is worth noting that the test network includes a `createChannel` subcommand that can be used to create a channel, but this tutorial explains how to do it manually, the process that is required when you do not use the test network.

Fabric v2.3 introduces the capability to create a channel without requiring a system channel, removing an extra layer of administration from the process. In this tutorial, we use the `configtxgen` tool to create a channel genesis block and then use the `osnadmin channel` command to create the channel.

Note:

- If you are *not* using the test network, you should follow the instructions for [how to deploy an ordering service without a system channel](#). In the Fabric v2.3 test network sample, the single-node ordering service is deployed without a system channel.
- If you prefer to learn how to create a channel on an ordering service that includes the system channel, you should refer to the [Create a channel tutorial](#) from Fabric v2.2. In the Fabric v2.2 test network sample, the single-node ordering service is deployed with a system channel.

To create a channel using the test network, this tutorial takes you through the following steps and concepts:

- *Prerequisites*
- *Step one: Generate the genesis block of the channel*
- *Step two: Create the application channel*
- *Next steps*

Before you begin

To run the test network, you need to clone the `fabric-samples` repository and download the latest production Fabric images. Make sure that you have installed the [Prerequisites](#) and [Installed the Samples, Binaries, and Docker Images](#).

Note: After you create a channel and join peers to it, you will need to add anchor peers to the channel, in order for service discovery and private data to work. Instructions on how to set an anchor peer on your channel are included in this tutorial, but require that the `jq` tool is installed on your local machine.

Prerequisites

Start the test network

We will use a running instance of the Fabric test network to create the new channel. Because it's important to operate from a known initial state, the following command destroys any active containers and removes any previously generated artifacts. For the purposes of this tutorial, we operate from the `test-network` directory inside `fabric-samples`. If you are not already there, navigate to that directory using the following command:

```
cd fabric-samples/test-network
```

Run the following command to bring down the network:

```
./network.sh down
```

You can then use the following command to start the test network:

```
./network.sh up
```

This command creates a Fabric network with the two peer organizations and the single ordering node ordering organization. The peer organizations will operate one peer each, while the ordering service administrator will operate a single ordering node. When you run the command, the script prints out the nodes being created:

```
Creating network "net_test" with the default driver
Creating volume "net_orderer.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating peer0.org2.example.com ... done
Creating orderer.example.com ... done
Creating peer0.org1.example.com ... done
Creating cli ... done
```

CONTAINER ID	IMAGE	COMMAND	CREATED	NAMES
1667543b5634	hyperledger/fabric-tools:latest	"/bin/bash"	1 second ago	cli
b6b117c81c7f	hyperledger/fabric-peer:latest	"peer node start"	2 seconds ago	peer0.org1.example.com
703ead770e05	hyperledger/fabric-orderer:latest	"orderer"	2 seconds ago	orderer.example.com
718d43f5f312	hyperledger/fabric-peer:latest	"peer node start"	2 seconds ago	peer0.org2.example.com

Notice that the peers are running on ports 7051 and 9051, while the orderer is running on port 7050. We will use these ports in subsequent commands.

By default, when you start the test network, it does not contain any channels. The following instructions demonstrate how to add a channel that is named `channel1` to this network.

Set up the configtxgen tool

Channels are created by generating a channel creation transaction in a genesis block, and then passing that genesis block to an ordering service node in a join request. The channel creation transaction specifies the initial configuration of the channel and can be created by the `configtxgen` tool. The tool reads the `configtx.yaml` file that defines the configuration of our channel, and then writes the relevant information into the channel creation transaction and outputs a genesis block including the channel creation transaction. When you installed Fabric, the `configtxgen` tool was installed in the `fabric-samples\bin` directory for you.

Ensure that you are still operating from the `test-network` directory of your local clone of `fabric-samples` and run this command:

```
export PATH=${PWD}/../bin:$PATH
```

Next, before you can use `configtxgen`, you need to set the `FABRIC_CFG_PATH` environment variable to the location of the test network folder that contains the `configtx.yaml` file. Because we are using the test network, we reference the `configtx` folder:

```
export FABRIC_CFG_PATH=${PWD}/configtx
```

Now verify that you can use the tool by printing the `configtxgen` help text:

```
configtxgen --help
```

The configtx.yaml file

For the test network, the `configtxgen` tool uses the channel profiles that are defined in the `configtx\configtx.yaml` file to create the channel configuration and write it to the `protobuf` format that can be read by Fabric.

This `configtx.yaml` file contains the following information that we will use to create our new channel:

- **Organizations:** The peer and ordering organizations that can become members of your channel. Each organization has a reference to the cryptographic material that is used to build the `channel MSP`.
- **Ordering service:** Which ordering nodes will form the ordering service of the network, and consensus method they will use to agree to a common order of transactions. This section also defines the ordering nodes that are part of the ordering service consenter set. In the test network sample, there is only a single ordering node, but in a production network we recommend **five** ordering nodes to allow for two ordering nodes to go down and still maintain consensus.

```
EtcdRaft:
  Consenters:
    - Host: orderer.example.com
      Port: 7050
      ClientTLS_CERT: ../organizations/ordererOrganizations/example.com/orderers/
        orderer.example.com/tls/server.crt
      ServerTLS_CERT: ../organizations/ordererOrganizations/example.com/orderers/
        orderer.example.com/tls/server.crt
```

- **Channel policies** Different sections of the file work together to define the policies that will govern how organizations interact with the channel and which organizations need to approve channel updates. For the purposes of this tutorial, we will use the default policies used by Fabric.
- **Channel profiles** Each channel profile references information from other sections of the `configtx.yaml` file to build a channel configuration. The profiles are used to create the genesis block of applica-

tion channel. Notice that the `configtx.yaml` file in the test network includes a single profile named `TwoOrgsApplicationGenesis` that we will use to generate the create channel transaction.

```
TwoOrgsApplicationGenesis:
  <<: *ChannelDefaults
  Orderer:
    <<: *OrdererDefaults
    Organizations:
      - *OrdererOrg
    Capabilities:
      <<: *OrdererCapabilities
  Application:
    <<: *ApplicationDefaults
    Organizations:
      - *Org1
      - *Org2
    Capabilities:
      <<: *ApplicationCapabilities
```

The profile includes both peer organizations, `Org1` and `Org2` as well as the ordering organization `OrdererOrg`. Additional ordering nodes and ordering organizations can be added or removed from the consenter set at a later time using a channel update transaction.

Want to learn more about this file and how to build your own channel application profiles? Visit [Using configtx.yaml to create a channel genesis block](#) tutorial for more details. For now, we will return to the operational aspects of creating the channel, though we will reference parts of this file in future steps.

Step one: Generate the genesis block of the channel

Because we have started the Fabric test network, we are ready to create a new channel. We have already set the environment variables that are required to use the `configtxgen` tool.

Run the following command to create the channel genesis block for `channel1`:

```
configtxgen -profile TwoOrgsApplicationGenesis -outputBlock ./channel-artifacts/
↳channel1.block -channelID channel1
```

- **-profile:** The command uses the `-profile` flag to reference the `TwoOrgsApplicationGenesis` profile from `configtx.yaml` that is used by the test network to create application channels.
- **-outputBlock:** The output of this command is the channel genesis block that is written to `-outputBlock ./channel-artifacts/channel1.block`.
- **-channelID:** The `-channelID` parameter will be the name of the future channel. You can specify any name you want for your channel but for illustration purposes in this tutorial we use `channel1`. Channel names must be all lowercase, fewer than 250 characters long and match the regular expression `[a-z][a-z0-9.-]*`.

When the command is successful, you can see the logs of `configtxgen` loading the `configtx.yaml` file and printing a channel creation transaction:

```
[common.tools.configtxgen] main -> INFO 001 Loading configuration
[common.tools.configtxgen.localconfig] completeInitialization -> INFO 002 orderer_
↳type: etcdraft
[common.tools.configtxgen.localconfig] completeInitialization -> INFO 003 Orderer.
↳EtcdRaft.Options unset, setting to tick_interval:"500ms" election_tick:10 heartbeat_
↳tick:1 max_inflight_blocks:5 snapshot_interval_size:16777216
[common.tools.configtxgen.localconfig] Load -> INFO 004 Loaded configuration: /Users/
↳fabric-samples/test-network/configtx/configtx.yaml
```

(continues on next page)

(continued from previous page)

```
[common.tools.configtxgen] doOutputBlock -> INFO 005 Generating genesis block
[common.tools.configtxgen] doOutputBlock -> INFO 006 Creating application channel_
↪ genesis block
[common.tools.configtxgen] doOutputBlock -> INFO 007 Writing genesis block
```

Step two: Create the application channel

Now that we have the channel genesis block, it is easy to use the `osnadmin channel join` command to create the channel. To simplify subsequent commands, we also need to set some environment variables to establish the locations of the certificates for the nodes in the test network:

```
export ORDERER_CA=${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪ orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
export ORDERER_ADMIN_TLS_SIGN_CERT=${PWD}/organizations/ordererOrganizations/example.
↪ com/orderers/orderer.example.com/tls/server.crt
export ORDERER_ADMIN_TLS_PRIVATE_KEY=${PWD}/organizations/ordererOrganizations/
↪ example.com/orderers/orderer.example.com/tls/server.key
```

Run the following command to create the channel named `channel1` on the ordering service.

```
osnadmin channel join --channelID channel1 --config-block ./channel-artifacts/
↪ channel1.block -o localhost:7053 --ca-file "$ORDERER_CA" --client-cert "$ORDERER_
↪ ADMIN_TLS_SIGN_CERT" --client-key "$ORDERER_ADMIN_TLS_PRIVATE_KEY"
```

- **--channelID:** Specify the name of the application channel that you provided when you created the channel genesis block.
- **--config-block:** Specify the location of the channel genesis block that you created with the `configtxgen` command, or the latest config block.
- **-o:** Specify the hostname and port of the orderer admin endpoint. For the test network ordering node this is set to `localhost:7053`.

In addition, because the `osnadmin channel` commands communicate with the ordering node using mutual TLS, you need to provide the following certificates:

- **--ca-file:** Specify the location and file name of the orderer organization TLS CA root certificate.
- **--client-cert:** Specify the location and file name of admin client signed certificate from the TLS CA.
- **--client-key:** Specify the location and file name of admin client private key from the TLS CA.

When successful, the output of the command contains the following:

```
Status: 201
{
  "name": "channel1",
  "url": "/participation/v1/channels/channel1",
  "consensusRelation": "consenter",
  "status": "active",
  "height": 1
}
```

The channel is active and ready for peers to join.

Consenter vs. Follower

Notice the ordering node was joined to the channel with a `consensusRelation`: `"consenter"`. If you ran the command against an ordering node that is not included in the list of `Consenters`: in the `configtx.yaml` file (or the channel configuration `consenter` set), it is added to the channel as a `follower`. To learn more about considerations when joining additional ordering nodes see the topic on [Joining additional ordering nodes](#).

Active vs. onboarding

An orderer can join the channel by providing the channel **genesis block**, or the **latest config block**. If joining from the latest config block, the orderer status is set to `onboarding` until the channel ledger has caught up to the specified config block, when it becomes `active`. At this point, you could then add the orderer to the channel `consenter` set by submitting a channel update transaction, which will cause the `consensusRelation` to change from `follower` to `consenter`.

Next steps

After you have created the channel, the next steps are to join peers to the channel and deploy smart contracts. This section walks you through those processes using the test network.

List channels on an orderer

Before you join peers to the channel, you might want to try to create additional channels. As you create more channels, the `osnadmin channel list` command is useful to view the channels that this orderer is a member of. The same parameters are used here as in the `osnadmin channel join` command from the previous step:

```
osnadmin channel list -o localhost:7053 --ca-file "$ORDERER_CA" --client-cert "  
↪$ORDERER_ADMIN_TLS_SIGN_CERT" --client-key "$ORDERER_ADMIN_TLS_PRIVATE_KEY"
```

The output of this command looks similar to:

```
Status: 200  
{  
  "systemChannel": null,  
  "channels": [  
    {  
      "name": "channel1",  
      "url": "/participation/v1/channels/channel1"  
    }  
  ]  
}
```

Join peers to the channel

The test network includes two peer organizations each with one peer. But before we can use the peer CLI, we need to set some environment variables to specify which user (client MSP) we are acting as and which peer we are targeting. Set the following environment variables to indicate that we are acting as the Org1 admin and targeting the Org1 peer.

```
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

In order to use the peer CLI, we also need to modify the FABRIC_CONFIG_PATH:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To join the test network peer from Org1 to the channel channel1 simply pass the genesis block in a join request:

```
peer channel join -b ./channel-artifacts/channel1.block
```

When successful, the output of this command contains the following:

```
[channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
[channelCmd] executeJoin -> INFO 002 Successfully submitted proposal to join channel
```

We repeat these steps for the Org2 peer. Set the following environment variables to operate the peer CLI as the Org2 admin. The environment variables will also set the Org2 peer, peer0.org2.example.com, as the target peer.

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

Now repeat the command to join the peer from Org2 to channel1:

```
peer channel join -b ./channel-artifacts/channel1.block
```

When successful, the output of this command contains the following:

```
[channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
[channelCmd] executeJoin -> INFO 002 Successfully submitted proposal to join channel
```

Set anchor peer

Finally, after an organization has joined their peers to the channel, they should select **at least one** of their peers to become an anchor peer. [Anchor peers](#) are required in order to take advantage of features such as private data and service discovery. Each organization should set multiple anchor peers on a channel for redundancy. For more information about gossip and anchor peers, see the [Gossip data dissemination protocol](#).

The endpoint information of the anchor peers of each organization is included in the channel configuration. Each channel member can specify their anchor peers by updating the channel. We will use the [configtxlator](#) tool to update the channel configuration and select an anchor peer for Org1 and Org2.

Note: If [jq](#) is not already installed on your local machine, you need to install it now to complete these steps.

We will start by selecting the peer from Org1 to be an anchor peer. The first step is to pull the most recent channel configuration block using the `peer channel fetch` command. Set the following environment variables to operate the peer CLI as the Org1 admin:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

You can use the following command to fetch the channel configuration:

```
peer channel fetch config channel-artifacts/config_block.pb -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com -c channel1 --tls --cafile "$ORDERER_
↪CA"
```

Because the most recent channel configuration block is the channel genesis block, the command returns block 0 from the channel.

```
[channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
[cli.common] readBlock -> INFO 002 Received block: 0
[channelCmd] fetch -> INFO 003 Retrieving last config block: 0
[cli.common] readBlock -> INFO 004 Received block: 0
```

The channel configuration block `config_block.pb` is stored in the `channel-artifacts` folder to keep the update process separate from other artifacts. Change into the `channel-artifacts` folder to complete the next steps:

```
cd channel-artifacts
```

We can now start using the `configtxlator` tool to start working with the channel configuration. The first step is to decode the block from protobuf into a JSON object that can be read and edited. We also strip away the unnecessary block data, leaving only the channel configuration.

```
configtxlator proto_decode --input config_block.pb --type common.Block --output_
↪config_block.json
jq '.data.data[0].payload.data.config' config_block.json > config.json
```

These commands convert the channel configuration block into a streamlined JSON, `config.json`, that will serve as the baseline for our update. Because we don't want to edit this file directly, we will make a copy that we can edit. We will use the original channel config in a future step.

```
cp config.json config_copy.json
```

You can use the `jq` tool to add the `Org1` anchor peer to the channel configuration.

```
jq '.channel_group.groups.Application.groups.Org1MSP.values += {"AnchorPeers":{"mod_
↪policy": "Admins","value":{"anchor_peers": [{"host": "peer0.org1.example.com","port
↪": 7051}]}},"version": "0"}}' config_copy.json > modified_config.json
```

After this step, we have an updated version of channel configuration in JSON format in the `modified_config.json` file. We can now convert both the original and modified channel configurations back into protobuf format and calculate the difference between them.

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
configtxlator proto_encode --input modified_config.json --type common.Config --output_
↪modified_config.pb
configtxlator compute_update --channel_id channel1 --original config.pb --updated_
↪modified_config.pb --output config_update.pb
```

The new protobuf named `config_update.pb` contains the anchor peer update that we need to apply to the channel configuration. We can wrap the configuration update in a transaction envelope to create the channel configuration update transaction.

```
configtxlator proto_decode --input config_update.pb --type common.ConfigUpdate --
↪output config_update.json
echo '{"payload":{"header":{"channel_header":{"channel_id":"channel1", "type":2}},
↪"data":{"config_update":"'$(cat config_update.json)'"}}}' | jq . > config_update_in_
↪envelope.json
configtxlator proto_encode --input config_update_in_envelope.json --type common.
↪Envelope --output config_update_in_envelope.pb
```

We can now use the final artifact, `config_update_in_envelope.pb`, that can be used to update the channel. Navigate back to the `test-network` directory:

```
cd ..
```

We can add the anchor peer by providing the new channel configuration to the `peer channel update` command. Because we are updating a section of the channel configuration that only affects `Org1`, other channel members do not need to approve the channel update.

```
peer channel update -f channel-artifacts/config_update_in_envelope.pb -c channel1 -o_
↪localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$
↪{PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/
↪msp/tlscacerts/tlsca.example.com-cert.pem"
```

When the channel update is successful, you should see the following response:

```
[channelCmd] update -> INFO 002 Successfully submitted channel update
```

We can also set the peer from `Org2` to be an anchor peer. Because we are going through the process a second time, we will go through the steps more quickly. Set the environment variables to operate the peer CLI as the `Org2` admin:

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

Pull the latest channel configuration block, which is now the second block on the channel:

```
peer channel fetch config channel-artifacts/config_block.pb -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com -c channel1 --tls --cafile "${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem"
```

Navigate back to the `channel-artifacts` directory:

```
cd channel-artifacts
```

You can then decode and copy the configuration block.

```
configtxlator proto_decode --input config_block.pb --type common.Block --output_
↪config_block.json
jq '.data.data[0].payload.data.config' config_block.json > config.json
cp config.json config_copy.json
```

Add the Org2 peer that is joined to the channel as the anchor peer in the channel configuration:

```
jq '.channel_group.groups.Application.groups.Org2MSP.values += {"AnchorPeers":{"mod_
↪policy": "Admins","value":{"anchor_peers": [{"host": "peer0.org2.example.com","port
↪": 9051}]}},"version": "0"}}' config_copy.json > modified_config.json
```

We can now convert both the original and updated channel configurations back into protobuf format and calculate the difference between them.

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
configtxlator proto_encode --input modified_config.json --type common.Config --output_
↪modified_config.pb
configtxlator compute_update --channel_id channel1 --original config.pb --updated_
↪modified_config.pb --output config_update.pb
```

Wrap the configuration update in a transaction envelope to create the channel configuration update transaction:

```
configtxlator proto_decode --input config_update.pb --type common.ConfigUpdate --
↪output config_update.json
echo '{"payload":{"header":{"channel_header":{"channel_id":"channel1", "type":2}},
↪data":{"config_update":"'$(cat config_update.json)'"}}}' | jq . > config_update_in_
↪envelope.json
configtxlator proto_encode --input config_update_in_envelope.json --type common.
↪Envelope --output config_update_in_envelope.pb
```

Navigate back to the test-network directory.

```
cd ..
```

Update the channel and set the Org2 anchor peer by issuing the following command:

```
peer channel update -f channel-artifacts/config_update_in_envelope.pb -c channel1 -o_
↪localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$
↪{PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/
↪msp/tlscacerts/tlsca.example.com-cert.pem"
```

If you want to learn more about how to submit a channel update request, see [update a channel configuration](#).

You can confirm that the channel has been updated successfully by running the `peer channel info` command:

```
peer channel getinfo -c channel1
```

Now that the channel has been updated by adding two channel configuration blocks to the channel genesis block, the height of the channel will have grown to three and the hashes are updated:

```
Blockchain info: {"height":3,"currentBlockHash":"GKqqk/HNi9x/
↪6YPnaIUpmBlb0Ew6ovUnSB5MEF7Y5Pc=", "previousBlockHash":"cl4TOQpZ30+d17OF5YOkX/
↪mtMjJpUXiJmtw8+sON8a8="}
```

Deploy a chaincode to the new channel

We can confirm that the channel was created successfully by deploying a chaincode to the channel. We can use the `network.sh` script to deploy the Basic asset transfer chaincode to any test network channel. Deploy a chaincode to our new channel using the following command:

```
./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-go/ -ccl go -
↪c channel1
```


After you run the command, you should see the chaincode being deployed to the channel in your logs.

```
Committed chaincode definition for chaincode 'basic' on channel 'channel1':
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc,
↳ Approvals: [Org1MSP: true, Org2MSP: true]
Query chaincode definition successful on peer0.org2 on channel 'channel1'
Chaincode initialization is not required
```

Then run the following command to initialize some assets on the ledger:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↳ com --tls --cafile "$ORDERER_CA" -C channel1 -n basic --peerAddresses
↳ localhost:7051 --tlsRootCertFiles "${PWD}/organizations/peerOrganizations/org1.
↳ example.com/peers/peer0.org1.example.com/tls/ca.crt" --peerAddresses localhost:9051
↳ --tlsRootCertFiles "${PWD}/organizations/peerOrganizations/org2.example.com/peers/
↳ peer0.org2.example.com/tls/ca.crt" -c '{"function":"InitLedger","Args":[]}'
```

When successful you will see:

```
[chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Chaincode invoke successful.
↳ result: status:200
```

Confirm the assets were added to the ledger by issuing the following query:

```
peer chaincode query -C channel1 -n basic -c '{"Args":["getAllAssets"]}'
```

You should see output similar to the following:

```
[{"ID":"asset1","color":"blue","size":5,"owner":"Tomoko","appraisedValue":300},
{"ID":"asset2","color":"red","size":5,"owner":"Brad","appraisedValue":400},
{"ID":"asset3","color":"green","size":10,"owner":"Jin Soo","appraisedValue":500},
{"ID":"asset4","color":"yellow","size":10,"owner":"Max","appraisedValue":600},
{"ID":"asset5","color":"black","size":15,"owner":"Adriana","appraisedValue":700},
{"ID":"asset6","color":"white","size":15,"owner":"Michel","appraisedValue":800}]
```

Create a channel without the test network

This tutorial has taken you through the basic steps to create a channel on the test network by using the `osnadmin channel join` command. When you are ready to build your own network, follow the steps in the [Create a channel](#) tutorial to learn more about using the `osnadmin channel` commands.

7.7.3 Using configtx.yaml to build a channel configuration

A channel is created by building a channel creation transaction artifact that specifies the initial configuration of the channel. The **channel configuration** is stored on the ledger, and governs all the subsequent blocks that are added to the channel. The channel configuration specifies which organizations are channel members, the ordering nodes that can add new blocks on the channel, as well as the policies that govern channel updates. The initial channel configuration, stored in the channel genesis block, can be updated through channel configuration updates. If a sufficient number of organizations approve a channel update, a new channel config block will govern the channel after it is committed to the channel.

While it is possible to build the channel creation transaction file manually, it is easier to create a channel by using the `configtx.yaml` file and the `configtxgen` tool. The `configtx.yaml` file contains the information that is required to build the channel configuration in a format that can be easily read and edited by humans. The `configtxgen` tool reads the information in the `configtx.yaml` file and writes it to the `protobuf` format that can be read by Fabric.

Overview

You can use this tutorial to learn how to use the `configtx.yaml` file to build the initial channel configuration that is stored in the genesis block. The tutorial will discuss the portion of channel configuration that is built by each section of file.

- *Organizations*
- *Capabilities*
- *Application*
- *Orderer*
- *Channel*
- *Profiles*

Because different sections of the file work together to create the policies that govern the channel, we will discuss channel policies in [their own tutorial](#).

Building off of the [Creating a channel tutorial](#), we will use the `configtx.yaml` file that is used to deploy the Fabric test network as an example. Open a command terminal on your local machine and navigate to the `test-network` directory in your local clone of the Fabric samples:

```
cd fabric-samples/test-network
```

The `configtx.yaml` file used by the test network is located in the `configtx` folder. Open the file in a text editor. You can refer back to this file as the tutorial goes through each section. You can find a more detailed version of the `configtx.yaml` file in the [Fabric sample configuration](#).

Organizations

The most important information contained in the channel configuration are the organizations that are channel members. Each organization is identified by an MSP ID and a [channel MSP](#). The channel MSP is stored in the channel configuration and contains the certificates that are used to identify the nodes, applications, and administrators of an organization. The **Organizations** section of `configtx.yaml` file is used to create the channel MSP and accompanying MSP ID for each member of the channel.

The `configtx.yaml` file used by the test network contains three organizations. Two organizations are peer organizations, Org1 and Org2, that can be added to application channels. One organization, OrdererOrg, is the administrator of the ordering service. Because it is a best practice to use different certificate authorities to deploy peer nodes and ordering nodes, organizations are often referred to as peer organizations or ordering organizations, even if they are in fact run by the same company.

You can see the part of `configtx.yaml` that defines Org1 of the test network below:

```
- &Org1
  # DefaultOrg defines the organization which is used in the sampleconfig
  # of the fabric.git development environment
  Name: Org1MSP

  # ID to load the MSP definition as
  ID: Org1MSP

  MSPDir: ../organizations/peerOrganizations/org1.example.com/msp

  # Policies defines the set of policies at this level of the config tree
  # For organization policies, their canonical path is usually
```

(continues on next page)

(continued from previous page)

```
# /Channel/<Application/Orderer>/<OrgName>/<PolicyName>
Policies:
  Readers:
    Type: Signature
    Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client')"
  Writers:
    Type: Signature
    Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
  Admins:
    Type: Signature
    Rule: "OR('Org1MSP.admin') "
  Endorsement:
    Type: Signature
    Rule: "OR('Org1MSP.peer') "
```

- The `Name` field is an informal name used to identify the organization.
- The `ID` field is the organization's MSP ID. The MSP ID acts as a unique identifier for your organization, and is referred to by channel policies and is included in the transactions submitted to the channel.
- The `MSPDir` is the path to an MSP folder that was created by the organization. The `configtxgen` tool will use this MSP folder to create the channel MSP. This MSP folder needs to contain the following information, which will be transferred to the channel MSP and stored in the channel configuration:
 - A CA root certificate that establishes the root of trust for the organization. The CA root cert is used to verify if an application, node, or administrator belongs to a channel member.
 - A root cert from the TLS CA that issued the TLS certificates of the peer or orderer nodes. The TLS root cert is used to identify the organization by the gossip protocol.
 - If Node OUs are enabled, the MSP folder needs to contain a `config.yaml` file that identifies the administrators, nodes, and clients based on the OUs of their x509 certificates.
 - If Node OUs are not enabled, the MSP needs to contain an `admincerts` folder that contains the signing certificates of the organizations administrator identities.

The MSP folder that is used to create the channel MSP only contains public certificates. As a result, you can build the MSP folder locally, and then send the MSP to the organization that is creating the channel.

- The `Policies` section is used to define a set of signature policies that reference the channel member. We will discuss these policies in more detail when we discuss [channel policies](#).

Capabilities

Fabric channels can be joined by orderer and peer nodes that are running different versions of Hyperledger Fabric. Channel capabilities allow organizations that are running different Fabric binaries to participate on the same channel by only enabling certain features. For example, organizations that are running Fabric v1.4 and organizations that are running Fabric v2.x can join the same channel as long as the channel capabilities levels are set to `V1_4_X` or below. None of the channel members will be able to use the features introduced in Fabric v2.0.

If you examine the `configtx.yaml` file, you will see three capability groups:

- **Application** capabilities govern the features that are used by peer nodes, such as the Fabric chaincode lifecycle, and set the minimum version of the Fabric binaries that can be run by peers joined to the channel.
- **Orderer** capabilities govern the features that are used by orderer nodes, such as Raft consensus, and set the minimum version of the Fabric binaries that can be run by ordering nodes that belong to the channel consenter set.

- **Channel** capabilities set the minimum version of the Fabric that can be run by peer and ordering nodes.

Because both of the peers and the ordering node of the Fabric test network run version v2.x, every capability group is set to V2_0. As a result, the test network cannot be joined by nodes that run a lower version of Fabric than v2.0. For more information, see the [capabilities](#) concept topic.

Application

The application section defines the policies that govern how peer organizations can interact with application channels. These policies govern the number of peer organizations that need to approve a chaincode definition or sign a request to update the channel configuration. These policies are also used to restrict access to channel resources, such as the ability to write to the channel ledger or to query channel events.

The test network uses the default application policies provided by Hyperledger Fabric. If you use the default policies, all peer organizations will be able to read and write data to the ledger. The default policies also require that a majority of channel members sign channel configuration updates and that a majority of channel members need to approve a chaincode definition before a chaincode can be deployed to a channel. The contents of this section are discussed in more detail in the [channel policies](#) tutorial.

Orderer

Each channel configuration includes the orderer nodes in the channel [consenter set](#). The consenter set is the group of ordering nodes that have the ability to create new blocks and distribute them to the peers joined to the channel. The endpoint information of each ordering node that is a member of the consenter set is stored in the channel configuration.

The test network uses the **Orderer** section of the `configtx.yaml` file to create a single node Raft ordering service.

- The `OrdererType` field is used to select Raft as the consensus type:

```
OrdererType: etcdraft
```

Raft ordering services are defined by the list of consenteres that can participate in the consensus process. Because the test network only uses a single ordering node, the consenteres list contains only one endpoint:

```
EtdcRaft:
  Consenteres:
    - Host: orderer.example.com
      Port: 7050
      ClientTLSCert: ../organizations/ordererOrganizations/example.com/orderers/
        ↳orderer.example.com/tls/server.crt
      ServerTLSCert: ../organizations/ordererOrganizations/example.com/orderers/
        ↳orderer.example.com/tls/server.crt
  Addressee:
    - orderer.example.com:7050
```

Each ordering node in the list of consenteres is identified by their endpoint address and their client and server TLS certificate. If you are deploying a multi-node ordering service, you would need to provide the hostname, port, and the path to the TLS certificates used by each node. You would also need to add the endpoint address of each ordering node to the list of `Addressee`.

- You can use the `BatchTimeout` and `BatchSize` fields to tune the latency and throughput of the channel by changing the maximum size of each block and how often a new block is created.
- The `Policies` section creates the policies that govern the channel consenter set. The test network uses the default policies provided by Fabric, which require that a majority of orderer administrators approve the addition or removal of ordering nodes, organizations, or an update to the block cutting parameters.

Because the test network is used for development and testing, it uses an ordering service that consists of a single ordering node. Networks that are deployed in production should use a multi-node ordering service for security and availability. To learn more, see [Configuring and operating a Raft ordering service](#).

Channel

The channel section defines the policies that govern the highest level of the channel configuration. For an application channel, these policies govern the hashing algorithm, the data hashing structure used to create new blocks, and the channel capability level. In the system channel, these policies also govern the creation or removal of consortiums of peer organizations.

The test network uses the default policies provided by Fabric, which require that a majority of orderer service administrators would need to approve updates to these values in the system channel. In an application channel, changes would need to be approved by a majority of orderer organizations and a majority of channel members. Most users will not need to change these values.

Profiles

The `configtxgen` tool reads the channel profiles in the **Profiles** section to build a channel configuration. Each profile uses YAML syntax to gather data from other sections of the file. The `configtxgen` tool uses this configuration to create a channel creation transaction for an applications channel, or to write the channel genesis block for a system channel. To learn more about YAML syntax, [Wikipedia](#) provides a good place to get started.

The `configtx.yaml` used by the test network contains two channel profiles, `TwoOrgsOrdererGenesis` and `TwoOrgsChannel`. If you want to create a channel without a system channel by using the `osnadmin` CLI then you can refer to the `SampleAppChannelEtcdRaft`

TwoOrgsOrdererGenesis

The `TwoOrgsOrdererGenesis` profile is used to create the system channel genesis block:

```
TwoOrgsOrdererGenesis:
  <<: *ChannelDefaults
  Orderer:
    <<: *OrdererDefaults
    Organizations:
      - *OrdererOrg
    Capabilities:
      <<: *OrdererCapabilities
  Consortiums:
    SampleConsortium:
      Organizations:
        - *Org1
        - *Org2
```

The system channel defines the nodes of the ordering service and the set of organizations that are ordering service administrators. The system channel also includes a set of peer organizations that belong to the blockchain [consortium](#). The channel MSP of each member of the consortium is included in the system channel, allowing them to create new application channels and add consortium members to the new channel.

The profile creates a consortium named `SampleConsortium` that contains the two peer organizations in the `configtx.yaml` file, `Org1` and `Org2`. The `Orderer` section of the profile uses the single node Raft ordering service defined in the **Orderer:** section of the file. The `OrdererOrg` from the **Organizations:** section is made the only administrator of the ordering service. Because our only ordering node is running Fabric 2.x, we can set the orderer

system channel capability to V2_0. The system channel uses default policies from the **Channel** section and enables V2_0 as the channel capability level.

TwoOrgsChannel

The TwoOrgsChannel profile is used by the test network to create application channels:

```
TwoOrgsChannel:
  Consortium: SampleConsortium
  <<: *ChannelDefaults
  Application:
    <<: *ApplicationDefaults
    Organizations:
      - *Org1
      - *Org2
    Capabilities:
      <<: *ApplicationCapabilities
```

The system channel is used by the ordering service as a template to create application channels. The nodes of the ordering service that are defined in the system channel become the default consenter set of new channels, while the administrators of the ordering service become the orderer administrators of the channel. The channel MSPs of channel members are transferred to the new channel from the system channel. After the channel is created, ordering nodes can be added or removed from the channel by updating the channel configuration. You can also update the channel configuration to [add other organizations as channel members](#).

The TwoOrgsChannel provides the name of the consortium, SampleConsortium, hosted by the test network system channel. As a result, the ordering service defined in the TwoOrgsOrdererGenesis profile becomes channel consenter set. In the Application section, both organizations from the consortium, Org1 and Org2, are included as channel members. The channel uses V2_0 as the application capabilities, and uses the default policies from the **Application** section to govern how peer organizations will interact with the channel. The application channel also uses the default policies from the **Channel** section and enables V2_0 as the channel capability level.

SampleAppChannelEtcdRaft

The SampleAppChannelEtcdRaft profile is provided for customers that prefer to create a channel without a system channel by using the `osnadmin` CLI. The major difference is that a consortium definition is no longer required. Check out the [Create a channel](#) tutorial to learn more about how to use this profile.

```
SampleAppChannelEtcdRaft:
  <<: *ChannelDefaults
  Orderer:
    <<: *OrdererDefaults
    OrdererType: etcdraft
    Organizations:
      - <<: *SampleOrg
        Policies:
          <<: *SampleOrgPolicies
        Admins:
          Type: Signature
          Rule: "OR('SampleOrg.member')"
  Application:
    <<: *ApplicationDefaults
    Organizations:
      - <<: *SampleOrg
```

(continues on next page)

(continued from previous page)

```

Policies:
  <<: *SampleOrgPolicies
Admins:
  Type: Signature
  Rule: "OR('SampleOrg.member') "

```

For simplicity, this snippet assumes that the peers and orderers belong to the same organization `SampleOrg`. For a production deployment however, it is recommended that the peer and ordering nodes belong to separate organizations.

7.7.4 Channel policies

Channels are a private method of communication between organizations. As a result, most changes to the channel configuration need to be agreed to by other members of the channel. A channel would not be useful if an organization could join the channel and read the data on the ledger without getting the approval of other organizations. Any changes to the channel **structure** need to be approved by a set of organizations that can satisfy the channel policies.

Policies also govern the **processes** of how users interact with the channel, such as the set of organizations that need to approve a chaincode before it can be deployed to a channel or which actions need to be completed by channel administrators.

Channel policies are important enough that they need to be discussed in their own topic. Unlike other parts of the channel configuration, the policies that govern the channel are determined by how different sections of the `configtx.yaml` file work together. While channel policies can be configured for any use case with few constraints, this topic will focus on how to use the default policies provided by Hyperledger Fabric. If you use the default policies used by the Fabric test network or the [Fabric sample configuration](#), each channel you create will use a combination of signature policies, ImplicitMeta policies, and Access Control Lists to determine how organizations interact with the channel and agree to update the channel structure. You can learn more about the role of policies in Hyperledger Fabric by visiting the [Policies concept topic](#).

Signature policies

By default, each channel member defines a set of signature policies that references their organization. When a proposal is submitted to a peer, or a transaction is submitted to the ordering nodes, the nodes read the signatures attached to the transaction and evaluate them against the signature policies defined in the channel configuration. Every signature policy has rule that specifies the set of organizations and identities whose signatures can satisfy the policy. You can see the signature policies defined by `Org1` in the **Organizations** section of `configtx.yaml` below:

```

- &Org1

...

Policies:
  Readers:
    Type: Signature
    Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client') "
  Writers:
    Type: Signature
    Rule: "OR('Org1MSP.admin', 'Org1MSP.client') "
  Admins:
    Type: Signature
    Rule: "OR('Org1MSP.admin') "
  Endorsement:
    Type: Signature
    Rule: "OR('Org1MSP.peer') "

```

All the policies above can be satisfied by signatures from Org1. However, each policy lists a different set of roles from within the organization that are able to satisfy the policy. The `Admins` policy can only be satisfied by transactions submitted by an identity with an admin role, while only identities with a peer role can satisfy the `Endorsement` policy. A set of signatures attached to a single transaction can satisfy multiple signature policies. For example, if the endorsements attached to a transaction were provided by both Org1 and Org2, then this signature set would satisfy the `Endorsement` policy of Org1 and Org2.

ImplicitMeta Policies

If your channel uses the default policies, the signature policies for each organization are evaluated by ImplicitMeta policies at higher levels of the channel configuration. Instead of directly evaluating the signatures that are submitted to the channel, ImplicitMeta policies have rules specify a set of other policies in the channel configuration that can satisfy the policy. A transaction can satisfy an ImplicitMeta policy if it can satisfy the underlying set of signature policies that are referenced by the policy.

You can see the ImplicitMeta policies defined in the **Application** section of `configtx.yaml` file below:

```
Policies:
  Readers:
    Type: ImplicitMeta
    Rule: "ANY Readers"
  Writers:
    Type: ImplicitMeta
    Rule: "ANY Writers"
  Admins:
    Type: ImplicitMeta
    Rule: "MAJORITY Admins"
  LifecycleEndorsement:
    Type: ImplicitMeta
    Rule: "MAJORITY Endorsement"
  Endorsement:
    Type: ImplicitMeta
    Rule: "MAJORITY Endorsement"
```

The ImplicitMeta policies in the **Application** section govern how peer organizations interact with the channel. Each policy references the signature policies associated with each channel member. You see the relationship between the policies in the **Application** section and the policies in the **Organization** section below:

Application policies <i>Policy path: Channel/Application</i>	Peer Organizations
Admins Policy Type: Implicit Meta Rule: Majority Admins Path: Channel/Application/Admins	Org1 Signature Policies: Admins Readers Writers Endorsement
Writers Policy Type: Implicit Meta Rule: Any Writers Path: Channel/Application/Writers	Org2 Signature Policies: Admins Readers Writers Endorsement
Readers Policy Type: Implicit Meta Rule: Any Readers Path: Channel/Application/Readers	
LifecycleEndorsement Policy Type: Implicit Meta Rule: Any Readers Path: Channel/Application/LifecycleEndorsement	Org3 Signature Policies: Admins Readers Writers Endorsement
Endorsement Policy Type: Implicit Meta Rule: Any Endorsement Path: Channel/Application/Endorsement	

Figure 1: The Admins ImplicitMeta policy can be satisfied by a majority of the Admins signature policies that are defined by each organization.

Each policy is referred to its path in the channel configuration. Because the policies in the **Application** section are located in the application group, which is located inside the channel group, they are referred to as Channel/Application policies. Since most places in the Fabric documentation refer to policies by their path, we will refer to policies by their path for the rest of the tutorial.

The Rule in each ImplicitMeta references the name of the signature policies that can satisfy the policy. For example, the Channel/Application/Admins ImplicitMeta policy references the Admins signature policies for each organization. Each Rule also contains the number of signature policies that are required to satisfy the ImplicitMeta policy. For example, the Channel/Application/Admins policy requires that a majority of the Admins signature policies be satisfied.

Application policies <i>Policy path: Channel/Application</i>	Peer Organizations
Admins Policy Type: Implicit Meta Rule: Majority Admins Path: Channel/Application/Admins	Org1 Signature Policies: Admins Readers Writers Endorsement
Writers Policy Type: Implicit Meta Rule: Any Writers Path: Channel/Application/Writers	Org2 Signature Policies: Admins Readers Writers Endorsement
Readers Policy Type: Implicit Meta Rule: Any Readers Path: Channel/Application/Readers	
LifecycleEndorsement Policy Type: Implicit Meta Rule: Any Readers Path: Channel/Application/LifecycleEndorsement	Org3 Signature Policies: Admins Readers Writers Endorsement
Endorsement Policy Type: Implicit Meta Rule: Any Endorsement Path: Channel/Application/Endorsement	

Figure 2: A channel update request submitted to the channel contains signatures from Org1, Org2, and Org3, satisfying

the signature policies for each organization. As a result, the request satisfies the Channel/Application/Admins policy. The Org3 check is in light green because the signature was not required to reach to a majority.

To provide another example, the Channel/Application/Endorsement policy can be satisfied by a majority of organization Endorsement policies, which require signatures from the peers of each organization. This policy is used by the Fabric chaincode lifecycle as the default chaincode endorsement policy. Unless you commit a chaincode definition with a different endorsement policy, transactions that invoke a chaincode need to be endorsed by a majority of channel members.

Application policies <i>Policy path: Channel/Application</i>	Peer Organizations
Admins Policy Type: Implicit Meta Rule: Majority Admins Path: Channel/Application/Admins	Org1 Signature Policies: Admins Readers Writers Endorsement ✓
Writers Policy Type: Implicit Meta Rule: Any Writers Path: Channel/Application/Writers	
Readers Policy Type: Implicit Meta Rule: Any Readers Path: Channel/Application/Readers	Org2 Signature Policies: Admins Readers Writers Endorsement ✓
LifecycleEndorsement Policy Type: Implicit Meta Rule: Any Readers Path: Channel/Application/LifecycleEndorsement	Org3 Signature Policies: Admins Readers Writers Endorsement
Endorsement Policy Type: Implicit Meta Rule: Any Endorsement Path: Channel/Application/Endorsement ✓	

Figure 3: A transaction from a client application invoked a chaincode on the peers of Org1 and Org2. The chaincode invoke was successful, and the application received an endorsement from the peers of both organizations. Because this transaction satisfies the Channel/Application/Endorsement policy, the transaction meets the default endorsement policy and can be added to the channel ledger.

The advantage of using ImplicitMeta policies and signature policies together is that you can set the rules for governance at the channel level, while allowing each channel member to select the identities that are required to sign for their organization. For example, a channel can specify that a majority of organization admins are required to sign a channel configuration update. However, each organization can use their signature policies to select which identities from their organization are admins, or even require that multiple identities from their organization need to sign in order to approve a channel update.

Another advantage of ImplicitMeta policies is that they do not need to be updated when an organization is added or removed from the channel. Using Figure 3 as an example, if two new organizations are added to the channel, the Channel/Application/Endorsement would require an endorsement from three organizations in order to validate a transaction.

A disadvantage of ImplicitMeta policies is that they do not explicitly read the signature policies used by the channel members (which is why they are called implicit policies). Instead, they assume that users have the required signature policies based on the configuration of the channel. The rule of the Channel/Application/Endorsement policy is based on the number of peer organizations in the channel. If two of the three organizations in Figure 3 do not possess the Endorsement signature policies, no transaction would be able to get the majority required to meet the Channel/Application/Endorsement ImplicitMeta policy.

Channel modification policies

The channel **structure** is governed by modification policies within the channel configuration. Each component of the channel configuration has a modification policy that needs to be satisfied in order to be updated by channel members. For example, the policies and channel MSP defined by each organization, the application group that contains the members of the channel, and the components of the configuration that define the channel consenter set each have a different modification policy.

Each modification policy can reference an ImplicitMeta policy or a signature policy. For example, if you use the default policies, the values that define each organization reference the Admins signature policy associated with that organization. As a result, an organization can update their channel MSP or set an anchor peer without approval from other channel members. The modification policy of the application group that defines the set of channel members is the Channel/Application/Admins ImplicitMeta policy. As a result, the default policy is that a majority of organizations need to approve the addition or removal of a channel member.

Channel policies and Access Control Lists

The policies within the channel configuration are also referenced by **Access Control Lists (ACLs)** that are used to restrict access to Fabric resources used by the channel. The ACLs extend the policies within the channel configuration to govern the **processes** of the channel. You can see the default ACLs in the [sample configtx.yaml](#) file. Each ACL refers to a channel policy using the path. For example, the following ACL restricts who can invoke a chaincode based on the /Channel/Application/Writers policy:

```
# ACL policy for invoking chaincodes on peer
peer/Propose: /Channel/Application/Writers
```

Most of the default ACLs point to the ImplicitMeta policies in the application section of the channel configuration. To extend the example above, an organization can invoke a chaincode if they can satisfy the /Channel/Application/Writers policy.

Application policies <i>Policy path: Channel/Application</i>	Peer Organizations
Admins Policy Type: Implicit Meta Rule: Majority Admins Path: Channel/Application/Admins	Org1 Signature Policies: Admins Readers Writers ✓ Endorsement
Writers Policy Type: Implicit Meta Rule: Any Writers Path: Channel/Application/Writers ✓	Org2 Signature Policies: Admins Readers Writers Endorsement
Readers Policy Type: Implicit Meta Rule: Any Readers Path: Channel/Application/Readers	Org3 Signature Policies: Admins Readers Writers Endorsement
LifecycleEndorsement Policy Type: Implicit Meta Rule: Any Readers Path: Channel/Application/LifecycleEndorsement	
Endorsement Policy Type: Implicit Meta Rule: Any Endorsement Path: Channel/Application/Endorsement	

Figure 4: The peer/Propose ACL is satisfied by the /Channel/Application/Writers policy. This policy can be satisfied by a transaction submitted by a client application from any organization with the writers signature policy.

Orderer policies

The ImplicitMeta policies in the **Orderer** section of `configtx.yaml` govern the ordering nodes of a channel in a similar way as the **Application** section governs the peer organizations. The ImplicitMeta policies point to the signature policies associated with the organizations that are ordering service administrators.

Orderer policies <i>Policy path: Channel/Orderer</i>	Ordering service Organizations
Admins Policy Type: Implicit Meta Rule: Majority Admins <i>Path: Channel/Orderer/Admins</i>	Org1 Signature Policies: Admins Readers Writers
Writers Policy Type: Implicit Meta Rule: Any Writers <i>Path: Channel/Orderer/Writers</i>	Org2 Signature Policies: Admins Readers Writers
Readers Policy Type: Implicit Meta Rule: Any Readers <i>Path: Channel/Orderer/Readers</i>	Org3 Signature Policies: Admins Readers Writers
BlockValidation Policy Type: Implicit Meta Rule: Any Writers <i>Path: Channel/Orderer/BlockValidation</i>	Org3 Signature Policies: Admins Readers Writers

Figure 5: The `Channel/Orderer/Admins` policy points to the Admins signature policies associated with the administrators of the ordering service.

If you use the default policies, a majority of orderer organizations are required to approve the addition or removal of an ordering node.

Orderer policies <i>Policy path: Channel/Orderer</i>	Ordering service Organizations
Admins Policy Type: Implicit Meta Rule: Majority Admins <i>Path: Channel/Orderer/Admins</i>	Org1 Signature Policies: Admins Readers Writers
Writers Policy Type: Implicit Meta Rule: Any Writers <i>Path: Channel/Orderer/Writers</i>	Org2 Signature Policies: Admins Readers Writers
Readers Policy Type: Implicit Meta Rule: Any Readers <i>Path: Channel/Orderer/Readers</i>	Org3 Signature Policies: Admins Readers Writers
BlockValidation Policy Type: Implicit Meta Rule: Any Writers <i>Path: Channel/Orderer/BlockValidation</i>	Org3 Signature Policies: Admins Readers Writers

Figure 6: A request submitted to remove an ordering node from the channel contains signatures from the three ordering organizations in the network, satisfying the `Channel/Orderer/Admins` policy. The Org3 check is in light green because the signature was not required to reach to a majority.

The `Channel/Orderer/BlockValidation` policy is used by peers to confirm that new blocks being added to the channel were generated by an ordering node that is part of the channel consenter set, and that the block was not tampered with or created by another peer organization. By default, any orderer organization with a `Writers` signature policy can create and validate blocks for the channel.

7.8 Adding an Org to a Channel

Note: Ensure that you have downloaded the appropriate images and binaries as outlined in [Install Samples, Binaries, and Docker Images](#) and [Prerequisites](#) that conform to the version of this documentation (which can be found at the bottom of the table of contents to the left).

This tutorial extends the Fabric test network by adding a new organization – Org3 – to an application channel.

While we will focus on adding a new organization to the channel, you can use a similar process to make other channel configuration updates (updating modification policies or altering batch size, for example). To learn more about the process and possibilities of channel config updates in general, check out [Updating a channel configuration](#). It's also worth noting that channel configuration updates like the one demonstrated here will usually be the responsibility of an organization admin (rather than a chaincode or application developer).

7.8.1 Setup the Environment

We will be operating from the root of the `test-network` subdirectory within your local clone of `fabric-samples`. Change into that directory now.

```
cd fabric-samples/test-network
```

First, use the `network.sh` script to tidy up. This command will kill any active or stale Docker containers and remove previously generated artifacts. It is by no means **necessary** to bring down a Fabric network in order to perform channel configuration update tasks. However, for the sake of this tutorial, we want to operate from a known initial state. Therefore let's run the following command to clean up any previous environments:

```
./network.sh down
```

You can now use the script to bring up the test network with one channel named `channel1`:

```
./network.sh up createChannel -c channel1
```

If the command was successful, you can see the following message printed in your logs:

```
Channel 'channel1' joined
```

Now that you have a clean version of the test network running on your machine, we can start the process of adding a new org to the channel we created. First, we are going use a script to add Org3 to the channel to confirm that the process works. Then, we will go through the step by step process of adding Org3 by updating the channel configuration.

7.8.2 Bring Org3 into the Channel with the Script

You should be in the `test-network` directory. To use the script, simply issue the following commands:

```
cd addOrg3
./addOrg3.sh up -c channel1
```

The output here is well worth reading. You'll see the Org3 crypto material being generated, the Org3 organization definition being created, and then the channel configuration being updated, signed, and then submitted to the channel.

If everything goes well, you'll get this message:

```
Org3 peer successfully added to network
```

Now that we have confirmed we can add Org3 to our channel, we can go through the steps to update the channel configuration that the script completed behind the scenes.

7.8.3 Bring Org3 into the Channel Manually

If you just used the `addOrg3.sh` script, you'll need to bring your network down. The following command will bring down all running components and remove the crypto material for all organizations:

```
cd ..  
./network.sh down
```

After the network is brought down, bring it back up again:

```
./network.sh up createChannel -c channel1
```

This will bring your network back to the same state it was in before you executed the `addOrg3.sh` script.

Now we're ready to add Org3 to the channel manually. As a first step, we'll need to generate Org3's crypto material.

7.8.4 Generate the Org3 Crypto Material

In another terminal, change into the `addOrg3` subdirectory from `test-network`.

```
cd addOrg3
```

First, we are going to create the certificates and keys for the Org3 peer, along with an application and admin user. Because we are updating an example channel, we are going to use the `cryptogen` tool instead of using a Certificate Authority. The following command uses `cryptogen` to read the `org3-crypto.yaml` file and generate the Org3 crypto material in a new `org3.example.com` folder:

```
../../bin/cryptogen generate --config=org3-crypto.yaml --output="../../organizations"
```

You can find the generated Org3 crypto material alongside the certificates and keys for Org1 and Org2 in the `test-network/organizations/peerOrganizations` directory.

Once we have created the Org3 crypto material, we can use the `configtxgen` tool to print out the Org3 organization definition. We will preface the command by telling the tool to look in the current directory for the `configtx.yaml` file that it needs to ingest.

```
export FABRIC_CFG_PATH=$PWD  
../../bin/configtxgen -printOrg Org3MSP > ../../organizations/peerOrganizations/org3.  
example.com/org3.json
```

The above command creates a JSON file – `org3.json` – and writes it to the `test-network/organizations/peerOrganizations/org3.example.com` folder. The organization definition contains the policy definitions for Org3, the NodeOU definitions for Org3, and two important certificates encoded in base64 format:

- a CA root cert, used to establish the organizations root of trust
- a TLS root cert, used by the gossip protocol to identify Org3 for block dissemination and service discovery

We will add Org3 to the channel by appending this organization definition to the channel configuration.

7.8.5 Bring up Org3 components

After we have created the Org3 certificate material, we can now bring up the Org3 peer. From the `addOrg3` directory, issue the following command:

```
docker-compose -f docker/docker-compose-org3.yaml up -d
```

If the command is successful, you will see the creation of the Org3 peer:

```
Creating peer0.org3.example.com ... done
```

This Docker Compose file has been configured to bridge across our initial network, so that the Org3 peer resolves with the existing peers and ordering node of the test network.

Note: the `./addOrg3.sh up` command uses a *fabric-tools* CLI container to perform the channel configuration update process demonstrated below. This is to avoid the *jq* dependency requirement for first-time users. However, it is recommended to follow the process below directly on your local machine instead of using the unnecessary CLI container.

7.8.6 Fetch the Configuration

Let's go fetch the most recent config block for the channel – `channel1`.

The reason why we have to pull the latest version of the config is because channel config elements are versioned. Versioning is important for several reasons. It prevents config changes from being repeated or replayed (for instance, reverting to a channel config with old CRLs would represent a security risk). Also it helps ensure concurrency (if you want to remove an Org from your channel, for example, after a new Org has been added, versioning will help prevent you from removing both Orgs, instead of just the Org you want to remove).

Navigate back to the `test-network` directory.

Because Org3 is not yet a member of the channel, we need to operate as the admin of another organization to fetch the channel config. Because Org1 is a member of the channel, the Org1 admin has permission to fetch the channel config from the ordering service. Issue the following commands to operate as the Org1 admin.

```
# you can issue all of these commands at once

export PATH=${PWD}/../bin:$PATH
export FABRIC_CFG_PATH=${PWD}/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

We can now issue the command to fetch the latest config block:

```
peer channel fetch config channel-artifacts/config_block.pb -o localhost:7050 --
ordererTLSHostnameOverride orderer.example.com -c channel1 --tls --cafile ${PWD}/
organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
tlscacerts/tlsca.example.com-cert.pem
```


This command saves the binary protobuf channel configuration block to `config_block.pb`. Note that the choice of name and file extension is arbitrary. However, following a convention which identifies both the type of object being represented and its encoding (protobuf or JSON) is recommended.

When you issued the `peer channel fetch` command, the following output is displayed in your logs:

```
2021-01-07 18:46:33.687 UTC [cli.common] readBlock -> INFO 004 Received block: 2
```

This is telling us that the most recent configuration block for `channel1` is actually block 2, **NOT** the genesis block. By default, the `peer channel fetch config` command returns the most **recent** configuration block for the targeted channel, which in this case is the third block. This is because the test network script, `network.sh`, defined anchor peers for our two organizations – `Org1` and `Org2` – in two separate channel update transactions. As a result, we have the following configuration sequence:

- block 0: genesis block
- block 1: `Org1` anchor peer update
- block 2: `Org2` anchor peer update

7.8.7 Convert the Configuration to JSON and Trim It Down

The channel configuration block was stored in the `channel-artifacts` folder to keep the update process separate from other artifacts. Change into the `channel-artifacts` folder to complete the next steps:

Now we will make use of the `configtxlator` tool to decode this channel configuration block into JSON format (which can be read and modified by humans). We also must strip away all of the headers, metadata, creator signatures, and so on that are irrelevant to the change we want to make. We accomplish this by means of the `jq` tool (you will need to install the [jq tool](#) on your local machine):

```
configtxlator proto_decode --input config_block.pb --type common.Block --output _  
↪ config_block.json  
jq .data.data[0].payload.data.config config_block.json > config.json
```

This command leaves us with a trimmed down JSON object – `config.json` – which will serve as the baseline for our config update.

Take a moment to open this file inside your text editor of choice (or in your browser). Even after you’re done with this tutorial, it will be worth studying it as it reveals the underlying configuration structure and the other kind of channel updates that can be made. We discuss them in more detail in [Updating a channel configuration](#).

7.8.8 Add the Org3 Crypto Material

Note: The steps you’ve taken up to this point will be nearly identical no matter what kind of config update you’re trying to make. We’ve chosen to add an org with this tutorial because it’s one of the most complex channel configuration updates you can attempt.

We’ll use the `jq` tool once more to append the `Org3` configuration definition – `org3.json` – to the channel’s application groups field, and name the output – `modified_config.json`.

```
jq -s '.[0] * {"channel_group":{"groups":{"Application":{"groups": {"Org3MSP":.[1]}}}}'  
↪ ' config.json ../organizations/peerOrganizations/org3.example.com/org3.json >_  
↪ modified_config.json
```


Now we have two JSON files of interest – `config.json` and `modified_config.json`. The initial file contains only Org1 and Org2 material, whereas the “modified” file contains all three Orgs. At this point it’s simply a matter of re-encoding these two JSON files and calculating the delta.

First, translate `config.json` back into a protobuf called `config.pb`:

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
```

Next, encode `modified_config.json` to `modified_config.pb`:

```
configtxlator proto_encode --input modified_config.json --type common.Config --output_
↪modified_config.pb
```

Now use `configtxlator` to calculate the delta between these two config protobufs. This command will output a new protobuf binary named `org3_update.pb`:

```
configtxlator compute_update --channel_id channel1 --original config.pb --updated_
↪modified_config.pb --output org3_update.pb
```

This new proto – `org3_update.pb` – contains the Org3 definitions and high level pointers to the Org1 and Org2 material. We are able to forgo the extensive MSP material and modification policy information for Org1 and Org2 because this data is already present within the channel’s genesis block. As such, we only need the delta between the two configurations.

Before submitting the channel update, we need to perform a few final steps. First, let’s decode this object into editable JSON format and call it `org3_update.json`:

```
configtxlator proto_decode --input org3_update.pb --type common.ConfigUpdate --output_
↪org3_update.json
```

Now, we have a decoded update file – `org3_update.json` – that we need to wrap in an envelope message. This step will give us back the header field that we stripped away earlier. We’ll name this file `org3_update_in_envelope.json`:

```
echo '{"payload":{"header":{"channel_header":{"channel_id":"'channel1'", "type":2}},
↪"data":{"config_update":"'$(cat org3_update.json)'"}}}' | jq . > org3_update_in_
↪envelope.json
```

Using our properly formed JSON – `org3_update_in_envelope.json` – we will leverage the `configtxlator` tool one last time and convert it into the fully fledged protobuf format that Fabric requires. We’ll name our final update object `org3_update_in_envelope.pb`:

```
configtxlator proto_encode --input org3_update_in_envelope.json --type common.
↪Envelope --output org3_update_in_envelope.pb
```

7.8.9 Sign and Submit the Config Update

Almost done!

We now have a protobuf binary – `org3_update_in_envelope.pb`. However, we need signatures from the requisite Admin users before the config can be written to the ledger. The modification policy (`mod_policy`) for our channel Application group is set to the default of “MAJORITY”, which means that we need a majority of existing org admins to sign it. Because we have only two orgs – Org1 and Org2 – and the majority of two is two, we need both of them to sign. Without both signatures, the ordering service will reject the transaction for failing to fulfill the policy.

First, let’s sign this update proto as Org1. Navigate back to the `test-network` directory:

Remember that we exported the necessary environment variables to operate as the Org1 admin. As a result, the following `peer channel signconfigtx` command will sign the update as Org1.

```
peer channel signconfigtx -f channel-artifacts/org3_update_in_envelope.pb
```

The final step is to switch the container's identity to reflect the Org2 Admin user. We do this by exporting four environment variables specific to the Org2 MSP.

Note: Switching between organizations to sign a config transaction (or to do anything else) is not reflective of a real-world Fabric operation. A single container would never be mounted with an entire network's crypto material. Rather, the config update would need to be securely passed out-of-band to an Org2 Admin for inspection and approval.

Export the Org2 environment variables:

```
# you can issue all of these commands at once

export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

Lastly, we will issue the `peer channel update` command. The Org2 Admin signature will be attached to this call so there is no need to manually sign the protobuf a second time:

Note: The upcoming update call to the ordering service will undergo a series of systematic signature and policy checks. As such you may find it useful to stream and inspect the ordering node's logs. You can issue a `docker logs -f orderer.example.com` command to display them.

Send the update call:

```
peer channel update -f channel-artifacts/org3_update_in_envelope.pb -c channel1 -o_
↪localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile $
↪{PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/
↪msp/tlscacerts/tlsca.example.com-cert.pem
```

You should see a message similar to the following if your update has been submitted successfully:

```
2021-01-07 18:51:48.015 UTC [channelCmd] update -> INFO 002 Successfully submitted_
↪channel update
```

The successful channel update call returns a new block – block 3 – to all of the peers on the channel. If you remember, blocks 0-2 are the initial channel configurations. Block 3 serves as the most recent channel configuration with Org3 now defined on the channel.

You can inspect the logs for `peer0.org1.example.com` by issuing the following command:

```
docker logs -f peer0.org1.example.com
```

7.8.10 Join Org3 to the Channel

At this point, the channel configuration has been updated to include our new organization – Org3 – meaning that peers attached to it can now join `channel1`.

Export the following environment variables to operate as the Org3 Admin:

```
# you can issue all of these commands at once

export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org3MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org3.
↪example.com/peers/peer0.org3.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org3.example.
↪com/users/Admin@org3.example.com/msp
export CORE_PEER_ADDRESS=localhost:11051
```

Org3 peers can join `channel1` by either the genesis block or a snapshot that is created after Org3 has joined the channel.

To join by the genesis block, send a call to the ordering service asking for the genesis block of `channel1`. As a result of the successful channel update, the ordering service will verify that Org3 can pull the genesis block and join the channel. If Org3 had not been successfully appended to the channel config, the ordering service would reject this request.

Note: Again, you may find it useful to stream the ordering node’s logs to reveal the sign/verify logic and policy checks.

Use the `peer channel fetch` command to retrieve this block:

```
peer channel fetch 0 channel-artifacts/channel1.block -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com -c channel1 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

Notice, that we are passing a 0 to indicate that we want the first block on the channel’s ledger; the genesis block. If we simply passed the `peer channel fetch config` command, then we would have received block 3 – the updated config with Org3 defined. However, we can’t begin our ledger with a downstream block – we must start with block 0.

If successful, the command returned the genesis block to a file named `channel1.block`. We can now use this block to join the peer to the channel. Issue the `peer channel join` command and pass in the genesis block to join the Org3 peer to the channel:

```
peer channel join -b channel-artifacts/channel1.block
```

To join by a snapshot, follow the instruction in [Taking a snapshot](#) to take a snapshot on an existing peer. The snapshot should be taken after Org3 has been added to `channel1` to ensure that the snapshot contains the updated channel configuration including Org3. Locate the snapshot directory, copy it to the filesystem of the new Org3 peer, and issue the `peer channel joinbysnapshot` command using the path to the snapshot on your file system.

```
peer channel joinbysnapshot --snapshotpath <path to snapshot>
```

7.8.11 Configuring Leader Election

Note: This section is included as a general reference for understanding the leader election settings when adding organizations to a network after the initial channel configuration has completed.

Newly joining peers are bootstrapped with the genesis block, which does not contain information about the organization that is being added in the channel configuration update. Therefore new peers are not able to utilize gossip as they cannot verify blocks forwarded by other peers from their own organization until they get the configuration transaction which added the organization to the channel. Newly added peers must therefore have one of the following configurations so that they receive blocks from the ordering service:

1. To ensure that peers always receive blocks directly from the ordering service, configure the peer to be an organization leader:

```
CORE_PEER_GOSSIP_USELEADERELECTION=false
CORE_PEER_GOSSIP_ORGLEADER=true
```

Note: This configuration is the default starting in Fabric v2.2 and must be the same for all new peers added to the channel.

2. To eventually utilize dynamic leader election within the organization, configure the peer to use leader election:

```
CORE_PEER_GOSSIP_USELEADERELECTION=true
CORE_PEER_GOSSIP_ORGLEADER=false
```

Note: Because peers of the newly added organization won't initially be able to form membership view, this option will be similar to the static configuration, as each peer will start proclaiming itself to be a leader. However, once they get updated with the configuration transaction that adds the organization to the channel, there will be only one active leader for the organization. Therefore, it is recommended to leverage this option if you eventually want the organization's peers to utilize leader election.

7.8.12 Install, define, and invoke chaincode

We can confirm that Org3 is a member of `channel1` by installing and invoking a chaincode on the channel. If the existing channel members have already committed a chaincode definition to the channel, a new organization can start using the chaincode by approving the chaincode definition.

Note: These instructions use the Fabric chaincode lifecycle introduced in the v2.0 release. If you would like to use the previous lifecycle to install and instantiate a chaincode, visit the v1.4 version of the [Adding an org to a channel tutorial](#).

Before we install a chaincode as Org3, we can use the `./network.sh` script to deploy the Basic chaincode on the channel. Open a new terminal and navigate to the `test-network` directory. You can then use the `test-network` script to deploy the Basic chaincode:

```
cd fabric-samples/test-network
./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-go/ -ccl go -
↪c channel1
```

The script will install the Basic chaincode on the Org1 and Org2 peers, approve the chaincode definition for Org1 and Org2, and then commit the chaincode definition to the channel. Once the chaincode definition has been committed

to the channel, the Basic chaincode is initialized and invoked to put initial data on the ledger. The commands below assume that we are still using the channel `channel1`.

After the chaincode has been deployed we can use the following steps to use invoke Basic chaincode as Org3. Copy and paste the following environment variables in your terminal in order to interact with the network as the Org3 admin:

```
export PATH=${PWD}/../bin:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org3MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org3.
↳example.com/peers/peer0.org3.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org3.example.
↳com/users/Admin@org3.example.com/msp
export CORE_PEER_ADDRESS=localhost:11051
```

The first step is to package the Basic chaincode:

```
peer lifecycle chaincode package basic.tar.gz --path ../asset-transfer-basic/
↳chaincode-go/ --lang golang --label basic_1
```

This command will create a chaincode package named `basic.tar.gz`, which we can install on the Org3 peer. Modify the command accordingly if the channel is running a chaincode written in Java or Node.js. Issue the following command to install the chaincode package `peer0.org3.example.com`:

```
peer lifecycle chaincode install basic.tar.gz
```

The next step is to approve the chaincode definition of Basic as Org3. Org3 needs to approve the same definition that Org1 and Org2 approved and committed to the channel. In order to invoke the chaincode, Org3 needs to include the package identifier in the chaincode definition. You can find the package identifier by querying your peer:

```
peer lifecycle chaincode queryinstalled
```

You should see output similar to the following:

```
Get installed chaincodes on peer:
Package ID: basic_1:5443b5b557efd3faece8723883d28d6f7026c0bf12245de109b89c5c4fe64887, _
↳Label: basic_1
```

We are going to need the package ID in a future command, so let's go ahead and save it as an environment variable. Paste the package ID returned by the `peer lifecycle chaincode queryinstalled` command into the command below. The package ID may not be the same for all users, so you need to complete this step using the package ID returned from your console.

```
export CC_PACKAGE_ID=basic_
↳1:5443b5b557efd3faece8723883d28d6f7026c0bf12245de109b89c5c4fe64887
```

Use the following command to approve a definition of the basic chaincode for Org3:

```
# use the --package-id flag to provide the package identifier
# use the --init-required flag to request the ``Init`` function be invoked to _
↳initialize the chaincode
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↳ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/
↳ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↳example.com-cert.pem --channelID channel1 --name basic --version 1.0 --package-id
↳$CC_PACKAGE_ID --sequence 1
```

You can use the `peer lifecycle chaincode querycommitted` command to check if the chaincode definition you have approved has already been committed to the channel.

```
# use the --name flag to select the chaincode whose definition you want to query
peer lifecycle chaincode querycommitted --channelID channel1 --name basic --cafile $
↪{PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/
↪msp/tlscacerts/tlsca.example.com-cert.pem
```

A successful command will return information about the committed definition:

```
Committed chaincode definition for chaincode 'basic' on channel 'channel1':
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc,
↪Approvals: [Org1MSP: true, Org2MSP: true, Org3MSP: true]
```

Org3 can use the basic chaincode after it approves the chaincode definition that was committed to the channel. The chaincode definition uses the default endorsement policy, which requires a majority of organizations on the channel endorse a transaction. This implies that if an organization is added to or removed from the channel, the endorsement policy will be updated automatically. We previously needed endorsements from Org1 and Org2 (2 out of 2). Now we need endorsements from two organizations out of Org1, Org2, and Org3 (2 out of 3).

Populate the ledger with some sample assets. We'll get endorsements from the Org2 peer and the new Org3 peer so that the endorsement policy is satisfied.

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C channel1 -n basic -
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt --
↪peerAddresses localhost:11051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org3.example.com/peers/peer0.org3.example.com/tls/ca.crt -c '{
↪"function": "InitLedger", "Args": []}'
```

You can query the chaincode to ensure that the Org3 peer committed the data.

```
peer chaincode query -C channel1 -n basic -c '{"Args":["GetAllAssets"]}'
```

You should see the initial list of assets that were added to the ledger as a response.

7.8.13 Conclusion

The channel configuration update process is indeed quite involved, but there is a logical method to the various steps. The endgame is to form a delta transaction object represented in protobuf binary format and then acquire the requisite number of admin signatures such that the channel configuration update transaction fulfills the channel's modification policy.

The `configtxlator` and `jq` tools, along with the `peer channel` commands, provide us with the functionality to accomplish this task.

7.8.14 Updating the Channel Config to include an Org3 Anchor Peer (Optional)

The Org3 peers were able to establish gossip connection to the Org1 and Org2 peers since Org1 and Org2 had anchor peers defined in the channel configuration. Likewise newly added organizations like Org3 should also define their anchor peers in the channel configuration so that any new peers from other organizations can directly discover an Org3 peer. In this section, we will make a channel configuration update to define an Org3 anchor peer. The process will be similar to the previous configuration update, therefore we'll go faster this time.

As before, we will fetch the latest channel configuration to get started. Fetch the most recent config block for the channel, using the `peer channel fetch` command.

```
peer channel fetch config channel-artifacts/config_block.pb -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com -c channel1 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

After fetching the config block we will want to convert it into JSON format. To do this we will use the `configtxlator` tool, as done previously when adding Org3 to the channel. First, change into the `channel-artifacts` folder:

When converting it we need to remove all the headers, metadata, and signatures that are not required to update Org3 to include an anchor peer by using the `jq` tool. This information will be reincorporated later before we proceed to update the channel configuration.

```
configtxlator proto_decode --input config_block.pb --type common.Block --output
↪config_block.json
jq .data.data[0].payload.data.config config_block.json > config.json
```

The `config.json` is the now trimmed JSON representing the latest channel configuration that we will update.

Using the `jq` tool again, we will update the configuration JSON with the Org3 anchor peer we want to add.

```
jq '.channel_group.groups.Application.groups.Org3MSP.values += {"AnchorPeers":{"mod_
↪policy": "Admins","value":{"anchor_peers": [{"host": "peer0.org3.example.com","port
↪": 11051}]}},"version": "0"}}' config.json > modified_anchor_config.json
```

We now have two JSON files, one for the current channel configuration, `config.json`, and one for the desired channel configuration `modified_anchor_config.json`. Next we convert each of these back into protobuf format and calculate the delta between the two.

Translate `config.json` back into protobuf format as `config.pb`

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
```

Translate the `modified_anchor_config.json` into protobuf format as `modified_anchor_config.pb`

```
configtxlator proto_encode --input modified_anchor_config.json --type common.Config --
↪output modified_anchor_config.pb
```

Calculate the delta between the two protobuf formatted configurations.

```
configtxlator compute_update --channel_id channel1 --original config.pb --updated
↪modified_anchor_config.pb --output anchor_update.pb
```

Now that we have the desired update to the channel we must wrap it in an envelope message so that it can be properly read. To do this we must first convert the protobuf back into a JSON that can be wrapped.

We will use the `configtxlator` command again to convert `anchor_update.pb` into `anchor_update.json`

```
configtxlator proto_decode --input anchor_update.pb --type common.ConfigUpdate --
↪output anchor_update.json
```

Next we will wrap the update in an envelope message, restoring the previously stripped away header, outputting it to `anchor_update_in_envelope.json`

```
echo '{"payload":{"header":{"channel_header":{"channel_id":"channel1", "type":2}},
↪"data":{"config_update":"'$(cat anchor_update.json)'"}}}' | jq . > anchor_update_in_
↪envelope.json
```

(continues on next page)

(continued from previous page)

Now that we have reincorporated the envelope we need to convert it to a protobuf so it can be properly signed and submitted to the orderer for the update.

```
configtxlator proto_encode --input anchor_update_in_envelope.json --type common.  
↪Envelope --output anchor_update_in_envelope.pb
```

Now that the update has been properly formatted it is time to sign off and submit it.

Navigate back to the `test-network` directory:

Since this is only an update to Org3 we only need to have Org3 sign off on the update. Run the following commands to make sure that we are operating as the Org3 admin:

```
# you can issue all of these commands at once  
  
export CORE_PEER_LOCALMSPID="Org3MSP"  
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org3.  
↪example.com/peers/peer0.org3.example.com/tls/ca.crt  
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org3.example.  
↪com/users/Admin@org3.example.com/msp  
export CORE_PEER_ADDRESS=localhost:11051
```

We can now just use the `peer channel update` command to sign the update as the Org3 admin before submitting it to the orderer.

```
peer channel update -f channel-artifacts/anchor_update_in_envelope.pb -c channel1 -o_↪  
↪localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile $  
↪{PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/  
↪msp/tlscacerts/tlsca.example.com-cert.pem
```

The orderer receives the config update request and cuts a block with the updated configuration. As peers receive the block, they will process the configuration updates.

Inspect the logs for one of the peers. While processing the configuration transaction from the new block, you will see gossip re-establish connections using the new anchor peer for Org3. This is proof that the configuration update has been successfully applied!

```
docker logs -f peer0.org1.example.com
```

```
2021-01-07 19:07:01.244 UTC [gossip.gossip] learnAnchorPeers -> INFO 05a Learning_↪  
↪about the configured anchor peers of Org1MSP for channel channel1: [{peer0.org1.  
↪example.com 7051}]  
2021-01-07 19:07:01.243 UTC [gossip.gossip] learnAnchorPeers -> INFO 05b Learning_↪  
↪about the configured anchor peers of Org2MSP for channel channel1: [{peer0.org2.  
↪example.com 9051}]  
2021-01-07 19:07:01.244 UTC [gossip.gossip] learnAnchorPeers -> INFO 05c Learning_↪  
↪about the configured anchor peers of Org3MSP for channel channel1: [{peer0.org3.  
↪example.com 11051}]
```

Congratulations, you have now made two configuration updates — one to add Org3 to the channel, and a second to define an anchor peer for Org3.

7.9 Updating a channel configuration

Audience: network administrators, node administrators

Note: this topic describes the process for updating a channel on a network that does not have a system channel. For a version of this topic that includes information about the system channel, check out [Updating a channel configuration](#).

7.9.1 What is a channel configuration?

Like many complex systems, Hyperledger Fabric networks are comprised of both **structure** and a number related of **processes**.

- **Structure:** encompassing users (like admins), organizations, peers, ordering nodes, CAs, smart contracts, and applications.
- **Process:** the way these structures interact. Most important of these are [Policies](#), the rules that govern which users can do what, and under what conditions.

Information identifying the structure of blockchain networks and the processes governing how structures interact are contained in **channel configurations**. These configurations are collectively decided upon by the members of channels and are contained in blocks that are committed to the ledger of a channel. Channel configurations can be built using a tool called `configtxgen`, which uses a `configtx.yaml` file as its input. You can look at a [sample configtx.yaml file here](#).

Because configurations are contained in blocks (the first of these is known as the genesis block with the latest representing the current configuration of the channel), the process for updating a channel configuration (changing the structure by adding members, for example, or processes by modifying channel policies) is known as a **configuration update transaction**.

In production networks, these configuration update transactions will normally be proposed by a single channel admin after an out of band discussion, just as the initial configuration of the channel will be decided on out of band by the initial members of the channel.

In this topic, we'll:

- Show a full sample configuration of an application channel.
- Discuss many of the channel parameters that can be edited.
- Show the process for updating a channel configuration, including the commands necessary to pull, translate, and scope a configuration into something that humans can read.
- Discuss the methods that can be used to edit a channel configuration.
- Show the process used to reformat a configuration and get the signatures necessary for it to be approved.

7.9.2 Channel parameters that can be updated

Channels are highly configurable, but not infinitely so. Once certain things about a channel (for example, the name of the channel) have been specified, they cannot be changed. And changing one of the parameters we'll talk about in this topic requires satisfying the relevant policy as specified in the channel configuration.

In this section, we'll look a sample channel configuration and show the configuration parameters that can be updated.

Sample channel configuration

To see what the configuration file of an application channel looks like after it has been pulled and scoped, click **Click here to see the config** below. For ease of readability, it might be helpful to put this config into a viewer that supports JSON folding, like atom or Visual Studio.

Click here to see the config

```
{
  "channel_group": {
    "groups": {
      "Application": {
        "groups": {
          "Org1MSP": {
            "groups": {},
            "mod_policy": "Admins",
            "policies": {
              "Admins": {
                "mod_policy": "Admins",
                "policy": {
                  "type": 1,
                  "value": {
                    "identities": [
                      {
                        "principal": {
                          "msp_identifier": "Org1MSP",
                          "role": "ADMIN"
                        },
                        "principal_classification": "ROLE"
                      }
                    ],
                    "rule": {
                      "n_out_of": {
                        "n": 1,
                        "rules": [
                          {
                            "signed_by": 0
                          }
                        ]
                      }
                    }
                  }
                },
                "version": 0
              }
            },
            "version": "0"
          },
          "Endorsement": {
            "mod_policy": "Admins",
            "policy": {
              "type": 1,
              "value": {
                "identities": [
                  {
                    "principal": {
                      "msp_identifier": "Org1MSP",
                      "role": "PEER"
                    },
                    "principal_classification": "ROLE"
                  }
                ]
              }
            }
          }
        ]
      }
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
  ],
  "rule": {
    "n_out_of": {
      "n": 1,
      "rules": [
        {
          "signed_by": 0
        }
      ]
    }
  },
  "version": 0
},
"version": "0"
},
"Readers": {
  "mod_policy": "Admins",
  "policy": {
    "type": 1,
    "value": {
      "identities": [
        {
          "principal": {
            "msp_identifier": "Org1MSP",
            "role": "ADMIN"
          },
          "principal_classification": "ROLE"
        },
        {
          "principal": {
            "msp_identifier": "Org1MSP",
            "role": "PEER"
          },
          "principal_classification": "ROLE"
        },
        {
          "principal": {
            "msp_identifier": "Org1MSP",
            "role": "CLIENT"
          },
          "principal_classification": "ROLE"
        }
      ]
    }
  },
  "rule": {
    "n_out_of": {
      "n": 1,
      "rules": [
        {
          "signed_by": 0
        },
        {
          "signed_by": 1
        },
        {
          "signed_by": 2
        }
      ]
    }
  }
}

```

(continues on next page)

(continued from previous page)

```

        }
      ]
    }
  },
  "version": 0
}
},
"version": "0"
},
"Writers": {
  "mod_policy": "Admins",
  "policy": {
    "type": 1,
    "value": {
      "identities": [
        {
          "principal": {
            "msp_identifier": "Org1MSP",
            "role": "ADMIN"
          },
          "principal_classification": "ROLE"
        },
        {
          "principal": {
            "msp_identifier": "Org1MSP",
            "role": "CLIENT"
          },
          "principal_classification": "ROLE"
        }
      ],
      "rule": {
        "n_out_of": {
          "n": 1,
          "rules": [
            {
              "signed_by": 0
            },
            {
              "signed_by": 1
            }
          ]
        }
      }
    }
  },
  "version": 0
}
},
"version": "0"
}
},
"values": {
  "AnchorPeers": {
    "mod_policy": "Admins",
    "value": {
      "anchor_peers": [
        {
          "host": "peer0.org1.example.com",
          "port": 7051

```

(continues on next page)

(continued from previous page)

```

    }
  ],
  "version": "0"
},
"MSP": {
  "mod_policy": "Admins",
  "value": {
    "config": {
      "admins": [],
      "crypto_config": {
        "identity_identifier_hash_function": "SHA256",
        "signature_hash_family": "SHA2"
      },
      "fabric_node_ous": {
        "admin_ou_identifier": {
          "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ01VYWVSeWNkQyt1R1lUTUNyWTg2UFVXUEdzQ
↪ ",
          "organizational_unit_identifier": "admin"
        },
        "client_ou_identifier": {
          "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ01VYWVSeWNkQyt1R1lUTUNyWTg2UFVXUEdzQ
↪ ",
          "organizational_unit_identifier": "client"
        },
        "enable": true,
        "orderer_ou_identifier": {
          "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ01VYWVSeWNkQyt1R1lUTUNyWTg2UFVXUEdzQ
↪ ",
          "organizational_unit_identifier": "orderer"
        },
        "peer_ou_identifier": {
          "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ01VYWVSeWNkQyt1R1lUTUNyWTg2UFVXUEdzQ
↪ ",
          "organizational_unit_identifier": "peer"
        }
      },
      "intermediate_certs": [],
      "name": "Org1MSP",
      "organizational_unit_identifiers": [],
      "revocation_list": [],
      "root_certs": [
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ01VYWVSeWNkQyt1R1lUTUNyWTg2UFVXUEdzQ
↪ "
    ],
    "signing_identity": null,
    "tls_intermediate_certs": [],
    "tls_root_certs": [
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ01VYWVSeWNkQyt1R1lUTUNyWTg2UFVXUEdzQ
↪ "
  ]
}

```

(continues on next page)

(continued from previous page)

```

        },
        "type": 0
    },
    "version": "0"
}
},
"version": "1"
},
"Org2MSP": {
    "groups": {},
    "mod_policy": "Admins",
    "policies": {
        "Admins": {
            "mod_policy": "Admins",
            "policy": {
                "type": 1,
                "value": {
                    "identities": [
                        {
                            "principal": {
                                "msp_identifier": "Org2MSP",
                                "role": "ADMIN"
                            },
                            "principal_classification": "ROLE"
                        }
                    ],
                    "rule": {
                        "n_out_of": {
                            "n": 1,
                            "rules": [
                                {
                                    "signed_by": 0
                                }
                            ]
                        }
                    }
                }
            },
            "version": 0
        }
    },
    "version": "0"
},
"Endorsement": {
    "mod_policy": "Admins",
    "policy": {
        "type": 1,
        "value": {
            "identities": [
                {
                    "principal": {
                        "msp_identifier": "Org2MSP",
                        "role": "PEER"
                    },
                    "principal_classification": "ROLE"
                }
            ],
            "rule": {
                "n_out_of": {

```

(continues on next page)

(continued from previous page)

```

        "n": 1,
        "rules": [
            {
                "signed_by": 0
            }
        ]
    },
    "version": 0
},
"version": "0"
},
"Readers": {
    "mod_policy": "Admins",
    "policy": {
        "type": 1,
        "value": {
            "identities": [
                {
                    "principal": {
                        "msp_identifier": "Org2MSP",
                        "role": "ADMIN"
                    },
                    "principal_classification": "ROLE"
                },
                {
                    "principal": {
                        "msp_identifier": "Org2MSP",
                        "role": "PEER"
                    },
                    "principal_classification": "ROLE"
                },
                {
                    "principal": {
                        "msp_identifier": "Org2MSP",
                        "role": "CLIENT"
                    },
                    "principal_classification": "ROLE"
                }
            ]
        },
        "rule": {
            "n_out_of": {
                "n": 1,
                "rules": [
                    {
                        "signed_by": 0
                    },
                    {
                        "signed_by": 1
                    },
                    {
                        "signed_by": 2
                    }
                ]
            }
        }
    }
},

```

(continues on next page)

(continued from previous page)

```

        "version": 0
      },
      "version": "0"
    },
    "Writers": {
      "mod_policy": "Admins",
      "policy": {
        "type": 1,
        "value": {
          "identities": [
            {
              "principal": {
                "msp_identifier": "Org2MSP",
                "role": "ADMIN"
              },
              "principal_classification": "ROLE"
            },
            {
              "principal": {
                "msp_identifier": "Org2MSP",
                "role": "CLIENT"
              },
              "principal_classification": "ROLE"
            }
          ],
          "rule": {
            "n_out_of": {
              "n": 1,
              "rules": [
                {
                  "signed_by": 0
                },
                {
                  "signed_by": 1
                }
              ]
            }
          }
        }
      },
      "version": 0
    }
  },
  "version": "0"
},
"values": {
  "AnchorPeers": {
    "mod_policy": "Admins",
    "value": {
      "anchor_peers": [
        {
          "host": "peer0.org2.example.com",
          "port": 9051
        }
      ]
    }
  },
  "version": "0"
}

```

(continues on next page)

(continued from previous page)

```

    },
    "MSP": {
      "mod_policy": "Admins",
      "value": {
        "config": {
          "admins": [],
          "crypto_config": {
            "identity_identifier_hash_function": "SHA256",
            "signature_hash_family": "SHA2"
          },
          "fabric_node_ous": {
            "admin_ou_identifier": {
              "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNiakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS0
↪ ",
              "organizational_unit_identifier": "admin"
            },
            "client_ou_identifier": {
              "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNiakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS0
↪ ",
              "organizational_unit_identifier": "client"
            },
            "enable": true,
            "orderer_ou_identifier": {
              "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNiakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS0
↪ ",
              "organizational_unit_identifier": "orderer"
            },
            "peer_ou_identifier": {
              "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNiakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS0
↪ ",
              "organizational_unit_identifier": "peer"
            }
          },
          "intermediate_certs": [],
          "name": "Org2MSP",
          "organizational_unit_identifiers": [],
          "revocation_list": [],
          "root_certs": [
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNiakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS0
↪ "
        ],
        "signing_identity": null,
        "tls_intermediate_certs": [],
        "tls_root_certs": [
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNiakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS0
↪ "
      ]
    },
    "type": 0
  },
  "version": "0"

```

(continues on next page)

(continued from previous page)

```

    }
  },
  "version": "1"
}
},
"mod_policy": "Admins",
"policies": {
  "Admins": {
    "mod_policy": "Admins",
    "policy": {
      "type": 3,
      "value": {
        "rule": "MAJORITY",
        "sub_policy": "Admins"
      }
    }
  },
  "version": "0"
},
"Endorsement": {
  "mod_policy": "Admins",
  "policy": {
    "type": 3,
    "value": {
      "rule": "MAJORITY",
      "sub_policy": "Endorsement"
    }
  }
},
"version": "0"
},
"LifecycleEndorsement": {
  "mod_policy": "Admins",
  "policy": {
    "type": 3,
    "value": {
      "rule": "MAJORITY",
      "sub_policy": "Endorsement"
    }
  }
},
"version": "0"
},
"Readers": {
  "mod_policy": "Admins",
  "policy": {
    "type": 3,
    "value": {
      "rule": "ANY",
      "sub_policy": "Readers"
    }
  }
},
"version": "0"
},
"Writers": {
  "mod_policy": "Admins",
  "policy": {
    "type": 3,
    "value": {
      "rule": "ANY",

```

(continues on next page)

(continued from previous page)

```

        "sub_policy": "Writers"
    }
},
    "version": "0"
}
},
    "values": {
        "Capabilities": {
            "mod_policy": "Admins",
            "value": {
                "capabilities": {
                    "V2_0": {}
                }
            },
            "version": "0"
        }
    },
    "version": "1"
},
    "Orderer": {
        "groups": {
            "OrdererOrg": {
                "groups": {},
                "mod_policy": "Admins",
                "policies": {
                    "Admins": {
                        "mod_policy": "Admins",
                        "policy": {
                            "type": 1,
                            "value": {
                                "identities": [
                                    {
                                        "principal": {
                                            "msp_identifier": "OrdererMSP",
                                            "role": "ADMIN"
                                        },
                                        "principal_classification": "ROLE"
                                    }
                                ],
                                "rule": {
                                    "n_out_of": {
                                        "n": 1,
                                        "rules": [
                                            {
                                                "signed_by": 0
                                            }
                                        ]
                                    }
                                }
                            }
                        },
                        "version": 0
                    }
                }
            },
            "version": "0"
        },
        "Readers": {
            "mod_policy": "Admins",
            "policy": {

```

(continues on next page)

(continued from previous page)

```

        "type": 1,
        "value": {
            "identities": [
                {
                    "principal": {
                        "msp_identifier": "OrdererMSP",
                        "role": "MEMBER"
                    },
                    "principal_classification": "ROLE"
                }
            ],
            "rule": {
                "n_out_of": {
                    "n": 1,
                    "rules": [
                        {
                            "signed_by": 0
                        }
                    ]
                }
            }
        },
        "version": 0
    },
    "version": "0"
},
"Writers": {
    "mod_policy": "Admins",
    "policy": {
        "type": 1,
        "value": {
            "identities": [
                {
                    "principal": {
                        "msp_identifier": "OrdererMSP",
                        "role": "MEMBER"
                    },
                    "principal_classification": "ROLE"
                }
            ],
            "rule": {
                "n_out_of": {
                    "n": 1,
                    "rules": [
                        {
                            "signed_by": 0
                        }
                    ]
                }
            }
        },
        "version": 0
    }
},
"version": "0"
}
},
"values": {

```

(continues on next page)

(continued from previous page)

```

    "MSP": {
      "mod_policy": "Admins",
      "value": {
        "config": {
          "admins": [],
          "crypto_config": {
            "identity_identifier_hash_function": "SHA256",
            "signature_hash_family": "SHA2"
          },
          "fabric_node_ous": {
            "admin_ou_identifier": {
              "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDeNDQWJHZ0F3SUJBZ0lVUkgyT0t1V1loaStFMkFHZ3IwWUdlVTRUOV
↪ ",
              "organizational_unit_identifier": "admin"
            },
            "client_ou_identifier": {
              "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDeNDQWJHZ0F3SUJBZ0lVUkgyT0t1V1loaStFMkFHZ3IwWUdlVTRUOV
↪ ",
              "organizational_unit_identifier": "client"
            },
            "enable": true,
            "orderer_ou_identifier": {
              "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDeNDQWJHZ0F3SUJBZ0lVUkgyT0t1V1loaStFMkFHZ3IwWUdlVTRUOV
↪ ",
              "organizational_unit_identifier": "orderer"
            },
            "peer_ou_identifier": {
              "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDeNDQWJHZ0F3SUJBZ0lVUkgyT0t1V1loaStFMkFHZ3IwWUdlVTRUOV
↪ ",
              "organizational_unit_identifier": "peer"
            }
          },
          "intermediate_certs": [],
          "name": "OrdererMSP",
          "organizational_unit_identifiers": [],
          "revocation_list": [],
          "root_certs": [
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDeNDQWJHZ0F3SUJBZ0lVUkgyT0t1V1loaStFMkFHZ3IwWUdlVTRUOV
↪ "
          ],
          "signing_identity": null,
          "tls_intermediate_certs": [],
          "tls_root_certs": [
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDeNDQWJHZ0F3SUJBZ0lVUkgyT0t1V1loaStFMkFHZ3IwWUdlVTRUOV
↪ "
          ]
        },
        "type": 0
      },
      "version": "0"
    }
  }

```

(continues on next page)

(continued from previous page)

```

    },
    "version": "0"
  },
  {
    "mod_policy": "Admins",
    "policies": {
      "Admins": {
        "mod_policy": "Admins",
        "policy": {
          "type": 3,
          "value": {
            "rule": "MAJORITY",
            "sub_policy": "Admins"
          }
        }
      },
      "version": "0"
    },
    "BlockValidation": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {
          "rule": "ANY",
          "sub_policy": "Writers"
        }
      },
      "version": "0"
    },
    "Readers": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {
          "rule": "ANY",
          "sub_policy": "Readers"
        }
      },
      "version": "0"
    },
    "Writers": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {
          "rule": "ANY",
          "sub_policy": "Writers"
        }
      },
      "version": "0"
    }
  },
  {
    "values": {
      "BatchSize": {
        "mod_policy": "Admins",
        "value": {
          "absolute_max_bytes": 103809024,
          "max_message_count": 10,

```

(continues on next page)

(continued from previous page)

```

        "preferred_max_bytes": 524288
    },
    "version": "0"
},
"BatchTimeout": {
    "mod_policy": "Admins",
    "value": {
        "timeout": "2s"
    },
    "version": "0"
},
"Capabilities": {
    "mod_policy": "Admins",
    "value": {
        "capabilities": {
            "V2_0": {}
        }
    },
    "version": "0"
},
"ChannelRestrictions": {
    "mod_policy": "Admins",
    "value": null,
    "version": "0"
},
"ConsensusType": {
    "mod_policy": "Admins",
    "value": {
        "metadata": {
            "consenters": [
                {
                    "client_tls_cert":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN3akNDQWlpZ0F3SUJBZ01VZG9JbWpzaW5vVnZua011bE5WUU8wbDRMb
↪ ",
                    "host": "orderer.example.com",
                    "port": 7050,
                    "server_tls_cert":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN3akNDQWlpZ0F3SUJBZ01VZG9JbWpzaW5vVnZua011bE5WUU8wbDRMb
↪ "
                }
            ],
            "options": {
                "election_tick": 10,
                "heartbeat_tick": 1,
                "max_inflight_blocks": 5,
                "snapshot_interval_size": 16777216,
                "tick_interval": "500ms"
            }
        },
        "state": "STATE_NORMAL",
        "type": "etcdraft"
    },
    "version": "0"
}
},
"version": "0"
}

```

(continues on next page)

(continued from previous page)

```

    },
    "mod_policy": "Admins",
    "policies": {
      "Admins": {
        "mod_policy": "Admins",
        "policy": {
          "type": 3,
          "value": {
            "rule": "MAJORITY",
            "sub_policy": "Admins"
          }
        }
      },
      "version": "0"
    },
    "Readers": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {
          "rule": "ANY",
          "sub_policy": "Readers"
        }
      },
      "version": "0"
    },
    "Writers": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {
          "rule": "ANY",
          "sub_policy": "Writers"
        }
      },
      "version": "0"
    },
    "values": {
      "BlockDataHashingStructure": {
        "mod_policy": "Admins",
        "value": {
          "width": 4294967295
        },
        "version": "0"
      },
      "Capabilities": {
        "mod_policy": "Admins",
        "value": {
          "capabilities": {
            "V2_0": {}
          }
        },
        "version": "0"
      },
      "Consortium": {
        "mod_policy": "Admins",
        "value": {

```

(continues on next page)

(continued from previous page)

```
    "name": "SampleConsortium"
  },
  "version": "0"
},
"HashingAlgorithm": {
  "mod_policy": "Admins",
  "value": {
    "name": "SHA256"
  },
  "version": "0"
},
"OrdererAddresses": {
  "mod_policy": "/Channel/Orderer/Admins",
  "value": {
    "addresses": [
      "orderer.example.com:7050"
    ]
  },
  "version": "0"
}
},
"version": "0"
},
"sequence": "3"
}
```

A config might look intimidating in this form, but once you study it you'll see that it has a logical structure.

For example, let's take a look at the config with a few of the tabs closed.

```

{
  "channel_group": {
    "groups": {
      "Application": {
        "groups": {
          "Org1MSP": {
          "Org2MSP": {
        },
        "mod_policy": "Admins",
        "policies": {
          "Admins": {
          "Readers": {
          "Writers": {
        },
        "values": {
          "Capabilities": {
        },
        "version": "1"
      },
      "Orderer": {
        "groups": {
          "OrdererOrg": {
        },
        "mod_policy": "Admins",
        "policies": {
          "Admins": {
          "BlockValidation": {
          "Readers": {
          "Writers": {
        },
        "values": {
          "BatchSize": {
          "BatchTimeout": {
          "Capabilities": {
          "ChannelValidation": {

```

The structure of the config should now be more obvious. You can see the config groupings: `Channel`, `Application`, and `Orderer`, and the configuration parameters related to each config grouping (we'll talk more about these in the next section), but also where the MSPs representing organizations are. Note that the `Channel` config grouping is below the `Orderer` group config values.

More about these parameters

In this section, we'll take a deeper look at the configurable values in the context of where they sit in the configuration.

First, there are config parameters that occur in multiple parts of the configuration:

- **Policies.** Policies are not just a configuration value (which can be updated as defined in a `mod_policy`), they define the circumstances under which all parameters can be changed. For more information, check out [Policies](#).
- **Capabilities.** Ensures that networks and channels process things in the same way, creating deterministic results for things like channel configuration updates and chaincode invocations. Without deterministic results, one peer on a channel might invalidate a transaction while another peer may validate it. For more information, check out [Capabilities](#).

Channel/Application

Governs the configuration parameters unique to application channels (for example, adding or removing channel members). By default, changing these parameters requires the signature of a majority of the application organization admins.

- **Add orgs to a channel.** To add an organization to a channel, their MSP and other organization parameters must be generated and added here (under `Channel/Application/groups`).
- **Organization-related parameters.** Any parameters specific to an organization, (identifying an anchor peer, for example, or the certificates of org admins), can be changed. Note that changing these values will by default not require the majority of application organization admins but only an admin of the organization itself.

Channel/Orderer

Governs configuration parameters unique to the ordering service and requires a majority of the ordering organizations' admins (by default there is only one ordering organization, though more can be added, for example when multiple organizations contribute nodes to the ordering service).

- **Batch size.** These parameters dictate the number and size of transactions in a block. No block will appear larger than `absolute_max_bytes` large or with more than `max_message_count` transactions inside the block. If it is possible to construct a block under `preferred_max_bytes`, then a block will be cut prematurely, and transactions larger than this size will appear in their own block.
- **Batch timeout.** The amount of time to wait after the first transaction arrives for additional transactions before cutting a block. Decreasing this value will improve latency, but decreasing it too much may decrease throughput by not allowing the block to fill to its maximum capacity.
- **Block validation.** This policy specifies the signature requirements for a block to be considered valid. By default, it requires a signature from some member of the ordering org.
- **Consensus type.** To enable the migration of Kafka based ordering services to Raft based ordering services, it is possible to change the consensus type of a channel. For more information, check out [Migrating from Kafka to Raft](#).
- **Raft ordering service parameters.** For a look at the parameters unique to a Raft ordering service, check out [Raft configuration](#).

- **Kafka brokers** (where applicable). When `ConsensusType` is set to `kafka`, the `brokers` list enumerates some subset (or preferably all) of the Kafka brokers for the orderer to initially connect to at startup.

Channel

Governs configuration parameters that both the peer orgs and the ordering service orgs need to consent to, requires both the agreement of a majority of application organization admins and orderer organization admins.

- **Orderer addresses.** A list of addresses where clients may invoke the orderer `Broadcast` and `Deliver` functions. The peer randomly chooses among these addresses and fails over between them for retrieving blocks.
- **Hashing structure.** The block data is an array of byte arrays. The hash of the block data is computed as a Merkle tree. This value specifies the width of that Merkle tree. For the time being, this value is fixed to `4294967295` which corresponds to a simple flat hash of the concatenation of the block data bytes.
- **Hashing algorithm.** The algorithm used for computing the hash values encoded into the blocks of the blockchain. In particular, this affects the data hash, and the previous block hash fields of the block. Note, this field currently only has one valid value (`SHA256`) and should not be changed.

7.9.3 Editing a config

Updating a channel configuration is a three step operation that's conceptually simple:

1. Get the latest channel config
2. Create a modified channel config
3. Create a config update transaction

However, as you'll see, this conceptual simplicity is wrapped in a somewhat convoluted process. As a result, some users might choose to script the process of pulling, translating, and scoping a config update. Users also have the option of how to modify the channel configuration itself, either manually or by using a tool like `jq`.

We have two tutorials that deal specifically with editing a channel configuration to achieve a specific end:

- [Adding an Org to a Channel](#): shows the process for adding an additional organization to an existing channel.
- [Updating channel capabilities](#): shows how to update channel capabilities.

In this topic, we'll show the process of editing a channel configuration independent of the end goal of the configuration update.

Set environment variables for your config update

Before you attempt to use the sample commands, make sure to export the following environment variables, which will depend on the way you have structured your deployment. Note that the channel name, `CH_NAME` will have to be set for every channel being updated, as channel configuration updates only apply to the configuration of the channel being updated.

- `CH_NAME`: the name of the channel being updated.
- `TLS_ROOT_CA`: the path to the root CA cert of the TLS CA of the organization proposing the update.
- `CORE_PEER_LOCALMSPID`: the name of your MSP.
- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP of your organization.

- `ORDERER_CONTAINER`: the name of an ordering node container. Note that when targeting the ordering service, you can target any active node in the ordering service. Your requests will be forwarded to the leader automatically.

Note: this topic will provide default names for the various JSON and protobuf files being pulled and modified (`config_block.pb`, `config_block.json`, etc). You are free to use whatever names you want. However, be aware that unless you go back and erase these files at the end of each config update, you will have to select different when making an additional update.

Step 1: Pull and translate the config

The first step in updating a channel configuration is getting the latest config block. This is a three step process. First, we'll pull the channel configuration in protobuf format, creating a file called `config_block.pb`.

Make sure you are in the peer container.

Now issue:

```
peer channel fetch config config_block.pb -o $ORDERER_CONTAINER -c $CH_NAME --tls --
↪cafile $TLS_ROOT_CA
```

Next, we'll convert the protobuf version of the channel config into a JSON version called `config_block.json` (JSON files are easier for humans to read and understand):

```
configtxlator proto_decode --input config_block.pb --type common.Block --output_
↪config_block.json
```

Finally, we'll scope out all of the unnecessary metadata from the config, which makes it easier to read. You are free to call this file whatever you want, but in this example we'll call it `config.json`.

```
jq .data.data[0].payload.data.config config_block.json > config.json
```

Now let's make a copy of `config.json` called `modified_config.json`. **Do not edit `config.json` directly**, as we will be using it to compute the difference between `config.json` and `modified_config.json` in a later step.

```
cp config.json modified_config.json
```

Step 2: Modify the config

At this point, you have two options of how you want to modify the config.

1. Open `modified_config.json` using the text editor of your choice and make edits. Online tutorials exist that describe how to copy a file from a container that does not have an editor, edit it, and add it back to the container.
2. Use `jq` to apply edits to the config.

Whether you choose to edit the config manually or using `jq` depends on your use case. Because `jq` is concise and scriptable (an advantage when the same configuration update will be made to multiple channels), it's the recommend method for performing a channel update. For an example on how `jq` can be used, check out [Updating channel capabilities](#), which shows multiple `jq` commands leveraging a capabilities config file called `capabilities.json`. If you are updating something other than the capabilities in your channel, you will have to modify your `jq` command and JSON file accordingly.

For more information about the content and structure of a channel configuration, check out our *sample channel config* above.

Step 3: Re-encode and submit the config

Whether you make your config updates manually or using a tool like `jq`, you now have to run the process you ran to pull and scope the config in reverse, along with a step to calculate the difference between the old config and the new one, before submitting the config update to the other administrators on the channel to be approved.

First, we'll turn our `config.json` file back to protobuf format, creating a file called `config.pb`. Then we'll do the same with our `modified_config.json` file. Afterwards, we'll compute the difference between the two files, creating a file called `config_update.pb`.

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb

configtxlator proto_encode --input modified_config.json --type common.Config --output _
↪modified_config.pb

configtxlator compute_update --channel_id $CH_NAME --original config.pb --updated _
↪modified_config.pb --output config_update.pb
```

Now that we have calculated the difference between the old config and the new one, we can apply the changes to the config.

```
configtxlator proto_decode --input config_update.pb --type common.ConfigUpdate --
↪output config_update.json

echo '{"payload":{"header":{"channel_header":{"channel_id":"'CH_NAME'", "type":2}},
↪"data":{"config_update":"'$(cat config_update.json)'"}}}' | jq . > config_update_in_
↪envelope.json

configtxlator proto_encode --input config_update_in_envelope.json --type common.
↪Envelope --output config_update_in_envelope.pb
```

Submit the config update transaction:

```
peer channel update -f config_update_in_envelope.pb -c $CH_NAME -o $ORDERER_CONTAINER_
↪--tls --cafile $TLS_ROOT_CA
```

Our config update transaction represents the difference between the original config and the modified one, but the ordering service will translate this into a full channel config.

7.9.4 Get the Necessary Signatures

Once you've successfully generated the new configuration protobuf file, it will need to satisfy the relevant policy for whatever it is you're trying to change, typically (though not always) by requiring signatures from other organizations.

Note: you may be able to script the signature collection, dependent on your application. In general, you may always collect more signatures than are required.

The actual process of getting these signatures will depend on how you've set up your system, but there are two main implementations. Currently, the Fabric command line defaults to a “pass it along” system. That is, the Admin of the Org proposing a config update sends the update to someone else (another Admin, typically) who needs to sign it. This Admin signs it (or doesn't) and passes it along to the next Admin, and so on, until there are enough signatures for the config to be submitted.

This has the virtue of simplicity — when there are enough signatures, the last admin can simply submit the config transaction (in Fabric, the `peer channel update` command includes a signature by default). However, this process will only be practical in smaller channels, since the “pass it along” method can be time consuming.

The other option is to submit the update to every Admin on a channel and wait for enough signatures to come back. These signatures can then be stitched together and submitted. This makes life a bit more difficult for the Admin who created the config update (forcing them to deal with a file per signer) but is the recommended workflow for users which are developing Fabric management applications.

Once the config has been added to the ledger, it will be a best practice to pull it and convert it to JSON to check to make sure everything was added correctly. This will also serve as a useful copy of the latest config.

7.10 Writing Your First Chaincode

7.10.1 What is Chaincode?

Chaincode is a program, written in [Go](#), [Node.js](#), or [Java](#) that implements a prescribed interface. Chaincode runs in a separate process from the peer and initializes and manages the ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it is similar to a “smart contract”. A chaincode can be invoked to update or query the ledger in a proposal transaction. Given the appropriate permission, a chaincode may invoke another chaincode, either in the same channel or in different channels, to access its state. Note that, if the called chaincode is on a different channel from the calling chaincode, only read query is allowed. That is, the called chaincode on a different channel is only a `Query`, which does not participate in state validation checks in subsequent commit phase.

In the following sections, we will explore chaincode through the eyes of an application developer. We’ll present a asset-transfer chaincode sample walkthrough, and the purpose of each method in the Fabric Contract API. If you are a network operator who is deploying a chaincode to running network, visit the [Deploying a smart contract to a channel](#) tutorial and the [Fabric chaincode lifecycle](#) concept topic.

This tutorial provides an overview of the high level APIs provided by the Fabric Contract API. To learn more about developing smart contracts using the Fabric contract API, visit the [Smart Contract Processing](#) topic.

7.10.2 Fabric Contract API

The `fabric-contract-api` provides the contract interface, a high level API for application developers to implement Smart Contracts. Within Hyperledger Fabric, Smart Contracts are also known as Chaincode. Working with this API provides a high level entry point to writing business logic. Documentation of the Fabric Contract API for different languages can be found at the links below:

- [Go](#)
- [Node.js](#)
- [Java](#)

Note that when using the contract api, each chaincode function that is called is passed a transaction context “ctx”, from which you can get the chaincode stub (`GetStub()`), which has functions to access the ledger (e.g. `GetState()`) and make requests to update the ledger (e.g. `PutState()`). You can learn more at the language-specific links below.

- [Go](#)
- [Node.js](#)
- [Java](#)

In this tutorial using Go chaincode, we will demonstrate the use of these APIs by implementing a asset-transfer chaincode application that manages simple “assets”.

7.10.3 Asset Transfer Chaincode

Our application is a basic sample chaincode to initialize a ledger with assets, create, read, update, and delete assets, check to see if an asset exists, and transfer assets from one owner to another.

Choosing a Location for the Code

If you haven't been doing programming in Go, you may want to make sure that you have [Go](#) installed and your system properly configured. We assume you are using a version that supports modules.

Now, you will want to create a directory for your chaincode application.

To keep things simple, let's use the following command:

```
// atcc is shorthand for asset transfer chaincode
mkdir atcc && cd atcc
```

Now, let's create the module and the source file that we'll fill in with code:

```
go mod init atcc
touch atcc.go
```

Housekeeping

First, let's start with some housekeeping. As with every chaincode, it implements the [fabric-contract-api interface](#), so let's add the Go import statements for the necessary dependencies for our chaincode. We'll import the fabric contract api package and define our SmartContract.

```
package main

import (
    "fmt"
    "log"
    "github.com/hyperledger/fabric-contract-api-go/contractapi"
)

// SmartContract provides functions for managing an Asset
type SmartContract struct {
    contractapi.Contract
}
```

Next, let's add a struct Asset to represent simple assets on the ledger. Note the JSON annotations, which will be used to marshal the asset to JSON which is stored on the ledger.

```
// Asset describes basic details of what makes up a simple asset
type Asset struct {
    ID            string `json:"ID"`
    Color         string `json:"color"`
    Size          int    `json:"size"`
    Owner         string `json:"owner"`
    AppraisedValue int    `json:"appraisedValue"`
}
```


Initializing the Chaincode

Next, we'll implement the `InitLedger` function to populate the ledger with some initial data.

```
// InitLedger adds a base set of assets to the ledger
func (s *SmartContract) InitLedger(ctx contractapi.TransactionContextInterface) error {
    assets := []Asset{
        {ID: "asset1", Color: "blue", Size: 5, Owner: "Tomoko", AppraisedValue: 300},
        {ID: "asset2", Color: "red", Size: 5, Owner: "Brad", AppraisedValue: 400},
        {ID: "asset3", Color: "green", Size: 10, Owner: "Jin Soo", AppraisedValue: 500},
        {ID: "asset4", Color: "yellow", Size: 10, Owner: "Max", AppraisedValue: 600},
        {ID: "asset5", Color: "black", Size: 15, Owner: "Adriana", AppraisedValue: 700},
        {ID: "asset6", Color: "white", Size: 15, Owner: "Michel", AppraisedValue: 800},
    }

    for _, asset := range assets {
        assetJSON, err := json.Marshal(asset)
        if err != nil {
            return err
        }

        err = ctx.GetStub().PutState(asset.ID, assetJSON)
        if err != nil {
            return fmt.Errorf("failed to put to world state. %v", err)
        }
    }

    return nil
}
```

Next, we write a function to create an asset on the ledger that does not yet exist. When writing chaincode, it is a good idea to check for the existence of something on the ledger prior to taking an action on it, as is demonstrated in the `CreateAsset` function below.

```
// CreateAsset issues a new asset to the world state with given details.
func (s *SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface, id_
    string, color string, size int, owner string, appraisedValue int) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if exists {
        return fmt.Errorf("the asset %s already exists", id)
    }

    asset := Asset{
        ID:          id,
        Color:       color,
        Size:        size,
        Owner:       owner,
        AppraisedValue: appraisedValue,
    }
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }
}
```

(continues on next page)

(continued from previous page)

```

return ctx.GetStub().PutState(id, assetJSON)
}

```

Now that we have populated the ledger with some initial assets and created an asset, let's write a function `ReadAsset` that allows us to read an asset from the ledger.

```

// ReadAsset returns the asset stored in the world state with given id.
func (s *SmartContract) ReadAsset(ctx contractapi.TransactionContextInterface, id_
↳string) (*Asset, error) {
    assetJSON, err := ctx.GetStub().GetState(id)
    if err != nil {
        return nil, fmt.Errorf("failed to read from world state: %v", err)
    }
    if assetJSON == nil {
        return nil, fmt.Errorf("the asset %s does not exist", id)
    }

    var asset Asset
    err = json.Unmarshal(assetJSON, &asset)
    if err != nil {
        return nil, err
    }

    return &asset, nil
}

```

Now that we have assets on our ledger we can interact with, let's write a chaincode function `UpdateAsset` that allows us to update attributes of the asset that we are allowed to change.

```

// UpdateAsset updates an existing asset in the world state with provided parameters.
func (s *SmartContract) UpdateAsset(ctx contractapi.TransactionContextInterface, id_
↳string, color string, size int, owner string, appraisedValue int) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if !exists {
        return fmt.Errorf("the asset %s does not exist", id)
    }

    // overwriting original asset with new asset
    asset := Asset{
        ID:          id,
        Color:       color,
        Size:        size,
        Owner:       owner,
        AppraisedValue: appraisedValue,
    }
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(id, assetJSON)
}

```

There may be cases where we need the ability to delete an asset from the ledger, so let's write a `DeleteAsset`

function to handle that requirement.

```
// DeleteAsset deletes an given asset from the world state.
func (s *SmartContract) DeleteAsset(ctx contractapi.TransactionContextInterface, id_
↳string) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if !exists {
        return fmt.Errorf("the asset %s does not exist", id)
    }

    return ctx.GetStub().DelState(id)
}
```

We said earlier that it was a good idea to check to see if an asset exists before taking an action on it, so let's write a function called `AssetExists` to implement that requirement.

```
// AssetExists returns true when asset with given ID exists in world state
func (s *SmartContract) AssetExists(ctx contractapi.TransactionContextInterface, id_
↳string) (bool, error) {
    assetJSON, err := ctx.GetStub().GetState(id)
    if err != nil {
        return false, fmt.Errorf("failed to read from world state: %v", err)
    }

    return assetJSON != nil, nil
}
```

Next, we'll write a function we'll call `TransferAsset` that enables the transfer of an asset from one owner to another.

```
// TransferAsset updates the owner field of asset with given id in world state.
func (s *SmartContract) TransferAsset(ctx contractapi.TransactionContextInterface, id_
↳string, newOwner string) error {
    asset, err := s.ReadAsset(ctx, id)
    if err != nil {
        return err
    }

    asset.Owner = newOwner
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(id, assetJSON)
}
```

Let's write a function we'll call `GetAllAssets` that enables the querying of the ledger to return all of the assets on the ledger.

```
// GetAllAssets returns all assets found in world state
func (s *SmartContract) GetAllAssets(ctx contractapi.TransactionContextInterface)
↳([]*Asset, error) {
    // range query with empty string for startKey and endKey does an
    // open-ended query of all assets in the chaincode namespace.
```

(continues on next page)

(continued from previous page)

```

resultsIterator, err := ctx.GetStub().GetStateByRange("", "")
if err != nil {
    return nil, err
}
defer resultsIterator.Close()

var assets []*Asset
for resultsIterator.HasNext() {
    queryResponse, err := resultsIterator.Next()
    if err != nil {
        return nil, err
    }

    var asset Asset
    err = json.Unmarshal(queryResponse.Value, &asset)
    if err != nil {
        return nil, err
    }
    assets = append(assets, &asset)
}

return assets, nil
}

```

Note: The full chaincode sample below is presented as a way to keep this tutorial as clear and straightforward as possible. In a real-world implementation, it is likely that packages will be segmented where a main package imports the chaincode package to allow for easy unit testing. To see what this looks like, see the [asset-transfer Go chaincode](#) in `fabric-samples`. If you look at `assetTransfer.go`, you will see that it contains package main and imports package `chaincode` defined in `smartcontract.go` and located at `fabric-samples/asset-transfer-basic/chaincode-go/chaincode/`.

Pulling it All Together

Finally, we need to add the main function, which will call the `ContractChaincode.Start` function. Here's the whole chaincode program source.

```

package main

import (
    "encoding/json"
    "fmt"
    "log"

    "github.com/hyperledger/fabric-contract-api-go/contractapi"
)

// SmartContract provides functions for managing an Asset
type SmartContract struct {
    contractapi.Contract
}

// Asset describes basic details of what makes up a simple asset
type Asset struct {

```

(continues on next page)

(continued from previous page)

```

    ID            string `json:"ID"`
    Color          string `json:"color"`
    Size           int    `json:"size"`
    Owner          string `json:"owner"`
    AppraisedValue int    `json:"appraisedValue"`
}

// InitLedger adds a base set of assets to the ledger
func (s *SmartContract) InitLedger(ctx contractapi.TransactionContextInterface) error {
    assets := []Asset{
        {ID: "asset1", Color: "blue", Size: 5, Owner: "Tomoko", AppraisedValue: 300},
        {ID: "asset2", Color: "red", Size: 5, Owner: "Brad", AppraisedValue: 400},
        {ID: "asset3", Color: "green", Size: 10, Owner: "Jin Soo", AppraisedValue: 500},
        {ID: "asset4", Color: "yellow", Size: 10, Owner: "Max", AppraisedValue: 600},
        {ID: "asset5", Color: "black", Size: 15, Owner: "Adriana", AppraisedValue: 700},
        {ID: "asset6", Color: "white", Size: 15, Owner: "Michel", AppraisedValue: 800},
    }

    for _, asset := range assets {
        assetJSON, err := json.Marshal(asset)
        if err != nil {
            return err
        }

        err = ctx.GetStub().PutState(asset.ID, assetJSON)
        if err != nil {
            return fmt.Errorf("failed to put to world state. %v", err)
        }
    }

    return nil
}

// CreateAsset issues a new asset to the world state with given details.
func (s *SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface, id_
    string, color string, size int, owner string, appraisedValue int) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if exists {
        return fmt.Errorf("the asset %s already exists", id)
    }

    asset := Asset{
        ID:            id,
        Color:          color,
        Size:           size,
        Owner:          owner,
        AppraisedValue: appraisedValue,
    }
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }
}

```

(continues on next page)

(continued from previous page)

```

    return ctx.GetStub().PutState(id, assetJSON)
}

// ReadAsset returns the asset stored in the world state with given id.
func (s *SmartContract) ReadAsset(ctx contractapi.TransactionContextInterface, id_
↳string) (*Asset, error) {
    assetJSON, err := ctx.GetStub().GetState(id)
    if err != nil {
        return nil, fmt.Errorf("failed to read from world state: %v", err)
    }
    if assetJSON == nil {
        return nil, fmt.Errorf("the asset %s does not exist", id)
    }

    var asset Asset
    err = json.Unmarshal(assetJSON, &asset)
    if err != nil {
        return nil, err
    }

    return &asset, nil
}

// UpdateAsset updates an existing asset in the world state with provided parameters.
func (s *SmartContract) UpdateAsset(ctx contractapi.TransactionContextInterface, id_
↳string, color string, size int, owner string, appraisedValue int) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if !exists {
        return fmt.Errorf("the asset %s does not exist", id)
    }

    // overwriting original asset with new asset
    asset := Asset{
        ID:          id,
        Color:       color,
        Size:        size,
        Owner:       owner,
        AppraisedValue: appraisedValue,
    }
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(id, assetJSON)
}

// DeleteAsset deletes an given asset from the world state.
func (s *SmartContract) DeleteAsset(ctx contractapi.TransactionContextInterface, id_
↳string) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
}

```

(continues on next page)

(continued from previous page)

```

    if !exists {
        return fmt.Errorf("the asset %s does not exist", id)
    }

    return ctx.GetStub().DelState(id)
}

// AssetExists returns true when asset with given ID exists in world state
func (s *SmartContract) AssetExists(ctx contractapi.TransactionContextInterface, id_
↳string) (bool, error) {
    assetJSON, err := ctx.GetStub().GetState(id)
    if err != nil {
        return false, fmt.Errorf("failed to read from world state: %v", err)
    }

    return assetJSON != nil, nil
}

// TransferAsset updates the owner field of asset with given id in world state.
func (s *SmartContract) TransferAsset(ctx contractapi.TransactionContextInterface, id_
↳string, newOwner string) error {
    asset, err := s.ReadAsset(ctx, id)
    if err != nil {
        return err
    }

    asset.Owner = newOwner
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(id, assetJSON)
}

// GetAllAssets returns all assets found in world state
func (s *SmartContract) GetAllAssets(ctx contractapi.TransactionContextInterface)
↳([]*Asset, error) {
    // range query with empty string for startKey and endKey does an
    // open-ended query of all assets in the chaincode namespace.
    resultsIterator, err := ctx.GetStub().GetStateByRange("", "")
    if err != nil {
        return nil, err
    }
    defer resultsIterator.Close()

    var assets []*Asset
    for resultsIterator.HasNext() {
        queryResponse, err := resultsIterator.Next()
        if err != nil {
            return nil, err
        }

        var asset Asset
        err = json.Unmarshal(queryResponse.Value, &asset)
        if err != nil {
            return nil, err
        }
    }
}

```

(continues on next page)

(continued from previous page)

```
    }
    assets = append(assets, &asset)
  }

  return assets, nil
}

func main() {
  assetChaincode, err := contractapi.NewChaincode(&SmartContract{})
  if err != nil {
    log.Panicf("Error creating asset-transfer-basic chaincode: %v", err)
  }

  if err := assetChaincode.Start(); err != nil {
    log.Panicf("Error starting asset-transfer-basic chaincode: %v", err)
  }
}
```

7.10.4 Chaincode access control

Chaincode can utilize the client (submitter) certificate for access control decisions with `ctx.GetStub().GetCreator()`. Additionally the Fabric Contract API provides extension APIs that extract client identity from the submitter's certificate that can be used for access control decisions, whether that is based on client identity itself, or the org identity, or on a client identity attribute.

For example an asset that is represented as a key/value may include the client's identity as part of the value (for example as a JSON attribute indicating that asset owner), and only this client may be authorized to make updates to the key/value in the future. The client identity library extension APIs can be used within chaincode to retrieve this submitter information to make such access control decisions.

7.10.5 Managing external dependencies for chaincode written in Go

Your Go chaincode depends on Go packages (like the chaincode shim) that are not part of the standard library. The source to these packages must be included in your chaincode package when it is installed to a peer. If you have structured your chaincode as a module, the easiest way to do this is to “vendor” the dependencies with `go mod vendor` before packaging your chaincode.

```
go mod tidy
go mod vendor
```

This places the external dependencies for your chaincode into a local `vendor` directory.

Once dependencies are vendored in your chaincode directory, `peer chaincode package` and `peer chaincode install` operations will then include code associated with the dependencies into the chaincode package.

7.11 Running chaincode in development mode

Audience: Smart contract developers that want to iteratively develop and test their chaincode packages without the overhead of the smart contract lifecycle process for every update.

During smart contract development, a developer needs a way to quickly and iteratively test a chaincode package without having to run the chaincode lifecycle commands for every modification. This tutorial uses the Fabric binaries and starts the peer in development mode (“DevMode”) and then connects the chaincode to the peer. It allows you to start a chaincode without ever installing the chaincode on the peer and after the chaincode is initially committed to a channel, you can bypass the peer lifecycle chaincode commands. This allows for rapid deployment, debugging, and update without the overhead of reissuing the peer lifecycle chaincode commands every time you make an update.

Note: In order to use the DevMode flag on a peer, TLS communications must be disabled on all the nodes in the network. And because TLS communications are strongly recommended for production networks, you should never run a production peer in DevMode. The network used in this tutorial should not be used as a template for any form of a production network. See [Deploying a production network](#) for instructions on how to deploy a production network. You can also refer to the [Fabric test network](#) to learn more about how to deploy and update a smart contract package on a channel using the Fabric chaincode lifecycle process.

Throughout this tutorial, all commands are performed from the `fabric/` folder. It uses all the default settings for the peer and orderer and overrides the configurations using environment variables from the command line as needed. No edits are required to the default peer `core.yaml` or orderer `orderer.yaml` files.

7.11.1 Set up environment

1. Clone the Fabric repository from [GitHub](#). Select the release branch according to your needs.
2. Run the following commands to build the binaries for orderer, peer, and configtxgen:

```
make orderer peer configtxgen
```

When successful you should see results similar to:

```
Building build/bin/orderer
GOBIN=/testDevMode/fabric/build/bin go install -tags "" -ldflags "-X github.com/
↳hyperledger/fabric/common/metadata.Version=2.3.0 -X github.com/hyperledger/
↳fabric/common/metadata.CommitSHA=298695ae2 -X github.com/hyperledger/fabric/
↳common/metadata.BaseDockerLabel=org.hyperledger.fabric -X github.com/
↳hyperledger/fabric/common/metadata.DockerNamespace=hyperledger" github.com/
↳hyperledger/fabric/cmd/orderer
Building build/bin/peer
GOBIN=/testDevMode/fabric/build/bin go install -tags "" -ldflags "-X github.com/
↳hyperledger/fabric/common/metadata.Version=2.3.0 -X github.com/hyperledger/
↳fabric/common/metadata.CommitSHA=298695ae2 -X github.com/hyperledger/fabric/
↳common/metadata.BaseDockerLabel=org.hyperledger.fabric -X github.com/
↳hyperledger/fabric/common/metadata.DockerNamespace=hyperledger" github.com/
↳hyperledger/fabric/cmd/peer
Building build/bin/configtxgen
GOBIN=/testDevMode/fabric/build/bin go install -tags "" -ldflags "-X github.com/
↳hyperledger/fabric/common/metadata.Version=2.3.0 -X github.com/hyperledger/
↳fabric/common/metadata.CommitSHA=298695ae2 -X github.com/hyperledger/fabric/
↳common/metadata.BaseDockerLabel=org.hyperledger.fabric -X github.com/
↳hyperledger/fabric/common/metadata.DockerNamespace=hyperledger" github.com/
↳hyperledger/fabric/cmd/configtxgen
```

3. Set the PATH environment variable to include orderer and peer binaries:

```
export PATH=$(pwd)/build/bin:$PATH
```

4. Set the FABRIC_CFG_PATH environment variable to point to the sampleconfig folder:

```
export FABRIC_CFG_PATH=$(pwd)/sampleconfig
```

5. Create the hyperledger subdirectory in the /var directory. This is the default location Fabric uses to store blocks as defined in the orderer `orderer.yaml` and peer `core.yaml` files. To create the hyperledger subdirectory, execute these commands, replacing the question marks with your username:

```
sudo mkdir /var/hyperledger
sudo chown ????? /var/hyperledger
```

6. Generate the genesis block for the ordering service. Run the following command to generate the genesis block and store it in `$(pwd)/sampleconfig/genesisblock` so that it can be used by the orderer in the next step when the orderer is started.

```
configtxgen -profile SampleDevModeSolo -channelID syschannel -outputBlock_
↳ genesisblock -configPath $FABRIC_CFG_PATH -outputBlock $(pwd)/sampleconfig/
↳ genesisblock
```

When successful you should see results similar to:

```
2020-09-14 17:36:37.295 EDT [common.tools.configtxgen] doOutputBlock -> INFO 004_
↳ Generating genesis block
2020-09-14 17:36:37.296 EDT [common.tools.configtxgen] doOutputBlock -> INFO 005_
↳ Writing genesis block
```

7.11.2 Start the orderer

Run the following command to start the orderer with the `SampleDevModeSolo` profile and start the ordering service:

```
ORDERER_GENERAL_GENESISPROFILE=SampleDevModeSolo orderer
```

When it is successful you should see results similar to:

```
2020-09-14 17:37:20.258 EDT [orderer.common.server] Main -> INFO 00b Starting orderer:
Version: 2.3.0
Commit SHA: 298695ae2
Go version: go1.15
OS/Arch: darwin/amd64
2020-09-14 17:37:20.258 EDT [orderer.common.server] Main -> INFO 00c Beginning to_
↳ serve requests
```

7.11.3 Start the peer in DevMode

Open another terminal window and set the required environment variables to override the peer configuration and start the peer node. Starting the peer with the `--peer-chaincodedev=true` flag puts the peer into DevMode.

```
export PATH=$(pwd)/build/bin:$PATH
export FABRIC_CFG_PATH=$(pwd)/sampleconfig
FABRIC_LOGGING_SPEC=chaincode=debug CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:7052_
↳ peer node start --peer-chaincodedev=true
```

Reminder: When running in DevMode, TLS cannot be enabled.

When it is successful you should see results similar to:

```
2020-09-14 17:38:45.324 EDT [nodeCmd] serve -> INFO 00e Running in chaincode_
↳development mode
...
2020-09-14 17:38:45.326 EDT [nodeCmd] serve -> INFO 01a Started peer with ID=[jdoe],
↳network ID=[dev], address=[192.168.1.134:7051]
```

7.11.4 Create channel and join peer

Open another terminal window and run the following commands to generate the channel creation transaction using the configtxgen tool. This command creates the channel ch1 with the SampleSingleMSPChannel profile:

```
export PATH=$(pwd)/build/bin:$PATH
export FABRIC_CFG_PATH=$(pwd)/sampleconfig
configtxgen -channelID ch1 -outputCreateChannelTx ch1.tx -profile_
↳SampleSingleMSPChannel -configPath $FABRIC_CFG_PATH
peer channel create -o 127.0.0.1:7050 -c ch1 -f ch1.tx
```

When it is successful you should see results similar to:

```
2020-09-14 17:42:56.931 EDT [cli.common] readBlock -> INFO 002 Received block: 0
```

Now join the peer to the channel by running the following command:

```
peer channel join -b ch1.block
```

When it is successful, you should see results similar to:

```
2020-09-14 17:43:34.976 EDT [channelCmd] executeJoin -> INFO 002 Successfully_
↳submitted proposal to join channel
```

The peer has now joined channel ch1.

7.11.5 Build the chaincode

We use the **simple** chaincode from the fabric/integration/chaincode directory to demonstrate how to run a chaincode package in DevMode. In the same terminal window as the previous step, run the following command to build the chaincode:

```
go build -o simpleChaincode ./integration/chaincode/simple/cmd
```

7.11.6 Start the chaincode

When DevMode is enabled on the peer, the CORE_CHAINCODE_ID_NAME environment variable must be set to <CHAINCODE_NAME>:<CHAINCODE_VERSION> otherwise, the peer is unable to find the chaincode. For this example, we set it to mycc:1.0. Run the following command to start the chaincode and connect it to the peer:

```
CORE_CHAINCODE_LOGLEVEL=debug CORE_PEER_TLS_ENABLED=false CORE_CHAINCODE_ID_
↳NAME=mycc:1.0 ./simpleChaincode -peer.address 127.0.0.1:7052
```

Because we set debug logging on the peer when we started it, you can confirm that the chaincode registration is successful. In your peer logs, you should see results similar to:

```
2020-09-14 17:53:43.413 EDT [chaincode] sendReady -> DEBU 045 Changed to state ready_
↳for chaincode mycc:1.0
```

7.11.7 Approve and commit the chaincode definition

Now you need to run the following Fabric chaincode lifecycle commands to approve and commit the chaincode definition to the channel:

```
peer lifecycle chaincode approveformyorg -o 127.0.0.1:7050 --channelID ch1 --name_
↳mycc --version 1.0 --sequence 1 --init-required --signature-policy "OR ('SampleOrg.
↳member')" --package-id mycc:1.0
peer lifecycle chaincode checkcommitreadiness -o 127.0.0.1:7050 --channelID ch1 --
↳name mycc --version 1.0 --sequence 1 --init-required --signature-policy "OR (
↳'SampleOrg.member')"
peer lifecycle chaincode commit -o 127.0.0.1:7050 --channelID ch1 --name mycc --
↳version 1.0 --sequence 1 --init-required --signature-policy "OR ('SampleOrg.member')
↳" --peerAddresses 127.0.0.1:7051
```

You should see results similar to:

```
2020-09-14 17:56:30.820 EDT [chaincodeCmd] ClientWait -> INFO 001 txid_
↳[f22b3c25dfea7fe0b28af9ee818056db81e29a9421c83fe00eb22fa41d1dle21] committed with_
↳status (VALID) at
Chaincode definition for chaincode 'mycc', version '1.0', sequence '1' on channel 'ch1
↳' approval status by org:
SampleOrg: true
2020-09-14 17:57:43.295 EDT [chaincodeCmd] ClientWait -> INFO 001 txid_
↳[fb803e8b0b4eae6b3a9ed35668f223753e1a34ffd2a7042f9e5bb516a383eb32] committed with_
↳status (VALID) at 127.0.0.1:7051
```

7.11.8 Next steps

You can issue CLI commands to invoke and query the chaincode as needed to verify your smart contract logic. For this example, we issue three commands. The first one initializes the smart contract, the second command moves 10 from asset a to asset b. And the final command queries the value of a to verify it was successfully changed from 100 to 90.

```
CORE_PEER_ADDRESS=127.0.0.1:7051 peer chaincode invoke -o 127.0.0.1:7050 -C ch1 -n_
↳mycc -c '{"Args":["init","a","100","b","200"]}' --isInit
CORE_PEER_ADDRESS=127.0.0.1:7051 peer chaincode invoke -o 127.0.0.1:7050 -C ch1 -n_
↳mycc -c '{"Args":["invoke","a","b","10"]}'
CORE_PEER_ADDRESS=127.0.0.1:7051 peer chaincode invoke -o 127.0.0.1:7050 -C ch1 -n_
↳mycc -c '{"Args":["query","a"]}'
```

You should see results similar to:

```
2020-09-14 18:15:00.034 EDT [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001_
↳Chaincode invoke successful. result: status:200
2020-09-14 18:16:29.704 EDT [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001_
↳Chaincode invoke successful. result: status:200
2020-09-14 18:17:42.101 EDT [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001_
↳Chaincode invoke successful. result: status:200 payload:"90"
```

The benefit of running the peer in DevMode is that you can now iteratively make updates to your smart contract, save your changes, *build* the chaincode, and then *start* it again using the steps above. You do not need to run the peer lifecycle commands to update the chaincode every time you make a change.

7.12 Videos

Refer to the Hyperledger Fabric channel on YouTube

This collection contains developers demonstrating various v1 features and components such as: ledger, channels, gossip, SDK, chaincode, MSP, and more...

Deploying a production network

This deployment guide is a high level overview of the proper sequence for setting up production Fabric network components, in addition to best practices and a few of the many considerations to keep in mind when deploying. Note that this topic will discuss “setting up the network” as a holistic process from the perspective of a single individual. More likely than not, real world production networks will not be set up by a single individual but as a collaborative effort directed by several individuals (a collection of banks each setting up their own components, for example) instead.

The process for deploying a Fabric network is complex and presumes an understanding of Public Key Infrastructure and managing distributed systems. If you are a smart contract or application developer, you should not need this level of expertise in deploying a production level Fabric network. However, you might need to be aware of how networks are deployed in order to develop effective smart contracts and applications.

If all you need is a development environment to test chaincode, smart contracts, and applications against, check out [Using the Fabric test network](#). The network you’ll deploy will include two organizations, each owning one peer, and a single ordering service organization that owns a single ordering node. **This test network is not meant to provide a blueprint for deploying production components, and should not be used as such, as it makes assumptions and decisions that production deployments will not make.**

The guide will give you an overview of the steps of setting up production components and a production network:

- *Step one: Decide on your network configuration*
- *Step two: Set up a cluster for your resources*
- *Step three: Set up your CAs*
- *Step four: Use the CA to create identities and MSPs*
- *Step five: Deploy peers and ordering nodes*
 - *Creating a peer*
 - *Creating an ordering node*

8.1 Step one: Decide on your network configuration

The structure of a blockchain network will be dictated by the use case it's serving. There are too many options to give definitive guidance on every point, but let's consider a few scenarios.

In contrast to development environments or proofs of concept, security, resource management, and high availability become a priority when operating in production. How many nodes do you need to satisfy high availability, and in what data centers do you wish to deploy them in to satisfy both the needs of disaster recovery and data residency? How will you ensure that your private keys and roots of trust remain secure?

In addition to the above, here is a sampling of the decisions you will need to make before deploying components:

- **Certificate Authority configuration.** As part of the overall decisions you have to make about your peers (how many, how many on each channel, and so on) and about your ordering service (how many nodes, who will own them), you also have to decide on how the CAs for your organization will be deployed. Production networks should be using Transport Layer Security (TLS), which will require setting up a TLS CA and using it to generate TLS certificates. This TLS CA will need to be deployed before your enrollment CA. We'll discuss this more in [Step three: Set up your CAs](#).
- **Use Organizational Units or not?** Some organizations might find it necessary to establish Organizational Units to create a separation between certain identities and MSPs created by a single CA (for example, a manufacturer might want one organizational unit for its shipping department and another for its quality control department). Note that this is separate from the concept of the "Node OU", in which identities can have roles coded into them (for example, "admin" or "peer").
- **Database type.** Some channels in a network might require all data to be modeled in a way [CouchDB as the State Database](#) can understand, while other networks, prioritizing speed, might decide that all peers will use LevelDB. Note that channels should not have peers that use both CouchDB and LevelDB on them, as CouchDB imposes some data restrictions on keys and values. Keys and values that are valid in LevelDB may not be valid in CouchDB.
- **Create a system channel or not.** Ordering nodes can be bootstrapped with a configuration block for an administrative channel known as the "system channel" (from which application channels can be created), or simply started and joined to application channels as needed. The recommended method is to bootstrap without a configuration block, which is the approach this deployment guide assumes you will take. For more information about creating a system channel genesis block and bootstrapping an ordering node with it, check out [Deploying a production network](#) from the Fabric v2.2 documentation.
- **Channels and private data.** Some networks might decide that [Channels](#) are the best way to ensure privacy and isolation for certain transactions. Others might decide that fewer channels, supplemented where necessary with [Private data](#) collections, better serves their privacy needs.
- **Container orchestration.** Different users might also make different decisions about their container orchestration, creating separate containers for their peer process, logging, CouchDB, gRPC communications, and chaincode, while other users might decide to combine some of these processes.
- **Chaincode deployment method.** Users have the option to deploy their chaincode using either the built in build and run support, a customized build and run using the [External Builders and Launchers](#), or using an [Chaincode as an external service](#).
- **Using firewalls.** In a production deployment, components belonging to one organization might need access to components from other organizations, necessitating the use of firewalls and advanced networking configuration. For example, applications using the Fabric SDK require access to all endorsing peers from all organizations and the ordering services for all channels. Similarly, peers need access to the ordering service on the channels that they are receiving new blocks from.

However and wherever your components are deployed, you will need a high degree of expertise in your management system of choice (such as Kubernetes) in order to efficiently operate your network. Similarly, the structure of the

network must be designed to fit the business use case and any relevant laws and regulations government of the industry in which the network will be designed to function.

This deployment guide will not go through every iteration and potential network configuration, but does give common guidelines and rules to consider.

8.2 Step two: Set up a cluster for your resources

Generally speaking, Fabric is agnostic to the methods used to deploy and manage it. It is possible, for example, to deploy and manage a peer on a laptop. For a number of reasons, this is likely to be unadvisable, but there is nothing in Fabric that prohibits it.

As long as you have the ability to deploy containers, whether locally (or behind a firewall), or in a cloud, it should be possible to stand up components and connect them to each other. However, Kubernetes features a number of helpful tools that have made it a popular container management platform for deploying and managing Fabric networks. For more information about Kubernetes, check out [the Kubernetes documentation](#). This topic will mostly limit its scope to the binaries and provide instructions that can be applied when using a Docker deployment or Kubernetes.

However and wherever you choose to deploy your components, you will need to make sure you have enough resources for the components to run effectively. The sizes you need will largely depend on your use case. If you plan to join a single peer to several high volume channels, it will need much more CPU and memory than if you only plan to join to a single channel. As a rough estimate, plan to dedicate approximately three times the resources to a peer as you plan to allocate to a single ordering node (as you will see below, it is recommended to deploy at least three and optimally five nodes in an ordering service). Similarly, you should need approximately a tenth of the resources for a CA as you will for a peer. You will also need to add storage to your cluster (some cloud providers may provide storage) as you cannot configure Persistent Volumes and Persistent Volume Claims without storage being set up with your cloud provider first. The use of persistent storage ensures that data such as MSPs, ledgers, and installed chaincodes are not stored on the container filesystem, preventing them from being destroyed if the containers are destroyed.

By deploying a proof of concept network and testing it under load, you will have a better sense of the resources you will require.

8.2.1 Managing your infrastructure

The exact methods and tools you use to manage your backend will depend on the backend you choose. However, here are some considerations worth noting.

- Using secret objects to securely store important configuration files in your cluster. For information about Kubernetes secrets, check out [Kubernetes secrets](#). You also have the option to use Hardware Security Modules (HSMs) or encrypted Persistent Volumes (PVs). Along similar lines, after deploying Fabric components, you will likely want to connect to a container on your own backend, for example using a private repo in a service like Docker Hub. In that case, you will need to code the login information in the form of a Kubernetes secret and include it in the YAML file when deploying components.
- Cluster considerations and node sizing. In step 2 above, we discussed a general outline for how to think about the sizings of nodes. Your use case, as well as a robust period of development, is the only way you will truly know how large your peers, ordering nodes, and CAs will need to be.
- How you choose to mount your volumes. It is a best practice to mount the volumes relevant to your nodes external to the place where your nodes are deployed. This will allow you to reference these volumes later on (for example, restarting a node or a container that has crashed) without having to redeploy or regenerate your crypto material.
- How you will monitor your resources. It is critical that you establish a strategy and method for monitoring the resources used by your individual nodes and the resources deployed to your cluster generally. As you join your

peers to more channels, you will need likely need to increase its CPU and memory allocation. Similarly, you will need to make sure you have enough storage space for your state database and blockchain.

8.3 Step three: Set up your CAs

The first component that must be deployed in a Fabric network is a CA. This is because the certificates associated with a node (not just for the node itself but also the certificates identifying who can administer the node) must be created before the node itself can be deployed. While it is not necessary to use the Fabric CA to create these certificates, the Fabric CA also creates MSP structures that are needed for components and organizations to be properly defined. If a user chooses to use a CA other than the Fabric CA, they will have to create the MSP folders themselves.

- One CA (or more, if you are using intermediate CAs — more on intermediate CAs below) is used to generate (through a process called “enrollment”) the certificates of the admin of an organization, the MSP of that organization, and any nodes owned by that organization. This CA will also generate the certificates for any additional users. Because of its role in “enrolling” identities, this CA is sometimes called the “enrollment CA” or the “ecert CA”.
- The other CA generates the certificates used to secure communications on Transport Layer Security (TLS). For this reason, this CA is often referred to as a “TLS CA”. These TLS certificates are attached to actions as a way of preventing “man in the middle” attacks. Note that the TLS CA is only used for issuing certificates for nodes and can be shut down when that activity is completed. Users have the option to use one way (client only) TLS as well as two way (server and client) TLS, with the latter also known as “mutual TLS”. Because specifying that your network will be using TLS (which is recommended) should be decided before deploying the “enrollment” CA (the YAML file specifying the configuration of this CA has a field for enabling TLS), you should deploy your TLS CA first and use its root certificate when bootstrapping your enrollment CA. This TLS certificate will also be used by the `fabric-ca client` when connecting to the enrollment CA to enroll identities for users and nodes.

While all of the non-TLS certificates associated with an organization can be created by a single “root” CA (that is, a CA that is its own root of trust), for added security organizations can decide to use “intermediate” CAs whose certificates are created by a root CA (or another intermediate CA that eventually leads back to a root CA). Because a compromise in the root CA leads to a collapse for its entire trust domain (the certs for the admins, nodes, and any CAs it has generated certificates for), intermediate CAs are a useful way to limit the exposure of the root CA. Whether you choose to use intermediate CAs will depend on the needs of your use case. They are not mandatory. Note that it is also possible to configure a Lightweight Directory Access Protocol (LDAP) to manage identities on a Fabric network for those enterprises that already have this implementation and do not want to add a layer of identity management to their existing infrastructure. The LDAP effectively pre registers all of the members of the directory and allows them to enroll based on the criteria given.

In a production network, it is recommended to deploy at least one CA per organization for enrollment purposes and another for TLS. For example, if you deploy three peers that are associated with one organization and an ordering node that is associated with an ordering organization, you will need at least four CAs. Two of the CAs will be for the peer organization (generating the enrollment and TLS certificates for the peer, admins, communications, and the folder structure of the MSP representing the organization) and the other two will be for the orderer organization. Note that users will generally only register and enroll with the enrollment CA, while nodes will register and enroll with both the enrollment CA (where the node will get its signing certificates that identify it when it attempts to sign its actions) and with the TLS CA (where it will get the TLS certificates it uses to authenticate its communications).

For an example of how to setup an organization CA and a TLS CA and enroll their admin identity, check out the [Fabric CA Deployment Guide](#). The deploy guide uses the Fabric CA client to register and enroll the identities that are required when setting up CAs.

8.4 Step four: Use the CA to create identities and MSPs

After you have created your CAs, you can use them to create the certificates for the identities and components related to your organization (which is represented by an MSP). For each organization, you will need to, at a minimum:

- **Register and enroll an admin identity and create an MSP.** After the CA that will be associated with an organization has been created, it can be used to first register a user and then enroll an identity (producing the certificate pair used by all entities on the network). In the first step, a username and password for the identity is assigned by the admin of the CA. Attributes and affiliations can also be given to the identity (for example, a `role` of `admin`, which is necessary for organization admins). After the identity has been registered, it can be enrolled by using the username and password. The CA will generate two certificates for this identity — a public certificate (also known as a “signcert” or “public cert”) known to the other members of the network, and the private key (stored in the `keystore` folder) used to sign actions taken by the identity. The CA will also generate a set of folders called an “MSP” containing the public certificate of the CA issuing the certificate and the root of trust for the CA (this may or may not be the same CA). This MSP can be thought of as defining the organization associated with the identity of the admin. In cases where the admin of the org will also be an admin of a node (which will be typical), **you must create the org admin identity before creating the local MSP of a node, since the certificate of the node admin must be used when creating the local MSP.**
- **Register and enroll node identities.** Just as an org admin identity is registered and enrolled, the identity of a node must be registered and enrolled with both an enrollment CA and a TLS CA (the latter generates certificates that are used to secure communications). Instead of giving a node a `role` of `admin` or `user` when registering it with the enrollment CA, give it a `role` of `peer` or `orderer`. As with the admin, attributes and affiliations for this identity can also be assigned. The MSP structure for a node is known as a “local MSP”, since the permissions assigned to the identities are only relevant at the local (node) level. This MSP is created when the node identity is created, and is used when bootstrapping the node.

For more conceptual information about identities and permissions in a Fabric-based blockchain network, see [Identity](#) and [Membership Service Provider \(MSP\)](#).

For more information about how to use a CA to register and enroll identities, including sample commands, check out [Registering and enrolling identities with a CA](#).

8.5 Step five: Deploy peers and ordering nodes

Once you have gathered all of the certificates and MSPs you need, you’re almost ready to create a node. As discussed above, there are a number of valid ways to deploy nodes.

Before any node can be deployed, its configuration file must be customized. For the peer, this file is called `core.yaml`, while the configuration file for ordering nodes is called `orderer.yaml`.

You have three main options for tuning your configuration.

1. Edit the YAML file bundled with the binaries.
2. Use environment variable overrides when deploying.
3. Specify flags on CLI commands.

Option 1 has the advantage of persisting your changes whenever you bring down and bring back up the node. The downside is that you will have to port the options you customized to the new YAML when upgrading to a new binary version (you should use the latest YAML when upgrading to a new version).

Note: You can extrapolate environment variables from the parameters in the relevant YAML file by using all capital letters, underscores between the relevant phrases, and a prefix. For example, the peer configuration variable called `peer.localMSPid` (which is the `localMSPid` variable inside the peer configuration section) in `core.yaml`

would be rendered as an environment variable called `CORE_PEER_LOCALMSPID`, while the ordering service environment variable `General.LocalMSPID` in the `General` section of the `orderer.yaml` configuration file would be rendered as an environment variable called `ORDERER_GENERAL_LOCALMSPID`.

8.5.1 Creating a peer

If you’ve read through the key concept topic on [Peers](#), you should have a good idea of the role peers play in a network and the nature of their interactions with other network components. Peers are owned by organizations that are members of a channel (for this reason, these organizations are sometimes called “peer organizations”). They connect to the ordering service and to other peers, have smart contracts installed on them, and are where ledgers are stored.

These roles are important to understand before you create a peer, as they will influence your customization and deployment decisions. For a look at the various decisions you will need to make, check out [Planning for a production peer](#).

The configuration values in a peer’s `core.yaml` file must be customized or overridden with environment variables. You can find the default `core.yaml` configuration file in the [sampleconfig](#) directory of Hyperledger Fabric. This configuration file is bundled with the peer image and is also included with the downloadable binaries. For information about how to download the production `core.yaml` along with the peer image, check out [Deploy the peer](#).

While there are many parameters in the default `core.yaml`, you will only need to customize a small percentage of them. In general, if you do not have the need to change a tuning value, keep the default value.

Among the parameters in `core.yaml`, there are:

- **Identifiers:** these include not just the paths to the relevant local MSP and Transport Layer Security (TLS) certificates, but also the name (known as the “peer ID”) of the peer and the MSP ID of the organization that owns the peer.
- **Addresses and paths:** because peers are not entities unto themselves but interact with other peers and components, you must specify a series of addresses in the configuration. These include addresses where the peer itself can be found by other components as well as the addresses where, for example, chaincodes can be found (if you are employing external chaincodes). Similarly, you will need to specify the location of your ledger (as well as your state database type) and the path to your external builders (again, if you intend to employ external chaincodes). These include **Operations and metrics**, which allow you to set up methods for monitoring the health and performance of your peer through the configuration of endpoints.
- **Gossip:** components in Fabric networks communicate with each other using the “gossip” protocol. Through this protocol, they can be discovered by the discovery service and disseminate blocks and private data to each other. Note that gossip communications are secured using TLS.

For more information about `core.yaml` and its specific parameters, check out [Checklist for a production peer](#).

When you’re comfortable with how your peer has been configured, how your volumes are mounted, and your backend configuration, you can run the command to launch the peer (this command will depend on your backend configuration).

Planning for a production peer

Audience: Architects, network operators, users setting up a production Fabric network who are familiar with Transport Layer Security (TLS), Public Key Infrastructure (PKI) and Membership Service Providers (MSPs).

Note: “chaincode” refers to the packages that are installed on peers, while “smart contracts” refers to the business logic that is agreed to by organizations.

Peer nodes are a fundamental element of a Fabric network because they host ledgers and smart contracts that are used to encapsulate the shared processes and shared information in a blockchain network. These instructions assume you are already familiar with the concept of a [peer](#) and provides guidance for the various decisions you will have to make

about a peer you will deploy and join to a production Fabric network channel. If you need to quickly stand up a network for education or testing purposes, check out the [Fabric test network](#).

Generate peer identities and Membership Service Providers (MSPs)

Before proceeding with this topic, you should have reviewed the process for a [Deploying a Certificate Authority \(CA\)](#) for your organization in order to generate the identities and MSPs for the admins and peers in your organization. To learn how to use a CA to create these identities, check out [Registering and enrolling identities with a CA](#)

Note that the “cryptogen” tool should never be used to generate any identities in a production scenario.

Folder management

While it is possible to bootstrap a peer using a number of folder structures for your MSPs and certificates, we do recommend a particular [folder structure](#) for the sake of consistency and repeatability. These instructions will presume that you have used that folder structure.

Certificates from a non-Fabric CA

While it is possible to use a non-Fabric CA to generate identities, this process requires that you manually construct the MSP folders the peer needs to be deployed. That process will not be covered here and will instead focus on using a Fabric CA to generate the identities and MSP folders for you.

Transport Layer Security (TLS) enablement

To prevent “man in the middle” attacks and otherwise secure communications, using TLS is a requirement for any production network. Therefore, in addition to registering your peer identities with your organization CA, you will also need to register your peer identities with the TLS CA for the organization. These TLS certificates will be used by the peer when communicating with the network.

State database

Each peer maintains a state database that tracks the current value for all of the assets (also known as “keys”) listed on the ledger. Two types of state databases are supported: External CouchDB (which allows JSON queries of the database) or embedded Goleveldb (which does not). The choice of database largely depends on whether you need the CouchDB JSON query support. If JSON query is not needed, Goleveldb improves performance and requires less management since it is embedded in the peer process. Because all of the peers on a channel must use the same state database, your choice of database might already be dictated by the channels you wish to join.

Beyond the ability to execute JSON queries when using CouchDB, the choice of the database is invisible to a smart contract.

You can review [State Database options](#) for more details.

Sizing your peer resources

A peer typically has multiple containers associated with it.

- **Peer container:** Encapsulates the peer process that validates and commits transactions for all channels a peer belongs to. The peer storage includes each channel's blockchain (in other words, the transaction history), local databases including the state database if using Goleveldb, and any chaincodes that are installed on the peer. The size of peer storage depends on the number of channels, and the number and size of transactions in each channel.
- **CouchDB container** (optional): If using CouchDB as the state database, the CouchDB container will be used to store the state database of each channel.
- **Chaincode launcher container** (optional): Used to launch a separate container for each chaincode, eliminating the need for a Docker-in-Docker container in the peer container. Note that the chaincode launcher container is not where smart contracts actually run, and is therefore given a smaller default resource than the "smart contracts" container that used to be deployed along with a peer. It only exists to help create the containers where a smart contract will run. You must make your own allowances in your cluster for the containers for the chaincodes deployed by the launcher.
- **Chaincode container:** The container where the chaincode runs. Note that the recommended process is to deploy each chaincode into a separate container, even if you have multiple peers on the same channel that have all installed the same chaincode. So if you have three peers on a channel, and install a smart contract on each one, you will have three smart contract containers running. However, if these three peers are on more than one channel using the exact same smart contract, you will still only have three pods running.

Storage considerations

Chaincodes and the ledger (one for each channel) are physically stored on a peer according to the `peer.fileSystemPath` parameter, while identities and MSP are stored according to the `peer.mspConfigPath` parameter (by default, both locations are at `/var/hyperledger/production`). **This file system needs to be protected, secured, and writable by authorized users only** and should also be regularly backed up. Note that the best practice is to use externally mounted volumes for both of these parameters, as they will therefore be easy to reference when restarting or upgrading the peer.

When you configure your peer, you need to decide if the state database will be stored in CouchDB or LevelDB (default) by configuring the `ledger.state.stateDatabase` parameter.

While this topic is focused on how to use the peer binary images, there are important storage considerations you need to be aware of when you run the Fabric images in Docker containers or use Kubernetes. Docker containers requires a volume bind mount that mounts the external folder pathing to your container. This is critical when the container restarts, so that the storage is not lost. Similarly, if you are using Kubernetes, you need to provision storage for the peer and then map it in your Kubernetes pod deployment YAML file.

High Availability

As part of planning to create a peer, you will need consider your strategy at an organization level in order to ensure zero downtime of your components. This means building redundant components, and specifically redundant peers. To ensure zero downtime, you need at least one redundant peer **in a separate virtual machine** so that peers can go down for maintenance while client applications go on submitting endorsement proposals uninterrupted.

Along similar lines, client applications should be configured to use Service Discovery to ensure that transactions are only submitted to peers that are currently available. As long as at least one peer from each organization is available, and service discovered is being used, any endorsement policy will be able to be satisfied. It is the responsibility of each organization to make sure their high availability strategy is robust enough to ensure that at least one peer owned by their organization is available at all times in every channel they're joined to.

Monitoring

All blockchain nodes require careful monitoring, but it is critically important to monitor the peer and ordering nodes. By virtue of being immutable, the ledger inevitably grows. As a result, storage must be monitored and extended as needed. If the storage for a peer is exhausted you also have the option to deploy a new peer with a larger storage allocation and let the ledger sync. In a production environment you should also monitor the CPU and memory allocated to a peer using widely available tooling. If you see the peer struggling to keep up with the transaction load or when performing relatively simple tasks (querying the ledger, for example), it is a sign that you might need to increase its resource allocation.

Chaincode

Prior to Hyperledger Fabric 2.0, the process used to build and launch chaincode was part of the peer implementation and could not be easily customized. All chaincode installed on the peer would be “built” using language specific logic hard coded in the peer. This build process would generate a Docker container image that would be launched to execute chaincode that connected as a client to the peer.

This approach limited chaincode implementations to a handful of languages, required Docker to be part of the deployment environment, prevented running chaincode as a long-running server process, and required that the peer have privileged access to the chaincode container.

Starting with Fabric 2.0, External Builders and Launchers enable operators to extend the peer with programs that can build, launch, and discover chaincode. To leverage this capability on peers that already exist you will need to create your own buildpack and then modify `core.yaml` to include a new externalBuilder configuration element which lets the peer know an external builder is available.

Gossip

Peers leverage the [gossip data dissemination protocol](#) to broadcast ledger and channel data in a scalable fashion. Gossip messaging is continuous, and each peer on a channel is constantly receiving data from multiple peers, including peers in other organizations (if cross-organization gossip is enabled).

For peer gossip to work you need to configure four parameters. Three of them — `peer.gossip.bootstrap`, `peer.gossip.endpoint`, `peer.gossip.externalEndpoint` — are in the peer’s `core.yaml` file. The fourth enables gossip between organization by specifying an anchor peer in the channel configuration.

To reduce network traffic, in Fabric v2.2 the default `core.yaml` is configured for peers to pull blocks from the ordering service instead of through gossip dissemination among peers (with the exception of private data, which are still sent from peer to peer using gossip). To get all blocks from the orderer, you must use the following parameters in the `core.yaml` file:

- `peer.gossip.useLeaderElection = false`
- `peer.gossip.orgLeader = true`
- `peer.gossip.state.enabled = false`

If all peers have `orgLeader=true` (recommended), then each peer will get blocks from the ordering service.

Service Discovery

In any network it is possible that peer nodes can be down for maintenance, unreachable due to network issues, or the peer ledger has fallen behind while being offline. For this reason, Fabric includes a “discovery service” that enables client applications that use the SDK to locate good candidate peers to target with endorsement requests. If service discovery is not enabled, when a client application targets a peer that is offline, the request fails and will need

to be resubmitted to another peer. The discovery service runs on peers and uses the network metadata information maintained by the gossip communication layer to find out which peers are online and can be targeted for requests.

Service discovery (and private data) requires that gossip is enabled, therefore you should configure the `peer.gossip.bootstrap`, `peer.gossip.endpoint`, and `peer.gossip.externalEndpoint` parameters, as well as anchor peers on each channel, to take advantage of this feature.

Checklist for a production peer

As you prepare to build a production peer, you need to customize the configuration by editing the `core.yaml` file, which is copied into the `/config` directory when downloading the Fabric binaries, and available within the Fabric peer image at `/etc/hyperledger/fabric/core.yaml`.

While in a production environment you could override the environment variables in the `core.yaml` file in your Docker container or your Kubernetes job, these instructions show how to edit `core.yaml` instead. It's important to understand the parameters in the configuration file and their dependencies on other parameter settings in the file. Blindly overriding one setting using an environment variable could affect the functionality of another setting. Therefore, the recommendation is that before starting the peer, you make the modifications to the settings in the configuration file to become familiar with the available settings and how they work. Afterwards, you may choose to override these parameters using environment variables.

This checklist covers key configuration parameters for setting up a production network. Of course, you can always refer to the `core.yaml` file for additional parameters or more information. It also provides guidance on which parameters should be overridden. The list of parameters that you need to understand and that are described in this topic include:

- *peer.id*
- *peer.networkId*
- *peer.listenAddress*
- *peer.chaincodeListenAddress*
- *peer.chaincodeAddress*
- *peer.address*
- *peer.mspConfigPath*
- *peer.localMspId*
- *peer.fileSystemPath*
- *peer.gossip.**
- *peer.tls.**
- *peer.bccsp.**
- *chaincode.externalBuilders.**
- *ledger.**
- *operations.**
- *metrics.**

peer.id


```
# The peer id provides a name for this peer instance and is used when
# naming docker resources.
id: jdoe
```

- **id:** (Default value should be overridden.) Start by giving your peer an ID (which is analogous to giving it a name). Often the name indicates the organization that the peer belongs to, for example `peer0.org1.example.com`. It is used for naming the peer's chaincode images and containers.

peer.networkId

```
# The networkId allows for logical separation of networks and is used when
# naming docker resources.
networkId: dev
```

- **networkId:** (Default value should be overridden.) Specify any name that you want. One recommendation would be to differentiate the network by naming it based on its planned usage (for example, “dev”, “staging”, “test”, “production”, etc). This value is also used to build the name of the chaincode images and containers.

peer.listenAddress

```
# The Address at local network interface this Peer will listen on.
# By default, it will listen on all network interfaces
listenAddress: 0.0.0.0:7051
```

- **listenAddress:** (Default value should be overridden.) Specify the address that the peer will listen on, for example, `0.0.0.0:7051`.

peer.chaincodeListenAddress

```
# The endpoint this peer uses to listen for inbound chaincode connections.
# If this is commented-out, the listen address is selected to be
# the peer's address (see below) with port 7052
chaincodeListenAddress: 0.0.0.0:7052
```

- **chaincodeListenAddress:** (Default value should be overridden.) Uncomment this parameter and specify the address where this peer listens for chaincode requests. It needs to be different than the `peer.listenAddress`, for example, `0.0.0.0:7052`.

peer.chaincodeAddress

```
# The endpoint the chaincode for this peer uses to connect to the peer.
# If this is not specified, the chaincodeListenAddress address is selected.
# And if chaincodeListenAddress is not specified, address is selected from
# peer address (see below). If specified peer address is invalid then it
# will fallback to the auto detected IP (local IP) regardless of the peer
# addressAutoDetect value.
chaincodeAddress: 0.0.0.0:7052
```

- **chaincodeAddress:** (Default value should be overridden.) Uncomment this parameter and specify the address that chaincode containers can use to connect to this peer, for example, `peer0.org1.example.com:7052`.

peer.address

```
# When used as peer config, this represents the endpoint to other peers
# in the same organization. For peers in other organization, see
# gossip.externalEndpoint for more info.
# When used as CLI config, this means the peer's endpoint to interact with
address: 0.0.0.0:7051
```

- **address:** (Default value should be overridden.) Specify the address that other peers in the organization use to connect to this peer, for example, `peer0.org1.example.com:7051`.

peer.mspConfigPath

```
mspConfigPath: msp
```

- **mspConfigPath:** (Default value should be overridden.) This is the path to the peer's local MSP, which must be created before the peer can be deployed. The path can be absolute or relative to `FABRIC_CFG_PATH` (by default, it is `/etc/hyperledger/fabric` in the peer image). Unless an absolute path is specified to a folder named something other than “msp”, the peer defaults to looking for a folder called “msp” at the path (in other words, `FABRIC_CFG_PATH/msp`) and when using the peer image: `/etc/hyperledger/fabric/msp`. If you are using the recommended folder structure described in the [Registering and enrolling identities with a CA](#) topic, it would be relative to the `FABRIC_CFG_PATH` as follows: `config/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/msp`. **The best practice is to store this data in persistent storage.** This prevents the MSP from being lost if your peer containers are destroyed for some reason.

peer.localMspId

```
# Identifier of the local MSP
# ----!!!IMPORTANT!!!-!!!IMPORTANT!!!-!!!IMPORTANT!!!!----
# Deployers need to change the value of the localMspId string.
# In particular, the name of the local MSP ID of a peer needs
# to match the name of one of the MSPs in each of the channel
# that this peer is a member of. Otherwise this peer's messages
# will not be identified as valid by other nodes.
localMspId: SampleOrg
```

- **localMspId:** (Default value should be overridden.) This is the value of the MSP ID of the organization the peer belongs to. Because peers can only be joined to a channel if the organization the peer belongs to is a channel member, this MSP ID must match the name of at least one of the MSPs in each of the channels that this peer is a member of.

peer.fileSystemPath

```
# Path on the file system where peer will store data (eg ledger). This
# location must be access control protected to prevent unintended
# modification that might corrupt the peer operations.
fileSystemPath: /var/hyperledger/production
```

- **fileSystemPath:** (Default value should be overridden.) This is the path to the ledger and installed chaincodes on the local filesystem of the peer. It can be an absolute path or relative to FABRIC_CFG_PATH. It defaults to /var/hyperledger/production. The user running the peer needs to own and have write access to this directory. **The best practice is to store this data in persistent storage.** This prevents the ledger and any installed chaincodes from being lost if your peer containers are destroyed for some reason. Note that ledger snapshots will be written to ledger.snapshots.rootDir, described in the *ledger.* section*.

peer.gossip.*

```
gossip:
  # Bootstrap set to initialize gossip with.
  # This is a list of other peers that this peer reaches out to at startup.
  # Important: The endpoints here have to be endpoints of peers in the same
  # organization, because the peer would refuse connecting to these endpoints
  # unless they are in the same organization as the peer.
  bootstrap: 127.0.0.1:7051

  # Overrides the endpoint that the peer publishes to peers
  # in its organization. For peers in foreign organizations
  # see 'externalEndpoint'
  endpoint:

  # This is an endpoint that is published to peers outside of the organization.
  # If this isn't set, the peer will not be known to other organizations.
  externalEndpoint:

  # NOTE: orgLeader and useLeaderElection parameters are mutual exclusive.
  # Setting both to true would result in the termination of the peer
  # since this is undefined state. If the peers are configured with
  # useLeaderElection=false, make sure there is at least 1 peer in the
  # organization that its orgLeader is set to true.

  # Defines whenever peer will initialize dynamic algorithm for
  # "leader" selection, where leader is the peer to establish
  # connection with ordering service and use delivery protocol
  # to pull ledger blocks from ordering service.
  useLeaderElection: false

  # Statically defines peer to be an organization "leader",
  # where this means that current peer will maintain connection
  # with ordering service and disseminate block across peers in
  # its own organization. Multiple peers or all peers in an organization
  # may be configured as org leaders, so that they all pull
  # blocks directly from ordering service.
  orgLeader: true

  # Gossip state transfer related configuration
  state:
    # indicates whenever state transfer is enabled or not
    # default value is true, i.e. state transfer is active
```

(continues on next page)

(continued from previous page)

```

    # and takes care to sync up missing blocks allowing
    # lagging peer to catch up to speed with rest network
    enabled: false

pvtData:
  implicitCollectionDisseminationPolicy:
    # requiredPeerCount defines the minimum number of eligible peers to which
    ↳the peer must successfully
    # disseminate private data for its own implicit collection during
    ↳endorsement. Default value is 0.
    requiredPeerCount: 0

    # maxPeerCount defines the maximum number of eligible peers to which the
    ↳peer will attempt to
    # disseminate private data for its own implicit collection during
    ↳endorsement. Default value is 1.
    maxPeerCount: 1

```

Peers leverage the Gossip data dissemination protocol to broadcast ledger and channel data in a scalable fashion. Gossip messaging is continuous, and each peer on a channel is constantly receiving current and consistent ledger data from multiple peers. While there are many Gossip parameters that can be customized, there are three groups of settings you need to pay attention to at a minimum:

- **Endpoints** Gossip is required for service discovery and private data dissemination. To use these features, you must configure the `gossip bootstrap`, `endpoint`, and `externalEndpoint` parameters in addition to **setting at least one anchor peer** in the peer's channel configuration.
 - **bootstrap**: (Default value should be overridden.) Provide the list of other peer *addresses* in this organization to discover.
 - **endpoint**: (Default value should be overridden.) Specify the address that other peers *in this organization* should use to connect to this peer. For example, `peer0.org1.example.com:7051`.
 - **externalEndpoint**: (Default value should be overridden.) Specify the address that peers in *other organizations* should use to connect to this peer, for example, `peer0.org1.example.com:7051`.
- **Block dissemination** In order to reduce network traffic, it is recommended that peers get their blocks from the ordering service instead of from other peers in their organization (the default configuration starting in Fabric v2.2). The combination of the `useLeaderElection`, `orgLeader`, and `state.enabled` parameters in this section ensures that peers will pull blocks from the ordering service.
 - **useLeaderElection**: (Defaults to `false` as of v2.2, which is recommended so that peers get blocks from ordering service.) When `useLeaderElection` is set to `false`, you must configure at least one peer to be the org leader by setting `peer.gossip.orgLeader` to `true`. Set `useLeaderElection` to `true` if you prefer that peers use Gossip for block dissemination among peers in the organization.
 - **orgLeader**: (Defaults to `true` as of v2.2, which is recommended so that peers get blocks from ordering service.) Set this value to `false` if you want to use Gossip for block dissemination among peers in the organization.
 - **state.enabled**: (Defaults to `false` as of v2.2 which is recommended so that peers get blocks from ordering service.) Set this value to `true` when you want to use Gossip to sync up missing blocks, which allows a lagging peer to catch up with other peers on the network.
- **Implicit data** Fabric v2.0 introduced the concept of private data implicit collections on a peer. If you'd like to utilize per-organization private data patterns, you don't need to define any collections when deploying chaincode in Fabric v2.*. Implicit organization-specific collections can be used without any upfront definition. When you plan to take advantage of this new feature, you need to configure the values of the

`pvtData.implicitCollectionDisseminationPolicy.requiredPeerCount` and `pvtData.implicitCollectionDisseminationPolicy.maxPeerCount`. For more details, review the [Private data tutorial](#).

- **`pvtData.implicitCollectionDisseminationPolicy.requiredPeerCount`**: (Recommended that you override this value when using private data implicit collections.) **New in Fabric 2.0.** It defaults to 0, but you will need to increase it based on the number of peers belonging to your organization. The value represents the required number of peers within your own organization that the data must be disseminated to, to ensure data redundancy in case a peer goes down after it endorses a transaction.
- **`pvtData.implicitCollectionDisseminationPolicy.maxPeerCount`**: (Recommended that you override this value when using private data implicit collections.) **New in Fabric 2.0.** This is an organization-specific collection setting that is used to ensure the private data is disseminated elsewhere in case this peer endorses a request and then goes down for some reason. While the `requiredPeerCount` specifies the number of peers that must get the data, the `maxPeerCount` is the number of attempted peer disseminations. The default is set to 1 but in a production environment with n peers in an organization, the recommended setting is $n-1$.

peer.tls.*

```
tls:
  # Require server-side TLS
  enabled: false
  # Require client certificates / mutual TLS for inbound connections.
  # Note that clients that are not configured to use a certificate will
  # fail to connect to the peer.
  clientAuthRequired: false
  # X.509 certificate used for TLS server
  cert:
    file: tls/server.crt
  # Private key used for TLS server
  key:
    file: tls/server.key
  # rootcert.file represents the trusted root certificate chain used for verifying
  ↪certificates
  # of other nodes during outbound connections.
  # It is not required to be set, but can be used to augment the set of TLS CA
  ↪certificates
  # available from the MSPs of each channel's configuration.
  rootcert:
    file: tls/ca.crt
  # If mutual TLS is enabled, clientRootCAs.files contains a list of additional
  ↪root certificates
  # used for verifying certificates of client connections.
  # It augments the set of TLS CA certificates available from the MSPs of each
  ↪channel's configuration.
  # Minimally, set your organization's TLS CA root certificate so that the peer can
  ↪receive join channel requests.
  clientRootCAs:
    files:
      - tls/ca.crt
```

Configure this section to enable TLS communications for the peer. After TLS is enabled, all nodes that transact with the peer will also need to enable TLS. Review the topic on [Registering and enrolling identities with a CA](#) for instructions on how to generate the peer TLS certificates.

- **enabled**: (Default value should be overridden.) To ensure your production environment is secure, TLS should

be enabled for all communications between nodes by setting `enabled: true` in the `tls` section of the config file. While this field is disabled by default, which may be acceptable for a test network, it should be enabled when in production. This setting will configure **server-side TLS**, meaning that TLS will guarantee the identity of the *server* to the client and provides a two-way encrypted channel between them.

- **cert.file:** (Default value should be overridden.) Every peer needs to register and enroll with its TLS CA before it can transact securely with other nodes in the organization. Therefore, before you can deploy a peer, you must first register a user for the peer and enroll the peer identity with the TLS CA to generate the peer's TLS signed certificate. If you are using the recommended folder structure from the [Registering and enrolling identities with a CA](#) topic, this file needs to be copied into `config/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls`
- **key.file:** (Default value should be overridden.) Similar to the `cert.file`, provide the name and location of the generated TLS private key for this peer, for example, `/msp/keystore/87bf5eff47d33b13d7aee81032b0e8e1e0ffc7a6571400493a7c_sk`. If you are using the recommended folder structure from the [Registering and enrolling identities with a CA](#) topic, this file needs to be copied into `config/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls`. If you are using an *HSM* to store the private key for the peer, this field will be blank.
- **rootcert.file:** (Default value can be unset.) This value contains the name and location of the root certificate chain used for verifying certificates of other nodes during outbound connections. It is not required to be set, but can be used to augment the set of TLS CA certificates available from the MSPs of each channel's configuration. If you are using the recommended folder structure from the [Registering and enrolling identities with a CA](#) topic, this file can be copied into `config/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls`.

The next two parameters only need to be provided when mutual TLS is required:

- **clientAuthRequired:** Defaults to `false`. Set to `true` for a higher level of security by using **mutual TLS**, which can be configured as an extra verification step of the client-side TLS certificate. Where server-side TLS is considered the minimally necessary level of security, mutual TLS is an additional and optional level of security.
- **clientRootCAs.files:** Contains a list of additional root certificates used for verifying certificates of client connections. It augments the set of TLS CA certificates available from the MSPs of each channel's configuration. Minimally, set your organization's TLS CA root certificate so that the peer can receive join channel requests.

peer.bccsp.*

```
BCCSP:
  Default: SW
  # Settings for the SW crypto provider (i.e. when DEFAULT: SW)
  SW:
    # TODO: The default Hash and Security level needs refactoring to be
    # fully configurable. Changing these defaults requires coordination
    # SHA2 is hardcoded in several places, not only BCCSP
    Hash: SHA2
    Security: 256
    # Location of Key Store
    FileKeyStore:
      # If "", defaults to 'mspConfigPath'/keystore
    KeyStore:
  # Settings for the PKCS#11 crypto provider (i.e. when DEFAULT: PKCS11)
  PKCS11:
    # Location of the PKCS11 module library
```

(continues on next page)

(continued from previous page)

```

Library:
# Token Label
Label:
# User PIN
Pin:
Hash:
Security:

```

(Optional) This section is used to configure the Blockchain crypto provider.

- **BCCSP.Default**: If you plan to use a Hardware Security Module (HSM), then this must be set to `PKCS11`.
- **BCCSP.PKCS11.***: Provide this set of parameters according to your HSM configuration. Refer to this [example](../hsm.html) of an HSM configuration for more information.

chaincode.externalBuilders.*

```

# List of directories to treat as external builders and launchers for
# chaincode. The external builder detection processing will iterate over the
# builders in the order specified below.
externalBuilders: []
# - path: /path/to/directory
#   name: descriptive-builder-name
#   propagateEnvironment:
#     - ENVVAR_NAME_TO_PROPAGATE_FROM_PEER
#     - GOPROXY

```

(Optional) **New in Fabric 2.0.** This section is used to configure a set of paths where your chaincode builders reside. Each external builder definition must include a name (used for logging) and the path to parent of the `bin` directory containing the builder scripts. Also, you can optionally specify a list of environment variable names to propagate from the peer when it invokes the external builder scripts. For details see [Configuring external builders and launchers](#).

- **externalBuilders.path**: Specify the path to the builder.
- **externalBuilders.name**: Give this builder a name.
- **externalBuilders.propagateEnvironment**: Specify the list of environment variables that you want to propagate to your peer.

ledger.*

```

ledger:

state:
# stateDatabase - options are "goleveldb", "CouchDB"
# goleveldb - default state database stored in goleveldb.
# CouchDB - store state database in CouchDB
stateDatabase: goleveldb

couchDBConfig:
# It is recommended to run CouchDB on the same server as the peer, and
# not map the CouchDB container port to a server port in docker-compose.
# Otherwise proper security must be provided on the connection between
# CouchDB client (on the peer) and server.

```

(continues on next page)

(continued from previous page)

```
couchDBAddress: 127.0.0.1:5984

# This username must have read and write authority on CouchDB
username:

# The password is recommended to pass as an environment variable
# during start up (eg CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD).
# If it is stored here, the file must be access control protected
# to prevent unintended users from discovering the password.
password:
```

This section is used to select your ledger database type, either `goleveldb` or `CouchDB`. To avoid errors all peers should use the same database **type**. `CouchDB` is an appropriate choice when JSON queries are required. While `CouchDB` runs in a separate operating system process, there is still a 1:1 relation between a peer node and a `CouchDB` instance, meaning that each peer will have a single database and that database will only be associated with that peer. Besides the additional JSON query capability of `CouchDB`, the choice of the database is invisible to a smart contract.

- **ledger.state.stateDatabase:** (Override this value when you plan to use `CouchDB`.) Defaults to `goleveldb` which is appropriate when ledger states are simple key-value pairs. A `LevelDB` database is embedded in the peer node process.
- **ledger.state.couchDBConfig.couchDBAddress:** (Required when using `CouchDB`.) Specify the address and port where `CouchDB` is running.
- **ledger.state.couchDBConfig.username:** (Required when using `CouchDB`.) Specify the `CouchDB` user with read and write authority to the database.
- **ledger.state.couchDBConfig.password:** (Required when using `CouchDB`.) Specify the password for the `CouchDB` user with read and write authority to the database.

The `ledger` section also contains your default snapshot directory where snapshots are stored. For more information about snapshots, check out [Taking ledger snapshots and using them to join channels](#).

```
snapshots:
# Path on the file system where peer will store ledger snapshots
rootDir: /var/hyperledger/production/snapshots
```

- **ledger.snapshots.rootDir:** (Default value should be overridden.) This is the path to where snapshots are stored on the local filesystem of the peer. It can be an absolute path or relative to `FABRIC_CFG_PATH` and defaults to `/var/hyperledger/production/snapshots`. When the snapshot is taken, it is automatically organized by the status, channel name, and block number of the snapshot. For more information, check out [Taking a snapshot](#). The user running the peer needs to own and have write access to this directory.

operations.*

```
operations:
# host and port for the operations server
listenAddress: 127.0.0.1:9443

# TLS configuration for the operations endpoint
tls:
# TLS enabled
enabled: false

# path to PEM encoded server certificate for the operations server
```

(continues on next page)

(continued from previous page)

```

cert:
  file:

  # path to PEM encoded server key for the operations server
key:
  file:

  # most operations service endpoints require client authentication when TLS
  # is enabled. clientAuthRequired requires client certificate authentication
  # at the TLS layer to access all resources.
clientAuthRequired: false

  # paths to PEM encoded ca certificates to trust for client authentication
clientRootCAs:
  files: []

```

The operations service is used for monitoring the health of the peer and relies on mutual TLS to secure its communication. Therefore, you need to set `operations.tls.clientAuthRequired` to `true`. When this parameter is set to `true`, clients attempting to ascertain the health of the node are required to provide a valid certificate for authentication. If the client does not provide a certificate or the service cannot verify the client's certificate, the request is rejected. This means that the clients will need to register with the peer's TLS CA and provide their TLS signing certificate on the requests. See [The Operations Service](#) to learn more.

If you plan to use Prometheus [metrics](#) to monitor your peer, you must configure the operations service here.

In the unlikely case where two peers are running on the same node, you need to modify the addresses for the second peer to use a different port. Otherwise, when you start the second peer, it will fail to start, reporting that the addresses are already in use.

- **`operations.listenAddress`**: (Required when using the operations service.) Specify the address and port of the operations server.
- **`operations.tls.cert.file*`**: (Required when using the operations service). Can be the same file as the `peer.tls.cert.file`.
- **`operations.tls.key.file*`**: (Required when using the operations service). Can be the same file as the `peer.tls.key.file`.
- **`operations.tls.clientAuthRequired*`**: (Required when using the operations service). Must be set to `true` to enable mutual TLS between the client and the server.
- **`operations.tls.clientRootCAs.files*`**: (Required when using the operations service). Similar to the `peer.tls.clientRootCAs.files`, it contains a list of client root CA certificates that can be used to verify client certificates. If the client enrolled with the peer organization CA, then this value is the peer organization root CA cert.

metrics.*

```

metrics:
  # metrics provider is one of statsd, prometheus, or disabled
  provider: disabled
  # statsd configuration
  statsd:
    # network type: tcp or udp
    network: udp

```

(continues on next page)

(continued from previous page)

```
# statsd server address
address: 127.0.0.1:8125
```

By default this is disabled, but if you want to monitor the metrics for the peer, you need to choose either `statsd` or `Prometheus` as your metric provider. `Statsd` uses a “push” model, pushing metrics from the peer to a `statsd` endpoint. Because of this, it does not require configuration of the operations service itself. See the list of [Available metrics for the peer](#).

- **provider:** (Required to use `statsd` or `Prometheus` metrics for the peer.) Because `Prometheus` utilizes a “pull” model there is not any configuration required, beyond making the operations service available. Rather, `Prometheus` will send requests to the operations URL to poll for available metrics.
- **address:** (Required when using `statsd`.) When `statsd` is enabled, you will need to configure the host-name and port of the `statsd` server so that the peer can push metric updates.

Next steps

After deciding on your peer configuration, you are ready to deploy your peers. Follow instructions in the [Deploy the peer](#) topic for instructions on how to deploy your peer.

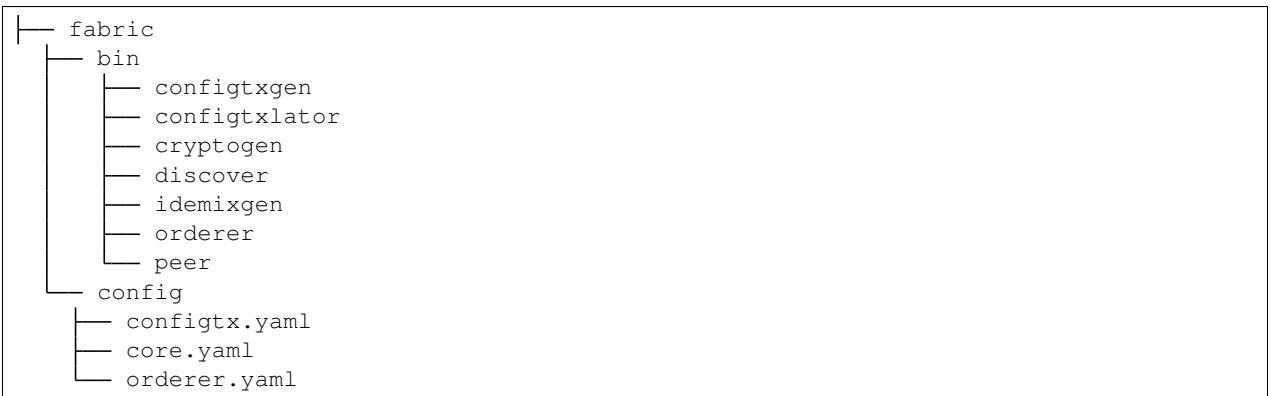
Deploy the peer

Before deploying a peer, make sure to digest the material in [Planning for a peer](#) and [Checklist for a production peer](#) which discusses all of the relevant decisions you need to make and parameters you need to configure before deploying a peer.

Note: in order for a peer to be a joined to a channel, the organization the peer belongs to must be joined to the channel. This means that you must have [created the MSP of your organization](#). The MSP ID of this organization must be the same as the ID specified at `peer.localMspId` in `core.yaml`.

Download the peer binary and configuration files

The Fabric peer binary and configuration files can be downloaded from [GitHub](#) to a folder on your local system for example `fabric/`. Scroll to the Fabric release you want to download, click the **Assets** twistie, and select the binary for your system type. Extract the ZIP file and you will find all of the Fabric binaries in the `/bin` folder and the associated configuration files in the `/config` folder. The resulting folder structure is similar to:



Along with the relevant binaries, you will receive both the peer binary executable and the peer configuration file, `core.yaml`, that is required to launch a peer on the network. The other files are not required for the peer deployment but will be useful when you attempt to create or edit channels, among other tasks.

Tip: Add the location of the peer binary to your `PATH` environment variable so that it can be picked up without fully qualifying the path to the binary executable, for example:

```
export PATH=<path to download location>/bin:$PATH
```

After you have mastered deploying and running a peer by using the peer binary and `core.yaml` configuration file, it is likely that you will want to use a peer container in a Kubernetes or Docker deployment. The Hyperledger Fabric project publishes a [peer image](#) that can be used for development and test, and various vendors provide supported peer images. For now though, the purpose of this topic is to teach you how to properly use the peer binary so you can take that knowledge and apply it to the production environment of your choice.

Prerequisites

Before you can launch a peer node in a production network, you need to make sure you've created and organized the necessary certificates, decided on storage, and configured `core.yaml`.

Certificates

Note: while **cryptogen** is a convenient utility that can be used to generate certificates for a test network, it should **never** be used on a production network. The core requirement for certificates for Fabric nodes is that they are Elliptic Curve (EC) certificates. You can use any tool you prefer to issue these certificates (for example, OpenSSL). However, the Fabric CA streamlines the process because it generates the Membership Service Providers (MSPs) for you.

Before you can deploy the peer, create the recommended folder structure for the peer certificates that is described in the [Registering and enrolling identities with a CA](#) topic to store the generated certificates and MSPs.

This folder structure isn't mandatory, but these instructions presume you have created it:

```
├─ organizations
│   └─ peerOrganizations
│       └─ org1.example.com
│           ├── msp
│           └─ peers
│               └─ peer0.org1.example.com
│                   ├── msp
│                   └─ tls
```

You should have already used your certificate authority of choice to generate the peer enrollment certificate, TLS certificate, private keys, and the MSPs that Fabric must consume. Refer to the [CA deployment](#) and [Registering and enrolling identities with a CA](#) topics for instructions on how to create a Fabric CA and how to generate these certificates. You need to generate the following sets of certificates:

- **Peer TLS CA certificates**
- **Peer local MSP (enrollment certificate and private key of the peer)**

You will either need to use the Fabric CA client to generate the certificates directly into the recommended folder structure or you will need to copy the generated certificates to their recommended folders after they are generated. Whichever method you choose, most users are ultimately likely to script this process so it can be repeatable as needed. A list of the certificates and their locations is provided here for your convenience.

TLS certificates

In order for the peer to launch successfully, make sure that the locations of the TLS certificates you specified in the [Checklist for a production peer](#) point to the correct certificates. To do this:

- Copy the TLS certificate that contains the public key that is associated with the signing (private) key certificate, which by default is called `ca-cert.pem`, to `organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/tls-cert.pem`. The path and name of the certificate corresponds to the `peer.tls.rootcert.file` parameter in `core.yaml`.
- After you have generated the peer TLS certificate, the certificate will have been generated in the `signcerts` directory, and the private key will have been generated in the `keystore` directory. Rename the generated private key in the `keystore` folder to `peer0-key.pem` so that it can more easily be recognized later on as being a private key.
- Copy the peer TLS certificate and private key to `organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls`. The path and name of the certificate and private key files correspond to the `peer.tls.cert.file` and `peer.tls.key.file` parameters in the `core.yaml`.
- If using mutual authentication (`clientAuthRequired` set to `true`), you need to indicate to the peer which TLS CA root certificates to use to authorize clients. Copy the organization's TLS CA root certificate `ca-cert.pem` to `organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca-cert.pem` so that the organization's clients will be authorized. The path and name of the certificate corresponds to `peer.tls.clientRootCAs.files` parameter in `core.yaml`. Note that multiple files can be configured, one for each client organization that will communicate with the peer (for example if other organizations will use this peer for endorsements). If `clientAuthRequired` is set to `false`, you can skip this step.

Peer local MSP (enrollment certificate and private key)

Similarly, you need to point to the [local MSP of your node](#) by copying it to `organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/msp`. This path corresponds to the value of the `peer.mspConfigPath` parameter in the `core.yaml` file. Because of the Fabric concept of “[Node Organization Unit \(OU\)](#)”, you do not need to specify an admin of the peer when bootstrapping. Rather, the role of “admin” is conferred onto an identity by setting an OU value of “admin” inside a certificate and enabled by the `config.yaml` file. When Node OUs are enabled, any organization admin will be able to administer the peer.

Note that the local MSP contains the signed certificate (public key) and the private key for the peer. The private key is used by the node to sign transactions, and is therefore not shared and must be secured. For maximum security, a Hardware Security Module (HSM) can be configured to generate and store this private key.

Storage

You must provision persistent storage for your ledger files. The following properties in `core.yaml` dictates where ledger files and snapshots are written:

- `peer.fileSystemPath` - defaults to `/var/hyperledger/production`
- `ledger.snapshots.rootDir` - defaults to `/var/hyperledger/production/snapshots`

Ensure that your peer has write access to these directories.

If you decide to use Kubernetes or Docker, recall that in a containerized environment local storage disappears when the container goes away, so you will need to provision or mount persistent storage for the ledger before you deploy a peer.

Configuration of `core.yaml`

Now you can use the [Checklist for a production peer](#) to modify the default settings in the `core.yaml` file. In the future, if you decide to deploy the peer through Kubernetes or Docker, you can override the same default settings by using environment variables instead. Check out the [note](#) in the deployment guide overview for instructions on how to construct the environment variable names for an override.

Make sure to set the value of the `FABRIC_CFG_PATH` to be the location of the `core.yaml` file. When you run the peer binary from the `fabric/bin` folder, it would point to the `/config` folder: `export FABRIC_CFG_PATH=../config`

Start the peer

After `core.yaml` has been configured and your deployment backend is ready, you can simply start the peer node with the following command:

```
cd bin
./peer node start
```

When the peer starts successfully, you should see a message similar to:

```
[nodeCmd] serve -> INFO 017 Started peer with ID=[peer0.org1.example.com], network_
↪ID=[prod], address=[peer0.org1.example.com:7060]
```

Next steps

In order to be able to transact on a network, the peer must be joined to a channel. The organization the peer belongs to must be a member of a channel before one of its peers can be joined to it. Note that if an organization wants to create a channel, it must be a member of the consortium hosted by the ordering service. If your organization has not specified at least one [anchor peer](#) on the channel, you should do so, as it will enable communication between organizations. See the [Create a channel tutorial](#) to learn more. Once a peer is joined to a channel, the Fabric chaincode lifecycle process can be used to install chaincode packages on the peer. Only peer admin identities can be used to install a package on a peer.

For high availability, you should consider deploying at least one other peer in the organization so that this peer can safely go offline for maintenance while transaction requests can continue to be addressed by the other peer. This redundant peer should be on a separate system or virtual machine in case the location where both peers are deployed goes down.

Troubleshooting peer deployment

Peer fails to start with `ERRO 001 Fatal error when initializing core config`

Problem: When launching the peer, it fails with:

```
InitCmd -> ERRO 001 Fatal error when initializing core config : Could not find config_
↪file. Please make sure that FABRIC_CFG_PATH is set to a path which contains core.
↪yaml
```

(continues on next page)

Solution:

This error occurs when the `FABRIC_CFG_PATH` is not set or is set incorrectly. Ensure that you have set the `FABRIC_CFG_PATH` environment variable to point to the location of the peer `core.yaml` file. Navigate to the folder where the `peer.exe` binary file resides and run the following command:

```
export FABRIC_CFG_PATH=../config
```

8.5.2 Creating an ordering node

Note: while it is possible to add additional nodes to an ordering service, only the process for creating an ordering service is covered in these tutorials.

If you’ve read through the key concept topic on [The Ordering Service](#), you should have a good idea of the role the ordering service plays in a network and the nature of its interactions with other network components. The ordering service is responsible for literally “ordering” endorsed transactions into blocks, which peers then validate and commit to their ledgers.

These roles are important to understand before you create an ordering service, as it will influence your customization and deployment decisions. Among the chief differences between a peer and ordering service is that in a production network, multiple ordering nodes work together to form the “ordering service” of a channel (these nodes are also known as the “consenter set”). This creates a series of important decisions that need to be made at both the node level and at the cluster level. Some of these cluster decisions are not made in individual ordering node `orderer.yaml` files but instead in the `configtx.yaml` file that is used to generate the genesis block for an application channel. For a look at the various decisions you will need to make, check out [Planning for an ordering service](#).

The configuration values in an ordering node’s `orderer.yaml` file must be customized or overridden with environment variables. You can find the default `orderer.yaml` configuration file in the [sampleconfig](#) directory of [Hyperledger Fabric](#).

This configuration file is bundled with the `orderer` image and is also included with the downloadable binaries. For information about how to download the production `orderer.yaml` along with the `orderer` image, check out [Deploy the ordering service](#).

While there are many parameters in the default `orderer.yaml`, you will only need to customize a small percentage of them. In general, if you do not have the need to change a tuning value, keep the default value.

Among the parameters in `orderer.yaml`, there are:

- **Identifiers:** these include not just the paths to the relevant local MSP and Transport Layer Security (TLS) certificates, but also the MSP ID of the organization that owns the ordering node.
- **Addresses and paths:** because ordering nodes interact with other components, you must specify a series of addresses in the configuration. These include addresses where the ordering node itself can be found by other components as well as **Operations and metrics**, which allow you to set up methods for monitoring the health and performance of your ordering node through the configuration of endpoints.

For more information about `orderer.yaml` and its specific parameters, check out [Checklist for a production ordering node](#).

Note: This tutorial assumes that a system channel genesis block will not be used when bootstrapping the `orderer`. Instead, these nodes (or a subset of them), will be joined to a channel using the process to [Create a channel](#). For information on how to create an `orderer` that will be bootstrapped with a system channel genesis block, check out [Deploy the ordering service](#) from the Fabric v2.2 documentation.

Planning for an ordering service

Audience: Architects, network operators, users setting up a production Fabric network who are familiar with Transport Layer Security (TLS), Public Key Infrastructure (PKI) and Membership Service Providers (MSPs).

Check out the conceptual topic on [The Ordering Service](#) for an overview on ordering service concepts, implementations, and the role an ordering service plays in a transaction.

In a Hyperledger Fabric network, a node or collection of nodes together form what's called an "ordering service", which literally orders transactions into blocks, which peers will then validate and commit to their ledgers. This separates Fabric from other distributed blockchains, such as Ethereum and Bitcoin, in which this ordering is done by any and all nodes.

Whereas Fabric networks that will only be used for testing and development purposes (such as our [test network](#)) often feature an ordering service made up of only one node (these nodes are typically referred to as "orderers" or "ordering nodes"), production networks require a more robust deployment of at least three nodes. For this reason, our deployment guide will feature instructions on how to create a three-node ordering service. For more guidance on the number of nodes you should deploy, check out [Cluster considerations](#).

This tutorial assumes that a system channel genesis block will not be used when bootstrapping the orderer. Instead, these nodes (or a subset of them), will be joined to a channel using the process to [Create a channel](#). For information on how to create an orderer that will be bootstrapped with a system channel genesis block, check out [Deploy the ordering service](#) from the Fabric v2.2 documentation.

Generate ordering node identities and Membership Service Providers (MSPs)

Before proceeding with this topic, you should have reviewed the process for a Deploying a Certificate Authority (CA) for your organization in order to generate the identities and MSPs for the admins and ordering nodes in your organization. To learn how to use a CA to create these identities, check out [Registering and enrolling identities with a CA](#). Note that the best practice is to register and enroll a separate node identity for each ordering node and to use distinct TLS certificates for each node.

Note that the `cryptogen` tool should never be used to generate any identities in a production scenario.

In this deployment guide, we'll assume that all ordering nodes will be created and owned by the same orderer organization. However, it is possible for multiple organizations to contribute nodes to an ordering service, both during the creation of the ordering service and after the ordering service has been created.

Folder management

While it is possible to bootstrap an ordering node using a number of folder structures for your MSPs and certificates, we do recommend the folder structure outlined in [Registering and enrolling identities with a CA](#) for the sake of consistency and repeatability. Although it is not required, these instructions will presume that you have used that folder structure.

Certificates from a non-Fabric CA

While it is possible to use a non-Fabric CA to generate identities, this process requires that you manually construct the MSP folders the ordering service and its organization need. That process will not be covered here and will instead focus on using a Fabric CA to generate the identities and MSP folders for you.

Transport Layer Security (TLS) enablement

To prevent “man in the middle” attacks and otherwise secure communications, the use of TLS is a requirement for any production network. Therefore, in addition to registering your ordering nodes identities with your organization CA, you will also need to create certificates for your ordering nodes with the TLS CA of your organization. These TLS certificates will be used by the ordering nodes when communicating with the network.

Sizing your ordering node resources

Because ordering nodes do not host a state database or chaincode, an ordering node will typically only have a single container associated with it. Like the “peer container” associated with the peer, this container encapsulates the ordering process that orders transactions into blocks for all channels on which the ordering node is a consenter (ordering nodes also validate actions in particular cases). The ordering node storage includes the blockchain for all of the channels on which the node is a consenter.

Note that, at a logical level, every “consenter set” for each channel is a separate ordering service, in which “alive” messages and other communications are duplicated. This affects the CPU and memory required for each node. Similarly, there is a direct relationship between the size of a consenter set and the amount of resources each node will need. This is because in a Raft ordering service, the nodes do not collaborate in ordering transactions. One node, a “leader” elected by the other nodes, performs all ordering and validation functions, and then replicates decisions to the other nodes. As a result, as consenter sets increase in size, there is more traffic and burden on the leader node and more communications across the consenter set.

More on this in *Cluster considerations*.

Cluster considerations

For more guidance on the number of nodes you should deploy, check out [Raft](#).

Raft is a leader based protocol, where a single leader validates transactions, orders blocks, and replicates the data out to the followers. Raft works based on the concept of a quorum in which as long as a majority of the Raft nodes are online, the Raft cluster stays available.

On the one hand, the more Raft nodes that are deployed, the more nodes can be lost while maintaining that a majority of the nodes are still available (unless a majority of nodes are available, the cluster will cease to process and create blocks). A five node cluster, for example, can tolerate two down nodes, while a seven node cluster can tolerate three down nodes.

However, more nodes means a larger communication overhead, as the leader must communicate with all of the nodes in order for the ordering service to function properly. If a node thinks it has lost connection with the leader, even if this loss of communication is only due to a networking or processing delay, it is designed to trigger a leader election. Unnecessary leader elections only add to the communications overhead for the leader, progressively escalating the burden on the cluster. And because, each channel an ordering node participates in is, logically, a separate Raft instance, an orderer participating in 100 channels is actually doing 100x the work as an ordering node in a single channel.

For these reasons, Raft clusters of more than a few dozen nodes begin to see noticeable performance degradation. Once clusters reach about 100 nodes, they begin having trouble maintaining quorum. The stage at which a deployment experiences issues is dependent on factors such as networking speeds and other resources available, and there are parameters such as the tick interval which can be used to mitigate the larger communications overhead.

The optimal number of ordering nodes for your ordering service ultimately depends on your use case, your resources, and your topology. However, clusters of three, five, seven, or nine nodes, are the most popular, with no more than about 50 channels per orderer.

Storage considerations and monitoring

The storage that should be allocated to an ordering node depends on factors such as the expected transaction throughput, the size of blocks, and number of channels the node will be joined to. Your needs will depend on your use case. However, the best practice is to monitor the storage available to your nodes closely. You may also decide to enable an autoscaler, which will allocate more resources to your node, if your infrastructure allows it.

If the storage for an ordering node is exhausted you also have the option to deploy a new node with a larger storage allocation and allow it to sync with the relevant ledgers. If you have several ordering nodes available to use, ensure that each node is a consenter on approximately the same number of channels.

In a production environment you should also monitor the CPU and memory allocated to an ordering node using widely available tooling. If you see an ordering node struggling to keep up (for example, it might be calling for leader elections when none is needed), it is a sign that you might need to increase its resource allocation.

Checklist for a production ordering node

As you prepare to build a production ordering service (or a single ordering node), you need to customize the configuration by editing the `orderer.yaml` file, which is copied into the `/config` directory when downloading the Fabric binaries, and available within the Fabric ordering node image at `/etc/hyperledger/fabric/orderer.yaml`.

While in a production environment you could override the environment variables in the `orderer.yaml` file in your Docker container or your Kubernetes job, these instructions show how to edit `orderer.yaml` instead. It's important to understand the parameters in the configuration file and their dependencies on other parameter settings in the file. Blindly overriding one setting using an environment variable could affect the functionality of another setting. Therefore, the recommendation is that before starting the ordering node, you make the modifications to the settings in the configuration file to become familiar with the available settings and how they work. Afterwards, you may choose to override these parameters using environment variables.

This checklist covers key configuration parameters for setting up a production ordering service. Of course, you can always refer to the `orderer.yaml` file for additional parameters or more information. It also provides guidance on which parameters should be overridden. The list of parameters that you need to understand and that are described in this topic include:

- *General.ListenAddress*
- *General.ListenPort*
- *General.TLS.**
- *General.Keepalive.**
- *General.Cluster.**
- *General.BootstrapMethod*
- *General.BootstrapFile*
- *General.LocalMSPDir*
- *General.LocalMSPID*
- *FileLedger.Location*
- *Operations.**
- *Metrics.**
- *Admin.**
- *ChannelParticipation.**

- *Consensus.**

General.ListenAddress

```
# Listen address: The IP on which to bind to listen.  
ListenAddress: 127.0.0.1
```

- **ListenAddress:** (default value should be overridden) This is the location where the orderer will listen, for example, 0.0.0.0. Note: unlike the peer, the `orderer.yaml` separates the address and the port, hence the *General.ListenPort* parameter.

General.ListenPort

```
# Listen port: The port on which to bind to listen.  
ListenPort: 7050
```

- **ListenPort:** (default value should be overridden) This is the port that the orderer listens on.

General.TLS

```
# Require server-side TLS  
Enabled: false  
# PrivateKey governs the file location of the private key of the TLS certificate.  
PrivateKey: tls/server.key  
# Certificate governs the file location of the server TLS certificate.  
Certificate: tls/server.crt  
# RootCAs contains a list additional root certificates used for verifying certificates  
# of other orderer nodes during outbound connections.  
# It is not required to be set, but can be used to augment the set of TLS CA_  
↪certificates  
# available from the MSPs of each channel's configuration.  
RootCAs:  
- tls/ca.crt  
# Require client certificates / mutual TLS for inbound connections.  
ClientAuthRequired: false  
# If mutual TLS is enabled, ClientRootCAs contains a list of additional root_  
↪certificates  
# used for verifying certificates of client connections.  
# It is not required to be set, but can be used to augment the set of TLS CA_  
↪certificates  
# available from the MSPs of each channel's configuration.  
ClientRootCAs:
```

- **Enabled:** (default value should be overridden) In a production network, you should be using TLS-secured communications. This value should be `true`.
- **PrivateKey:** (default value should be overridden). Provide the path to, and filename of, the private key generated by your TLS CA for this node.
- **Certificate:** (default value should be overridden) Provide the path to, and filename of, the public certificate (also known as the sign certificate) generated by your TLS CA for this node.

- **RootCAs:** (should be commented out) This parameter is typically unset for normal use. It is a list of the paths to additional root certificates used for verifying certificates of other orderer nodes during outbound connections. It can be used to augment the set of TLS CA certificates available from the MSPs of each channel's configuration.
- **ClientAuthRequired:** (Mutual TLS only) Setting this value to "true" will enable mutual TLS on your network, and must be done for the entire network, not just one node.
- **ClientRootCAs:** (Mutual TLS only) Can be left blank if mutual TLS is disabled. If mutual TLS is enabled, this is a list of the paths to additional root certificates used for verifying certificates of client connections. It can be used to augment the set of TLS CA certificates available from the MSPs of each channel's configuration.

General.KeepAlive

The KeepAlive values might need to be tuned for compatibility with any networking devices or software (like firewalls or proxies) in between components of your network. Ideally, these settings would be manipulated if needed in a test or pre-prod environment and then set accordingly for your production environment.

```
# ServerMinInterval is the minimum permitted time between client pings.
# If clients send pings more frequently, the server will
# disconnect them.
ServerMinInterval: 60s
# ServerInterval is the time between pings to clients.
ServerInterval: 7200s
# ServerTimeout is the duration the server waits for a response from
# a client before closing the connection.
ServerTimeout: 20s
```

- **ServerMinInterval:** (default value should not be overridden, unless determined necessary through testing)
- **ServerInterval:** (default value should not be overridden, unless determined necessary through testing)
- **ServerTimeout:** (default value should not be overridden, unless determined necessary through testing)

General.Cluster

```
# SendBufferSize is the maximum number of messages in the egress buffer.
# Consensus messages are dropped if the buffer is full, and transaction
# messages are waiting for space to be freed.
SendBufferSize: 10
# ClientCertificate governs the file location of the client TLS certificate
# If not set, the server General.TLS.Certificate is re-used.
ClientCertificate:
# If not set, the server General.TLS.PrivateKey is re-used.
ClientPrivateKey:
# The below 4 properties should be either set together, or be unset together.
# If they are set, then the orderer node uses a separate listener for intra-cluster
# communication. If they are unset, then the general orderer listener is used.
# This is useful if you want to use a different TLS server certificates on the
# client-facing and the intra-cluster listeners.

# ListenPort defines the port on which the cluster listens to connections.
ListenPort:
# ListenAddress defines the IP on which to listen to intra-cluster communication.
ListenAddress:
# ServerCertificate defines the file location of the server TLS certificate used for
↪ intra-cluster
```

(continues on next page)

(continued from previous page)

```
# communication.
ServerCertificate:
# ServerPrivateKey defines the file location of the private key of the TLS_
↪certificate.
ServerPrivateKey:
```

If unset, the `ClientCertificate` and `ClientPrivateKey` default to the server `General.TLS.Certificate` and `General.TLS.PrivateKey` when the orderer is not configured to use a separate cluster port.

- **ClientCertificate:** Provide the path to, and filename of, the public certificate (also known as a signed certificate) generated by your TLS CA for this node.
- **ClientPrivateKey:** Provide the path to, and filename of, the private key generated by your TLS CA for this node.

In general, these four parameters would only need to be configured if you want to configure a separate listener and TLS certificates for intra-cluster communication (with other Raft orderers), as opposed to using the listener that peer clients and application clients utilize. This is an advanced deployment option. These four parameters should be set together or left unset, and if they are set, note that the `ClientCertificate` and `ClientPrivateKey` must be set as well.

- **ListenPort**
- **ListenAddress**
- **ServerCertificate**
- **ServerPrivateKey**

General.BootstrapMethod

```
# Bootstrap method: The method by which to obtain the bootstrap block
# system channel is specified. The option can be one of:
# "file" - path to a file containing the genesis block or config block of system_
↪channel
# "none" - allows an orderer to start without a system channel configuration
BootstrapMethod: file
```

- **BootstrapMethod:** If you plan to create this node on a network that is not using a system channel, override this value to `none` and then ensure that `ChannelParticipation.Enabled` is set to `true`, otherwise you will get an error when you attempt to start the node. If you are creating a node to be joined to a system channel, unless you plan to use a file type other than “file”, this value should be left as is.

General.BootstrapFile

```
# Bootstrap file: The file containing the bootstrap block to use when
# initializing the orderer system channel and BootstrapMethod is set to
# "file". The bootstrap file can be the genesis block, and it can also be
# a config block for late bootstrap of some consensus methods like Raft.
# Generate a genesis block by updating $FABRIC_CFG_PATH/configtx.yaml and
# using configtxgen command with "-outputBlock" option.
# Defaults to file "genesisblock" (in $FABRIC_CFG_PATH directory) if not specified.
BootstrapFile:
```

- **BootstrapFile:** (if you are creating this node to be joined to a system channel, the default value should be overridden) Specify the location and name of the system channel genesis block to use when this node is created. If you are creating this node without using a system channel, this value will not be used, and can therefore be left blank.

General.LocalMSPDir

```
# LocalMSPDir is where to find the private crypto material needed by the
# orderer. It is set relative here as a default for dev environments but
# should be changed to the real location in production.
LocalMSPDir: msp
```

LocalMSPDir: (default value will often be overridden) This is the path to the ordering node's local MSP, which must be created before it can be deployed. The path can be absolute or relative to FABRIC_CFG_PATH (by default, it is /etc/hyperledger/fabric in the orderer image). Unless an absolute path is specified to a folder named something other than "msp", the ordering node defaults to looking for a folder called "msp" at the path (in other words, FABRIC_CFG_PATH/msp) and when using the orderer image: /etc/hyperledger/fabric/msp. If you are using the recommended folder structure described in the [Registering and enrolling identities with a CA](#) topic, it would be relative to the FABRIC_CFG_PATH as follows: config/organizations/ordererOrganizations/org0.example.com/orderers/orderer0.org0.example.com/msp. **The best practice is to store this data in persistent storage.** This prevents the MSP from being lost if your orderer containers are destroyed for some reason.

General.LocalMSPID

```
# LocalMSPID is the identity to register the local MSP material with the MSP
# manager. IMPORTANT: The local MSP ID of an orderer needs to match the MSP
# ID of one of the organizations defined in the orderer system channel's
# /Channel/Orderer configuration. The sample organization defined in the
# sample configuration provided has an MSP ID of "SampleOrg".
LocalMSPID: SampleOrg
```

- **LocalMSPID:** (default value should be overridden) This identifies the organization this ordering node belongs to. The MSP ID must match the orderer organization MSP ID that exists in the configuration of any channel this joined will be joined to.

General.BCCSP.*

```
# Default specifies the preferred blockchain crypto service provider
# to use. If the preferred provider is not available, the software
# based provider ("SW") will be used.
# Valid providers are:
# - SW: a software based crypto provider
# - PKCS11: a CA hardware security module crypto provider.
Default: SW

# SW configures the software based blockchain crypto provider.
SW:
# TODO: The default Hash and Security level needs refactoring to be
# fully configurable. Changing these defaults requires coordination
# SHA2 is hardcoded in several places, not only BCCSP
```

(continues on next page)

(continued from previous page)

```

Hash: SHA2
Security: 256
# Location of key store. If this is unset, a location will be
# chosen using: 'LocalMSPDir'/keystore
FileKeyStore:
    KeyStore:

```

(Optional) This section is used to configure the Blockchain crypto provider.

- **BCCSP.Default**: If you plan to use a Hardware Security Module (HSM), then this must be set to PKCS11.

```

# Settings for the PKCS#11 crypto provider (i.e. when DEFAULT: PKCS11)
PKCS11:
    # Location of the PKCS11 module library
    Library:
    # Token Label
    Label:
    # User PIN
    Pin:
    Hash:
    Security:
    FileKeyStore:
        KeyStore:

```

- **BCCSP.PKCS11.***: Provide this set of parameters according to your HSM configuration. Refer to this [example](#) of an HSM configuration for more information.

FileLedger.Location

```

# Location: The directory to store the blocks in.
Location: /var/hyperledger/production/orderer

```

- **Location**: (default value should be overridden in the unlikely event where two ordering nodes are running on the same node) Every channel on which the node is a consenter will have its own subdirectory at this location. The user running the orderer needs to own and have write access to this directory. **The best practice is to store this data in persistent storage.** This prevents the ledger from being lost if your orderer containers are destroyed for some reason.

Operations.*

The operations service is used for monitoring the health of the ordering node and relies on mutual TLS to secure its communication. Therefore, you need to set `operations.tls.clientAuthRequired` to `true`. When this parameter is set to `true`, clients attempting to ascertain the health of the node are required to provide a valid certificate for authentication. If the client does not provide a certificate or the service cannot verify the client's certificate, the request is rejected. This means that the clients will need to register with the ordering node's TLS CA and provide their TLS signing certificate on the requests. See [The Operations Service](#) to learn more.

If you plan to use Prometheus [metrics](#) to monitor your ordering node, you must configure the operations service here.

In the unlikely case where two ordering nodes are running on the same node on your infrastructure, you need to modify the addresses for the second ordering node to use a different port. Otherwise, when you start the second ordering node, it will fail to start, reporting that the addresses are already in use.

```
# host and port for the operations server
ListenAddress: 127.0.0.1:8443

# TLS configuration for the operations endpoint
TLS:
  # TLS enabled
  Enabled: false

  # Certificate is the location of the PEM encoded TLS certificate
  Certificate:

  # PrivateKey points to the location of the PEM-encoded key
  PrivateKey:

  # Most operations service endpoints require client authentication when TLS
  # is enabled. ClientAuthRequired requires client certificate authentication
  # at the TLS layer to access all resources.
  ClientAuthRequired: false

  # Paths to PEM encoded ca certificates to trust for client authentication
  ClientRootCAs: []
```

- **ListenAddress:** (required when using the operations service) Specify the address and port of the operations server.
- **Enabled:** (required when using the operations service) Must be `true` if the operations service is being used.
- **Certificate:** (required when using the operations service) Can be the same file as the `General.TLS.Certificate`.
- **PrivateKey:** (required when using the operations service) Can be the same file as the `General.TLS.PrivateKey`.
- **ClientAuthRequired:** (required when using the operations service) Must be set to `true` to enable mutual TLS between the client and the server.
- **ClientRootCAs:** (required when using the operations service) Similar to the client root CA cert file in TLS, it contains a list of client root CA certificates that can be used to verify client certificates. If the client enrolled with the orderer organization CA, then this value is the orderer organization root CA cert.

Metrics.*

By default this is disabled, but if you want to monitor the metrics for the orderer, you need to use StatsD or Prometheus as your metric provider. StatsD uses a “push” model, pushing metrics from the ordering node to a StatsD endpoint. Because of this, it does not require configuration of the operations service itself. Prometheus metrics, by contrast, are pulled from an ordering node.

For more information about the available Prometheus metrics, check out [Prometheus](#)

For more information about the available StatsD metrics, check out [StatsD](#).

Because Prometheus utilizes a “pull” model there is not any configuration required, beyond making the operations service available. Rather, Prometheus will send requests to the operations URL to poll for available metrics.

```
# The metrics provider is one of statsd, prometheus, or disabled
Provider: disabled

# The statsd configuration
```

(continues on next page)

(continued from previous page)

```
Statsd:
  # network type: tcp or udp
  Network: udp

  # the statsd server address
  Address: 127.0.0.1:8125

  # The interval at which locally cached counters and gauges are pushed
  # to statsd; timings are pushed immediately
  WriteInterval: 30s

  # The prefix is prepended to all emitted statsd metrics
  Prefix:
```

- **Provider:** Set this value to statsd if using StatsD or prometheus if using Prometheus.
- **Statsd.Address:** (required to use StatsD metrics for the ordering node) When StatsD is enabled, you will need to configure the hostname and port of the StatsD server so that the ordering node can push metric updates.

Admin.*

```
Admin:
  # host and port for the admin server
  ListenAddress: 127.0.0.1:9443

  # TLS configuration for the admin endpoint
  TLS:
    # TLS enabled
    Enabled: false

    # Certificate is the location of the PEM encoded TLS certificate
    Certificate:

    # PrivateKey points to the location of the PEM-encoded key
    PrivateKey:

    # Most admin service endpoints require client authentication when TLS
    # is enabled. ClientAuthRequired requires client certificate authentication
    # at the TLS layer to access all resources.
    #
    # NOTE: When TLS is enabled, the admin endpoint requires mutual TLS. The
    # orderer will panic on startup if this value is set to false.
    ClientAuthRequired: true

    # Paths to PEM encoded ca certificates to trust for client authentication
    ClientRootCAs: []
```

- **ListenAddress:** The orderer admin server address (host and port) that can be used by the `osnadmin` command to configure channels on the ordering service. This value should be a unique `host:port` combination to avoid conflicts.
- **TLS.Enabled:** Technically this can be set to `false`, but this is not recommended. In general, you should always set this value to `true`.
- **TLS.Certificate:** The path to and file name of the orderer signed certificate issued by the TLS CA.

- **TLS.PrivateKey:** The path to and file name of the orderer private key issued by the TLS CA.
- **TLS.ClientAuthRequired:** This value must be set to `true`. Note that while mutual TLS is required for all operations on the orderer Admin endpoint, the entire network is not required to use mutual TLS.
- **TLS.ClientRootCAs:** The path to and file name of the admin client TLS CA root certificate.

ChannelParticipation.*

```
ChannelParticipation:
    # Channel participation API is enabled.
    Enabled: false

    # The maximum size of the request body when joining a channel.
    MaxRequestBodySize: 1 MB
```

- **Enabled:** If you are bootstrapping the ordering node with a system channel genesis block, this value can be set to either `true` or `false` (setting the value to `true` allows you to list channels and to migrate away from the system channel in the future). If you are **not** bootstrapping the ordering node with a system channel genesis block, this value must be set to `true` and the `General.BootstrapMethod` should be set to `none`.
- **MaxRequestBodySize:** (default value should not be overridden) This value controls the maximum size a configuration block can be and be accepted by this ordering node. Most configuration blocks are smaller than 1 MB, but if for some reason a configuration block is too large to be accept, bring down the node, increase this value, and restart the node.

Consensus.*

The values of this section vary by consensus plugin. The values below are for the etcdraft consensus plugin. If you are using a different consensus plugin, refer to its documentation for allowed keys and recommended values.

```
# The allowed key-value pairs here depend on consensus plugin. For etcd/raft,
# we use following options:

# WALDir specifies the location at which Write Ahead Logs for etcd/raft are
# stored. Each channel will have its own subdir named after channel ID.
WALDir: /var/hyperledger/production/orderer/etcdraft/wal

# SnapDir specifies the location at which snapshots for etcd/raft are
# stored. Each channel will have its own subdir named after channel ID.
SnapDir: /var/hyperledger/production/orderer/etcdraft/snapshot
```

- **WALDir:** (default value should be overridden) This is the path to the write ahead logs on the local filesystem of the ordering node. It can be an absolute path or relative to `FABRIC_CFG_PATH`. It defaults to `/var/hyperledger/production/orderer/etcdraft/wal`. Each channel will have its own subdirectory named after the channel ID. The user running the ordering node needs to own and have write access to this directory. **The best practice is to store this data in persistent storage.** This prevents the write ahead log from being lost if your orderer containers are destroyed for some reason.
- **SnapDir:** (default value should be overridden) This is the path to the snapshots on the local filesystem of the ordering node. For more information about how snapshots work in a Raft ordering service, check out [Snapshots](#). It can be an absolute path or relative to `FABRIC_CFG_PATH`. It defaults to `/var/hyperledger/production/orderer/etcdraft/wal`. Each channel will have its own subdirectory named after the channel ID. The user running the ordering node needs to own and have write access to this directory. **The best**

practice is to store this data in persistent storage. This prevents snapshots from being lost if your orderer containers are destroyed for some reason.

For more information about ordering node configuration, including how to set parameters that are not available in `orderer.yaml`, check out [Configuring and operating a Raft ordering service](#).

Deploy the ordering service

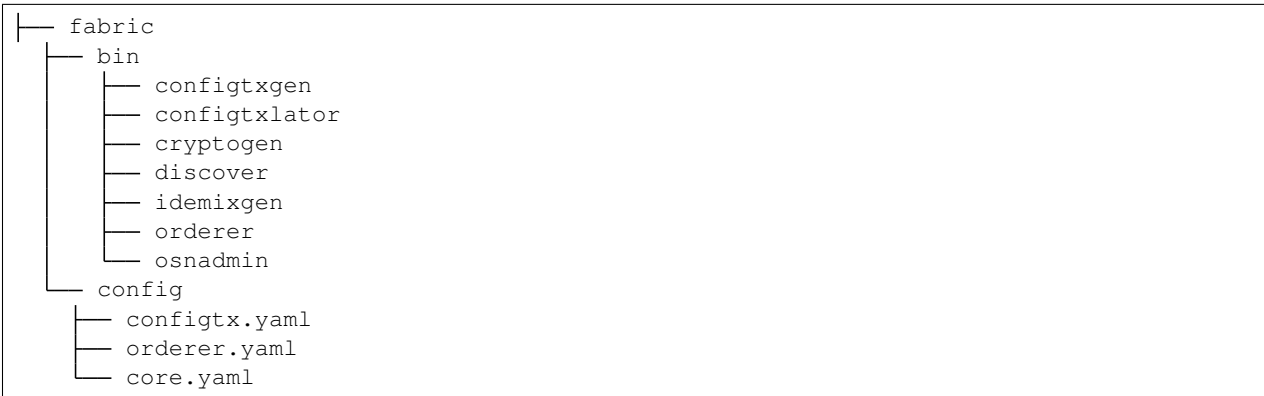
Before deploying an ordering service, review the material in [Planning for an ordering service](#) and [Checklist for a production ordering service](#), which discusses all of the relevant decisions you need to make and parameters you need to configure before deploying an ordering service.

This tutorial is based on the Raft consensus protocol and can be used to build an ordering service, which is comprised of ordering nodes, or “orderers”. It describes the process to create a three-node Raft ordering service where all of the ordering nodes belong to the same organization. This tutorial assumes that a system channel genesis block will not be used when bootstrapping the orderer. Instead, these nodes (or a subset of them), will be joined to a channel using the process to [Create a channel](#).

For information on how to create an orderer that will be bootstrapped with a system channel genesis block, check out [Deploy the ordering service](#) from the Fabric v2.2 documentation.

Download the ordering service binary and configuration files

To get started, download the Fabric binaries from [GitHub](#) to a folder on your local system, for example `fabric/`. In GitHub, scroll to the Fabric release you want to download, click the **Assets** twistie, and select the binary for your system type. After you extract the `.zip` file, you will find all of the Fabric binaries in the `/bin` folder and the associated configuration files in the `/config` folder. The resulting folder structure is similar to:



Along with the relevant binary file, the orderer configuration file, `orderer.yaml` is required to launch an orderer on the network. The other files are not required for the orderer deployment but are useful when you attempt to create or edit channels, among other tasks.

Tip: Add the location of the orderer binary to your `PATH` environment variable so that it can be picked up without fully qualifying the path to the binary executable, for example:

```
export PATH=<path to download location>/bin:$PATH
```

After you have mastered deploying and running an ordering service by using the orderer binary and `orderer.yaml` configuration file, it is likely that you will want to use an orderer container in a Kubernetes or Docker deployment. The Hyperledger Fabric project publishes an [orderer image](#) that can be used for development and test, and various vendors provide supported orderer images. For now though, the purpose of this topic is to teach you how to properly use the orderer binary so you can take that knowledge and apply it to the production environment of your choice.

Prerequisites

Before you can launch an orderer in a production network, you need to make sure you've created and organized the necessary certificates, generate the genesis block, decided on storage, and configured `orderer.yaml`.

Certificates

While **cryptogen** is a convenient utility that can be used to generate certificates for a test environment, it should **never** be used on a production network. The core requirement for certificates for Fabric nodes is that they are Elliptic Curve (EC) certificates. You can use any tool you prefer to issue these certificates (for example, OpenSSL). However, the Fabric CA streamlines the process because it generates the Membership Service Providers (MSPs) for you.

Before you can deploy the orderer, create the recommended folder structure for the orderer or orderer certificates that is described in the [Registering and enrolling identities with a CA](#) topic to store the generated certificates and MSPs.

This folder structure isn't mandatory, but these instructions presume you have created it:

```

├─ organizations
│   └─ ordererOrganizations
│       └─ ordererOrg1.example.com
│           ├── msp
│           │   ├── cacerts
│           │   └─ tlscacerts
│           └─ orderers
│               └─ orderer0.ordererOrg1.example.com
│                   ├── msp
│                   └─ tls

```

You should have already used your certificate authority of choice to generate the orderer enrollment certificate, TLS certificate, private keys, and the MSPs that Fabric must consume. Refer to the [CA deployment guide](#) and [Registering and enrolling identities with a CA](#) topics for instructions on how to create a Fabric CA and how to generate these certificates. You need to generate the following sets of certificates:

- **Orderer organization MSP**
- **Orderer TLS CA certificates**
- **Orderer local MSP (enrollment certificate and private key of the orderer)**

You will either need to use the Fabric CA client to generate the certificates directly into the recommended folder structure or you will need to copy the generated certificates to their recommended folders after they are generated. Whichever method you choose, most users are ultimately likely to script this process so it can be repeated as needed. A list of the certificates and their locations is provided here for your convenience.

If you are using a containerized solution for running your network (which for obvious reasons is a popular choice), **it is a best practice to mount volumes for the certificate directories external to the container where the node itself is running. This will allow the certificates to be used by an ordering node container, regardless whether the ordering node container goes down, becomes corrupted, or is restarted.**

TLS certificates

For the ordering node to launch successfully, the locations of the TLS certificates you specified in the [Checklist for a production ordering node](#) must point to the correct certificates. To do this:

- Copy the **TLS CA Root certificate**, which by default is called `ca-cert.pem`, to the orderer organization MSP definition `organizations/ordererOrganizations/ordererOrg1.example.com/msp/tlscacerts/tls-cert.pem`.

- Copy the **CA Root certificate**, which by default is called `ca-cert.pem`, to the orderer organization MSP definition `organizations/ordererOrganizations/ordererOrg1.example.com/msp/cacerts/ca-cert.pem`.
- When you enroll the orderer identity with the TLS CA, the public key is generated in the `signcerts` folder, and the private key is located in the `keystore` directory. Rename the private key in the `keystore` folder to `orderer0-tls-key.pem` so that it can be easily recognized later as the TLS private key for this node.
- Copy the orderer TLS certificate and private key files to `organizations/ordererOrganizations/ordererOrg1.example.com/orderers/orderer0.ordererOrg1.example.com/tls`. The path and name of the certificate and private key files correspond to the values of the `General.TLS.Certificate` and `General.TLS.PrivateKey` parameters in the `orderer.yaml`.

Note: Don't forget to create the `config.yaml` file and add it to the organization MSP and local MSP folder for each ordering node. This file enables Node OU support for the MSP, an important feature that allows the MSP's admin to be identified based on an "admin" OU in an identity's certificate. Learn more in the [Fabric CA](#) documentation.

If you are using a containerized solution for running your network (which for obvious reasons is a popular choice), **it is a best practice to mount volumes for the certificate directories external to the container where the node itself is running**. This will allow the certificates to be used by an ordering node container, regardless whether the ordering node container goes down, becomes corrupted, or is restarted.

Orderer local MSP (enrollment certificate and private key)

Similarly, you need to point to the [local MSP of your orderer](#) by copying the MSP folder to `organizations/ordererOrganizations/ordererOrg1.example.com/orderers/orderer0.ordererOrg1.example.com/msp`. This path corresponds to the value of the `General.LocalMSPDir` parameter in the `orderer.yaml` file. Because of the Fabric concept of "[Node Organization Unit \(OU\)](#)", you do not need to specify an admin of the orderer when bootstrapping. Rather, the role of "admin" is conferred onto an identity by setting an OU value of "admin" inside a certificate and enabled by the `config.yaml` file. When Node OUs are enabled, any admin identity from this organization will be able to administer the orderer.

Note that the local MSP contains the signed certificate (public key) and the private key for the orderer. The private key is used by the node to sign transactions, and is therefore not shared and must be secured. For maximum security, a Hardware Security Module (HSM) can be configured to generate and store this private key.

Storage

You must provision persistent storage for your ledgers. The default location for the ledger is located at `/var/hyperledger/production/orderer`. Ensure that your orderer has write access to the folder. If you choose to use a different location, provide that path in the `FileLedger:` parameter in the `orderer.yaml` file. If you decide to use Kubernetes or Docker, recall that in a containerized environment, local storage disappears when the container goes away, so you will need to provision or mount persistent storage for the ledger before you deploy an orderer.

Configuration of `orderer.yaml`

Now you can use the [Checklist for a production orderer](#) to modify the default settings in the `orderer.yaml` file. In the future, if you decide to deploy the orderer through Kubernetes or Docker, you can override the same default settings by using environment variables instead. Check out the [note](#) in the deployment guide overview for instructions on how to construct the environment variable names for an override.

At a minimum, you need to configure the following parameters:

- `General.ListenAddress` - Hostname that the ordering node listens on.

- `General.ListenPort` - Port that the ordering node listens on.
- `General.TLS.Enabled: true` - Server-side TLS should be enabled in all production networks.
- `General.TLS.PrivateKey` - Ordering node private key from TLS CA.
- `General.TLS.Certificate` - Ordering node signed certificate (public key) from the TLS CA.
- `General.TLS.RootCAS` - This value should be unset.
- `General.BootstrapMethod:none` - This allows the orderer to start without needing a system channel configuration block.
- `General.LocalMSPDir` - Path to the ordering node MSP folder.
- `General.LocalMSPID` - MSP ID of the ordering organization as specified in the channel configuration.
- `FileLedger.Location` - Location on the file system to the ledgers of the channels this orderer will be servicing.
- `ChannelParticipation.Enabled` - Set to `true`. This allows the orderer to be joined to an application channel without joining a system channel first.

Because this tutorial assumes that a system channel genesis block will not be used when bootstrapping the orderer, the following additional parameters are required if you want to create an application channel with the `osnadmin` command.

- `Admin.ListenAddress` - The orderer admin server address (host and port) that can be used by the `osnadmin` command to configure channels on the ordering service. This value should be a unique `host:port` combination to avoid conflicts.
- `Admin.TLS.Enabled:` - Technically this can be set to `false`, but this is not recommended. In general, you should always set this value to `true`.
- `Admin.TLS.PrivateKey:` - The path to and file name of the orderer private key issued by the TLS CA.
- `Admin.TLS.Certificate:` - The path to and file name of the orderer signed certificate issued by the TLS CA.
- `Admin.TLS.ClientAuthRequired:` This value must be set to `true`. Note that while mutual TLS is required for all operations on the orderer Admin endpoint, the entire network is not required to use Mutual TLS.
- `Admin.TLS.ClientRootCAs:` - The path to and file name of the admin client TLS CA Root certificate. In the folder structure above, this is `admin-client/client-tls-ca-cert.pem`.

Start the orderer

Make sure you have set the value of the `FABRIC_CFG_PATH` to be the location of the `orderer.yaml` file relative to where you are invoking the orderer binary. For example, if you run the orderer binary from the `fabric/bin` folder, it would point to the `/config` folder: `export FABRIC_CFG_PATH=../config`

After `orderer.yaml` has been configured and your deployment backend is ready, you can simply start the orderer node with the following command:

```
cd bin
./orderer start
```

When the orderer starts successfully, you should see a message similar to:

```
INFO 01d Registrar initializing without a system channel, number of application_
↪channels: 0, with 0 consensus.Chain(s) and 0 follower.Chain(s)
INFO 01f Starting orderer:
```

You have successfully started one node, you now need to repeat these steps to configure and start the other two orderers.

Next steps

Your ordering service is started and ready to order transactions into blocks. Depending on your use case, you may need to add or remove orderers from the consenter set, or other organizations may want to contribute their own orderers to the cluster. See the topic on ordering service [reconfiguration](#) for considerations and instructions.

Once your nodes have been created, you are ready to join them to a channel. Check out [Create a channel](#) for more information.

Troubleshooting

When you start the orderer, it fails with the following error:

```
ERRO 001 failed to parse config: Error reading configuration: Unsupported Config_
↪Type ""
```

Solution:

Your `FABRIC_CFG_PATH` is not set. Run the following command to set it to the location of your `orderer.yaml` file.

```
export FABRIC_CFG_PATH=<PATH_TO_ORDERER_YAML>
```

When you start the orderer, it fails with the following error:

```
PANI 003 Failed to setup local msp with config: administrators must be declared when_
↪no admin ou classification is set
```

Solution:

Your local MSP definition is missing the `config.yaml` file. Create the file and copy it into the local MSP `/msp` folder of orderer. See the [Fabric CA](#) documentation for more instructions.

When you start the orderer, it fails with the following error:

```
PANI 27c Failed creating a block puller: client certificate isn't in PEM format: _
↪channel=mychannel node=3
```

Solution:

Your `orderer.yaml` file is missing the `General.Cluster.ClientCertificate` and `General.Cluster.ClientPrivateKey` definitions. Provide the path to and filename of the public certificate (also known as a signed certificate) and private key generated by your TLS CA for the orderer in these two fields and restart the node.

8.6 Next steps

Blockchain networks are all about connection, so once you've deployed nodes, you'll obviously want to connect them to other nodes! If you have a peer organization and a peer, you'll want to join your organization to a consortium and join or [Create a channel](#). If you have an ordering node, you will want to add peer organizations to your consortium. You'll also want to learn how to develop chaincode, which you can learn about in the topics [The scenario](#) and [Writing Your First Chaincode](#).

Part of the process of connecting nodes and creating channels will involve modifying policies to fit the use cases of business networks. For more information about policies, check out [Policies](#).

One of the common tasks in a Fabric will be the editing of existing channels. For a tutorial about that process, check out [Updating a channel configuration](#). One popular channel update is to add an org to an existing channel. For a tutorial about that specific process, check out [Adding an Org to a Channel](#). For information about upgrading nodes after they have been deployed, check out [Upgrading your components](#).

9.1 Membership Service Providers (MSP)

For a conceptual overview of the Membership Service Provider (MSP), check out [Membership Service Provider \(MSP\)](#).

This topic elaborates on the setup of the MSP implementation supported by Hyperledger Fabric and discusses best practices concerning its use.

9.1.1 MSP configuration

To setup an instance of the MSP, its configuration needs to be specified locally at each peer and orderer (to enable peer and orderer signing), and on the channels to enable peer, orderer, client identity validation, and respective signature verification (authentication) by and for all channel members.

Firstly, for each MSP a name needs to be specified in order to reference that MSP in the network (e.g. `msp1`, `org2`, and `org3.divA`). This is the name under which membership rules of an MSP representing a consortium, organization or organization division is to be referenced in a channel. This is also referred to as the *MSP Identifier* or *MSP ID*. MSP Identifiers are required to be unique per MSP instance.

In the case of the default MSP implementation, a set of parameters need to be specified to allow for identity (certificate) validation and signature verification. These parameters are deduced by [RFC5280](#), and include:

- A list of self-signed (X.509) CA certificates to constitute the *root of trust*
- A list of X.509 certificates to represent intermediate CAs this provider considers for certificate validation; these certificates ought to be certified by exactly one of the certificates in the root of trust; intermediate CAs are optional parameters
- A list of X.509 certificates representing the administrators of this MSP with a verifiable certificate path to exactly one of the CA certificates of the root of trust; owners of these certificates are authorized to request changes to this MSP configuration (e.g. root CAs, intermediate CAs)

- A list of Organizational Units that valid members of this MSP should include in their X.509 certificate; this is an optional configuration parameter, used when, e.g., multiple organizations leverage the same root of trust, and intermediate CAs, and have reserved an OU field for their members
- A list of certificate revocation lists (CRLs) each corresponding to exactly one of the listed (intermediate or root) MSP Certificate Authorities; this is an optional parameter
- A list of self-signed (X.509) certificates to constitute the *TLS root of trust* for TLS certificates.
- A list of X.509 certificates to represent intermediate TLS CAs this provider considers; these certificates ought to be certified by exactly one of the certificates in the TLS root of trust; intermediate CAs are optional parameters.

Valid identities for this MSP instance are required to satisfy the following conditions:

- They are in the form of X.509 certificates with a verifiable certificate path to exactly one of the root of trust certificates;
- They are not included in any CRL;
- And they *list* one or more of the Organizational Units of the MSP configuration in the OU field of their X.509 certificate structure.

For more information on the validity of identities in the current MSP implementation, we refer the reader to `msp-identity-validity-rules`.

In addition to verification related parameters, for the MSP to enable the node on which it is instantiated to sign or authenticate, one needs to specify:

- The signing key used for signing by the node (currently only ECDSA keys are supported), and
- The node's X.509 certificate, that is a valid identity under the verification parameters of this MSP.

It is important to note that MSP identities never expire; they can only be revoked by adding them to the appropriate CRLs. Additionally, there is currently no support for enforcing revocation of TLS certificates.

9.1.2 How to generate MSP certificates and their signing keys?

`Openssl` can be used to generate X.509 certificates and keys. Please note that Hyperledger Fabric does not support RSA key and certificates.

The Hyperledger Fabric CA can also be used to generate the keys and certificates needed to configure an MSP. Check out [Registering and enrolling identities with a CA](#) for more information about how to generate MSPs for nodes and organizations.

9.1.3 Organizational Units

In order to configure the list of Organizational Units that valid members of this MSP should include in their X.509 certificate, the `config.yaml` file needs to specify the organizational unit (OU, for short) identifiers. You can find an example below:

```
OrganizationalUnitIdentifiers:
- Certificate: "cacerts/cacert1.pem"
  OrganizationalUnitIdentifier: "commercial"
- Certificate: "cacerts/cacert2.pem"
  OrganizationalUnitIdentifier: "administrators"
```

The above example declares two organizational unit identifiers: **commercial** and **administrators**. An MSP identity is valid if it carries at least one of these organizational unit identifiers. The `Certificate` field refers to the CA

or intermediate CA certificate path under which identities, having that specific OU, should be validated. The path is relative to the MSP root folder and cannot be empty.

9.1.4 Identity classification

The default MSP implementation allows organizations to further classify identities into clients, admins, peers, and orderers based on the OUs of their x509 certificates.

- An identity should be classified as a **client** if it transacts on the network.
- An identity should be classified as an **admin** if it handles administrative tasks such as joining a peer to a channel or signing a channel configuration update transaction.
- An identity should be classified as a **peer** if it endorses or commits transactions.
- An identity should be classified as an **orderer** if belongs to an ordering node.

In order to define the clients, admins, peers, and orderers of a given MSP, the `config.yaml` file needs to be set appropriately. You can find an example NodeOU section of the `config.yaml` file below:

```
NodeOUs:
  Enable: true
  # For each identity classification that you would like to utilize, specify
  # an OU identifier.
  # You can optionally configure that the OU identifier must be issued by a specific_
  ↪CA
  # or intermediate certificate from your organization. However, it is typical to NOT
  # configure a specific Certificate. By not configuring a specific Certificate, you_
  ↪will be
  # able to add other CA or intermediate certs later, without having to reissue all_
  ↪credentials.
  # For this reason, the sample below comments out the Certificate field.
  ClientOUIdentifier:
    # Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "client"
  AdminOUIdentifier:
    # Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "admin"
  PeerOUIdentifier:
    # Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "peer"
  OrdererOUIdentifier:
    # Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "orderer"
```

Identity classification is enabled when `NodeOUs.Enable` is set to `true`. Then the client (admin, peer, orderer) organizational unit identifier is defined by setting the properties of the `NodeOUs.ClientOUIdentifier` (`NodeOUs.AdminOUIdentifier`, `NodeOUs.PeerOUIdentifier`, `NodeOUs.OrdererOUIdentifier`) key:

1. `OrganizationalUnitIdentifier`: Is the OU value that the x509 certificate needs to contain to be considered a client (admin, peer, orderer respectively). If this field is empty, then the classification is not applied.
2. `Certificate`: (Optional) Set this to the path of the CA or intermediate CA certificate under which client (peer, admin or orderer) identities should be validated. The field is relative to the MSP root folder. Only a single Certificate can be specified. If you do not set this field, then the identities are validated under any CA defined in the organization's MSP configuration, which could be desirable in the future if you need to add other CA or intermediate certificates.

Notice that if the `NodeOUs.ClientOUIdentifier` section (`NodeOUs.AdminOUIdentifier`, `NodeOUs.PeerOUIdentifier`, `NodeOUs.OrdererOUIdentifier`) is missing, then the classification is not applied. If `NodeOUs.Enable` is set to `true` and no classification keys are defined, then identity classification is assumed to be disabled.

Identities can use organizational units to be classified as either a client, an admin, a peer, or an orderer. The four classifications are mutually exclusive. The 1.1 channel capability needs to be enabled before identities can be classified as clients or peers. The 1.4.3 channel capability needs to be enabled for identities to be classified as an admin or orderer.

Classification allows identities to be classified as admins (and conduct administrator actions) without the certificate being stored in the `admincerts` folder of the MSP. Instead, the `admincerts` folder can remain empty and administrators can be created by enrolling identities with the admin OU. Certificates in the `admincerts` folder will still grant the role of administrator to their bearer, provided that they possess the client or admin OU.

9.1.5 Adding MSPs to channels

For information about how to add MSPs to a channel (including the decision of whether to bootstrap ordering nodes with a system channel genesis block), check out [Creating a channel](#).

9.1.6 Best practices

In this section we elaborate on best practices for MSP configuration in commonly met scenarios.

1) Mapping between organizations/corporations and MSPs

We recommend that there is a one-to-one mapping between organizations and MSPs. If a different type of mapping is chosen, the following needs to be considered:

- **One organization employing various MSPs.** This corresponds to the case of an organization including a variety of divisions each represented by its MSP, either for management independence reasons, or for privacy reasons. In this case a peer can only be owned by a single MSP, and will not recognize peers with identities from other MSPs as peers of the same organization. The implication of this is that peers may share through gossip organization-scoped data with a set of peers that are members of the same subdivision, and NOT with the full set of providers constituting the actual organization.
- **Multiple organizations using a single MSP.** This corresponds to a case of a consortium of organizations that are governed by similar membership architecture. One needs to know here that peers would propagate organization-scoped messages to the peers that have an identity under the same MSP regardless of whether they belong to the same actual organization. This is a limitation of the granularity of MSP definition, and/or of the peer's configuration.

2) One organization has different divisions (say organizational units), to which it wants to grant access to different channels.

Two ways to handle this:

- **Define one MSP to accommodate membership for all organization's members.** Configuration of that MSP would consist of a list of root CAs, intermediate CAs and admin certificates; and membership identities would include the organizational unit (OU) a member belongs to. Policies can then be defined to capture members of a specific `role` (should be one of: `peer`, `admin`, `client`, `orderer`, `member`), and these policies may constitute the read/write policies of a channel or endorsement policies of a chaincode. Specifying custom OUs in the profile section of `configtx.yaml` is currently not configured. A limitation of this approach is that gossip peers would consider peers with membership identities under their local MSP as members of the same organization, and would consequently gossip with them organization-scoped data (e.g. their status).
- **Defining one MSP to represent each division.** This would involve specifying for each division, a set of certificates for root CAs, intermediate CAs, and admin Certs, such that there is no overlapping certification path

across MSPs. This would mean that, for example, a different intermediate CA per subdivision is employed. Here the disadvantage is the management of more than one MSPs instead of one, but this circumvents the issue present in the previous approach. One could also define one MSP for each division by leveraging an OU extension of the MSP configuration.

3) Separating clients from peers of the same organization.

In many cases it is required that the “type” of an identity is retrievable from the identity itself (e.g. it may be needed that endorsements are guaranteed to have derived by peers, and not clients or nodes acting solely as orderers).

There is limited support for such requirements.

One way to allow for this separation is to create a separate intermediate CA for each node type - one for clients and one for peers/orderers; and configure two different MSPs - one for clients and one for peers/orderers. Channels this organization should be accessing would need to include both MSPs, while endorsement policies will leverage only the MSP that refers to the peers. This would ultimately result in the organization being mapped to two MSP instances, and would have certain consequences on the way peers and clients interact.

Gossip would not be drastically impacted as all peers of the same organization would still belong to one MSP. Peers can restrict the execution of certain system chaincodes to local MSP based policies. For example, peers would only execute “joinChannel” request if the request is signed by the admin of their local MSP who can only be a client (end-user should be sitting at the origin of that request). We can go around this inconsistency if we accept that the only clients to be members of a peer/orderer MSP would be the administrators of that MSP.

Another point to be considered with this approach is that peers authorize event registration requests based on membership of request originator within their local MSP. Clearly, since the originator of the request is a client, the request originator is always deemed to belong to a different MSP than the requested peer and the peer would reject the request.

4) Admin and CA certificates.

It is important to set MSP admin certificates to be different than any of the certificates considered by the MSP for root of trust, or intermediate CAs. This is a common (security) practice to separate the duties of management of membership components from the issuing of new certificates, and/or validation of existing ones.

5) Blocking an intermediate CA.

As mentioned in previous sections, reconfiguration of an MSP is achieved by reconfiguration mechanisms (manual reconfiguration for the local MSP instances, and via properly constructed `config_update` messages for MSP instances of a channel). Clearly, there are two ways to ensure an intermediate CA considered in an MSP is no longer considered for that MSP’s identity validation:

1. Reconfigure the MSP to no longer include the certificate of that intermediate CA in the list of trusted intermediate CA certs. For the locally configured MSP, this would mean that the certificate of this CA is removed from the `intermediatecerts` folder.
2. Reconfigure the MSP to include a CRL produced by the root of trust which denounces the mentioned intermediate CA’s certificate.

In the current MSP implementation we only support method (1) as it is simpler and does not require blocking the no longer considered intermediate CA.

6) CAs and TLS CAs

MSP identities’ root CAs and MSP TLS certificates’ root CAs (and relative intermediate CAs) need to be declared in different folders. This is to avoid confusion between different classes of certificates. It is not forbidden to reuse the same CAs for both MSP identities and TLS certificates but best practices suggest to avoid this in production.

9.2 Taking ledger snapshots and using them to join channels

For a peer to process transactions on a channel, it must contain the minimum ledger data necessary to endorse and validate transactions consistent with other peers on a channel. This includes the “world state”, maintained in the state database, which represents the current value of all of the keys on the ledger (who owns a particular asset, for example) as of the most recently committed block. There are two ways for a peer to get a copy of the necessary ledger data.

1. Join the channel starting with the initial configuration block (known as the “genesis block”), and continue pulling blocks from the ordering service and processing them locally until the latest block that has been written to the ledger is reached. In this scenario, the world state is built from the blocks.
2. Join the channel from a “snapshot”, which contains the minimum ledger data as of a particular block number, without needing to pull and process the individual blocks.

While the first method represents a more comprehensive way of joining a channel, because of the size of established channels (which can reach many thousands of blocks), it can take a long time for peers to pull and process all the blocks already committed to the channel. Peers that join a channel this way must also store every block since the creation of the channel, increasing storage costs for an organization. Additionally, joining by snapshot will provide a peer with the latest channel configuration, which may be important if the channel configuration has changed since the genesis block. For example, the peer may need the orderer endpoints or CA certificates from the latest channel configuration before it can successfully pull blocks from the ordering service.

In this topic, we’ll describe the process for joining a peer to a channel using a snapshot.

9.2.1 Limitations

While creating a snapshot and using it to join a peer to a channel from a snapshot (also known as a “checkpoint”) will take less time and saves on storage costs compared to processing and storing every block on the ledger, there are a few limitations to consider:

- It is not possible for a peer that joins from a snapshot to query blocks (or transactions within blocks) that were committed before the ledger height of the snapshot. Similarly, it is not possible to query the history of a key prior to the snapshot. If the snapshot the peer uses to join a channel from was taken at block 1000, for example, none of the blocks between 0-999 can be queried. Applications attempting to query for the data in these blocks will have to target a peer that contains the relevant block. Because of this, it is likely that organizations will want to keep at least one peer that has all of the historical data, and target this peer for historical queries.
- While endorsements will continue and queries can be submitted, peers taking a snapshot at a particular height will not commit blocks on the channel while the snapshot is being generated. Because taking a snapshot is a resource-intensive operation, the peer might also be slow to endorse transactions or commit blocks on other channels. For these reasons, it is anticipated that snapshots will only be taken when necessary (for example, when a new peer needs a snapshot to join a channel, or when organizations want to verify that no ledger forks have occurred).
- Because the private data between organizations in a channel is likely to be at least somewhat different, private data is not included in the snapshot (hashes of the private data are included, but not the data itself). Peers that join a channel using a snapshot will discover the collections it is a member of and pull the relevant private data from peers that are members of those collections directly. This private data reconciliation will begin after the peer joins the channel, and may take some time.
- The snapshot process does not archive and prune the ledgers of peers that are already joined to the channel. Similarly, it is not meant as a method to take a full backup of a peer, as private data, and peer configuration information, such as MSPs, are not included in a snapshot).
- It is not possible to use the `reset`, `rollback`, or `rebuild-dbs` commands on peers that have joined a channel using a snapshot since the peer would not have all the block files required for the operations. Instead

of these administrative commands, it is expected that peers that have joined a channel using a snapshot can be entirely rebuilt from the same or newer snapshots.

9.2.2 Considerations

- When deciding whether to join peers from a genesis block or from a snapshot, consider the time it may take to join a peer to a channel based on the number of blocks since the genesis block, whether the peer will be able to pull blocks from the ordering service based on the original channel configuration in the genesis block, and whether you will need to query the entire history of a channel (historical blocks, transactions, or state).
- If your endorsement requests or queries don't require the latest block commits, you can target a peer that generates snapshots.
- If your endorsement requests or queries require the latest block commits, you can utilize service discovery to identify and target peers with the highest block heights on a channel, thereby avoiding any peer that is generating a snapshot. Alternatively, you could utilize dedicated peers for snapshot, and not make these peers available for endorsements and queries, for example by not setting `peer.gossip.externalEndpoint` so that the peer does not participate in service discovery.
- You may not want to make a peer available for endorsements and queries until it has joined all the expected channels, and has reconciled all the private data that it is authorized to receive.

9.2.3 Overview

Snapshots can be used by organizations that already have peers on a channel or by organizations new to a channel. Whatever the use case, the process is largely the same.

1. **Schedule a snapshot.** These snapshots must be taken at **exactly the same ledger height** on each peer. This will allow an organization to evaluate the snapshots to make sure they contain the same data. This ledger height must be equal or higher than the current block height (snapshots scheduled for a higher block height will be taken when the block height is reached). They cannot be taken from a lower block height. If you attempt to schedule a snapshot at a height lower than the current height you will get an error. Note that a peer that already used a snapshot to join a channel can also be used to take a snapshot. Snapshots can be scheduled as needed or there can be an agreed among organizations to take them at a regular cadence, for example every 10,000 blocks. This ensures that consistent and recent snapshots are always available. Note that it is not possible to schedule recurring snapshots. Each snapshot has to be scheduled independently. However, there is no limit to the number of future snapshots that can be scheduled. When joining a peer from a snapshot, it is a good practice to use a snapshot more recent than the latest channel config block height. This ensures that the peer will have the most recent channel configuration including the latest ordering service endpoints and CA certificates.
2. **When the ledger height is reached, the snapshot is taken by the peer.** The snapshot is comprised of a directory that includes files that contain the public state, hashes of private state, transaction IDs, and the collection config history. A file containing metadata relating to these files is also included. For more information, check out *contents of a snapshot*.
3. **If the snapshot will be used by a new organization, the snapshot is sent to them.** This must be completed out of band. Because snapshot files are not compressed, it is likely that peer administrators will want to compress these files before sending them. In a typical scenario, the administrator will receive the snapshot from one of the existing organizations but will want to receive the snapshot metadata from more than one organizations in order to verify the snapshot received.

The organization that will use the snapshot to join the channel will then:

1. **Evaluate the snapshot or snapshots.** An administrator of the peer organization attempting to use the snapshot to join the peer to the channel should independently compute the hashes of the snapshot files and match these with the hashes present in the metadata file. In addition, the administrator may want to match the metadata files

from more than one organization, depending on the trust model established by the network. In some scenarios, the administrator may want the administrators of other organizations to sign the metadata file for its records.

2. **Join the peer to the channel using the snapshot.** When the peer has finished joining the channel using the snapshot, it will begin pulling private data according to the collections it is a member of. It will also start committing blocks as normal, starting with any blocks greater than the snapshot height that are available from the ordering service.
3. **Verify the peer has joined the channel successfully.** For more information, check out *Joining a channel using a snapshot*.

If an organization that is already joined to the channel wants to join a new peer using a snapshot, it might decide to skip the process of having other organizations take snapshots and evaluate them, though it is a best practice for an organization to periodically take snapshots of all its peers and compare them to ensure the no ledger forks have occurred. In that case, the organization can take a snapshot immediately and then use the snapshot to join the new peer to the channel.

Note: snapshots can also be used to verify that the public state between peers is identical (in other words, that no ledger fork has occurred), even if no new peer will use the snapshot to join the channel. This can be done by ensuring that the `snapshot_hash` in the file `_snapshot_additional_metadata.json` in the snapshots generated across peers is the same.

9.2.4 Taking a snapshot

For the full list of snapshot-related commands, check out `peer snapshot commands`.

Before taking a snapshot, it is a best practice to confirm the current ledger height by issuing a command similar to:

```
peer channel getinfo -c <name of channel>
```

You will see a response similar to:

```
Blockchain info: {"height":970,"currentBlockHash":"JgK9lcaPUNmFb5Mp1qe1SVMsx3o/
↳22Ct4+n5tejXCw=", "previousBlockHash": "f8lZXoAn3gF86zrFq7L1DzW2aKuabH9Ow6SIE5Y04a4=
↳" }
```

In this example, the ledger height is 970.

A snapshot request can be submitted by issuing a command similar to:

```
peer snapshot submitrequest -c <name of channel> -b <ledger height where snapshot_
↳will be taken> --peerAddress <address of peer> --tlsRootCertFile <path to root_
↳certificate of the TLS CA>
```

For example:

```
peer snapshot submitrequest -c testchannel -b 1000 --peerAddress 127.0.0.1:22509 --
↳tlsRootCertFile tls/cert.pem
```

If you give a ledger height of 0, the snapshot is taken immediately. This is useful for cases when an organization is interested in generating a snapshot that will be used by one of its own peers and does not intend to share the data with another organization. Do not take these “immediate” snapshots in cases when snapshots will be evaluated from multiple peers, as it increases the likelihood that the snapshots will be taken at different ledger heights.

If the request is successful, you will see a `Snapshot request submitted successfully` message.

You can list the pending snapshots by issuing a command similar to:


```
peer snapshot listpending -c testchannel --peerAddress 127.0.0.1:22509 --
↪tlsRootCertFile tls/cert.pem
```

You will see a response similar to:

```
Successfully got pending snapshot requests [1000]
```

When a snapshot has been generated for a particular block height, the pending request for that block height will no longer appear in the list. You can also verify that a snapshot has been created successfully by looking at the peer logs.

Snapshots will be written to a directory based on the `core.yaml` `ledger.snapshots.rootDir` property. Completed snapshots are written to a subdirectory based on the channel name and block number of the snapshot: `{ledger.snapshots.rootDir}/completed/{channelName}/{lastBlockNumberInSnapshot}`. If the `ledger.snapshots.rootDir` property is not specified in the `core.yaml`, then the default value is `{peer.fileSystemPath}/snapshots`. If you expect a snapshot will be large, or you expect to share snapshots in the location that they are generated, consider setting the snapshot directory to a different volume than the peer's `fileSystemPath`.

To delete a snapshot request, simply exchange `submitrequest` with `cancelrequest`. For example:

```
peer snapshot cancelrequest -c testchannel -b 1000 --peerAddress 127.0.0.1:22509 --
↪tlsRootCertFile tls/cert.pem
```

If you submit the `listpending` command again, the snapshot should no longer appear.

Contents of a snapshot

Once the peer generates a snapshot to the `{ledger.snapshots.rootDir}/completed/{channelName}/{lastBlockNumberInSnapshot}` directory, the peer does not use that directory for any purpose and it is safe to compress and transfer the snapshot using external tools, and to delete it when no longer needed.

As mentioned above, the completed snapshot directory contains files for the different data items listed below:

- **Public state**
 - This includes the latest value of all of the keys on the channel. For example, the public state would show an asset (a key) and its current owner (the value), but not any of its historical owners.
- **Private data hashes**
 - This includes hashes of private data transactions on the channel. Recall from our documentation on private data that while the actual transaction data of a private data transaction is not stored on the public ledger of the channel, hashes of the data are stored. This allows the private data to be verified against the hash, for example by an organization that is added to a private data collection. These hashes are included in the snapshot so that new organizations can verify them against the private data they receive for the collections they are a member of.
- **Transactions IDs**
 - This consists of the transaction IDs that have been used in the channel until the last block in the snapshot. The transaction IDs are included so that peers can verify that a transaction ID is not later re-used for another transaction.
- **Collection config history**
 - This contains the history of the collection configurations for all chaincodes. Recall from our documentation on private data that the collection configuration derives the endorsement and dissemination policies for private data collections.

In addition, the snapshot contains two metadata files that help the data in the snapshot to be verified. This snapshot metadata file is expected to be same across snapshots of a channel for a particular height.

One of these files contains a JSON record called `_snapshot_signable_metadata.json` with the following fields:

- `channel_name`: the name of the channel.
- `last_block_number`: the block height when the snapshot was taken.
- `last_block_hash`: a hash of the block at the height the snapshot was taken.
- `previous_block_hash`: a hash of the block prior to the `last_block`.
- `state_db_type` (the value of this field will be either CouchDB or SimpleKeyValueDB (also known as LevelDB)).
- `snapshot_files_raw_hashes`, is a JSON record that contains the hashes of the files above.

This metadata file is also a JSON record with the following two fields:

- `snapshot_hash`, the hash of the file `_snapshot_signable_metadata.json` and can be treated as a hash of the snapshot.
- `last_block_commit_hash`, which is included if the snapshot generating peer is equipped to compute the block commit hashes.

Note that the file types explained here is a superset of all of the files that might be included in a snapshot. If admins find some of these file types missing in their snapshots (for example, the collection config history) this is not mean the snapshot is incomplete. The channel might not have any collections.

9.2.5 Joining a channel using a snapshot

When joining a channel using the genesis block, a command similar to `peer channel join --blockpath mychannel.block` is issued. When joining the peer to the channel using a snapshot, issue a command similar to:

```
peer channel joinbysnapshot --snapshotpath <path to snapshot>
```

To verify that the peer has joined the channel successfully, issue a command similar to:

```
peer channel getinfo -c <name of channel joined by snapshot>
```

Additionally, if the peer has not already installed a chaincode being used on the channel, do so, and then issue a query. A successful return of data indicates that the peer has successfully joined using the snapshot. You can then install all of the chaincodes being used on the channel. If the snapshot is used by a new organization, and the channel contains the definitions for the chaincodes that are defined using the [new chaincode lifecycle](#), the new organization will need to approve the definition of these chaincodes before they can be invoked on the peer.

9.2.6 Try it out

If you want to try out the ledger snapshotting process, you'll first need a network with a running channel. If you don't have a network, deploy the [test network](#). This will create a network with two orgs, which both have a single peer, and an application channel.

Next, follow the [Adding an Org to a Channel](#) to add a new org to your network and application channel. When you reach the section where you are asked to [Join Org3 to the Channel](#), select the peer you want to use to take the snapshot and follow the instructions above to take the snapshot. Then locate the snapshot on the peer and copy it somewhere else on your filesystem. Taking the snapshot at this step ensures that the new peer joins the channel using a snapshot taken after a point when its organization has already been joined to the channel.

After you have taken the snapshot and copied it, instead of issuing the `peer channel join -b mychannel.block` command, substitute `peer channel joinbysnapshot --snapshotpath <path to snapshot>` using the path to the snapshot on your filesystem.

9.2.7 Troubleshooting

There are a few reasons why a peer might fail to join a channel using a snapshot:

- The snapshot is not at the location that was specified. Check to make sure the snapshot is in the location you have specified in the `joinbysnapshot` command.
- The hash of the data does not match the data. This can indicate that there was an undetected error during the creation of the snapshot or that the data in the snapshot has been corrupted somehow.

9.3 Using a Hardware Security Module (HSM)

The cryptographic operations performed by Fabric nodes can be delegated to a Hardware Security Module (HSM). An HSM protects your private keys and handles cryptographic operations, allowing your peers and orderer nodes to sign and endorse transactions without exposing their private keys. If you require compliance with government standards such as FIPS 140-2, there are multiple certified HSMs from which to choose.

Fabric currently leverages the PKCS11 standard to communicate with an HSM.

9.3.1 Configuring an HSM

To use an HSM with your Fabric node, you need to update the `bccsp` (Crypto Service Provider) section of the node configuration file such as `core.yaml` or `orderer.yaml`. In the `bccsp` section, you need to select PKCS11 as the provider and enter the path to the PKCS11 library that you would like to use. You also need to provide the `Label` and `PIN` of the token that you created for your cryptographic operations. You can use one token to generate and store multiple keys.

The prebuilt Hyperledger Fabric Docker images are not enabled to use PKCS11. If you are deploying Fabric using docker, you need to build your own images and enable PKCS11 using the following command:

```
make docker GO_TAGS=pkcs11
```

You also need to ensure that the PKCS11 library is available to be used by the node by installing it or mounting it inside the container.

Example

The following example demonstrates how to configure a Fabric node to use an HSM.

First, you will need to install an implementation of the PKCS11 interface. This example uses the `softhsm` open source implementation. After downloading and configuring `softhsm`, you will need to set the `SOFTHSM2_CONF` environment variable to point to the `softhsm2` configuration file.

You can then use `softhsm` to create the token that will handle the cryptographic operations of your Fabric node inside an HSM slot. In this example, we create a token labelled “fabric” and set the pin to “71811222”. After you have created the token, update the configuration file to use PKCS11 and your token as the crypto service provider. You can find an example `bccsp` section below:

```
#####
# BCCSP (BlockChain Crypto Service Provider) section is used to select which
# crypto library implementation to use
#####
bccsp:
  default: PKCS11
  pkcs11:
    Library: /etc/hyperledger/fabric/libsofthsm2.so
    Pin: "71811222"
    Label: fabric
    hash: SHA2
    security: 256
    Immutable: false
```

By default, when private keys are generated using the HSM, the private key is mutable, meaning PKCS11 private key attributes can be changed after the key is generated. Setting `Immutable` to `true` means that the private key attributes cannot be altered after key generation. Before you configure immutability by setting `Immutable: true`, ensure that PKCS11 object copy is supported by the HSM.

If you are using AWS HSM there is an additional step required:

- Add the parameter, `AltID` to the `pkcs11` section of the `bccsp` block. When AWS HSM is being used, this parameter is used to assign a unique value for the Subject Key Identifier (SKI). Create a long secure string outside of Fabric and assign it to the `AltID` parameter. For example:

```
#####
# BCCSP (BlockChain Crypto Service Provider) section is used to select which
# crypto library implementation to use
#####
bccsp:
  default: PKCS11
  pkcs11:
    Library: /etc/hyperledger/fabric/libsofthsm2.so
    Pin: 71811222
    Label: fabric
    hash: SHA2
    security: 256
    Immutable: false
    AltID: ↵
↵4AMfmFMtLY6B6vN3q4SQtCkCQ6UY5f6gUF3rDRE4wqD4YDUrunuZbmZpVk8zszkt86yenPBUGE2aCQCZmQFcmnj3UaxyLz
```

You can also use environment variables to override the relevant fields of the configuration file. If you are connecting to `softhsm2` using the Fabric CA server, you could set the following environment variables or directly set the corresponding values in the CA server config file:

```
FABRIC_CA_SERVER_BCCSP_DEFAULT=PKCS11
FABRIC_CA_SERVER_BCCSP_PKCS11_LIBRARY=/etc/hyperledger/fabric/libsofthsm2.so
FABRIC_CA_SERVER_BCCSP_PKCS11_PIN=71811222
FABRIC_CA_SERVER_BCCSP_PKCS11_LABEL=fabric
```

If you are connecting to `softhsm2` using the Fabric peer, you could set the following environment variables or directly set the corresponding values in the peer config file:

```
CORE_PEER_BCCSP_DEFAULT=PKCS11
CORE_PEER_BCCSP_PKCS11_LIBRARY=/etc/hyperledger/fabric/libsofthsm2.so
CORE_PEER_BCCSP_PKCS11_PIN=71811222
CORE_PEER_BCCSP_PKCS11_LABEL=fabric
```

If you are connecting to softhsm2 using the Fabric orderer, you could set the following environment variables or directly set the corresponding values in the orderer config file:

```
ORDERER_GENERAL_BCCSP_DEFAULT=PKCS11
ORDERER_GENERAL_BCCSP_PKCS11_LIBRARY=/etc/hyperledger/fabric/libsofthsm2.so
ORDERER_GENERAL_BCCSP_PKCS11_PIN=71811222
ORDERER_GENERAL_BCCSP_PKCS11_LABEL=fabric
```

If you are deploying your nodes using docker compose, after building your own images, you can update your docker compose files to mount the softhsm library and configuration file inside the container using volumes. As an example, you would add the following environment and volumes variables to your docker compose file:

```
environment:
  - SOFTHSM2_CONF=/etc/hyperledger/fabric/config.file
volumes:
  - /home/softhsm/config.file:/etc/hyperledger/fabric/config.file
  - /usr/local/Cellar/softhsm/2.1.0/lib/softhsm/libsofthsm2.so:/etc/hyperledger/
  ↪ fabric/libsofthsm2.so
```

9.3.2 Setting up a network using HSM

If you are deploying Fabric nodes using an HSM, your private keys need to be generated and stored inside the HSM rather than inside the `keystore` folder of the node's local MSP folder. The `keystore` folder of the MSP will remain empty. Instead, the Fabric node will use the subject key identifier of the signing certificate in the `signcerts` folder to retrieve the private key from inside the HSM. The process for creating the node MSP folder differs depending on whether you are using a Fabric Certificate Authority (CA) your own CA.

Before you begin

Before configuring a Fabric node to use an HSM, you should have completed the following steps:

1. Created a partition on your HSM Server and recorded the `Label` and `PIN` of the partition.
2. Followed instructions in the documentation from your HSM provider to configure an HSM Client that communicates with your HSM server.

Using an HSM with a Fabric CA

You can set up a Fabric CA to use an HSM by making the same edits to the CA server configuration file as you would make to a peer or ordering node. Because you can use the Fabric CA to generate keys inside an HSM, the process of creating the local MSP folders is straightforward. Use the following steps:

1. Modify the `bccsp` section of the Fabric CA server configuration file and point to the `Label` and `PIN` that you created for your HSM. When the Fabric CA server starts, the private key is generated and stored in the HSM. If you are not concerned about exposing your CA signing certificate, you can skip this step and only configure an HSM for your peer or ordering nodes, described in the next steps.
2. Use the Fabric CA client to register the peer or ordering node identities with your CA.
3. Before you deploy a peer or ordering node with HSM support, you need to enroll the node identity by storing its private key in the HSM. Edit the `bccsp` section of the Fabric CA client config file or use the associated environment variables to point to the HSM configuration for your peer or ordering node. In the Fabric CA Client configuration file, replace the default `SW` configuration with the `PKCS11` configuration and provide the values for your own HSM:

```
bccsp:
  default: PKCS11
  pkcs11:
    Library: /etc/hyperledger/fabric/libsofthsm2.so
    Pin: "71811222"
    Label: fabric
    hash: SHA2
    security: 256
    Immutable: false
```

Then for each node, use the Fabric CA client to generate the peer or ordering node's MSP folder by enrolling against the node identity that you registered in step 2. Instead of storing the private key in the `keystore` folder of the associated MSP, the enroll command uses the node's HSM to generate and store the private key for the peer or ordering node. The `keystore` folder remains empty.

1. To configure a peer or ordering node to use the HSM, similarly update the `bccsp` section of the peer or orderer configuration file to use PKCS11 and provide the `Label` and `PIN`. Also, edit the value of the `mspConfigPath` (for a peer node) or the `LocalMSPDir` (for an ordering node) to point to the MSP folder that was generated in the previous step using the Fabric CA client. Now that the peer or ordering node is configured to use HSM, when you start the node it will be able sign and endorse transactions with the private key protected by the HSM.

Using an HSM with your own CA

If you are using your own Certificate Authority to deploy Fabric components, you can use an HSM using the following steps:

1. Configure your CA to communicate with an HSM using PKCS11 and create a `Label` and `PIN`. Then use your CA to generate the private key and signing certificate for each node, with the private key generated inside the HSM.
2. Use your CA to build the peer or ordering node MSP folder. Place the signing certificate that you generated in step 1 inside the `signcerts` folder. You can leave the `keystore` folder empty.
3. To configure a peer or ordering node to use the HSM, similarly update the `bccsp` section of the peer or orderer configuration file to use PKCS11 and provide the `Label` and `PIN`. Edit the value of the `mspConfigPath` (for a peer node) or the `LocalMSPDir` (for an ordering node) to point to the MSP folder that was generated in the previous step using the Fabric CA client. Now that the peer or ordering node is configured to use HSM, when you start the node it will be able sign and endorse transactions with the private key protected by the HSM.

9.4 Channel Configuration (configtx)

Note: This topic describes how channels are configured when the network has not been bootstrapped using a system channel genesis block. For information about the structure of configurations, including the configuration of the system channel, check out [Channel Configuration \(configtx\)](#) from the v2.2 documentation.

Shared configuration for a Hyperledger Fabric blockchain network is stored in a collection configuration transactions, one per channel. Each configuration transaction is usually referred to by the shorter name *configtx*.

Channel configuration has the following important properties:

1. **Versioned:** All elements of the configuration have an associated version which is advanced with every modification. Further, every committed configuration receives a sequence number.

2. **Permissioned:** Each element of the configuration has an associated policy which governs whether or not modification to that element is permitted. Anyone with a copy of the previous configtx (and no additional info) may verify the validity of a new config based on these policies.
3. **Hierarchical:** A root configuration group contains sub-groups, and each group of the hierarchy has associated values and policies. These policies can take advantage of the hierarchy to derive policies at one level from policies of lower levels.

9.4.1 Anatomy of a configuration

Configuration is stored as a transaction of type `HeaderType_CONFIG` in a block with no other transactions. These blocks are referred to as *Configuration Blocks*, the first of which is referred to as the *Genesis Block*.

The proto structures for configuration are stored in `fabric-protos/common/configtx.proto`. The Envelope of type `HeaderType_CONFIG` encodes a `ConfigEnvelope` message as the `Payload` data field. The proto for `ConfigEnvelope` is defined as follows:

```
message ConfigEnvelope {
    Config config = 1;
    Envelope last_update = 2;
}
```

The `last_update` field is defined below in the **Updates to configuration** section, but is only necessary when validating the configuration, not reading it. Instead, the currently committed configuration is stored in the `config` field, containing a `Config` message.

```
message Config {
    uint64 sequence = 1;
    ConfigGroup channel_group = 2;
}
```

The sequence number is incremented by one for each committed configuration. The `channel_group` field is the root group which contains the configuration. The `ConfigGroup` structure is recursively defined, and builds a tree of groups, each of which contains values and policies. It is defined as follows:

```
message ConfigGroup {
    uint64 version = 1;
    map<string, ConfigGroup> groups = 2;
    map<string, ConfigValue> values = 3;
    map<string, ConfigPolicy> policies = 4;
    string mod_policy = 5;
}
```

Because `ConfigGroup` is a recursive structure, it has hierarchical arrangement. The following example is expressed for clarity in Go syntax.

```
// Assume the following groups are defined
var root, child1, child2, grandChild1, grandChild2, grandChild3 *ConfigGroup

// Set the following values
root.Groups["child1"] = child1
root.Groups["child2"] = child2
child1.Groups["grandChild1"] = grandChild1
child2.Groups["grandChild2"] = grandChild2
child2.Groups["grandChild3"] = grandChild3
```

(continues on next page)

(continued from previous page)

```
// The resulting config structure of groups looks like:
// root:
//   child1:
//     grandChild1
//   child2:
//     grandChild2
//     grandChild3
```

Each group defines a level in the config hierarchy, and each group has an associated set of values (indexed by string key) and policies (also indexed by string key).

Values are defined by:

```
message ConfigValue {
    uint64 version = 1;
    bytes value = 2;
    string mod_policy = 3;
}
```

Policies are defined by:

```
message ConfigPolicy {
    uint64 version = 1;
    Policy policy = 2;
    string mod_policy = 3;
}
```

Note that Values, Policies, and Groups all have a version and a mod_policy. The version of an element is incremented each time that element is modified. The mod_policy is used to govern the required signatures to modify that element. For Groups, modification is adding or removing elements to the Values, Policies, or Groups maps (or changing the mod_policy). For Values and Policies, modification is changing the Value and Policy fields respectively (or changing the mod_policy). Each element's mod_policy is evaluated in the context of the current level of the config. Consider the following example mod policies defined at Channel.Groups["Application"] (Here, we use the Go map reference syntax, so Channel.Groups["Application"].Policies["policy1"] refers to the base Channel group's Application group's Policies map's policy1 policy.)

- policy1 maps to Channel.Groups["Application"].Policies["policy1"]
- Org1/policy2 maps to Channel.Groups["Application"].Groups["Org1"].Policies["policy2"]
- /Channel/policy3 maps to Channel.Policies["policy3"]

Note that if a mod_policy references a policy which does not exist, the item cannot be modified.

9.4.2 Configuration updates

Configuration updates are submitted as an Envelope message of type HeaderType_CONFIG_UPDATE. The Payload data of the transaction is a marshaled ConfigUpdateEnvelope. The ConfigUpdateEnvelope is defined as follows:

```
message ConfigUpdateEnvelope {
    bytes config_update = 1;
    repeated ConfigSignature signatures = 2;
}
```


The `signatures` field contains the set of signatures which authorizes the config update. Its message definition is:

```
message ConfigSignature {
    bytes signature_header = 1;
    bytes signature = 2;
}
```

The `signature_header` is as defined for standard transactions, while the `signature` is over the concatenation of the `signature_header` bytes and the `config_update` bytes from the `ConfigUpdateEnvelope` message.

The `ConfigUpdateEnvelope` `config_update` bytes are a marshaled `ConfigUpdate` message which is defined as follows:

```
message ConfigUpdate {
    string channel_id = 1;
    ConfigGroup read_set = 2;
    ConfigGroup write_set = 3;
}
```

The `channel_id` is the channel ID the update is bound for, this is necessary to scope the signatures which support this reconfiguration.

The `read_set` specifies a subset of the existing configuration, specified sparsely where only the `version` field is set and no other fields must be populated. The particular `ConfigValue` value or `ConfigPolicy` policy fields should never be set in the `read_set`. The `ConfigGroup` may have a subset of its map fields populated, so as to reference an element deeper in the config tree. For instance, to include the `Application` group in the `read_set`, its parent (the `Channel` group) must also be included in the `read_set`, but, the `Channel` group does not need to populate all of the keys, such as the `Orderer` group key, or any of the values or policies keys.

The `write_set` specifies the pieces of configuration which are modified. Because of the hierarchical nature of the configuration, a write to an element deep in the hierarchy must contain the higher level elements in its `write_set` as well. However, for any element in the `write_set` which is also specified in the `read_set` at the same version, the element should be specified sparsely, just as in the `read_set`.

For example, given the configuration:

```
Channel: (version 0)
  Orderer (version 0)
  Application (version 3)
    Org1 (version 2)
```

To submit a configuration update which modifies `Org1`, the `read_set` would be:

```
Channel: (version 0)
  Application: (version 3)
```

and the `write_set` would be

```
Channel: (version 0)
  Application: (version 3)
    Org1 (version 3)
```

When the `CONFIG_UPDATE` is received, the orderer computes the resulting `CONFIG` by doing the following:

1. Verifies the `channel_id` and `read_set`. All elements in the `read_set` must exist at the given versions.
2. Computes the update set by collecting all elements in the `write_set` which do not appear at the same version in the `read_set`.
3. Verifies that each element in the update set increments the version number of the element update by exactly 1.

4. Verifies that the signature set attached to the `ConfigUpdateEnvelope` satisfies the `mod_policy` for each element in the update set.
5. Computes a new complete version of the config by applying the update set to the current config.
6. Writes the new config into a `ConfigEnvelope` which includes the `CONFIG_UPDATE` as the `last_update` field and the new config encoded in the `config` field, along with the incremented sequence value.
7. Writes the new `ConfigEnvelope` into a `Envelope` of type `CONFIG`, and ultimately writes this as the sole transaction in a new configuration block.

When the peer (or any other receiver for `Deliver`) receives this configuration block, it should verify that the config was appropriately validated by applying the `last_update` message to the current config and verifying that the orderer-computed `config` field contains the correct new configuration.

9.4.3 Permitted configuration groups and values

Any valid configuration is a subset of the following configuration. Here we use the notation `peer.<MSG>` to define a `ConfigValue` whose value field is a marshaled proto message of name `<MSG>` defined in `fabric-protos/peer/configuration.proto`. The notations `common.<MSG>`, `msp.<MSG>`, and `orderer.<MSG>` correspond similarly, but with their messages defined in `fabric-protos/common/configuration.proto`, `fabric-protos/msp/mspconfig.proto`, and `fabric-protos/orderer/configuration.proto` respectively.

Note, that the keys `{{org_name}}` and `{{consortium_name}}` represent arbitrary names, and indicate an element which may be repeated with different names.

```
&ConfigGroup{
  Groups: map<string, *ConfigGroup> {
    "Application": &ConfigGroup{
      Groups: map<String, *ConfigGroup> {
        {{org_name}}: &ConfigGroup{
          Values: map<string, *ConfigValue>{
            "MSP": msp.MSPConfig,
            "AnchorPeers": peer.AnchorPeers,
          },
        },
      },
    },
    "Orderer": &ConfigGroup{
      Groups: map<String, *ConfigGroup> {
        {{org_name}}: &ConfigGroup{
          Values: map<string, *ConfigValue>{
            "MSP": msp.MSPConfig,
          },
        },
      },
    },
  },
  Values: map<string, *ConfigValue> {
    "ConsensusType": orderer.ConsensusType,
    "BatchSize": orderer.BatchSize,
    "BatchTimeout": orderer.BatchTimeout,
    "KafkaBrokers": orderer.KafkaBrokers,
  },
},
"Consortiums": &ConfigGroup{
  Groups: map<String, *ConfigGroup> {
    {{consortium_name}}: &ConfigGroup{
```

(continues on next page)

(continued from previous page)

```

        Groups:map<string, *ConfigGroup> {
            {{org_name}}:&ConfigGroup{
                Values:map<string, *ConfigValue>{
                    "MSP":msp.MSPConfig,
                },
            },
        },
        Values:map<string, *ConfigValue> {
            "ChannelCreationPolicy":common.Policy,
        }
    },
},
},
},
Values: map<string, *ConfigValue> {
    "HashingAlgorithm":common.HashingAlgorithm,
    "BlockHashingDataStructure":common.BlockDataHashingStructure,
    "Consortium":common.Consortium,
    "OrdererAddresses":common.OrdererAddresses,
},
}

```

9.4.4 Channel configuration

Application configuration is for channels which are designed for application type transactions. It is defined as follows:

```

&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Application":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                        "AnchorPeers":peer.AnchorPeers,
                    },
                },
            },
        },
    },
},
}

```

Just like with the `Orderer` section, each organization is encoded as a group. However, instead of only encoding the MSP identity information, each org additionally encodes a list of `AnchorPeers`. This list allows the peers of different organizations to contact each other for peer gossip networking.

9.4.5 Channel creation

For information about how to create a channel, check out [Create a channel](#).

9.5 Endorsement policies

Every chaincode has an endorsement policy which specifies the set of peers on a channel that must execute chaincode and endorse the execution results in order for the transaction to be considered valid. These endorsement policies define the organizations (through their peers) who must “endorse” (i.e., approve of) the execution of a proposal.

Note: Recall that **state**, represented by key-value pairs, is separate from blockchain data. For more on this, check out our [Ledger](#) documentation.

As part of the transaction validation step performed by the peers, each validating peer checks to make sure that the transaction contains the appropriate **number** of endorsements and that they are from the expected sources (both of these are specified in the endorsement policy). The endorsements are also checked to make sure they’re valid (i.e., that they are valid signatures from valid certificates).

9.5.1 Multiple ways to require endorsement

By default, endorsement policies are specified in the chaincode definition, which is agreed to by channel members and then committed to a channel (that is, one endorsement policy covers all of the state associated with a chaincode).

For private data collections, you can also specify an endorsement policy at the private data collection level, which would override the chaincode level endorsement policy for any keys in the private data collection, thereby further restricting which organizations can write to a private data collection.

Finally, there are cases where it may be necessary for a particular public channel state or private data collection state (a particular key-value pair, in other words) to have a different endorsement policy. This **state-based endorsement** allows the chaincode-level or collection-level endorsement policies to be overridden by a different policy for the specified keys.

To illustrate the circumstances in which the various types of endorsement policies might be used, consider a channel on which cars are being exchanged. The “creation” — also known as “issuance” — of a car as an asset that can be traded (putting the key-value pair that represents it into the world state, in other words) would have to satisfy the chaincode-level endorsement policy. To see how to set a chaincode-level endorsement policy, check out the section below.

If the key representing the car requires a specific endorsement policy, it can be defined either when the car is created or afterwards. There are a number of reasons why it might be necessary or preferable to set a state-specific endorsement policy. The car might have historical importance or value that makes it necessary to have the endorsement of a licensed appraiser. Also, the owner of the car (if they’re a member of the channel) might also want to ensure that their peer signs off on a transaction. In both cases, **an endorsement policy is required for a particular asset that is different from the default endorsement policies for the other assets associated with that chaincode.**

We’ll show you how to define a state-based endorsement policy in a subsequent section. But first, let’s see how we set a chaincode-level endorsement policy.

9.5.2 Setting chaincode-level endorsement policies

Chaincode-level endorsement policies are agreed to by channel members when they approve a chaincode definition for their organization. A sufficient number of channel members need to approve a chaincode definition to meet the `Channel/Application/LifecycleEndorsement` policy, which by default is set to a majority of channel members, before the definition can be committed to the channel. Once the definition has been committed, the chaincode is ready to use. Any invoke of the chaincode that writes data to the ledger will need to be validated by enough channel members to meet the endorsement policy.

You can create an endorsement policy from your CLI when you approve and commit a chaincode definition with the Fabric peer binaries by using the `--signature-policy` flag.

Note: Don't worry about the policy syntax ('Org1.member', et all) right now. We'll talk more about the syntax in the next section.

For example:

```
peer lifecycle chaincode approveformyorg --channelID mychannel --signature-policy
↪ "AND('Org1.member', 'Org2.member')" --name mycc --version 1.0 --package-id mycc_
↪ 1:3a8c52d70c36313cfefbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173 --sequence 1 --
↪ tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪ example.com-cert.pem --waitForEvent
```

The above command approves the chaincode definition of mycc with the policy `AND('Org1.member', 'Org2.member')` which would require that a member of both Org1 and Org2 sign the transaction. After a sufficient number of channel members approve a chaincode definition for mycc, the definition and endorsement policy can be committed to the channel using the command below:

```
peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID mychannel --
↪ signature-policy "AND('Org1.member', 'Org2.member')" --name mycc --version 1.0 --
↪ sequence 1 --init-required --tls --cafile /opt/gopath/src/github.com/hyperledger/
↪ fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/
↪ msp/tlscacerts/tlsca.example.com-cert.pem --waitForEvent --peerAddresses peer0.org1.
↪ example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/
↪ peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.
↪ crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles /opt/gopath/src/
↪ github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/
↪ peer0.org2.example.com/tls/ca.crt
```

Notice that, if the identity classification is enabled (see *Membership Service Providers (MSP)*), one can use the PEER role to restrict endorsement to only peers.

For example:

```
peer lifecycle chaincode approveformyorg --channelID mychannel --signature-policy
↪ "AND('Org1.peer', 'Org2.peer')" --name mycc --version 1.0 --package-id mycc_
↪ 1:3a8c52d70c36313cfefbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173 --sequence 1 --
↪ tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪ example.com-cert.pem --waitForEvent
```

In addition to the specifying an endorsement policy from the CLI or SDK, a chaincode can also use policies in the channel configuration as endorsement policies. You can use the `--channel-config-policy` flag to select a channel policy with format used by the channel configuration and by ACLs.

For example:

```
peer lifecycle chaincode approveformyorg --channelID mychannel --channel-config-
↪ policy Channel/Application/Admins --name mycc --version 1.0 --package-id mycc_
↪ 1:3a8c52d70c36313cfefbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173 --sequence 1 --
↪ tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪ example.com-cert.pem --waitForEvent
```

If you do not specify a policy, the chaincode definition will use the `Channel/Application/Endorsement`

policy by default, which requires that a transaction be validated by a majority of channel members. This policy depends on the membership of the channel, so it will be updated automatically when organizations are added or removed from a channel. One advantage of using channel policies is that they can be written to be updated automatically with channel membership.

If you specify an endorsement policy using the `--signature-policy` flag, you will need to update the policy when organizations join or leave the channel. A new organization added to the channel after the chaincode has been defined will be able to query a chaincode (provided the query has appropriate authorization as defined by channel policies and any application level checks enforced by the chaincode) but will not be able to execute or endorse the chaincode. Only organizations listed in the endorsement policy syntax will be able sign transactions.

Endorsement policy syntax

As you can see above, policies are expressed in terms of principals (“principals” are identities matched to a role). Principals are described as `'MSP.ROLE'`, where MSP represents the required MSP ID and ROLE represents one of the four accepted roles: `member`, `admin`, `client`, and `peer`.

Here are a few examples of valid principals:

- `'Org0.admin'`: any administrator of the `Org0` MSP
- `'Org1.member'`: any member of the `Org1` MSP
- `'Org1.client'`: any client of the `Org1` MSP
- `'Org1.peer'`: any peer of the `Org1` MSP

The syntax of the language is:

```
EXPR(E[, E...])
```

Where EXPR is either AND, OR, or OutOf, and E is either a principal (with the syntax described above) or another nested call to EXPR.

For example:

- `AND('Org1.member', 'Org2.member', 'Org3.member')` requests one signature from each of the three principals.
- `OR('Org1.member', 'Org2.member')` requests one signature from either one of the two principals.
- `OR('Org1.member', AND('Org2.member', 'Org3.member'))` requests either one signature from a member of the `Org1` MSP or one signature from a member of the `Org2` MSP and one signature from a member of the `Org3` MSP.
- `OutOf(1, 'Org1.member', 'Org2.member')`, which resolves to the same thing as `OR('Org1.member', 'Org2.member')`.
- Similarly, `OutOf(2, 'Org1.member', 'Org2.member')` is equivalent to `AND('Org1.member', 'Org2.member')`, and `OutOf(2, 'Org1.member', 'Org2.member', 'Org3.member')` is equivalent to `OR(AND('Org1.member', 'Org2.member'), AND('Org1.member', 'Org3.member'), AND('Org2.member', 'Org3.member'))`.

9.5.3 Setting collection-level endorsement policies

Similar to chaincode-level endorsement policies, when you approve and commit a chaincode definition, you can also specify the chaincode’s private data collections and corresponding collection-level endorsement policies. If a collection-level endorsement policy is set, transactions that write to a private data collection key will require that the specified organization peers have endorsed the transaction.

You can use collection-level endorsement policies to restrict which organization peers can write to the private data collection key namespace, for example to ensure that non-authorized organizations cannot write to a collection, and to have confidence that any state in a private data collection has been endorsed by the required collection organization(s).

The collection-level endorsement policy may be less restrictive or more restrictive than the chaincode-level endorsement policy and the collection's private data distribution policy. For example a majority of organizations may be required to endorse a chaincode transaction, but a specific organization may be required to endorse a transaction that includes a key in a specific collection.

The syntax for collection-level endorsement policies exactly matches the syntax for chaincode-level endorsement policies — in the collection configuration you can specify an `endorsementPolicy` with either a `signaturePolicy` or `channelConfigPolicy`. For more details see [Private Data](#).

9.5.4 Setting key-level endorsement policies

Setting regular chaincode-level or collection-level endorsement policies is tied to the lifecycle of the corresponding chaincode. They can only be set or modified when defining the chaincode on a channel.

In contrast, key-level endorsement policies can be set and modified in a more granular fashion from within a chaincode. The modification is part of the read-write set of a regular transaction.

The shim API provides the following functions to set and retrieve an endorsement policy for/from a regular key.

Note: `ep` below stands for the “endorsement policy”, which can be expressed either by using the same syntax described above or by using the convenience function described below. Either method will generate a binary version of the endorsement policy that can be consumed by the basic shim API.

```
SetStateValidationParameter(key string, ep []byte) error
GetStateValidationParameter(key string) ([]byte, error)
```

For keys that are part of *Private data* in a collection the following functions apply:

```
SetPrivateDataValidationParameter(collection, key string, ep []byte) error
GetPrivateDataValidationParameter(collection, key string) ([]byte, error)
```

To help set endorsement policies and marshal them into validation parameter byte arrays, the Go shim provides an extension with convenience functions that allow the chaincode developer to deal with endorsement policies in terms of the MSP identifiers of organizations, see [KeyEndorsementPolicy](#):

```
type KeyEndorsementPolicy interface {
    // Policy returns the endorsement policy as bytes
    Policy() ([]byte, error)

    // AddOrgs adds the specified orgs to the list of orgs that are required
    // to endorse
    AddOrgs(roleType RoleType, organizations ...string) error

    // DelOrgs delete the specified channel orgs from the existing key-level_
    ↪endorsement
    // policy for this KVS key. If any org is not present, an error will be returned.
    DelOrgs(organizations ...string) error

    // ListOrgs returns an array of channel orgs that are required to endorse changes
    ListOrgs() ([]string)
}
```

For example, to set an endorsement policy for a key where two specific orgs are required to endorse the key change, pass both org MSPIDs to `AddOrgs()`, and then call `Policy()` to construct the endorsement policy byte array that can be passed to `SetStateValidationParameter()`.

To add the shim extension to your chaincode as a dependency, see [Managing external dependencies for chaincode written in Go](#).

9.5.5 Validation

At commit time, setting a value of a key is no different from setting the endorsement policy of a key — both update the state of the key and are validated based on the same rules.

Validation	no validation parameter set	validation parameter set
modify value	check chaincode or collection ep	check key-level ep
modify key-level ep	check chaincode or collection ep	check key-level ep

As we discussed above, if a key is modified and no key-level endorsement policy is present, the chaincode-level or collection-level endorsement policy applies by default. This is also true when a key-level endorsement policy is set for a key for the first time — the new key-level endorsement policy must first be endorsed according to the pre-existing chaincode-level or collection-level endorsement policy.

If a key is modified and a key-level endorsement policy is present, the key-level endorsement policy overrides the chaincode-level or collection-level endorsement policy. In practice, this means that the key-level endorsement policy can be either less restrictive or more restrictive than the chaincode-level or collection-level endorsement policies. Because the chaincode-level or collection-level endorsement policy must be satisfied in order to set a key-level endorsement policy for the first time, no trust assumptions have been violated.

If a key's endorsement policy is removed (set to nil), the chaincode-level or collection-level endorsement policy becomes the default again.

If a transaction modifies multiple keys with different associated key-level endorsement policies, all of these policies need to be satisfied in order for the transaction to be valid.

9.6 Pluggable transaction endorsement and validation

9.6.1 Motivation

When a transaction is validated at time of commit, the peer performs various checks before applying the state changes that come with the transaction itself:

- Validating the identities that signed the transaction.
- Verifying the signatures of the endorsers on the transaction.
- Ensuring the transaction satisfies the endorsement policies of the namespaces of the corresponding chaincodes.

There are use cases which demand custom transaction validation rules different from the default Fabric validation rules, such as:

- **UTXO (Unspent Transaction Output):** When the validation takes into account whether the transaction doesn't double spend its inputs.
- **Anonymous transactions:** When the endorsement doesn't contain the identity of the peer, but a signature and a public key are shared that can't be linked to the peer's identity.

9.6.2 Pluggable endorsement and validation logic

Fabric allows for the implementation and deployment of custom endorsement and validation logic into the peer to be associated with chaincode handling. This logic can be compiled into the peer or built with the peer and deployed alongside it as a [Go plugin](#).

Note: Go plugins have a number of practical restrictions that require them to be compiled and linked in the same build environment as the peer. Differences in Go package versions, compiler versions, tags, and even GOPATH values will result in runtime failures when loading or executing the plugin logic.

By default, A chaincode will use the built in endorsement and validation logic. However, users have the option of selecting custom endorsement and validation plugins as part of the chaincode definition. An administrator can extend the endorsement/validation logic available to the peer by customizing the peer's local configuration.

9.6.3 Configuration

Each peer has a local configuration (`core.yaml`) that declares a mapping between the endorsement/validation logic name and the implementation that is to be run.

The default logic are called ESCC (with the “E” standing for endorsement) and VSCC (validation), and they can be found in the peer local configuration in the `handlers` section:

```
handlers:
  endorsers:
    escc:
      name: DefaultEndorsement
  validators:
    vsccl:
      name: DefaultValidation
```

When the endorsement or validation implementation is compiled into the peer, the `name` property represents the initialization function that is to be run in order to obtain the factory that creates instances of the endorsement/validation logic.

The function is an instance method of the `HandlerLibrary` construct under `core/handlers/library/library.go` and in order for custom endorsement or validation logic to be added, this construct needs to be extended with any additional methods.

If the custom code is built as a Go plugin, the `library` property must be provided and set to the location of the shared library.

For example, if we have custom endorsement and validation logic which is implemented as a plugin, we would have the following entries in the configuration in `core.yaml`:

```
handlers:
  endorsers:
    escc:
      name: DefaultEndorsement
    custom:
      name: customEndorsement
      library: /etc/hyperledger/fabric/plugins/customEndorsement.so
  validators:
    vsccl:
      name: DefaultValidation
    custom:
```

(continues on next page)

(continued from previous page)

```
name: customValidation
library: /etc/hyperledger/fabric/plugins/customValidation.so
```

And we'd have to place the `.so` plugin files in the peer's local file system.

The name of the custom plugin needs to be referenced by the chaincode definition to be used by the chaincode. If you are using the peer CLI to approve the chaincode definition, use the `--esc` and `--vsc` flag to select the name of the custom endorsement or validation library. If you are using the Fabric SDK for Node.js, visit [How to install and start your chaincode](#). For more information, see *Fabric chaincode lifecycle*.

Note: Hereafter, custom endorsement or validation logic implementation is going to be referred to as “plugins”, even if they are compiled into the peer.

9.6.4 Endorsement plugin implementation

To implement an endorsement plugin, one must implement the `Plugin` interface found in `core/handlers/endorsement/api/endorsement.go`:

```
// Plugin endorses a proposal response
type Plugin interface {
    // Endorse signs the given payload(ProposalResponsePayload bytes), and optionally
    // mutates it.
    // Returns:
    // The Endorsement: A signature over the payload, and an identity that is used to
    // verify the signature
    // The payload that was given as input (could be modified within this function)
    // Or error on failure
    Endorse(payload []byte, sp *peer.SignedProposal) (*peer.Endorsement, []byte,
    error)

    // Init injects dependencies into the instance of the Plugin
    Init(dependencies ...Dependency) error
}
```

An endorsement plugin instance of a given plugin type (identified either by the method name as an instance method of the `HandlerLibrary` or by the plugin `.so` file path) is created for each channel by having the peer invoke the `New` method in the `PluginFactory` interface which is also expected to be implemented by the plugin developer:

```
// PluginFactory creates a new instance of a Plugin
type PluginFactory interface {
    New() Plugin
}
```

The `Init` method is expected to receive as input all the dependencies declared under `core/handlers/endorsement/api/`, identified as embedding the `Dependency` interface.

After the creation of the `Plugin` instance, the `Init` method is invoked on it by the peer with the dependencies passed as parameters.

Currently Fabric comes with the following dependencies for endorsement plugins:

- `SigningIdentityFetcher`: Returns an instance of `SigningIdentity` based on a given signed proposal:

```
// SigningIdentity signs messages and serializes its public identity to bytes
type SigningIdentity interface {
    // Serialize returns a byte representation of this identity which is used to
    // verify
    // messages signed by this SigningIdentity
    Serialize() ([]byte, error)

    // Sign signs the given payload and returns a signature
    Sign([]byte) ([]byte, error)
}
```

- StateFetcher: Fetches a **State** object which interacts with the world state:

```
// State defines interaction with the world state
type State interface {
    // GetPrivateDataMultipleKeys gets the values for the multiple private data items
    // in a single call
    GetPrivateDataMultipleKeys(namespace string, collection string, keys []string) ([][]byte,
    error)

    // GetStateMultipleKeys gets the values for multiple keys in a single call
    GetStateMultipleKeys(namespace string, keys []string) ([][]byte, error)

    // GetTransientByTXID gets the values private data associated with the given txID
    GetTransientByTXID(txID string) (*rwset.TxPvtReadWriteSet, error)

    // Done releases resources occupied by the State
    Done()
}
```

9.6.5 Validation plugin implementation

To implement a validation plugin, one must implement the Plugin interface found in `core/handlers/validation/api/validation.go`:

```
// Plugin validates transactions
type Plugin interface {
    // Validate returns nil if the action at the given position inside the transaction
    // at the given position in the given block is valid, or an error if not.
    Validate(block *common.Block, namespace string, txPosition int, actionPosition
    int, contextData ...ContextDatum) error

    // Init injects dependencies into the instance of the Plugin
    Init(dependencies ...Dependency) error
}
```

Each ContextDatum is additional runtime-derived metadata that is passed by the peer to the validation plugin. Currently, the only ContextDatum that is passed is one that represents the endorsement policy of the chaincode:

```
// SerializedPolicy defines a serialized policy
type SerializedPolicy interface {
    validation.ContextDatum

    // Bytes returns the bytes of the SerializedPolicy
    Bytes() []byte
}
```

A validation plugin instance of a given plugin type (identified either by the method name as an instance method of the `HandlerLibrary` or by the plugin .so file path) is created for each channel by having the peer invoke the `New` method in the `PluginFactory` interface which is also expected to be implemented by the plugin developer:

```
// PluginFactory creates a new instance of a Plugin
type PluginFactory interface {
    New() Plugin
}
```

The `Init` method is expected to receive as input all the dependencies declared under `core/handlers/validation/api/`, identified as embedding the `Dependency` interface.

After the creation of the `Plugin` instance, the **Init** method is invoked on it by the peer with the dependencies passed as parameters.

Currently Fabric comes with the following dependencies for validation plugins:

- `IdentityDeserializer`: Converts byte representation of identities into `Identity` objects that can be used to verify signatures signed by them, be validated themselves against their corresponding MSP, and see whether they satisfy a given **MSP Principal**. The full specification can be found in `core/handlers/validation/api/identities/identities.go`.
- `PolicyEvaluator`: Evaluates whether a given policy is satisfied:

```
// PolicyEvaluator evaluates policies
type PolicyEvaluator interface {
    validation.Dependency

    // Evaluate takes a set of SignedData and evaluates whether this set of
    // signatures satisfies
    // the policy with the given bytes
    Evaluate(policyBytes []byte, signatureSet []*common.SignedData) error
}
```

- `StateFetcher`: Fetches a `State` object which interacts with the world state:

```
// State defines interaction with the world state
type State interface {
    // GetStateMultipleKeys gets the values for multiple keys in a single call
    GetStateMultipleKeys(namespace string, keys []string) ([][]byte, error)

    // GetStateRangeScanIterator returns an iterator that contains all the key-values
    // between given key ranges.
    // startKey is included in the results and endKey is excluded. An empty startKey
    // refers to the first available key
    // and an empty endKey refers to the last available key. For scanning all the
    // keys, both the startKey and the endKey
    // can be supplied as empty strings. However, a full scan should be used
    // judiciously for performance reasons.
    // The returned ResultsIterator contains results of type *KV which is defined in
    // fabric-protos/ledger/queryresult.
    GetStateRangeScanIterator(namespace string, startKey string, endKey string)
    (ResultsIterator, error)

    // GetStateMetadata returns the metadata for given namespace and key
    GetStateMetadata(namespace, key string) (map[string][]byte, error)

    // GetPrivateDataMetadata gets the metadata of a private data item identified by
    // a tuple <namespace, collection, key>
```

(continues on next page)

(continued from previous page)

```

    GetPrivateDataMetadata(namespace, collection, key string) (map[string][]byte,
↳error)

    // Done releases resources occupied by the State
    Done()
}

```

9.6.6 Important notes

- **Validation plugin consistency across peers:** In future releases, the Fabric channel infrastructure would guarantee that the same validation logic is used for a given chaincode by all peers in the channel at any given blockchain height in order to eliminate the chance of mis-configuration which would might lead to state divergence among peers that accidentally run different implementations. However, for now it is the sole responsibility of the system operators and administrators to ensure this doesn't happen.
- **Validation plugin error handling:** Whenever a validation plugin can't determine whether a given transaction is valid or not, because of some transient execution problem like inability to access the database, it should return an error of type **ExecutionFailureError** that is defined in `core/handlers/validation/api/validation.go`. Any other error that is returned, is treated as an endorsement policy error and marks the transaction as invalidated by the validation logic. However, if an **ExecutionFailureError** is returned, the chain processing halts instead of marking the transaction as invalid. This is to prevent state divergence between different peers.
- **Error handling for private metadata retrieval:** In case a plugin retrieves metadata for private data by making use of the `StateFetcher` interface, it is important that errors are handled as follows: `CollConfigNotDefinedError` and `InvalidCollNameError`, signalling that the specified collection does not exist, should be handled as deterministic errors and should not lead the plugin to return an **ExecutionFailureError**.
- **Importing Fabric code into the plugin:** Importing code that belongs to Fabric other than protobufs as part of the plugin is highly discouraged, and can lead to issues when the Fabric code changes between releases, or can cause inoperability issues when running mixed peer versions. Ideally, the plugin code should only use the dependencies given to it, and should import the bare minimum other than protobufs.

9.7 Access Control Lists (ACL)

9.7.1 What is an Access Control List?

Note: This topic deals with access control and policies on a channel administration level. To learn about access control within a chaincode, check out our [chaincode for developers tutorial](#).

Fabric uses access control lists (ACLs) to manage access to resources by associating a [Policy](#) with a resource. Fabric contains a number of default ACLs. In this document, we'll talk about how they're formatted and how the defaults can be overridden.

But before we can do that, it's necessary to understand a little about resources and policies.

Resources

Users interact with Fabric by targeting a [user chaincode](#), or an [events stream source](#), or system chaincode that are called in the background. As such, these endpoints are considered "resources" on which access control should be exercised.

Application developers need to be aware of these resources and the default policies associated with them. The complete list of these resources are found in `configtx.yaml`. You can look at a [sample configtx.yaml file here](#).

The resources named in `configtx.yaml` is an exhaustive list of all internal resources currently defined by Fabric. The loose convention adopted there is `<component>/<resource>`. So `csc/GetConfigBlock` is the resource for the `GetConfigBlock` call in the `CSCC` component.

Policies

Policies are fundamental to the way Fabric works because they allow the identity (or set of identities) associated with a request to be checked against the policy associated with the resource needed to fulfill the request. Endorsement policies are used to determine whether a transaction has been appropriately endorsed. The policies defined in the channel configuration are referenced as modification policies as well as for access control, and are defined in the channel configuration itself.

Policies can be structured in one of two ways: as `Signature` policies or as an `ImplicitMeta` policy.

Signature policies

These policies identify specific users who must sign in order for a policy to be satisfied. For example:

```
Policies:
  MyPolicy:
    Type: Signature
    Rule: "OR('Org1.peer', 'Org2.peer')"
```

This policy construct can be interpreted as: *the policy named `MyPolicy` can only be satisfied by the signature of an identity with role of “a peer from Org1” or “a peer from Org2”*.

Signature policies support arbitrary combinations of `AND`, `OR`, and `NOutOf`, allowing the construction of extremely powerful rules like: “An admin of org A and two other admins, or 11 of 20 org admins”.

ImplicitMeta policies

`ImplicitMeta` policies aggregate the result of policies deeper in the configuration hierarchy that are ultimately defined by `Signature` policies. They support default rules like “A majority of the organization admins”. These policies use a different but still very simple syntax as compared to `Signature` policies: `<ALL|ANY|MAJORITY> <sub_policy>`.

For example: `ANY Readers` or `MAJORITY Admins`.

Note that in the default policy configuration Admins have an operational role. Policies that specify that only Admins — or some subset of Admins — have access to a resource will tend to be for sensitive or operational aspects of the network (such as instantiating chaincode on a channel). Writers will tend to be able to propose ledger updates, such as a transaction, but will not typically have administrative permissions. Readers have a passive role. They can access information but do not have the permission to propose ledger updates nor do can they perform administrative tasks. These default policies can be added to, edited, or supplemented, for example by the new peer and client roles (if you have NodeOU support).

Here’s an example of an `ImplicitMeta` policy structure:

```
Policies:
  AnotherPolicy:
    Type: ImplicitMeta
    Rule: "MAJORITY Admins"
```

Here, the policy `AnotherPolicy` can be satisfied by the MAJORITY of Admins, where Admins is eventually being specified by lower level Signature policy.

Where is access control specified?

Access control defaults exist inside `configtx.yaml`, the file that `configtxgen` uses to build channel configurations.

Access control can be updated in one of two ways, either by editing `configtx.yaml` itself, which will be used when creating new channel configurations, or by updating access control in the channel configuration of an existing channel.

9.7.2 How ACLs are formatted in `configtx.yaml`

ACLs are formatted as a key-value pair consisting of a resource function name followed by a string. To see what this looks like, reference this [sample configtx.yaml file](#).

Two excerpts from this sample:

```
# ACL policy for invoking chaincodes on peer
peer/Propose: /Channel/Application/Writers
```

```
# ACL policy for sending block events
event/Block: /Channel/Application/Readers
```

These ACLs define that access to `peer/Propose` and `event/Block` resources is restricted to identities satisfying the policy defined at the canonical path `/Channel/Application/Writers` and `/Channel/Application/Readers`, respectively.

Updating ACL defaults in `configtx.yaml`

In cases where it will be necessary to override ACL defaults when bootstrapping a network, or to change the ACLs before a channel has been bootstrapped, the best practice will be to update `configtx.yaml`.

Let's say you want to modify the `peer/Propose` ACL default — which specifies the policy for invoking chaincodes on a peer — from `/Channel/Application/Writers` to a policy called `MyPolicy`.

This is done by adding a policy called `MyPolicy` (it could be called anything, but for this example we'll call it `MyPolicy`). The policy is defined in the `Application.Policies` section inside `configtx.yaml` and specifies a rule to be checked to grant or deny access to a user. For this example, we'll be creating a Signature policy identifying `SampleOrg.admin`.

```
Policies: &ApplicationDefaultPolicies
  Readers:
    Type: ImplicitMeta
    Rule: "ANY Readers"
  Writers:
    Type: ImplicitMeta
    Rule: "ANY Writers"
  Admins:
    Type: ImplicitMeta
    Rule: "MAJORITY Admins"
  MyPolicy:
    Type: Signature
    Rule: "OR('SampleOrg.admin')"
```

Then, edit the `Application: ACLs` section inside `configtx.yaml` to change `peer/Propose` from this:

```
peer/Propose: /Channel/Application/Writers
```

To this:

```
peer/Propose: /Channel/Application/MyPolicy
```

Once these fields have been changed in `configtx.yaml`, the `configtxgen` tool will use the policies and ACLs defined when creating a channel creation transaction. When appropriately signed and submitted by one of the admins of the consortium members, a new channel with the defined ACLs and policies is created.

Once `MyPolicy` has been bootstrapped into the channel configuration, it can also be referenced to override other ACL defaults. For example:

```
SampleSingleMSPChannel:
  Consortium: SampleConsortium
  Application:
    <<: *ApplicationDefaults
    ACLs:
      <<: *ACLsDefault
      event/Block: /Channel/Application/MyPolicy
```

This would restrict the ability to subscribe to block events to `SampleOrg.admin`.

If channels have already been created that want to use this ACL, they'll have to update their channel configurations one at a time using the following flow:

Updating ACL defaults in the channel config

If channels have already been created that want to use `MyPolicy` to restrict access to `peer/Propose` — or if they want to create ACLs they don't want other channels to know about — they'll have to update their channel configurations one at a time through config update transactions.

Note: Channel configuration transactions are an involved process we won't delve into here. If you want to read more about them check out our document on [channel configuration updates](#) and our [“Adding an Org to a Channel” tutorial](#).

After pulling, translating, and stripping the configuration block of its metadata, you would edit the configuration by adding `MyPolicy` under `Application: policies`, where the `Admins`, `Writers`, and `Readers` policies already live.

```
"MyPolicy": {
  "mod_policy": "Admins",
  "policy": {
    "type": 1,
    "value": {
      "identities": [
        {
          "principal": {
            "msp_identifier": "SampleOrg",
            "role": "ADMIN"
          },
          "principal_classification": "ROLE"
        }
      ],
      "rule": {
        "n_out_of": {
          "n": 1,
          "rules": [
```

(continues on next page)

(continued from previous page)

```

        {
            "signed_by": 0
        }
    ]
}
},
"version": 0
}
},
"version": "0"
},

```

Note in particular the `msp_idenfifer` and `role` here.

Then, in the ACLs section of the config, change the `peer/Propose` ACL from this:

```

"peer/Propose": {
    "policy_ref": "/Channel/Application/Writers"

```

To this:

```

"peer/Propose": {
    "policy_ref": "/Channel/Application/MyPolicy"

```

Note: If you do not have ACLs defined in your channel configuration, you will have to add the entire ACL structure.

Once the configuration has been updated, it will need to be submitted by the usual channel update process.

Satisfying an ACL that requires access to multiple resources

If a member makes a request that calls multiple system chaincodes, all of the ACLs for those system chaincodes must be satisfied.

For example, `peer/Propose` refers to any proposal request on a channel. If the particular proposal requires access to two system chaincodes that requires an identity satisfying `Writers` and one system chaincode that requires an identity satisfying `MyPolicy`, then the member submitting the proposal must have an identity that evaluates to “true” for both `Writers` and `MyPolicy`.

In the default configuration, `Writers` is a signature policy whose rule is `SampleOrg.member`. In other words, “any member of my organization”. `MyPolicy`, listed above, has a rule of `SampleOrg.admin`, or “any admin of my organization”. To satisfy these ACLs, the member would have to be both an administrator and a member of `SampleOrg`. By default, all administrators are members (though not all administrators are members), but it is possible to overwrite these policies to whatever you want them to be. As a result, it’s important to keep track of these policies to ensure that the ACLs for peer proposals are not impossible to satisfy (unless that is the intention).

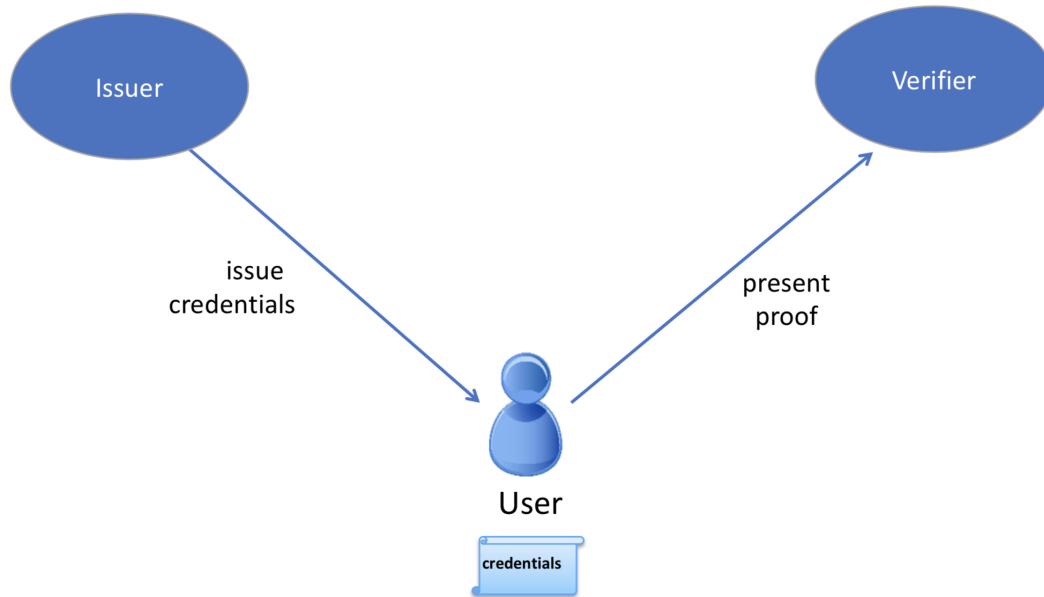
9.8 MSP Implementation with Identity Mixer

9.8.1 What is Idemix?

Idemix is a cryptographic protocol suite, which provides strong authentication as well as privacy-preserving features such as **anonymity**, the ability to transact without revealing the identity of the transactor, and **unlinkability**, the ability of a single identity to send multiple transactions without revealing that the transactions were sent by the same identity.

There are three actors involved in an Idemix flow: **user**, **issuer**, and **verifier**.

Identity Mixer Overview



- An issuer certifies a set of user’s attributes are issued in the form of a digital certificate, hereafter called “credential”.
- The user later generates a “[zero-knowledge proof](#)” of possession of the credential and also selectively discloses only the attributes the user chooses to reveal. The proof, because it is zero-knowledge, reveals no additional information to the verifier, issuer, or anyone else.

As an example, suppose “Alice” needs to prove to Bob (a store clerk) that she has a driver’s license issued to her by the DMV.

In this scenario, Alice is the user, the DMV is the issuer, and Bob is the verifier. In order to prove to Bob that Alice has a driver’s license, she could show it to him. However, Bob would then be able to see Alice’s name, address, exact age, etc. — much more information than Bob needs to know.

Instead, Alice can use Idemix to generate a “zero-knowledge proof” for Bob, which only reveals that she has a valid driver’s license and nothing else.

So from the proof:

- Bob does not learn any additional information about Alice other than the fact that she has a valid license (anonymity).
- If Alice visits the store multiple times and generates a proof each time for Bob, Bob would not be able to tell from the proof that it was the same person (unlinkability).

Idemix authentication technology provides the trust model and security guarantees that are similar to what is ensured by standard X.509 certificates but with underlying cryptographic algorithms that efficiently provide advanced privacy features including the ones described above. We’ll compare Idemix and X.509 technologies in detail in the technical section below.

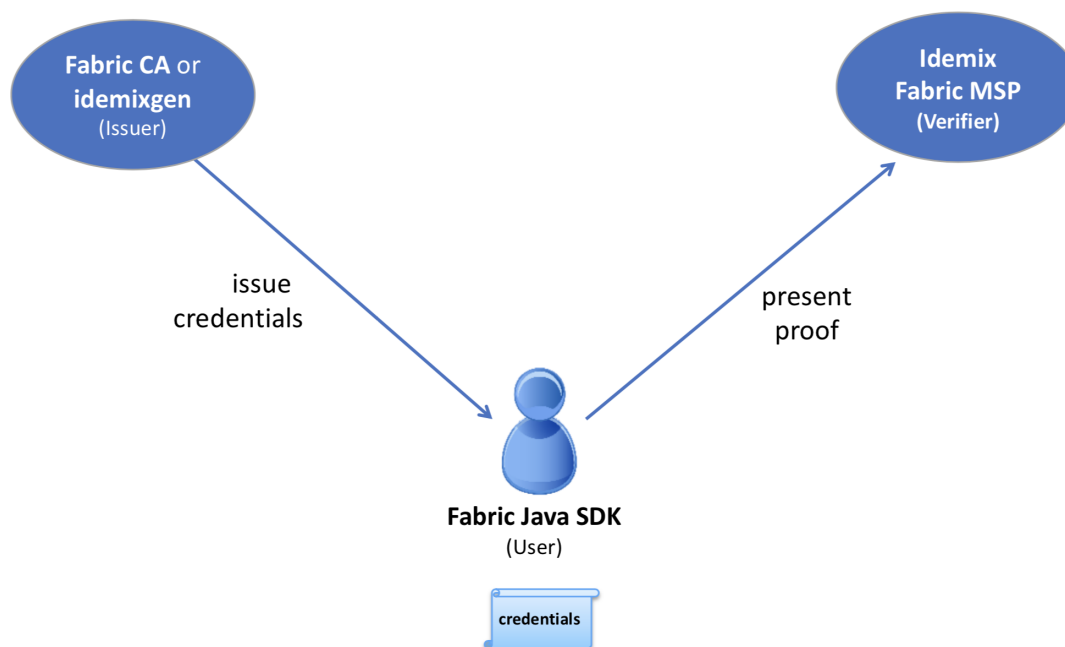
9.8.2 How to use Idemix

To understand how to use Idemix with Hyperledger Fabric, we need to see which Fabric components correspond to the user, issuer, and verifier in Idemix.

- The Fabric Java SDK is the API for the **user**. In the future, other Fabric SDKs will also support Idemix.
- Fabric provides two possible Idemix **issuers**:
 1. Fabric CA for production environments or development, and
 2. the *idemixgen* tool for development environments.
- The **verifier** is an Idemix MSP in Fabric.

In order to use Idemix in Hyperledger Fabric, the following three basic steps are required:

Identity Mixer In Hyperledger Fabric



Compare the roles in this image to the ones above.

1. Consider the issuer.

Fabric CA (version 1.3 or later) has been enhanced to automatically function as an Idemix issuer. When `fabric-ca-server` is started (or initialized via the `fabric-ca-server init` command), the following two files are automatically created in the home directory of the `fabric-ca-server`: `IssuerPublicKey` and `IssuerRevocationPublicKey`. These files are required in step 2.

For a development environment and if you are not using Fabric CA, you may use `idemixgen` to create these files.

2. Consider the verifier.

You need to create an Idemix MSP using the `IssuerPublicKey` and `IssuerRevocationPublicKey` from step 1.

For example, consider the following excerpt from `configtx.yaml` in the Hyperledger Java SDK sample:

```
- &Org1Idemix
  # defaultorg defines the organization which is used in the sampleconfig
  # of the fabric.git development environment
  name: idemixMSP1

  # id to load the msp definition as
  id: idemixMSPID1

  msptype: idemix
  mspdir: crypto-config/peerOrganizations/org3.example.com
```

The `msptype` is set to `idemix` and the contents of the `mspdir` directory (`crypto-config/peerOrganizations/org3.example.com/msp` in this example) contains the `IssuerPublicKey` and `IssuerRevocationPublicKey` files.

Note that in this example, `Org1Idemix` represents the Idemix MSP for `Org1` (not shown), which would also have an X509 MSP.

3. Consider the user. Recall that the Java SDK is the API for the user.

There is only a single additional API call required in order to use Idemix with the Java SDK: the `idemixEnroll` method of the `org.hyperledger.fabric_ca.sdk.HFCAClient` class. For example, assume `hfcaClient` is your `HFCAClient` object and `x509Enrollment` is your `org.hyperledger.fabric.sdk.Enrollment` associated with your X509 certificate.

The following call will return an `org.hyperledger.fabric.sdk.Enrollment` object associated with your Idemix credential.

```
IdemixEnrollment idemixEnrollment = hfcaClient.idemixEnroll(x509enrollment,
    ↪ "idemixMSPID1");
```

Note also that `IdemixEnrollment` implements the `org.hyperledger.fabric.sdk.Enrollment` interface and can, therefore, be used in the same way that one uses the X509 enrollment object, except, of course, that this automatically provides the privacy enhancing features of Idemix.

9.8.3 Idemix and chaincode

From a verifier perspective, there is one more actor to consider: chaincode. What can chaincode learn about the transactor when an Idemix credential is used?

The `cid` ([Client Identity](#)) library (for Go only) has been extended to support the `GetAttributeValue` function when an Idemix credential is used. However, as mentioned in the “Current limitations” section below, there are only two attributes which are disclosed in the Idemix case: `ou` and `role`.

If Fabric CA is the credential issuer:

- the value of the `ou` attribute is the identity’s **affiliation** (e.g. “org1.department1”);
- the value of the `role` attribute will be either ‘member’ or ‘admin’. A value of ‘admin’ means that the identity is an MSP administrator. By default, identities created by Fabric CA will return the ‘member’ role. In order to create an ‘admin’ identity, register the identity with the `role` attribute and a value of 2.

For an example of setting an affiliation in the Java SDK see this [sample](#).

For an example of using the CID library in go chaincode to retrieve attributes, see this [go chaincode](#).

Idemix organizations cannot be used to endorse a chaincode or approve a chaincode definition. This needs to be taken into account when you set the `LifecycleEndorsement` and `Endorsement` policies on your channels. For more information, see the limitations section below.

9.8.4 Current limitations

The current version of Idemix does have a few limitations.

- **Idemix organizations and endorsement policies**

Idemix organizations cannot be used to endorse a chaincode transaction or approve a chaincode definition. By default, the `Channel/Application/LifecycleEndorsement` and `Channel/Application/Endorsement` policies will require signatures from a majority of organizations active on the channel. This implies that a channel that contains a large number of Idemix organizations may not be able to reach the majority needed to fulfill the default policy. For example, if a channel has two MSP Organizations and two Idemix organizations, the channel policy will require that three out of four organizations approve a chaincode definition to commit that definition to the channel. Because Idemix organizations cannot approve a chaincode definition, the policy will only be able to validate two out of four signatures.

If your channel contains a sufficient number of Idemix organizations to affect the endorsement policy, you can use a signature policy to explicitly specify the required MSP organizations.

- **Fixed set of attributes**

It is not yet possible to issue or use an Idemix credential with custom attributes. Custom attributes will be supported in a future release.

The following four attributes are currently supported:

1. Organizational Unit attribute (“ou”):
 - Usage: same as X.509
 - Type: String
 - Revealed: always
2. Role attribute (“role”):
 - Usage: same as X.509
 - Type: integer
 - Revealed: always
3. Enrollment ID attribute
 - Usage: uniquely identify a user — same in all enrollment credentials that belong to the same user (will be used for auditing in the future releases)
 - Type: BIG
 - Revealed: never in the signature, only when generating an authentication token for Fabric CA
4. Revocation Handle attribute
 - Usage: uniquely identify a credential (will be used for revocation in future releases)
 - Type: integer
 - Revealed: never

- **Revocation is not yet supported**

Although much of the revocation framework is in place as can be seen by the presence of a revocation handle attribute mentioned above, revocation of an Idemix credential is not yet supported.

- **Peers do not use Idemix for endorsement**

Currently, Idemix MSP is used by the peers only for signature verification. Signing with Idemix is only done via Client SDK. More roles (including a ‘peer’ role) will be supported by Idemix MSP.

9.8.5 Technical summary

Comparing Idemix credentials to X.509 certificates

The certificate/credential concept and the issuance process are very similar in Idemix and X.509 certs: a set of attributes is digitally signed with a signature that cannot be forged and there is a secret key to which a credential is cryptographically bound.

The main difference between a standard X.509 certificate and an Identity Mixer credential is the signature scheme that is used to certify the attributes. The signatures underlying the Identity Mixer system allow for efficient proofs of the possession of a signature and the corresponding attributes without revealing the signature and (selected) attribute values themselves. We use zero-knowledge proofs to ensure that such “knowledge” or “information” is not revealed while ensuring that the signature over some attributes is valid and the user is in possession of the corresponding credential secret key.

Such proofs, like X.509 certificates, can be verified with the public key of the authority that originally signed the credential and cannot be successfully forged. Only the user who knows the credential secret key can generate the proofs about the credential and its attributes.

With regard to unlinkability, when an X.509 certificate is presented, all attributes have to be revealed to verify the certificate signature. This implies that all certificate usages for signing transactions are linkable.

To avoid such linkability, fresh X.509 certificates need to be used every time, which results in complex key management and communication and storage overhead. Furthermore, there are cases where it is important that not even the CA issuing the certificates is able to link all the transactions to the user.

Idemix helps to avoid linkability with respect to both the CA and verifiers, since even the CA is not able to link proofs to the original credential. Neither the issuer nor a verifier can tell whether two proofs were derived from the same credential (or from two different ones).

More details on the concepts and features of the Identity Mixer technology are described in the paper [Concepts and Languages for Privacy-Preserving Attribute-Based Authentication](#).

Topology Information

Given the above limitations, it is recommended to have only one Idemix-based MSP per channel or, at the extreme, per network. Indeed, for example, having multiple Idemix-based MSPs per channel would allow a party, reading the ledger of that channel, to tell apart transactions signed by parties belonging to different Idemix-based MSPs. This is because, each transaction leak the MSP-ID of the signer. In other words, Idemix currently provides only anonymity of clients among the same organization (MSP).

In the future, Idemix could be extended to support anonymous hierarchies of Idemix-based Certification Authorities whose certified credentials can be verified by using a unique public-key, therefore achieving anonymity across organizations (MSPs). This would allow multiple Idemix-based MSPs to coexist in the same channel.

In principal, a channel can be configured to have a single Idemix-based MSP and multiple X.509-based MSPs. Of course, the interaction between these MSP can potential leak information. An assessment of the leaked information need to be done case by case.wq

Underlying cryptographic protocols

Idemix technology is built from a blind signature scheme that supports multiple messages and efficient zero-knowledge proofs of signature possession. All of the cryptographic building blocks for Idemix were published at the top conferences and journals and verified by the scientific community.

This particular Idemix implementation for Fabric uses a pairing-based signature scheme that was briefly proposed by [Camenisch and Lysyanskaya](#) and described in detail by [Au et al.](#). The ability to prove knowledge of a signature in a zero-knowledge proof [Camenisch et al.](#) was used.

9.9 Identity Mixer MSP configuration generator (idemixgen)

This document describes the usage for the `idemixgen` utility, which can be used to create configuration files for the identity mixer based MSP. Two commands are available, one for creating a fresh CA key pair, and one for creating an MSP config using a previously generated CA key.

9.9.1 Directory Structure

The `idemixgen` tool will create directories with the following structure:

```
- /ca/
  IssuerSecretKey
  IssuerPublicKey
  RevocationKey
- /msp/
  IssuerPublicKey
  RevocationPublicKey
- /user/
  SignerConfig
```

The `ca` directory contains the issuer secret key (including the revocation key) and should only be present for a CA. The `msp` directory contains the information required to set up an MSP verifying idemix signatures. The `user` directory specifies a default signer.

9.9.2 CA Key Generation

CA (issuer) keys suitable for identity mixer can be created using command `idemixgen ca-keygen`. This will create directories `ca` and `msp` in the working directory.

9.9.3 Adding a Default Signer

After generating the `ca` and `msp` directories with `idemixgen ca-keygen`, a default signer specified in the `user` directory can be added to the config with `idemixgen signerconfig`.

```
$ idemixgen signerconfig -h
usage: idemixgen signerconfig [<flags>]

Generate a default signer for this Idemix MSP

Flags:
  -h, --help                Show context-sensitive help (also try --help-long and --
↪help-man) .
  -u, --org-unit=ORG-UNIT  The Organizational Unit of the default signer
  -a, --admin               Make the default signer admin
  -e, --enrollment-id=ENROLLMENT-ID
                           The enrollment id of the default signer
  -r, --revocation-handle=REVOCATION-HANDLE
                           The handle used to revoke this signer
```

For example, we can create a default signer that is a member of organizational unit “OrgUnit1”, with enrollment identity “johndoe”, revocation handle “1234”, and that is an admin, with the following command:

```
idemixgen signerconfig -u OrgUnit1 --admin -e "johndoe" -r 1234
```

9.10 The Operations Service

The peer and the orderer host an HTTP server that offers a RESTful “operations” API. This API is unrelated to the Fabric network services and is intended to be used by operators, not administrators or “users” of the network.

The API exposes the following capabilities:

- Log level management
- Health checks
- Prometheus target for operational metrics (when configured)
- Endpoint for retrieving version information

9.10.1 Configuring the Operations Service

The operations service requires two basic pieces of configuration:

- The **address** and **port** to listen on.
- The **TLS certificates** and **keys** to use for authentication and encryption. Note, **these certificates should be generated by a separate and dedicated CA**. Do not use a CA that has generated certificates for any organizations in any channels.

Peer

For each peer, the operations server can be configured in the `operations` section of `core.yaml`:

```
operations:
  # host and port for the operations server
  listenAddress: 127.0.0.1:9443

  # TLS configuration for the operations endpoint
  tls:
    # TLS enabled
    enabled: true

    # path to PEM encoded server certificate for the operations server
    cert:
      file: tls/server.crt

    # path to PEM encoded server key for the operations server
    key:
      file: tls/server.key

    # most operations service endpoints require client authentication when TLS
    # is enabled. clientAuthRequired requires client certificate authentication
    # at the TLS layer to access all resources.
    clientAuthRequired: false
```

(continues on next page)

(continued from previous page)

```
# paths to PEM encoded ca certificates to trust for client authentication
clientRootCAs:
  files: []
```

The `listenAddress` key defines the host and port that the operation server will listen on. If the server should listen on all addresses, the host portion can be omitted.

The `tls` section is used to indicate whether or not TLS is enabled for the operations service, the location of the service's certificate and private key, and the locations of certificate authority root certificates that should be trusted for client authentication. When `enabled` is `true`, most of the operations service endpoints require client authentication, therefore `clientRootCAs.files` must be set. When `clientAuthRequired` is `true`, the TLS layer will require clients to provide a certificate for authentication on every request. See Operations Security section below for more details.

Orderer

For each orderer, the operations server can be configured in the *Operations* section of `orderer.yaml`:

```
Operations:
  # host and port for the operations server
  ListenAddress: 127.0.0.1:8443

  # TLS configuration for the operations endpoint
  TLS:
    # TLS enabled
    Enabled: true

    # PrivateKey: PEM-encoded tls key for the operations endpoint
    PrivateKey: tls/server.key

    # Certificate governs the file location of the server TLS certificate.
    Certificate: tls/server.crt

    # Paths to PEM encoded ca certificates to trust for client authentication
    ClientRootCAs: []

    # Most operations service endpoints require client authentication when TLS
    # is enabled. ClientAuthRequired requires client certificate authentication
    # at the TLS layer to access all resources.
    ClientAuthRequired: false
```

The `ListenAddress` key defines the host and port that the operations server will listen on. If the server should listen on all addresses, the host portion can be omitted.

The `TLS` section is used to indicate whether or not TLS is enabled for the operations service, the location of the service's certificate and private key, and the locations of certificate authority root certificates that should be trusted for client authentication. When `Enabled` is `true`, most of the operations service endpoints require client authentication, therefore `RootCAs` must be set. When `ClientAuthRequired` is `true`, the TLS layer will require clients to provide a certificate for authentication on every request. See Operations Security section below for more details.

Operations Security

As the operations service is focused on operations and intentionally unrelated to the Fabric network, it does not use the Membership Services Provider for access control. Instead, the operations service relies entirely on mutual TLS

with client certificate authentication.

When TLS is disabled, authorization is bypassed and any client that can connect to the operations endpoint will be able to use the API.

When TLS is enabled, a valid client certificate must be provided in order to access all resources unless explicitly noted otherwise below.

When `clientAuthRequired` is also enabled, the TLS layer will require a valid client certificate regardless of the resource being accessed.

Log Level Management

The operations service provides a `/logspec` resource that operators can use to manage the active logging spec for a peer or orderer. The resource is a conventional REST resource and supports GET and PUT requests.

When a GET `/logspec` request is received by the operations service, it will respond with a JSON payload that contains the current logging specification:

```
{ "spec": "info" }
```

When a PUT `/logspec` request is received by the operations service, it will read the body as a JSON payload. The payload must consist of a single attribute named `spec`.

```
{ "spec": "chaincode=debug:info" }
```

If the spec is activated successfully, the service will respond with a 204 "No Content" response. If an error occurs, the service will respond with a 400 "Bad Request" and an error payload:

```
{ "error": "error message" }
```

9.10.2 Health Checks

The operations service provides a `/healthz` resource that operators can use to help determine the liveness and health of peers and orderers. The resource is a conventional REST resource that supports GET requests. The implementation is intended to be compatible with the liveness probe model used by Kubernetes but can be used in other contexts.

When a GET `/healthz` request is received, the operations service will call all registered health checkers for the process. When all of the health checkers return successfully, the operations service will respond with a 200 "OK" and a JSON body:

```
{
  "status": "OK",
  "time": "2009-11-10T23:00:00Z"
}
```

If one or more of the health checkers returns an error, the operations service will respond with a 503 "Service Unavailable" and a JSON body that includes information about which health checker failed:

```
{
  "status": "Service Unavailable",
  "time": "2009-11-10T23:00:00Z",
  "failed_checks": [
    {
      "component": "docker",
      "reason": "failed to connect to Docker daemon: invalid endpoint"
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

    }
  ]
}
```

In the current version, the only health check that is registered is for Docker. Future versions will be enhanced to add additional health checks.

When TLS is enabled, a valid client certificate is not required to use this service unless `clientAuthRequired` is set to `true`.

9.10.3 Metrics

Some components of the Fabric peer and orderer expose metrics that can help provide insight into the behavior of the system. Operators and administrators can use this information to better understand how the system is performing over time.

Configuring Metrics

Fabric provides two ways to expose metrics: a **pull** model based on Prometheus and a **push** model based on StatsD.

Prometheus

A typical Prometheus deployment scrapes metrics by requesting them from an HTTP endpoint exposed by instrumented targets. As Prometheus is responsible for requesting the metrics, it is considered a pull system.

When configured, a Fabric peer or orderer will present a `/metrics` resource on the operations service.

Peer

A peer can be configured to expose a `/metrics` endpoint for Prometheus to scrape by setting the metrics provider to `prometheus` in the `metrics` section of `core.yaml`.

```
metrics:
  provider: prometheus
```

Orderer

An orderer can be configured to expose a `/metrics` endpoint for Prometheus to scrape by setting the metrics provider to `prometheus` in the `Metrics` section of `orderer.yaml`.

```
Metrics:
  Provider: prometheus
```

StatsD

StatsD is a simple statistics aggregation daemon. Metrics are sent to a `statsd` daemon where they are collected, aggregated, and pushed to a backend for visualization and alerting. As this model requires instrumented processes to send metrics data to StatsD, this is considered a push system.

Peer

A peer can be configured to send metrics to StatsD by setting the metrics provider to `statsd` in the `metrics` section of `core.yaml`. The `statsd` subsection must also be configured with the address of the StatsD daemon, the network type to use (`tcp` or `udp`), and how often to send the metrics. An optional `prefix` may be specified to help differentiate the source of the metrics — for example, differentiating metrics coming from separate peers — that would be prepended to all generated metrics.

```
metrics:
  provider: statsd
  statsd:
    network: udp
    address: 127.0.0.1:8125
    writeInterval: 10s
    prefix: peer-0
```

Orderer

An orderer can be configured to send metrics to StatsD by setting the metrics provider to `statsd` in the `Metrics` section of `orderer.yaml`. The `Statsd` subsection must also be configured with the address of the StatsD daemon, the network type to use (`tcp` or `udp`), and how often to send the metrics. An optional `prefix` may be specified to help differentiate the source of the metrics.

```
Metrics:
  Provider: statsd
  Statsd:
    Network: udp
    Address: 127.0.0.1:8125
    WriteInterval: 30s
    Prefix: org-orderer
```

For a look at the different metrics that are generated, check out [Metrics Reference](#).

9.10.4 Version

The orderer and peer both expose a `/version` endpoint. This endpoint serves a JSON document containing the orderer or peer version and the commit SHA on which the release was created.

9.11 Metrics Reference

9.11.1 Orderer Metrics

Prometheus

The following orderer metrics are exported for consumption by Prometheus.

Name	Type	Description
blockcutter_block_fill_duration	histogram	The time from first transaction enqueueing to the block being cut in seconds

Table 1 – continued from previous

Name	Type	Description
broadcast_enqueue_duration	histogram	The time to enqueue a transaction in seconds.
broadcast_processed_count	counter	The number of transactions processed.
broadcast_validate_duration	histogram	The time to validate a transaction in seconds.
cluster_comm_egress_queue_capacity	gauge	Capacity of the egress queue.
cluster_comm_egress_queue_length	gauge	Length of the egress queue.
cluster_comm_egress_queue_workers	gauge	Count of egress queue workers.
cluster_comm_egress_stream_count	gauge	Count of streams to other nodes.
cluster_comm_egress_tls_connection_count	gauge	Count of TLS connections to other nodes.
cluster_comm_ingress_stream_count	gauge	Count of streams from other nodes.
cluster_comm_msg_dropped_count	counter	Count of messages dropped.
cluster_comm_msg_send_time	histogram	The time it takes to send a message in seconds.
consensus_etcdraft_active_nodes	gauge	Number of active nodes in this channel.
consensus_etcdraft_cluster_size	gauge	Number of nodes in this channel.
consensus_etcdraft_committed_block_number	gauge	The block number of the latest block committed.
consensus_etcdraft_config_proposals_received	counter	The total number of proposals received for config type transactions.
consensus_etcdraft_data_persist_duration	histogram	The time taken for etcd/raft data to be persisted in storage (in seconds).
consensus_etcdraft_is_leader	gauge	The leadership status of the current node: 1 if it is the leader else 0.
consensus_etcdraft_leader_changes	counter	The number of leader changes since process start.
consensus_etcdraft_normal_proposals_received	counter	The total number of proposals received for normal type transactions.
consensus_etcdraft_proposal_failures	counter	The number of proposal failures.
consensus_etcdraft_snapshot_block_number	gauge	The block number of the latest snapshot.
consensus_kafka_batch_size	gauge	The mean batch size in bytes sent to topics.
consensus_kafka_compression_ratio	gauge	The mean compression ratio (as percentage) for topics.
consensus_kafka_incoming_byte_rate	gauge	Bytes/second read off brokers.
consensus_kafka_last_offset_persisted	gauge	The offset specified in the block metadata of the most recently committed.
consensus_kafka_outgoing_byte_rate	gauge	Bytes/second written to brokers.
consensus_kafka_record_send_rate	gauge	The number of records per second sent to topics.
consensus_kafka_records_per_request	gauge	The mean number of records sent per request to topics.
consensus_kafka_request_latency	gauge	The mean request latency in ms to brokers.
consensus_kafka_request_rate	gauge	Requests/second sent to brokers.
consensus_kafka_request_size	gauge	The mean request size in bytes to brokers.
consensus_kafka_response_rate	gauge	Requests/second sent to brokers.
consensus_kafka_response_size	gauge	The mean response size in bytes from brokers.
deliver_blocks_sent	counter	The number of blocks sent by the deliver service.
deliver_requests_completed	counter	The number of deliver requests that have been completed.

Table 1 – continued from previous

Name	Type	Description
deliver_requests_received	counter	The number of deliver requests that have been received.
deliver_streams_closed	counter	The number of GRPC streams that have been closed for the deliver service.
deliver_streams_opened	counter	The number of GRPC streams that have been opened for the deliver service.
fabric_version	gauge	The active version of Fabric.
grpc_comm_conn_closed	counter	gRPC connections closed. Open minus closed is the active number of connections.
grpc_comm_conn_opened	counter	gRPC connections opened. Open minus closed is the active number of connections.
grpc_server_stream_messages_received	counter	The number of stream messages received.
grpc_server_stream_messages_sent	counter	The number of stream messages sent.
grpc_server_stream_request_duration	histogram	The time to complete a stream request.
grpc_server_stream_requests_completed	counter	The number of stream requests completed.
grpc_server_stream_requests_received	counter	The number of stream requests received.
grpc_server_unary_request_duration	histogram	The time to complete a unary request.
grpc_server_unary_requests_completed	counter	The number of unary requests completed.
grpc_server_unary_requests_received	counter	The number of unary requests received.
ledger_blockchain_height	gauge	Height of the chain in blocks.
ledger_blockstorage_commit_time	histogram	Time taken in seconds for committing the block to storage.
logging_entries_checked	counter	Number of log entries checked against the active logging level.
logging_entries_written	counter	Number of log entries that are written.
participation_consensus_relation	gauge	The channel participation consensus relation of the node: 0 if other, 1 if self.
participation_status	gauge	The channel participation status of the node: 0 if inactive, 1 if active, 2 if pending.

StatsD

The following orderer metrics are emitted for consumption by StatsD. The `%{variable_name}` nomenclature represents segments that vary based on context.

For example, `%{channel}` will be replaced with the name of the channel associated with the metric.

Bucket	Type	Description
blockcutter.block_fill_duration.%{channel}	histogram	The time from first transaction enqueue
broadcast.enqueue_duration.%{channel}.%{type}.%{status}	histogram	The time to enqueue a transaction in s
broadcast.processed_count.%{channel}.%{type}.%{status}	counter	The number of transactions processed
broadcast.validate_duration.%{channel}.%{type}.%{status}	histogram	The time to validate a transaction in s
cluster.comm.egress_queue_capacity.%{host}.%{msg_type}.%{channel}	gauge	Capacity of the egress queue.
cluster.comm.egress_queue_length.%{host}.%{msg_type}.%{channel}	gauge	Length of the egress queue.
cluster.comm.egress_queue_workers.%{channel}	gauge	Count of egress queue workers.
cluster.comm.egress_stream_count.%{channel}	gauge	Count of streams to other nodes.
cluster.comm.egress_tls_connection_count	gauge	Count of TLS connections to other no
cluster.comm.ingress_stream_count	gauge	Count of streams from other nodes.
cluster.comm.msg_dropped_count.%{host}.%{channel}	counter	Count of messages dropped.
cluster.comm.msg_send_time.%{host}.%{channel}	histogram	The time it takes to send a message in
consensus.etcdraft.active_nodes.%{channel}	gauge	Number of active nodes in this channe
consensus.etcdraft.cluster_size.%{channel}	gauge	Number of nodes in this channel.
consensus.etcdraft.committed_block_number.%{channel}	gauge	The block number of the latest block c
consensus.etcdraft.config_proposals_received.%{channel}	counter	The total number of proposals receive
consensus.etcdraft.data_persist_duration.%{channel}	histogram	The time taken for etcd/raft data to be
consensus.etcdraft.is_leader.%{channel}	gauge	The leadership status of the current no
consensus.etcdraft.leader_changes.%{channel}	counter	The number of leader changes since p
consensus.etcdraft.normal_proposals_received.%{channel}	counter	The total number of proposals receive
consensus.etcdraft.proposal_failures.%{channel}	counter	The number of proposal failures.
consensus.etcdraft.snapshot_block_number.%{channel}	gauge	The block number of the latest snapsh
consensus.kafka.batch_size.%{topic}	gauge	The mean batch size in bytes sent to t
consensus.kafka.compression_ratio.%{topic}	gauge	The mean compression ratio (as perce
consensus.kafka.incoming_byte_rate.%{broker_id}	gauge	Bytes/second read off brokers.
consensus.kafka.last_offset_persisted.%{channel}	gauge	The offset specified in the block meta
consensus.kafka.outgoing_byte_rate.%{broker_id}	gauge	Bytes/second written to brokers.
consensus.kafka.record_send_rate.%{topic}	gauge	The number of records per second sen
consensus.kafka.records_per_request.%{topic}	gauge	The mean number of records sent per
consensus.kafka.request_latency.%{broker_id}	gauge	The mean request latency in ms to bro
consensus.kafka.request_rate.%{broker_id}	gauge	Requests/second sent to brokers.
consensus.kafka.request_size.%{broker_id}	gauge	The mean request size in bytes to brok
consensus.kafka.response_rate.%{broker_id}	gauge	Requests/second sent to brokers.
consensus.kafka.response_size.%{broker_id}	gauge	The mean response size in bytes from
deliver.blocks_sent.%{channel}.%{filtered}.%{data_type}	counter	The number of blocks sent by the deli
deliver.requests_completed.%{channel}.%{filtered}.%{data_type}.%{success}	counter	The number of deliver requests that ha
deliver.requests_received.%{channel}.%{filtered}.%{data_type}	counter	The number of deliver requests that ha
deliver.streams_closed	counter	The number of GRPC streams that hav
deliver.streams_opened	counter	The number of GRPC streams that hav
fabric_version.%{version}	gauge	The active version of Fabric.
grpc.comm.conn_closed	counter	gRPC connections closed. Open minu
grpc.comm.conn_opened	counter	gRPC connections opened. Open minu
grpc.server.stream_messages_received.%{service}.%{method}	counter	The number of stream messages receiv
grpc.server.stream_messages_sent.%{service}.%{method}	counter	The number of stream messages sent.
grpc.server.stream_request_duration.%{service}.%{method}.%{code}	histogram	The time to complete a stream request
grpc.server.stream_requests_completed.%{service}.%{method}.%{code}	counter	The number of stream requests compl
grpc.server.stream_requests_received.%{service}.%{method}	counter	The number of stream requests receive
grpc.server.unary_request_duration.%{service}.%{method}.%{code}	histogram	The time to complete a unary request.
grpc.server.unary_requests_completed.%{service}.%{method}.%{code}	counter	The number of unary requests comple
grpc.server.unary_requests_received.%{service}.%{method}	counter	The number of unary requests receive

Table 2 – continued from previous page

Bucket	Type	Description
ledger.blockchain_height.{channel}	gauge	Height of the chain in blocks.
ledger.blockstorage_commit_time.{channel}	histogram	Time taken in seconds for committing
logging.entries_checked.{level}	counter	Number of log entries checked against
logging.entries_written.{level}	counter	Number of log entries that are written
participation.consensus_relation.{channel}	gauge	The channel participation consensus r
participation.status.{channel}	gauge	The channel participation status of the

9.11.2 Peer Metrics

Prometheus

The following peer metrics are exported for consumption by Prometheus.

Name	Type	Description
chaincode_execute_timeouts	counter	The number of chaincode executions (Init or Invoke) that have timed
chaincode_launch_duration	histogram	The time to launch a chaincode.
chaincode_launch_failures	counter	The number of chaincode launches that have failed.
chaincode_launch_timeouts	counter	The number of chaincode launches that have timed out.
chaincode_shim_request_duration	histogram	The time to complete chaincode shim requests.
chaincode_shim_requests_completed	counter	The number of chaincode shim requests completed.
chaincode_shim_requests_received	counter	The number of chaincode shim requests received.
couchdb_processing_time	histogram	Time taken in seconds for the function to complete request to CouchDB
deliver_blocks_sent	counter	The number of blocks sent by the deliver service.
deliver_requests_completed	counter	The number of deliver requests that have been completed.
deliver_requests_received	counter	The number of deliver requests that have been received.
deliver_streams_closed	counter	The number of GRPC streams that have been closed for the deliver service
deliver_streams_opened	counter	The number of GRPC streams that have been opened for the deliver service
dockercontroller_chaincode_container_build_duration	histogram	The time to build a chaincode image in seconds.

Table 3 – continued from previous page

Name	Type	Description
endorser_chaincode_instantiation_failures	counter	The number of chaincode instantiations or upgrade that have failed.
endorser_duplicate_transaction_failures	counter	The number of failed proposals due to duplicate transaction ID.
endorser_endorsement_failures	counter	The number of failed endorsements.
endorser_proposal_acl_failures	counter	The number of proposals that failed ACL checks.
endorser_proposal_duration	histogram	The time to complete a proposal.
endorser_proposal_simulation_failures	counter	The number of failed proposal simulations
endorser_proposal_validation_failures	counter	The number of proposals that have failed initial validation.
endorser_proposals_received	counter	The number of proposals received.
endorser_successful_proposals	counter	The number of successful proposals.
fabric_version	gauge	The active version of Fabric.
gossip_comm_messages_received	counter	Number of messages received
gossip_comm_messages_sent	counter	Number of messages sent
gossip_comm_overflow_count	counter	Number of outgoing queue buffer overflows
gossip_leader_election_leader	gauge	Peer is leader (1) or follower (0)
gossip_membership_total_peers_known	gauge	Total known peers
gossip_payload_buffer_size	gauge	Size of the payload buffer
gossip_privdata_commit_block_duration	histogram	Time it takes to commit private data and the corresponding block (in seconds)
gossip_privdata_fetch_duration	histogram	Time it takes to fetch missing private data from peers (in seconds)
gossip_privdata_list_missing_duration	histogram	Time it takes to list the missing private data (in seconds)
gossip_privdata_pull_duration	histogram	Time it takes to pull a missing private data element (in seconds)
gossip_privdata_purge_duration	histogram	Time it takes to purge private data (in seconds)
gossip_privdata_reconciliation_duration	histogram	Time it takes for reconciliation to complete (in seconds)
gossip_privdata_retrieve_duration	histogram	Time it takes to retrieve missing private data elements from the ledger
gossip_privdata_send_duration	histogram	Time it takes to send a missing private data element (in seconds)
gossip_privdata_validation_duration	histogram	Time it takes to validate a block (in seconds)
gossip_state_commit_duration	histogram	Time it takes to commit a block in seconds
gossip_state_height	gauge	Current ledger height
grpc_comm_conn_closed	counter	gRPC connections closed. Open minus closed is the active number of
grpc_comm_conn_opened	counter	gRPC connections opened. Open minus closed is the active number of
grpc_server_stream_messages_received	counter	The number of stream messages received.
grpc_server_stream_messages_sent	counter	The number of stream messages sent.
grpc_server_stream_request_duration	histogram	The time to complete a stream request.

Table 3 – continued from previous page

Name	Type	Description
grpc_server_stream_requests_completed	counter	The number of stream requests completed.
grpc_server_stream_requests_received	counter	The number of stream requests received.
grpc_server_unary_request_duration	histogram	The time to complete a unary request.
grpc_server_unary_requests_completed	counter	The number of unary requests completed.
grpc_server_unary_requests_received	counter	The number of unary requests received.
ledger_block_processing_time	histogram	Time taken in seconds for ledger block processing.
ledger_blockchain_height	gauge	Height of the chain in blocks.
ledger_blockstorage_and_pvtdata_commit_time	histogram	Time taken in seconds for committing the block and private data to storage.
ledger_blockstorage_commit_time	histogram	Time taken in seconds for committing the block to storage.
ledger_statedb_commit_time	histogram	Time taken in seconds for committing block changes to state db.
ledger_transaction_count	counter	Number of transactions processed.
logging_entries_checked	counter	Number of log entries checked against the active logging level
logging_entries_written	counter	Number of log entries that are written

StatsD

The following peer metrics are emitted for consumption by StatsD. The `%{variable_name}` nomenclature represents segments that vary based on context.

For example, `%{channel}` will be replaced with the name of the channel associated with the metric.

Bucket	Type	Description
chaincode.execute_timeouts.%{chaincode}	counter	The number of chaincode execute timeouts.
chaincode.launch_duration.%{chaincode}.%{success}	histogram	The time to launch a chaincode.
chaincode.launch_failures.%{chaincode}	counter	The number of chaincode launch failures.
chaincode.launch_timeouts.%{chaincode}	counter	The number of chaincode launch timeouts.
chaincode.shim_request_duration.%{type}.%{channel}.%{chaincode}.%{success}	histogram	The time to complete a shim request.
chaincode.shim_requests_completed.%{type}.%{channel}.%{chaincode}.%{success}	counter	The number of shim requests completed.
chaincode.shim_requests_received.%{type}.%{channel}.%{chaincode}	counter	The number of shim requests received.
couchdb.processing_time.%{database}.%{function_name}.%{result}	histogram	Time taken in seconds for couchdb processing.
deliver.blocks_sent.%{channel}.%{filtered}.%{data_type}	counter	The number of blocks sent.
deliver.requests_completed.%{channel}.%{filtered}.%{data_type}.%{success}	counter	The number of deliver requests completed.
deliver.requests_received.%{channel}.%{filtered}.%{data_type}	counter	The number of deliver requests received.
deliver.streams_closed	counter	The number of GRPC streams closed.

Table 4 – continued from previous page

Bucket	Type	Description
deliver.streams_opened	counter	The number of GRPC
dockercontroller.chaincode_container_build_duration.{chaincode}.{success}	histogram	The time to build a cha
endorser.chaincode_instantiation_failures.{channel}.{chaincode}	counter	The number of chained
endorser.duplicate_transaction_failures.{channel}.{chaincode}	counter	The number of failed p
endorser.endorsement_failures.{channel}.{chaincode}.{chaincodeerror}	counter	The number of failed e
endorser.proposal_acl_failures.{channel}.{chaincode}	counter	The number of propos
endorser.proposal_duration.{channel}.{chaincode}.{success}	histogram	The time to complete a
endorser.proposal_simulation_failures.{channel}.{chaincode}	counter	The number of failed p
endorser.proposal_validation_failures	counter	The number of propos
endorser.proposals_received	counter	The number of propos
endorser.successful_proposals	counter	The number of success
fabric_version.{version}	gauge	The active version of F
gossip.comm.messages_received	counter	Number of messages r
gossip.comm.messages_sent	counter	Number of messages s
gossip.comm.overflow_count	counter	Number of outgoing q
gossip.leader_election.leader.{channel}	gauge	Peer is leader (1) or fo
gossip.membership.total_peers_known.{channel}	gauge	Total known peers
gossip.payload_buffer.size.{channel}	gauge	Size of the payload bu
gossip.privdata.commit_block_duration.{channel}	histogram	Time it takes to comm
gossip.privdata.fetch_duration.{channel}	histogram	Time it takes to fetch r
gossip.privdata.list_missing_duration.{channel}	histogram	Time it takes to list the
gossip.privdata.pull_duration.{channel}	histogram	Time it takes to pull a
gossip.privdata.purge_duration.{channel}	histogram	Time it takes to purge
gossip.privdata.reconciliation_duration.{channel}	histogram	Time it takes for recom
gossip.privdata.retrieve_duration.{channel}	histogram	Time it takes to retriev
gossip.privdata.send_duration.{channel}	histogram	Time it takes to send a
gossip.privdata.validation_duration.{channel}	histogram	Time it takes to validat
gossip.state.commit_duration.{channel}	histogram	Time it takes to comm
gossip.state.height.{channel}	gauge	Current ledger height
grpc.comm.conn_closed	counter	gRPC connections clos
grpc.comm.conn_opened	counter	gRPC connections ope
grpc.server.stream_messages_received.{service}.{method}	counter	The number of stream
grpc.server.stream_messages_sent.{service}.{method}	counter	The number of stream
grpc.server.stream_request_duration.{service}.{method}.{code}	histogram	The time to complete a
grpc.server.stream_requests_completed.{service}.{method}.{code}	counter	The number of stream
grpc.server.stream_requests_received.{service}.{method}	counter	The number of stream
grpc.server.unary_request_duration.{service}.{method}.{code}	histogram	The time to complete a
grpc.server.unary_requests_completed.{service}.{method}.{code}	counter	The number of unary r
grpc.server.unary_requests_received.{service}.{method}	counter	The number of unary r
ledger.block_processing_time.{channel}	histogram	Time taken in seconds
ledger.blockchain_height.{channel}	gauge	Height of the chain in
ledger.blockstorage_and_pvtdata_commit_time.{channel}	histogram	Time taken in seconds
ledger.blockstorage_commit_time.{channel}	histogram	Time taken in seconds
ledger.statedb_commit_time.{channel}	histogram	Time taken in seconds
ledger.transaction_count.{channel}.{transaction_type}.{chaincode}.{validation_code}	counter	Number of transaction
logging.entries_checked.{level}	counter	Number of log entries
logging.entries_written.{level}	counter	Number of log entries

9.12 External Builders and Launchers

Prior to Hyperledger Fabric 2.0, the process used to build and launch chaincode was part of the peer implementation and could not be easily customized. All chaincode installed on the peer would be “built” using language specific logic hard coded in the peer. This build process would generate a Docker container image that would be launched to execute chaincode that connected as a client to the peer.

This approach limited chaincode implementations to a handful of languages, required Docker to be part of the deployment environment, and prevented running chaincode as a long running server process.

Starting with Fabric 2.0, External Builders and Launchers address these limitations by enabling operators to extend the peer with programs that can build, launch, and discover chaincode. To leverage this capability you will need to create your own buildpack and then modify the peer core.yaml to include a new `externalBuilder` configuration element which lets the peer know an external builder is available. The following sections describe the details of this process.

Note that if no configured external builder claims a chaincode package, the peer will attempt to process the package as if it were created with the standard Fabric packaging tools such as the peer CLI or node SDK.

Note: This is an advanced feature which unless your builders and launchers are simple enough, such as those used in the [basic asset transfer external chaincode Fabric sample](#), will likely require custom packaging of the peer image with everything your builders and launchers depend on. For example, the following samples use `go` and `bash`, which are not included in the current official `fabric-peer` image.

9.12.1 External builder model

Hyperledger Fabric External Builders and Launchers are loosely based on Heroku [Buildpacks](#). A buildpack implementation is simply a collection of programs or scripts that transform application artifacts into something that can run. The buildpack model has been adapted for chaincode packages and extended to support chaincode execution and discovery.

External builder and launcher API

An external builder and launcher consists of four programs or scripts:

- `bin/detect`: Determine whether or not this buildpack should be used to build the chaincode package and launch it.
- `bin/build`: Transform the chaincode package into executable chaincode.
- `bin/release` (optional): Provide metadata to the peer about the chaincode.
- `bin/run` (optional): Run the chaincode.

`bin/detect`

The `bin/detect` script is responsible for determining whether or not a buildpack should be used to build a chaincode package and launch it. The peer invokes `detect` with two arguments:

```
bin/detect CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR
```

When `detect` is invoked, `CHAINCODE_SOURCE_DIR` contains the chaincode source and `CHAINCODE_METADATA_DIR` contains the `metadata.json` file from the chaincode package installed to the peer. The `CHAINCODE_SOURCE_DIR` and `CHAINCODE_METADATA_DIR` should be treated as read only

inputs. If the buildpack should be applied to the chaincode source package, `detect` must return an exit code of 0; any other exit code will indicate that the buildpack should not be applied.

The following is an example of a simple `detect` script for go chaincode:

```
#!/bin/bash

CHAINCODE_METADATA_DIR="$2"

# use jq to extract the chaincode type from metadata.json and exit with
# success if the chaincode type is golang
if [ "$(jq -r .type "$CHAINCODE_METADATA_DIR/metadata.json" | tr '[:upper:]'
→ '[:lower:]')" = "golang" ]; then
    exit 0
fi

exit 1
```

bin/build

The `bin/build` script is responsible for building, compiling, or transforming the contents of a chaincode package into artifacts that can be used by `release` and `run`. The peer invokes `build` with three arguments:

```
bin/build CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR BUILD_OUTPUT_DIR
```

When `build` is invoked, `CHAINCODE_SOURCE_DIR` contains the chaincode source and `CHAINCODE_METADATA_DIR` contains the `metadata.json` file from the chaincode package installed to the peer. `BUILD_OUTPUT_DIR` is the directory where `build` must place artifacts needed by `release` and `run`. The build script should treat the input directories `CHAINCODE_SOURCE_DIR` and `CHAINCODE_METADATA_DIR` as read only, but the `BUILD_OUTPUT_DIR` is writeable.

When `build` completes with an exit code of 0, the contents of `BUILD_OUTPUT_DIR` will be copied to the persistent storage maintained by the peer; any other exit code will be considered a failure.

The following is an example of a simple `build` script for go chaincode:

```
#!/bin/bash

CHAINCODE_SOURCE_DIR="$1"
CHAINCODE_METADATA_DIR="$2"
BUILD_OUTPUT_DIR="$3"

# extract package path from metadata.json
GO_PACKAGE_PATH="$(jq -r .path "$CHAINCODE_METADATA_DIR/metadata.json")"
if [ -f "$CHAINCODE_SOURCE_DIR/src/go.mod" ]; then
    cd "$CHAINCODE_SOURCE_DIR/src"
    go build -v -mod=readonly -o "$BUILD_OUTPUT_DIR/chaincode" "$GO_PACKAGE_PATH"
else
    GO11MODULE=off go build -v -o "$BUILD_OUTPUT_DIR/chaincode" "$GO_PACKAGE_PATH"
fi

# save statedb index metadata to provide at release
if [ -d "$CHAINCODE_SOURCE_DIR/META-INF" ]; then
    cp -a "$CHAINCODE_SOURCE_DIR/META-INF" "$BUILD_OUTPUT_DIR/"
fi
```

bin/release

The `bin/release` script is responsible for providing chaincode metadata to the peer. `bin/release` is optional. If it is not provided, this step is skipped. The peer invokes `release` with two arguments:

```
bin/release BUILD_OUTPUT_DIR RELEASE_OUTPUT_DIR
```

When `release` is invoked, `BUILD_OUTPUT_DIR` contains the artifacts populated by the build program and should be treated as read only input. `RELEASE_OUTPUT_DIR` is the directory where `release` must place artifacts to be consumed by the peer.

When `release` completes, the peer will consume two types of metadata from `RELEASE_OUTPUT_DIR`:

- state database index definitions for CouchDB
- external chaincode server connection information (`chaincode/server/connection.json`)

If CouchDB index definitions are required for the chaincode, `release` is responsible for placing the indexes into the `statedb/couchdb/indexes` directory under `RELEASE_OUTPUT_DIR`. The indexes must have a `.json` extension. See the [CouchDB indexes](#) documentation for details.

In cases where a chaincode server implementation is used, `release` is responsible for populating `chaincode/server/connection.json` with the address of the chaincode server and any TLS assets required to communicate with the chaincode. When server connection information is provided to the peer, `run` will not be called. See the [Chaincode Server](#) documentation for details.

The following is an example of a simple `release` script for go chaincode:

```
#!/bin/bash

BUILD_OUTPUT_DIR="$1"
RELEASE_OUTPUT_DIR="$2"

# copy indexes from META-INF/* to the output directory
if [ -d "$BUILD_OUTPUT_DIR/META-INF" ] ; then
    cp -a "$BUILD_OUTPUT_DIR/META-INF/"* "$RELEASE_OUTPUT_DIR/"
fi
```

bin/run

The `bin/run` script is responsible for running chaincode. The peer invokes `run` with two arguments:

```
bin/run BUILD_OUTPUT_DIR RUN_METADATA_DIR
```

When `run` is called, `BUILD_OUTPUT_DIR` contains the artifacts populated by the build program and `RUN_METADATA_DIR` is populated with a file called `chaincode.json` that contains the information necessary for chaincode to connect and register with the peer. Note that the `bin/run` script should treat these `BUILD_OUTPUT_DIR` and `RUN_METADATA_DIR` directories as read only input. The keys included in `chaincode.json` are:

- `chaincode_id`: The unique ID associated with the chaincode package.
- `peer_address`: The address in `host:port` format of the ChaincodeSupport gRPC server endpoint hosted by the peer.
- `client_cert`: The PEM encoded TLS client certificate generated by the peer that must be used when the chaincode establishes its connection to the peer.

- `client_key`: The PEM encoded client key generated by the peer that must be used when the chaincode establishes its connection to the peer.
- `root_cert`: The PEM encoded TLS root certificate for the ChaincodeSupport gRPC server endpoint hosted by the peer.
- `mspid`: The local mspid of the peer.

When `run` terminates, the peer considers the chaincode terminated. If another request arrives for the chaincode, the peer will attempt to start another instance of the chaincode by invoking `run` again. The contents of `chaincode.json` must not be cached across invocations.

The following is an example of a simple `run` script for go chaincode:

```
#!/bin/bash

BUILD_OUTPUT_DIR="$1"
RUN_METADATA_DIR="$2"

# setup the environment expected by the go chaincode shim
export CORE_CHAINCODE_ID_NAME="$(jq -r .chaincode_id "$RUN_METADATA_DIR/chaincode.json"
↪)"
export CORE_PEER_TLS_ENABLED="true"
export CORE_TLS_CLIENT_CERT_FILE="$RUN_METADATA_DIR/client.crt"
export CORE_TLS_CLIENT_KEY_FILE="$RUN_METADATA_DIR/client.key"
export CORE_PEER_TLS_ROOTCERT_FILE="$RUN_METADATA_DIR/root.crt"
export CORE_PEER_LOCALMSPID="$(jq -r .mspid "$RUN_METADATA_DIR/chaincode.json")"

# populate the key and certificate material used by the go chaincode shim
jq -r .client_cert "$RUN_METADATA_DIR/chaincode.json" > "$CORE_TLS_CLIENT_CERT_FILE"
jq -r .client_key "$RUN_METADATA_DIR/chaincode.json" > "$CORE_TLS_CLIENT_KEY_FILE"
jq -r .root_cert "$RUN_METADATA_DIR/chaincode.json" > "$CORE_PEER_TLS_ROOTCERT_FILE"
if [ -z "$(jq -r .client_cert "$RUN_METADATA_DIR/chaincode.json")" ]; then
    export CORE_PEER_TLS_ENABLED="false"
fi

# exec the chaincode to replace the script with the chaincode process
exec "$BUILD_OUTPUT_DIR/chaincode" -peer.address="$(jq -r .peer_address "$ARTIFACTS/
↪chaincode.json")"
```

9.12.2 Configuring external builders and launchers

Configuring the peer to use external builders involves adding an `externalBuilder` element under the chaincode configuration block in the `core.yaml` that defines external builders. Each external builder definition must include a name (used for logging) and the path to parent of the `bin` directory containing the builder scripts.

An optional list of environment variable names to propagate from the peer when invoking the external builder scripts can also be provided.

The following example defines two external builders:

```
chaincode:
  externalBuilders:
    - name: my-golang-builder
      path: /builders/golang
      propagateEnvironment:
        - GOPROXY
        - GONOPROXY
```

(continues on next page)

(continued from previous page)

```
- GOSUMDB
- GONOSUMDB
- name: noop-builder
  path: /builders/binary
```

In this example, the implementation of “my-golang-builder” is contained within the `/builders/golang` directory and its build scripts are located in `/builders/golang/bin`. When the peer invokes any of the build scripts associated with “my-golang-builder”, it will propagate only the values of the environment variables in the `propagateEnvironment`.

Note: The following environment variables are always propagated to external builders:

- `LD_LIBRARY_PATH`
- `LIBPATH`
- `PATH`
- `TMPDIR`

When an `externalBuilder` configuration is present, the peer will iterate over the list of builders in the order provided, invoking `bin/detect` until one completes successfully. If no builder completes `detect` successfully, the peer will fallback to using the legacy Docker build process implemented within the peer. This means that external builders are completely optional.

In the example above, the peer will attempt to use “my-golang-builder”, followed by “noop-builder”, and finally the peer internal build process.

9.12.3 Chaincode packages

As part of the new lifecycle introduced with Fabric 2.0, the chaincode package format changed from serialized protocol buffer messages to a gzip compressed POSIX tape archive. Chaincode packages created with `peer lifecycle chaincode package` use this new format.

Lifecycle chaincode package contents

A lifecycle chaincode package contains two files. The first file, `code.tar.gz` is a gzip compressed POSIX tape archive. This file includes the source artifacts for the chaincode. Packages created by the peer CLI will place the chaincode implementation source under the `src` directory and chaincode metadata (like CouchDB indexes) under the `META-INF` directory.

The second file, `metadata.json` is a JSON document with three keys:

- `type`: the chaincode type (e.g. GOLANG, JAVA, NODE)
- `path`: for go chaincode, the `GOPATH` or `GOMOD` relative path to the main chaincode package; undefined for other types
- `label`: the chaincode label that is used to generate the package-id by which the package is identified within the new chaincode lifecycle process.

Note that the `type` and `path` fields are only utilized by docker platform builds.

Chaincode packages and external builders

When a chaincode package is installed to a peer, the contents of `code.tar.gz` and `metadata.json` are not processed prior to calling external builders, except for the `label` field that is used by the new lifecycle process to

compute the package id. This affords users a great deal of flexibility in how they package source and metadata that will be processed by external builders and launchers.

For example, a custom chaincode package could be constructed that contains a pre-compiled, implementation of chaincode in `code.tar.gz` with a `metadata.json` that allows a *binary buildpack* to detect the custom package, validate the hash of the binary, and run the program as chaincode.

Another example would be a chaincode package that only contains state database index definitions and the data necessary for an external launcher to connect to a running chaincode server. In this case, the `build` process would simply extract the metadata from the process and `release` would present it to the peer.

The only requirements are that `code.tar.gz` can only contain regular file and directory entries, and that the entries cannot contain paths that would result in files being written outside of the logical root of the chaincode package.

9.13 Chaincode as an external service

Fabric v2.0 supports chaincode deployment and execution outside of Fabric that enables users to manage a chaincode runtime independently of the peer. This facilitates deployment of chaincode on Fabric cloud deployments such as Kubernetes. Instead of building and launching the chaincode on every peer, chaincode can now run as a service whose lifecycle is managed outside of Fabric. This capability leverages the Fabric v2.0 external builder and launcher functionality which enables operators to extend a peer with programs to build, launch, and discover chaincode. Before reading this topic you should become familiar with the [External Builder and Launcher](#) content.

Prior to the availability of the external builders, the chaincode package content was required to be a set of source code files for a particular language which could be built and launched as a chaincode binary. The new external build and launcher functionality now allows users to optionally customize the build process. With respect to running the chaincode as an external service, the build process allows you to specify the endpoint information of the server where the chaincode is running. Hence the package simply consists of the externally running chaincode server endpoint information and TLS artifacts for secure connection. TLS is optional but highly recommended for all environments except a simple test environment.

The rest of this topic describes how to configure chaincode as an external service:

- *Packaging chaincode*
- *Configuring a peer to process external chaincode*
- *External builder and launcher sample scripts*
- *Writing chaincode to run as an external service*
- *Deploying the chaincode*
- *Running the chaincode as an external service*

Note: This is an advanced feature that will likely require custom packaging of the peer image. For example, the following samples use `jq` and `bash`, which are not included in the current official `fabric-peer` image.

9.13.1 Packaging chaincode

With the Fabric v2.0 chaincode lifecycle, chaincode is [packaged](#) and installed in a `.tar.gz` format. The following `myccpackage.tgz` archive demonstrates the required structure:

```
$ tar xvfz myccpackage.tgz
metadata.json
code.tar.gz
```

The chaincode package should be used to provide two pieces of information to the external builder and launcher process

- identify if the chaincode is an external service. The `bin/detect` section describes an approach using the `metadata.json` file
- provide chaincode endpoint information in a `connection.json` file placed in the release directory. The `bin/run` section describes the `connection.json` file

There is plenty of flexibility to gathering the above information. The sample scripts in the *External builder and launcher sample scripts* illustrate a simple approach to providing the information. As an example of flexibility, consider packaging couchdb index files (see [Add the index to your chaincode folder](#)). Sample scripts below describe an approach to packaging the files into `myccpackage.tar.gz`.

```
tar cfz code.tar.gz connection.json metadata
tar cfz myccpackage.tgz metadata.json code.tar.gz
```

9.13.2 Configuring a peer to process external chaincode

In this section we go over the configuration needed

- to detect if the chaincode package identifies an external chaincode service
- to create the `connection.json` file in the release directory

Modify the peer `core.yaml` to include the `externalBuilder`

Assume the scripts are on the peer in the `bin` directory as follows

```
<fully qualified path on the peer's env>
└─ bin
    ├── build
    ├── detect
    └── release
```

Modify the chaincode stanza of the peer `core.yaml` file to include the `externalBuilders` configuration element:

```
externalBuilders:
  - name: myexternal
    path: <fully qualified path on the peer's env>
```

External builder and launcher sample scripts

To help understand what each script needs to contain to work with the chaincode as an external service, this section contains samples of `bin/detect`, `bin/build`, `bin/release`, and `bin/run` scripts.

Note: These samples use the `jq` command to parse json. You can run `jq --version` to check if you have it installed. Otherwise, install `jq` or suitably modify the scripts.

`bin/detect`

The `bin/detect` script is responsible for determining whether or not a buildpack should be used to build a chaincode package and launch it. For chaincode as an external service, the sample script looks for a `type` property set to `external` in the `metadata.json` file:

```
{ "path": "", "type": "external", "label": "mycc" }
```

The peer invokes detect with two arguments:

```
bin/detect CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR
```

A sample bin/detect script could contain:

```
#!/bin/bash

set -euo pipefail

METADIR=$2
#check if the "type" field is set to "external"
if [ "$(jq -r .type "$METADIR/metadata.json")" == "external" ]; then
    exit 0
fi

exit 1
```

bin/build

For chaincode as an external service, the sample build script assumes the chaincode package's `code.tar.gz` file contains `connection.json` which it simply copies to the `BUILD_OUTPUT_DIR`. The peer invokes the build script with three arguments:

```
bin/build CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR BUILD_OUTPUT_DIR
```

A sample bin/build script could contain:

```
#!/bin/bash

set -euo pipefail

SOURCE=$1
OUTPUT=$3

#external chaincodes expect connection.json file in the chaincode package
if [ ! -f "$SOURCE/connection.json" ]; then
    >&2 echo "$SOURCE/connection.json not found"
    exit 1
fi

#simply copy the endpoint information to specified output location
cp $SOURCE/connection.json $OUTPUT/connection.json

if [ -d "$SOURCE/metadata" ]; then
    cp -a $SOURCE/metadata $OUTPUT/metadata
fi

exit 0
```

bin/release

For chaincode as an external service, the `bin/release` script is responsible for providing the `connection.json` to the peer by placing it in the `RELEASE_OUTPUT_DIR`. The `connection.json` file has the following JSON structure

- **address** - chaincode server endpoint accessible from peer. Must be specified in “:” format.
- **dial_timeout** - interval to wait for connection to complete. Specified as a string qualified with time units (e.g, “10s”, “500ms”, “1m”). Default is “3s” if not specified.
- **tls_required** - true or false. If false, “client_auth_required”, “client_key”, “client_cert”, and “root_cert” are not required. Default is “true”.
- **client_auth_required** - if true, “client_key” and “client_cert” are required. Default is false. It is ignored if `tls_required` is false.
- **client_key** - PEM encoded string of the client private key.
- **client_cert** - PEM encoded string of the client certificate.
- **root_cert** - PEM encoded string of the server (peer) root certificate.

For example:

```
{
  "address": "your.chaincode.host.com:9999",
  "dial_timeout": "10s",
  "tls_required": "true",
  "client_auth_required": "true",
  "client_key": "-----BEGIN EC PRIVATE KEY----- ... -----END EC PRIVATE KEY-----",
  "client_cert": "-----BEGIN CERTIFICATE----- ... -----END CERTIFICATE-----",
  "root_cert": "-----BEGIN CERTIFICATE----- ... -----END CERTIFICATE-----"
}
```

As noted in the `bin/build` section, this sample assumes the chaincode package directly contains the `connection.json` file which the build script copies to the `BUILD_OUTPUT_DIR`. The peer invokes the release script with two arguments:

```
bin/release BUILD_OUTPUT_DIR RELEASE_OUTPUT_DIR
```

A sample `bin/release` script could contain:

```
#!/bin/bash

set -euo pipefail

BLD="$1"
RELEASE="$2"

if [ -d "$BLD/metadata" ]; then
  cp -a "$BLD/metadata/"* "$RELEASE/"
fi

#external chaincodes expect artifacts to be placed under "$RELEASE"/chaincode/server
if [ -f $BLD/connection.json ]; then
  mkdir -p "$RELEASE"/chaincode/server
  cp $BLD/connection.json "$RELEASE"/chaincode/server

  #if tls_required is true, copy TLS files (using above example, the fully qualified
  ↪path for these files would be "$RELEASE"/chaincode/server/tls)
  (continues on next page)
```

(continued from previous page)

```

    exit 0
fi
exit 1

```

9.13.3 Writing chaincode to run as an external service

Currently, the chaincode as an external service model is supported by Go chaincode shim and Node.js chaincode shim.

Go

In Fabric v2.0, the Go shim API provides a `ChaincodeServer` type that developers should use to create a chaincode server. The `Invoke` and `Query` APIs are unaffected. Developers should write to the `shim.ChaincodeServer` API, then build the chaincode and run it in the external environment of choice. Here is a simple sample chaincode program to illustrate the pattern:

```

package main

import (
    "fmt"

    "github.com/hyperledger/fabric-chaincode-go/shim"
    pb "github.com/hyperledger/fabric-protos-go/peer"
)

// SimpleChaincode example simple Chaincode implementation
type SimpleChaincode struct {
}

func (s *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    // init code
}

func (s *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    // invoke code
}

//NOTE - parameters such as ccid and endpoint information are hard coded here for
//illustration. This can be passed in in a variety of standard ways
func main() {
    //The ccid is assigned to the chaincode on install (using the "peer lifecycle
    //chaincode install <package>" command) for instance
    ccid := "mycc:fcbf8724572d42e859a7dd9a7cd8e2efb84058292017df6e3d89178b64e6c831"

    server := &shim.ChaincodeServer{
        CCID: ccid,
        Address: "myhost:9999",
        CC: new(SimpleChaincode),
        TLSProps: shim.TLSProperties{
            Disabled: true,
        },
    }
}

```

(continues on next page)

(continued from previous page)

```

err := server.Start()
if err != nil {
    fmt.Printf("Error starting Simple chaincode: %s", err)
}
}

```

The key to running the chaincode as an external service is the use of `shim.ChaincodeServer`. This uses the new shim API `shim.ChaincodeServer` with the chaincode service properties described below:

- **CCID** (string)- CCID should match chaincode's package name on peer. This is the CCID associated with the installed chaincode as returned by the peer lifecycle chaincode install <package> CLI command. This can be obtained post-install using the "peer lifecycle chaincode queryinstalled" command.
- **Address** (string) - Address is the listen address of the chaincode server
- **CC** (Chaincode) - CC is the chaincode that handles Init and Invoke
- **TLSProps** (TLSProperties) - TLSProps is the TLS properties passed to chaincode server
- **KaOpts** (keepalive.ServerParameters) - KaOpts keepalive options, sensible defaults provided if nil

Then build the chaincode as suitable to your Go environment.

Node.js

`fabric-shim` package for Node.js chaincode provides the `shim.server` API to run chaincode as an external service. If you are using contract APIs, you may want to use the `server` command provided by `fabric-chaincode-node` CLI to run a contract in the external service mode.

The following is a sample chaincode using `fabric-shim`:

```

const shim = require('fabric-shim');

class SimpleChaincode extends shim.ChaincodeInterface {
    async Init(stub) {
        // ... Init code
    }
    async Invoke(stub) {
        // ... Invoke code
    }
}

const server = shim.server(new SimpleChaincode(), {
    ccid: "mycc:fcbf8724572d42e859a7dd9a7cd8e2efb84058292017df6e3d89178b64e6c831",
    address: "0.0.0.0:9999"
});

server.start();

```

To run a chaincode with the `fabric-contract` API as an external service, simply use `fabric-chaincode-node server` instead of `fabric-chaincode-node start`. Here is a sample for `package.json`:

```

{
  "scripts": {
    "start": "fabric-chaincode-node server"
  },

```

(continues on next page)

(continued from previous page)

```

    ...
}

```

When `fabric-chaincode-node server` is used, the following options should be set as either arguments or environment variables:

- **CORE_CHAINCODE_ID** (`--chaincode-id`): See **CCID** in the Go chaincode above.
- **CORE_CHAINCODE_ADDRESS** (`--chaincode-address`): See **Address** in the Go chaincode above.

If TLS is enabled, the following additional options are required:

- **CORE_CHAINCODE_TLS_CERT_FILE** (`--chaincode-tls-cert-file`): path to a certificate
- **CORE_CHAINCODE_TLS_KEY_FILE** (`--chaincode-tls-key-file`): path to a private key

When mutual TLS is enabled, **CORE_CHAINCODE_TLS_CLIENT_CACERT_FILE** (`--chaincode-tls-client-cacert-file`) option should be set to specify the path to the CA certificate for acceptable client certificates.

9.13.4 Deploying the chaincode

When the chaincode is ready for deployment, you can package the chaincode as explained in the [Packaging chaincode](#) section and deploy the chaincode as explained in the [Fabric chaincode lifecycle](#) concept topic.

9.13.5 Running the chaincode as an external service

Create the chaincode as specified in the [Writing chaincode to run as an external service](#) section. Run the built executable in your environment of choice, such as Kubernetes or directly as a process on the peer machine.

Using this chaincode as an external service model, installing the chaincode on each peer is no longer required. With the chaincode endpoint deployed to the peer instead and the chaincode running, you can continue the normal process of committing the chaincode definition to the channel and invoking the chaincode.

9.14 Error handling

9.14.1 General Overview

Hyperledger Fabric code should use the vendored package [github.com/pkg/errors](#) in place of the standard error type provided by Go. This package allows easy generation and display of stack traces with error messages.

9.14.2 Usage Instructions

[github.com/pkg/errors](#) should be used in place of all calls to `fmt.Errorf()` or `errors.New()`. Using this package will generate a call stack that will be appended to the error message.

Using this package is simple and will only require easy tweaks to your code.

First, you'll need to import [github.com/pkg/errors](#).

Next, update all errors that are generated by your code to use one of the error creation functions (`errors.New()`, `errors.Errorf()`, `errors.WithMessage()`, `errors.Wrap()`, `errors.Wrapf()`).

Note: See <https://godoc.org/github.com/pkg/errors> for complete documentation of the available error creation function. Also, refer to the General guidelines section below for more specific guidelines for using the package for Fabric code.

Finally, change the formatting directive for any logger or `fmt.Printf()` calls from `%s` to `%+v` to print the call stack along with the error message.

9.14.3 General guidelines for error handling in Hyperledger Fabric

- If you are servicing a user request, you should log the error and return it.
- If the error comes from an external source, such as a Go library or vendored package, wrap the error using `errors.Wrap()` to generate a call stack for the error.
- If the error comes from another Fabric function, add further context, if desired, to the error message using `errors.WithMessage()` while leaving the call stack unaffected.
- A panic should not be allowed to propagate to other packages.

9.14.4 Example program

The following example program provides a clear demonstration of using the package:

```
package main

import (
    "fmt"

    "github.com/pkg/errors"
)

func wrapWithStack() error {
    err := createError()
    // do this when error comes from external source (go lib or vendor)
    return errors.Wrap(err, "wrapping an error with stack")
}

func wrapWithoutStack() error {
    err := createError()
    // do this when error comes from internal Fabric since it already has stack trace
    return errors.WithMessage(err, "wrapping an error without stack")
}

func createError() error {
    return errors.New("original error")
}

func main() {
    err := createError()
    fmt.Printf("print error without stack: %s\n\n", err)
    fmt.Printf("print error with stack: %+v\n\n", err)
    err = wrapWithoutStack()
    fmt.Printf("%+v\n\n", err)
    err = wrapWithStack()
    fmt.Printf("%+v\n\n", err)
}
```


9.15 Logging Control

9.15.1 Overview

Logging in the `peer` and `orderer` is provided by the `common/flogging` package. This package supports

- Logging control based on the severity of the message
- Logging control based on the software *logger* generating the message
- Different pretty-printing options based on the severity of the message

All logs are currently directed to `stderr`. Global and logger-level control of logging by severity is provided for both users and developers. There are currently no formalized rules for the types of information provided at each severity level. When submitting bug reports, developers may want to see full logs down to the `DEBUG` level.

In pretty-printed logs the logging level is indicated both by color and by a four-character code, e.g. “ERRO” for `ERROR`, “DEBU” for `DEBUG`, etc. In the logging context a *logger* is an arbitrary name (string) given by developers to groups of related messages. In the pretty-printed example below, the loggers `ledgermgmt`, `kvledger`, and `peer` are generating logs.

```
2018-11-01 15:32:38.268 UTC [ledgermgmt] initialize -> INFO 002 Initializing ledger_
↪mgmt
2018-11-01 15:32:38.268 UTC [kvledger] NewProvider -> INFO 003 Initializing ledger_
↪provider
2018-11-01 15:32:38.342 UTC [kvledger] NewProvider -> INFO 004 ledger provider_
↪Initialized
2018-11-01 15:32:38.357 UTC [ledgermgmt] initialize -> INFO 005 ledger mgmt_
↪initialized
2018-11-01 15:32:38.357 UTC [peer] func1 -> INFO 006 Auto-detected peer address: 172.
↪24.0.3:7051
2018-11-01 15:32:38.357 UTC [peer] func1 -> INFO 007 Returning peer0.org1.example.
↪com:7051
```

An arbitrary number of loggers can be created at runtime, therefore there is no “master list” of loggers, and logging control constructs can not check whether logging loggers actually do or will exist.

9.15.2 Logging specification

The logging levels of the `peer` and `orderer` commands are controlled by a logging specification, which is set via the `FABRIC_LOGGING_SPEC` environment variable.

The full logging level specification is of the form

```
[<logger>[,<logger>...]=]<level>[:[<logger>[,<logger>...]=]<level>...]
```

Logging severity levels are specified using case-insensitive strings chosen from

```
FATAL | PANIC | ERROR | WARNING | INFO | DEBUG
```

A logging level by itself is taken as the overall default. Otherwise, overrides for individual or groups of loggers can be specified using the

```
<logger>[,<logger>...]=<level>
```

syntax. Examples of specifications:

info	- Set default to INFO
warning:msp,gossip=warning:chaincode=info	- Default WARNING; Override for msp, and gossip, and chaincode
chaincode=info:msp,gossip=warning:warning	- Same as above

Note: Logging specification terms are separated by a colon. If a term does not include a specific logger, for example *info*: then it is applied as the default log level across all loggers on the component. The string *info:dockercontroller,endorser,chaincode,chaincode.platform=debug* sets the default log level to *INFO* for all loggers and then the *dockercontroller*, *endorser*, *chaincode*, and *chaincode.platform* loggers are set to *DEBUG*. The order of the terms does not matter. In the examples above, the second and third options produce the same result although the order of the terms is reversed.

9.15.3 Logging format

The logging format of the *peer* and *orderer* commands is controlled via the `FABRIC_LOGGING_FORMAT` environment variable. This can be set to a format string, such as the default

```
"{%color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] {%shortfunc} -> {%level:.4s}
↳{%id:03x}%{color:reset} {%message}"
```

to print the logs in a human-readable console format. It can be also set to `json` to output logs in JSON format.

9.15.4 Chaincode

Chaincode logging is the responsibility of the chaincode developer.

As independently executed programs, user-provided chaincodes may technically also produce output on stdout/stderr. While naturally useful for “devmode”, these channels are normally disabled on a production network to mitigate abuse from broken or malicious code. However, it is possible to enable this output even for peer-managed containers (e.g. “netmode”) on a per-peer basis via the `CORE_VM_DOCKER_ATTACHSTDOUT=true` configuration option.

Once enabled, each chaincode will receive its own logging channel keyed by its container-id. Any output written to either stdout or stderr will be integrated with the peer’s log on a per-line basis. It is not recommended to enable this for production.

Stdout and stderr not forwarded to the peer container can be viewed from the chaincode container using standard commands for your container platform.

```
docker logs <chaincode_container_id>
kubectl logs -n <namespace> <pod_name>
oc logs -n <namespace> <pod_name>
```

9.16 Securing Communication With Transport Layer Security (TLS)

Fabric supports for secure communication between nodes using TLS. TLS communication can use both one-way (server only) and two-way (server and client) authentication.

9.16.1 Configuring TLS for peers nodes

A peer node is both a TLS server and a TLS client. It is the former when another peer node, application, or the CLI makes a connection to it and the latter when it makes a connection to another peer node or orderer.

To enable TLS on a peer node set the following peer configuration properties:

- `peer.tls.enabled=true`
- `peer.tls.cert.file` = fully qualified path of the file that contains the TLS server certificate
- `peer.tls.key.file` = fully qualified path of the file that contains the TLS server private key

By default, TLS client authentication is turned off when TLS is enabled on a peer node. This means that the peer node will not verify the certificate of a client (another peer node, application, or the CLI) during a TLS handshake. To enable TLS client authentication on a peer node, set the peer configuration property `peer.tls.clientAuthRequired` to `true` and set the `peer.tls.clientRootCAs.files` property to the CA chain file(s) that contain(s) the CA certificate chain(s) that issued TLS certificates for your organization's clients.

By default, a peer node will use the same certificate and private key pair when acting as a TLS server and client. To use a different certificate and private key pair for the client side, set the `peer.tls.clientCert.file` and `peer.tls.clientKey.file` configuration properties to the fully qualified path of the client certificate and key file, respectively.

TLS with client authentication can also be enabled by setting the following environment variables:

- `CORE_PEER_TLS_ENABLED=true`
- `CORE_PEER_TLS_CERT_FILE` = fully qualified path of the server certificate
- `CORE_PEER_TLS_KEY_FILE` = fully qualified path of the server private key
- `CORE_PEER_TLS_CLIENTAUTHREQUIRED=true`
- `CORE_PEER_TLS_CLIENTROOTCAS_FILES` = fully qualified path of the CA chain file
- `CORE_PEER_TLS_CLIENTCERT_FILE` = fully qualified path of the client certificate
- `CORE_PEER_TLS_CLIENTKEY_FILE` = fully qualified path of the client key

When client authentication is enabled on a peer node, a client is required to send its certificate during a TLS handshake. If the client does not send its certificate, the handshake will fail and the peer will close the connection.

When a peer joins a channel, root CA certificate chains of the channel members are read from the config block of the channel and are added to the TLS server and client root CAs data structure. So, peer to peer communication, peer to orderer communication should work seamlessly. However, if needed, you can add additional root CA certificates by setting `peer.tls.rootcert.file` and `peer.tls.clientRootCAs.files`.

9.16.2 Configuring TLS for orderer nodes

To enable TLS on an orderer node, set the following orderer configuration properties:

- `General.TLS.Enabled=true`
- `General.TLS.PrivateKey` = fully qualified path of the file that contains the server private key
- `General.TLS.Certificate` = fully qualified path of the file that contains the server certificate

By default, TLS client authentication is turned off on orderer, as is the case with peer. To enable TLS client authentication, set the following config property:

- `General.TLS.ClientAuthRequired=true`

TLS with client authentication can also be enabled by setting the following environment variables:

- `ORDERER_GENERAL_TLS_ENABLED = true`
- `ORDERER_GENERAL_TLS_PRIVATEKEY` = fully qualified path of the file that contains the server private key
- `ORDERER_GENERAL_TLS_CERTIFICATE` = fully qualified path of the file that contains the server certificate
- `ORDERER_GENERAL_TLS_CLIENTAUTHREQUIRED = true`

When an orderer participates in a channel, root CA certificate chains of the channel members are read from the config block of the channel and are added to the TLS server and client root CAs data structure. So, orderer to orderer communication should work seamlessly. However, if needed, you can add additional root CA certificates by setting `General.TLS.RootCAs` and `General.TLS.ClientRootCAs`.

9.16.3 Configuring TLS for the peer CLI

The following environment variables must be set when running peer CLI commands against a TLS enabled peer node:

- `CORE_PEER_TLS_ENABLED = true`
- `CORE_PEER_TLS_ROOTCERT_FILE` = fully qualified path of the file that contains cert chain of the CA that issued the TLS server cert

If TLS client authentication is also enabled on the remote server, the following variables must to be set in addition to those above:

- `CORE_PEER_TLS_CLIENTAUTHREQUIRED = true`
- `CORE_PEER_TLS_CLIENTCERT_FILE` = fully qualified path of the client certificate
- `CORE_PEER_TLS_CLIENTKEY_FILE` = fully qualified path of the client private key

When running a command that connects to orderer service, like *peer channel <create|update|fetch>* or *peer chaincode <invoke>*, following command line arguments must also be specified if TLS is enabled on the orderer:

- `-tls`
- `-cafile` <fully qualified path of the file that contains cert chain of the orderer CA>

If TLS client authentication is enabled on the orderer, the following arguments must be specified as well:

- `-clientauth`
- `-keyfile` <fully qualified path of the file that contains the client private key>
- `-certfile` <fully qualified path of the file that contains the client certificate>

9.16.4 Debugging TLS issues

Before debugging TLS issues, it is advisable to enable `GRPC debug` on both the TLS client and the server side to get additional information. To enable `GRPC debug`, set the environment variable `FABRIC_LOGGING_SPEC` to include `grpc=debug`. For example, to set the default logging level to `INFO` and the `GRPC` logging level to `DEBUG`, set the logging specification to `grpc=debug:info`.

If you see the error message `remote error: tls: bad certificate` on the client side, it usually means that the TLS server has enabled client authentication and the server either did not receive the correct client certificate or it received a client certificate that it does not trust. Make sure the client is sending its certificate and that it has been signed by one of the CA certificates trusted by the peer or orderer node.

If you see the error message `remote error: tls: bad certificate` in your chaincode logs, ensure that your chaincode has been built using the chaincode shim provided with Fabric v1.1 or newer.

9.17 Configuring and operating a Raft ordering service

Audience: *Raft ordering node admins*

Note: this topic describes the process for configuring a Raft ordering service that has not been bootstrapped with a system channel genesis block. For a version of this topic that includes information about the system channel, check out [Configuring and operating a Raft ordering service](#).

9.17.1 Conceptual overview

For a high level overview of the concept of ordering and how the supported ordering service implementations (including Raft) work at a high level, check out our conceptual documentation on the [Ordering Service](#).

To learn about the process of setting up an ordering node — including the creation of a local MSP and the creation of a genesis block — check out our documentation on [Setting up an ordering node](#).

9.17.2 Configuration

A Raft cluster is configured in two places:

- **Local configuration:** Governs node specific aspects, such as TLS communication, replication behavior, and file storage.
- **Channel configuration:** Defines the membership of the Raft cluster for the corresponding channel, as well as protocol specific parameters such as heartbeat frequency, leader timeouts, and more.

Raft nodes identify each other using TLS pinning, so in order to impersonate a Raft node, an attacker needs to obtain the **private key** of its TLS certificate. As a result, it is not possible to run a Raft node without a valid TLS configuration.

Recall, each channel has its own instance of a Raft protocol running. Thus, a Raft node must be referenced in the configuration of each channel it belongs to by adding its server and client TLS certificates (in PEM format) to the channel config. This ensures that when other nodes receive a message from it, they can securely confirm the identity of the node that sent the message.

The following section from `configtx.yaml` shows three Raft nodes (also called “consenters”) in the channel:

```
Consenters:
- Host: raft0.example.com
  Port: 7050
  ClientTLSCert: path/to/ClientTLSCert0
  ServerTLSCert: path/to/ServerTLSCert0
- Host: raft1.example.com
  Port: 7050
  ClientTLSCert: path/to/ClientTLSCert1
  ServerTLSCert: path/to/ServerTLSCert1
- Host: raft2.example.com
  Port: 7050
  ClientTLSCert: path/to/ClientTLSCert2
  ServerTLSCert: path/to/ServerTLSCert2
```

When the channel config block is created, the `configtxgen` tool reads the paths to the TLS certificates, and replaces the paths with the corresponding bytes of the certificates.

Note: it is possible to remove and add an ordering node from a channel dynamically without affecting the other nodes, a process described in the Reconfiguration section below.

Local configuration

The `orderer.yaml` has two configuration sections that are relevant for Raft orderers:

Cluster, which determines the TLS communication configuration. And **consensus**, which determines where Write Ahead Logs and Snapshots are stored.

Cluster parameters:

By default, the Raft service is running on the same gRPC server as the client facing server (which is used to send transactions or pull blocks), but it can be configured to have a separate gRPC server with a separate port.

This is useful for cases where you want TLS certificates issued by the organizational CAs, but used only by the cluster nodes to communicate among each other, and TLS certificates issued by a public TLS CA for the client facing API.

- `ClientCertificate, ClientPrivateKey`: The file path of the client TLS certificate and corresponding private key.
- `ListenPort`: The port the cluster listens on. It must be same as `consenters[i].Port` in Channel configuration. If blank, the port is the same port as the orderer general port (`general.listenPort`)
- `ListenAddress`: The address the cluster service is listening on.
- `ServerCertificate, ServerPrivateKey`: The TLS server certificate key pair which is used when the cluster service is running on a separate gRPC server (different port).

Note: `ListenPort, ListenAddress, ServerCertificate, ServerPrivateKey` must be either set together or unset together. If they are unset, they are inherited from the general TLS section, in example `general.tls.{privateKey, certificate}`. When general TLS is disabled:

- Use a different `ListenPort` than the orderer general port
- Properly configure TLS root CAs in the channel configuration.

There are also hidden configuration parameters for `general.cluster` which can be used to further fine tune the cluster communication or replication mechanisms:

- `SendBufferSize`: Regulates the number of messages in the egress buffer.
- `DialTimeout, RPCTimeout`: Specify the timeouts of creating connections and establishing streams.
- `ReplicationBufferSize`: the maximum number of bytes that can be allocated for each in-memory buffer used for block replication from other cluster nodes. Each channel has its own memory buffer. Defaults to 20971520 which is 20MB.
- `PullTimeout`: the maximum duration the ordering node will wait for a block to be received before it aborts. Defaults to five seconds.
- `ReplicationRetryTimeout`: The maximum duration the ordering node will wait between two consecutive attempts. Defaults to five seconds.
- `ReplicationBackgroundRefreshInterval`: the time between two consecutive attempts to replicate existing channels that this node was added to, or channels that this node failed to replicate in the past. Defaults to five minutes.
- `TLSHandshakeTimeShift`: If the TLS certificates of the ordering nodes expire and are not replaced in time (see TLS certificate rotation below), communication between them cannot be established, and it will be impossible to send new transactions to the ordering service. To recover from such a scenario, it is possible to make TLS handshakes between ordering nodes consider the time to be shifted backwards a given amount that is configured to `TLSHandshakeTimeShift`. This setting only applies when a separate cluster listener is in use. If the cluster service is sharing the orderer's main gRPC server, then instead specify `TLSHandshakeTimeShift` in the `General.TLS` section.

Consensus parameters:

- `WALDir`: the location at which Write Ahead Logs for `etcd/raft` are stored. Each channel will have its own subdirectory named after the channel ID.
- `SnapDir`: specifies the location at which snapshots for `etcd/raft` are stored. Each channel will have its own subdirectory named after the channel ID.

There are also two hidden configuration parameters that can each be set by adding them the consensus section in the `orderer.yaml`:

- `EvictionSuspicion`: The cumulative period of time of channel eviction suspicion that triggers the node to pull blocks from other nodes and see if it has been evicted from the channel in order to confirm its suspicion. If the suspicion is confirmed (the inspected block doesn't contain the node's TLS certificate), the node halts its operation for that channel. A node suspects its channel eviction when it doesn't know about any elected leader nor can be elected as leader in the channel. Defaults to 10 minutes.
- `TickIntervalOverride`: If set, this value will be preferred over the tick interval configured in all channels where this ordering node is a consenter. This value should be set only with great care, as a mismatch in tick interval across orderers could result in a loss of quorum for one or more channels.

Channel configuration

Apart from the (already discussed) consenters, the Raft channel configuration has an `Options` section which relates to protocol specific knobs. It is currently not possible to change these values dynamically while a node is running. The node have to be reconfigured and restarted.

The only exceptions is `SnapshotIntervalSize`, which can be adjusted at runtime.

Note: It is recommended to avoid changing the following values, as a misconfiguration might lead to a state where a leader cannot be elected at all (i.e, if the `TickInterval` and `ElectionTick` are extremely low). Situations where a leader cannot be elected are impossible to resolve, as leaders are required to make changes. Because of such dangers, we suggest not tuning these parameters for most use cases.

- `TickInterval`: The time interval between two `Node.Tick` invocations.
- `ElectionTick`: The number of `Node.Tick` invocations that must pass between elections. That is, if a follower does not receive any message from the leader of current term before `ElectionTick` has elapsed, it will become candidate and start an election.
- `ElectionTick` must be greater than `HeartbeatTick`.
- `HeartbeatTick`: The number of `Node.Tick` invocations that must pass between heartbeats. That is, a leader sends heartbeat messages to maintain its leadership every `HeartbeatTick` ticks.
- `MaxInflightBlocks`: Limits the max number of in-flight append blocks during optimistic replication phase.
- `SnapshotIntervalSize`: Defines number of bytes per which a snapshot is taken.

9.17.3 Reconfiguration

The Raft orderer supports dynamic (meaning, while the channel is being serviced) addition and removal of nodes as long as only one node is added or removed at a time. Note that your cluster must be operational and able to achieve consensus before you attempt to reconfigure it. For instance, if you have three nodes, and two nodes fail, you will not be able to reconfigure your cluster to remove those nodes. Similarly, if you have one failed node in a channel with three nodes, you should not attempt to rotate a certificate, as this would induce a second fault. As a rule, you should never attempt any configuration changes to the Raft consenters, such as adding or removing a consenter, or rotating a consenter's certificate unless all consenters are online and healthy.

If you do decide to change these parameters, it is recommended to only attempt such a change during a maintenance cycle. Problems are most likely to occur when a configuration is attempted in clusters with only a few nodes while a

node is down. For example, if you have three nodes in your consenter set and one of them is down, it means you have two out of three nodes alive. If you extend the cluster to four nodes while in this state, you will have only two out of four nodes alive, which is not a quorum. The fourth node won't be able to onboard because nodes can only onboard to functioning clusters (unless the total size of the cluster is one or two).

So by extending a cluster of three nodes to four nodes (while only two are alive) you are effectively stuck until the original offline node is resurrected.

To add a new node to the ordering service:

1. **Ensure the orderer organization that owns the new node is one of the orderer organizations on the channel.** If the orderer organization is not an administrator, the node will be unable to pull blocks as a follower or be joined to the consenter set.
2. **Start the new ordering node.** For information about how to deploy an ordering node, check out [Planning for an ordering service](#). Note that when you use the `osnadmin` CLI to create and join a channel, you do not need to point to a configuration block when starting the node.
3. **Use the `osnadmin` CLI to add the first orderer to the channel.** For more information, check out the [Create a channel](#) tutorial.
4. **Wait for the Raft node to replicate the blocks** from existing nodes for all channels its certificates have been added to. When an ordering node is added to a channel, it is added as a “follower”, a state in which it can replicate blocks but is not part of the “consenter set” actively servicing the channel. When the node finishes replicating the blocks, its status should change from “onboarding” to “active”. Note that an “active” ordering node is still not part of the consenter set.
5. **Add the new ordering node to the consenter set.** For more information, check out the [Create a channel](#) tutorial.

It is possible to add a node that is already running (and participates in some channels already) to a channel while the node itself is running. To do this, simply add the node's certificate to the channel config of the channel. The node will autonomously detect its addition to the new channel (the default value here is five minutes, but if you want the node to detect the new channel more quickly, reboot the node) and will pull the channel blocks from an orderer in the channel, and then start the Raft instance for that chain.

After it has successfully done so, the channel configuration can be updated to include the endpoint of the new Raft orderer.

To remove an ordering node from the consenter set of a channel, use the `osnadmin channel remove` command to remove its endpoint and certificates from the channel. For more information, check out [Add or remove orderers from existing channels](#).

Once an ordering node is removed from the channel, the other ordering nodes stop communicating with the removed orderer in the context of the removed channel. They might still be communicating on other channels.

The node that is removed from the channel automatically detects its removal either immediately or after `EvictionSuspicion` time has passed (10 minutes by default) and shuts down its Raft instance on that channel.

If the intent is to delete the node entirely, remove it from all channels before shutting down the node.

TLS certificate rotation for an orderer node

All TLS certificates have an expiration date that is determined by the issuer. These expiration dates can range from 10 years from the date of issuance to as little as a few months, so check with your issuer. Before the expiration date, you will need to rotate these certificates on the node itself and every channel the node is joined to.

Note: In case the public key of the TLS certificate remains the same, there is no need to issue channel configuration updates.

For each channel the node participates in:

1. Update the channel configuration with the new certificates.
2. Replace its certificates in the file system of the node.
3. Restart the node.

Because a node can only have a single TLS certificate key pair, the node will be unable to service channels its new certificates have not been added to during the update process, degrading the capacity of fault tolerance. Because of this, **once the certificate rotation process has been started, it should be completed as quickly as possible.**

If for some reason the rotation of the TLS certificates has started but cannot complete in all channels, it is advised to rotate TLS certificates back to what they were and attempt the rotation later.

Certificate expiration related authentication

Whenever a client with an identity that has an expiration date (such as an identity based on an x509 certificate) sends a transaction to the orderer, the orderer checks whether its identity has expired, and if so, rejects the transaction submission.

However, it is possible to configure the orderer to ignore expiration of identities via enabling the `General.Authentication.NoExpirationChecks` configuration option in the `orderer.yaml`.

This should be done only under extreme circumstances, where the certificates of the administrators have expired, and due to this it is not possible to send configuration updates to replace the administrator certificates with renewed ones, because the config transactions signed by the existing administrators are now rejected because they have expired. After updating the channel it is recommended to change back to the default configuration which enforces expiration checks on identities.

9.17.4 Metrics

For a description of the Operations Service and how to set it up, check out [our documentation on the Operations Service](#).

For a list at the metrics that are gathered by the Operations Service, check out our [reference material on metrics](#).

While the metrics you prioritize will have a lot to do with your particular use case and configuration, there are two metrics in particular you might want to monitor:

- `consensus_etcdraft_is_leader`: identifies which node in the cluster is currently leader. If no nodes have this set, you have lost quorum.
- `consensus_etcdraft_data_persist_duration`: indicates how long write operations to the Raft cluster's persistent write ahead log take. For protocol safety, messages must be persisted durably, calling `fsync` where appropriate, before they can be shared with the consenter set. If this value begins to climb, this node may not be able to participate in consensus (which could lead to a service interruption for this node and possibly the network).
- `consensus_etcdraft_cluster_size` and `consensus_etcdraft_active_nodes`: these channel metrics help track the “active” nodes (which, as it sounds, are the nodes that are currently contributing to the cluster, as compared to the total number of nodes in the cluster). If the number of active nodes falls below a majority of the nodes in the cluster, quorum will be lost and the ordering service will stop processing blocks on the channel.

9.17.5 Troubleshooting

- The more stress you put on your nodes, the more you might have to change certain parameters. As with any system, computer or mechanical, stress can lead to a drag in performance. As we noted in the conceptual

documentation, leader elections in Raft are triggered when follower nodes do not receive either a “heartbeat” messages or an “append” message that carries data from the leader for a certain amount of time. Because Raft nodes share the same communication layer across channels (this does not mean they share data — they do not!), if a Raft node is part of the consenter set in many channels, you might want to lengthen the amount of time it takes to trigger an election to avoid inadvertent leader elections.

9.18 Migrating from Kafka to Raft

Note: this document presumes a high degree of expertise with channel configuration update transactions. As the process for migration involves several channel configuration update transactions, do not attempt to migrate from Kafka to Raft without first familiarizing yourself with the [Add an Organization to a Channel](#) tutorial, which describes the channel update process in detail.

For users who want to transition channels from using Kafka-based ordering services to [Raft-based](#) ordering services, nodes at v1.4.2 or higher allow this to be accomplished through a series of configuration update transactions on each channel in the network.

This tutorial will describe this process at a high level, calling out specific details where necessary, rather than show each command in detail.

9.18.1 Assumptions and considerations

Before attempting migration, take the following into account:

1. This process is solely for migration from Kafka to Raft. Migrating between any other orderer consensus types is not currently supported.
2. Migration is one way. Once the ordering service is migrated to Raft, and starts committing transactions, it is not possible to go back to Kafka.
3. Because the ordering nodes must go down and be brought back up, downtime must be allowed during the migration.
4. Recovering from a botched migration is possible only if a backup is taken at the point in migration prescribed later in this document. If you do not take a backup, and migration fails, you will not be able to recover your previous state.
5. All channels must be migrated during the same maintenance window. It is not possible to migrate only some channels before resuming operations.
6. At the end of the migration process, every channel will have the same consenter set of Raft nodes. This is the same consenter set that will exist in the ordering system channel. This makes it possible to diagnose a successful migration.
7. Migration is done in place, utilizing the existing ledgers for the deployed ordering nodes. Addition or removal of orderers should be performed after the migration.

9.18.2 High level migration flow

Migration is carried out in five phases.

1. The system is placed into a maintenance mode where application transactions are rejected and only ordering service admins can make changes to the channel configuration.
2. The system is stopped, and a backup is taken in case an error occurs during migration.
3. The system is started, and each channel has its consensus type and metadata modified.

4. The system is restarted and is now operating on Raft consensus; each channel is checked to confirm that it has successfully achieved a quorum.
5. The system is moved out of maintenance mode and normal function resumes.

9.18.3 Preparing to migrate

There are several steps you should take before attempting to migrate.

- Design the Raft deployment, deciding which ordering service nodes are going to remain as Raft consenters. You should deploy at least three ordering nodes in your cluster, but note that deploying a consenter set of at least five nodes will maintain high availability should a node goes down, whereas a three node configuration will lose high availability once a single node goes down for any reason (for example, as during a maintenance cycle).
- Prepare the material for building the Raft Metadata configuration. **Note: all the channels should receive the same Raft Metadata configuration.** Refer to the [Raft configuration guide](#) for more information on these fields. Note: you may find it easiest to bootstrap a new ordering network with the Raft consensus protocol, then copy and modify the consensus metadata section from its config. In any case, you will need (for each ordering node):
 - hostname
 - port
 - server certificate
 - client certificate
- Compile a list of all channels (system and application) in the system. Make sure you have the correct credentials to sign the configuration updates. For example, the relevant ordering service admin identities.
- Ensure all ordering service nodes are running the same version of Fabric, and that this version is v1.4.2 or greater.
- Ensure all peers are running at least v1.4.2 of Fabric. Make sure all channels are configured with the channel capability that enables migration.
 - Orderer capability V1_4_2 (or above).
 - Channel capability V1_4_2 (or above).

Entry to maintenance mode

Prior to setting the ordering service into maintenance mode, it is recommended that the peers and clients of the network be stopped. Leaving peers or clients up and running is safe, however, because the ordering service will reject all of their requests, their logs will fill with benign but misleading failures.

Follow the process in the [Add an Organization to a Channel](#) tutorial to pull, translate, and scope the configuration of **each channel, starting with the system channel**. The only field you should change during this step is in the channel configuration at `/Channel/Orderer/ConsensusType`. In a JSON representation of the channel configuration, this would be `.channel_group.groups.Orderer.values.ConsensusType`.

The `ConsensusType` is represented by three values: `Type`, `Metadata`, and `State`, where:

- `Type` is either `kafka` or `etcdraft` (Raft). This value can only be changed while in maintenance mode.
- `Metadata` will be empty if the `Type` is `kafka`, but must carry valid Raft metadata if the `ConsensusType` is `etcdraft`. More on this below.
- `State` is either `STATE_NORMAL`, when the channel is processing transactions, or `STATE_MAINTENANCE`, during the migration process.

In the first step of the channel configuration update, only change the `State` from `STATE_NORMAL` to `STATE_MAINTENANCE`. Do not change the `Type` or the `Metadata` field yet. Note that the `Type` should currently be `kafka`.

While in maintenance mode, normal transactions, config updates unrelated to migration, and `Deliver` requests from the peers used to retrieve new blocks are rejected. This is done in order to prevent the need to both backup, and if necessary restore, peers during migration, as they only receive updates once migration has successfully completed. In other words, we want to keep the ordering service backup point, which is the next step, ahead of the peer's ledger, in order to be able to perform rollback if needed. However, ordering node admins can issue `Deliver` requests (which they need to be able to do in order to continue the migration process).

Verify that each ordering service node has entered maintenance mode on each of the channels. This can be done by fetching the last config block and making sure that the `Type`, `Metadata`, `State` on each channel is `kafka`, empty (recall that there is no metadata for Kafka), and `STATE_MAINTENANCE`, respectively.

If the channels have been updated successfully, the ordering service is now ready for backup.

Backup files and shut down servers

Shut down all ordering nodes, Kafka servers, and Zookeeper servers. It is important to **shutdown the ordering service nodes first**. Then, after allowing the Kafka service to flush its logs to disk (this typically takes about 30 seconds, but might take longer depending on your system), the Kafka servers should be shut down. Shutting down the Kafka brokers at the same time as the orderers can result in the filesystem state of the orderers being more recent than the Kafka brokers which could prevent your network from starting.

Create a backup of the file system of these servers. Then restart the Kafka service and then the ordering service nodes.

Switch to Raft in maintenance mode

The next step in the migration process is another channel configuration update for each channel. In this configuration update, switch the `Type` to `etcdraft` (for Raft) while keeping the `State` in `STATE_MAINTENANCE`, and fill in the `Metadata` configuration. It is highly recommended that the `Metadata` configuration be identical on all channels. If you want to establish different consenter sets with different nodes, you will be able to reconfigure the `Metadata` configuration after the system is restarted into `etcdraft` mode. Supplying an identical metadata object, and hence, an identical consenter set, means that when the nodes are restarted, if the system channel forms a quorum and can exit maintenance mode, other channels will likely be able to do the same. Supplying different consenter sets to each channel can cause one channel to succeed in forming a cluster while another channel will fail.

Then, validate that each ordering service node has committed the `ConsensusType` change configuration update by pulling and inspecting the configuration of each channel.

Note: For each channel, the transaction that changes the `ConsensusType` must be the last configuration transaction before restarting the nodes (in the next step). If some other configuration transaction happens after this step, the nodes will most likely crash on restart, or result in undefined behavior.

Restart and validate leader

Note: exit of maintenance mode **must** be done **after** restart.

After the `ConsensusType` update has been completed on each channel, stop all ordering service nodes, stop all Kafka brokers and Zookeepers, and then restart only the ordering service nodes. They should restart as Raft nodes, form a cluster per channel, and elect a leader on each channel.

Note: Since the Raft-based ordering service uses client and server TLS certificates for authentication between orderer nodes, **additional configurations** are required before you start them again, see [Section: Local Configuration](#) for more details.

After restart process finished, make sure to **validate** that a leader has been elected on each channel by inspecting the node logs (you can see what to look for below). This will confirm that the process has been completed successfully.

When a leader is elected, the log will show, for each channel:

```
"Raft leader changed: 0 -> node-number channel=channel-name
node=node-number "
```

For example:

```
2019-05-26 10:07:44.075 UTC [orderer.consensus.etcdraft] serveRequest ->
INFO 047 Raft leader changed: 0 -> 1 channel=testchannel1 node=2
```

In this example node 2 reports that a leader was elected (the leader is node 1) by the cluster of channel testchannel1.

Switch out of maintenance mode

Perform another channel configuration update on each channel (sending the config update to the same ordering node you have been sending configuration updates to until now), switching the State from STATE_MAINTENANCE to STATE_NORMAL. Start with the system channel, as usual. If it succeeds on the ordering system channel, migration is likely to succeed on all channels. To verify, fetch the last config block of the system channel from the ordering node, verifying that the State is now STATE_NORMAL. For completeness, verify this on each ordering node.

When this process is completed, the ordering service is now ready to accept all transactions on all channels. If you stopped your peers and application as recommended, you may now restart them.

9.18.4 Abort and rollback

If a problem emerges during the migration process **before exiting maintenance mode**, simply perform the rollback procedure below.

1. Shut down the ordering nodes and the Kafka service (servers and Zookeeper ensemble).
2. Rollback the file system of these servers to the backup taken at maintenance mode before changing the ConsensusType.
3. Restart said servers, the ordering nodes will bootstrap to Kafka in maintenance mode.
4. Send a configuration update exiting maintenance mode to continue using Kafka as your consensus mechanism, or resume the instructions after the point of backup and fix the error which prevented a Raft quorum from forming and retry migration with corrected Raft configuration Metadata.

There are a few states which might indicate migration has failed:

1. Some nodes crash or shutdown.
2. There is no record of a successful leader election per channel in the logs.
3. The attempt to flip to STATE_NORMAL mode on the system channel fails.

9.19 Bringing up a Kafka-based Ordering Service

9.19.1 Caveat emptor

This document assumes that the reader knows how to set up a Kafka cluster and a ZooKeeper ensemble, and keep them secure for general usage by preventing unauthorized access. The sole purpose of this guide is to identify the steps you need to take so as to have a set of Hyperledger Fabric ordering service nodes (OSNs) use your Kafka cluster and provide an ordering service to your blockchain network.

For information about the role orderers play in a network and in a transaction flow, checkout our [The Ordering Service](#) documentation.

For information on how to set up an ordering node, check out our [orderer_deploy](#) documentation.

For information about configuring Raft ordering services, check out [Configuring and operating a Raft ordering service](#).

9.19.2 Big picture

Each channel maps to a separate single-partition topic in Kafka. When an OSN receives transactions via the Broadcast RPC, it checks to make sure that the broadcasting client has permissions to write on the channel, then relays (i.e. produces) those transactions to the appropriate partition in Kafka. This partition is also consumed by the OSN which groups the received transactions into blocks locally, persists them in its local ledger, and serves them to receiving clients via the Deliver RPC. For low-level details, refer to [the document that describes how we came to this design](#). **Figure 8** is a schematic representation of the process described above.

9.19.3 Steps

Let K and Z be the number of nodes in the Kafka cluster and the ZooKeeper ensemble respectively:

1. At a minimum, K should be set to 4. (As we will explain in Step 4 below, this is the minimum number of nodes necessary in order to exhibit crash fault tolerance, i.e. with 4 brokers, you can have 1 broker go down, all channels will continue to be writeable and readable, and new channels can be created.)
2. Z will either be 3, 5, or 7. It has to be an odd number to avoid split-brain scenarios, and larger than 1 in order to avoid single point of failures. Anything beyond 7 ZooKeeper servers is considered overkill.

Then proceed as follows:

3. Orderers: **Encode the Kafka-related information in the network's genesis block.** If you are using `configtxgen`, edit `configtx.yaml`. Alternatively, pick a preset profile for the system channel's genesis block— so that:
 - `Orderer.OrdererType` is set to `kafka`.
 - `Orderer.Kafka.Brokers` contains the address of *at least two* of the Kafka brokers in your cluster in `IP:port` notation. The list does not need to be exhaustive. (These are your bootstrap brokers.)
4. Orderers: **Set the maximum block size.** Each block will have at most `Orderer.AbsoluteMaxBytes` bytes (not including headers), a value that you can set in `configtx.yaml`. Let the value you pick here be A and make note of it — it will affect how you configure your Kafka brokers in Step 6.
5. Orderers: **Create the genesis block.** Use `configtxgen`. The settings you picked in Steps 3 and 4 above are system-wide settings, i.e. they apply across the network for all the OSNs. Make note of the genesis block's location.
6. Kafka cluster: **Configure your Kafka brokers appropriately.** Ensure that every Kafka broker has these keys configured:

- `unclean.leader.election.enable = false` — Data consistency is key in a blockchain environment. We cannot have a channel leader chosen outside of the in-sync replica set, or we run the risk of overwriting the offsets that the previous leader produced, and —as a result— rewrite the blockchain that the orderers produce.
- `min.insync.replicas = M` — Where you pick a value M such that $1 < M < N$ (see `default.replication.factor` below). Data is considered committed when it is written to at least M replicas (which are then considered in-sync and belong to the in-sync replica set, or ISR). In any other case, the write operation returns an error. Then:
 - If up to $N-M$ replicas —out of the N that the channel data is written to become unavailable, operations proceed normally.
 - If more replicas become unavailable, Kafka cannot maintain an ISR set of M , so it stops accepting writes. Reads work without issues. The channel becomes writeable again when M replicas get in-sync.
- `default.replication.factor = N` — Where you pick a value N such that $N < K$. A replication factor of N means that each channel will have its data replicated to N brokers. These are the candidates for the ISR set of a channel. As we noted in the `min.insync.replicas` section above, not all of these brokers have to be available all the time. N should be set *strictly smaller* to K because channel creations cannot go forward if less than N brokers are up. So if you set $N = K$, a single broker going down means that no new channels can be created on the blockchain network — the crash fault tolerance of the ordering service is non-existent.

Based on what we’ve described above, the minimum allowed values for M and N are 2 and 3 respectively. This configuration allows for the creation of new channels to go forward, and for all channels to continue to be writeable.

- `message.max.bytes` and `replica.fetch.max.bytes` should be set to a value larger than A , the value you picked in `Orderer.AbsoluteMaxBytes` in Step 4 above. Add some buffer to account for headers — 1 MiB is more than enough. The following condition applies:

```
Orderer.AbsoluteMaxBytes < replica.fetch.max.bytes <= message.max.bytes
```

(For completeness, we note that `message.max.bytes` should be strictly smaller to `socket.request.max.bytes` which is set by default to 100 MiB. If you wish to have blocks larger than 100 MiB you will need to edit the hard-coded value in `brokerConfig.Producer.MaxMessageBytes` in `fabric/orderer/kafka/config.go` and rebuild the binary from source. This is not advisable.)

- `log.retention.ms = -1`. Until the ordering service adds support for pruning of the Kafka logs, you should disable time-based retention and prevent segments from expiring. (Size-based retention — see `log.retention.bytes` — is disabled by default in Kafka at the time of this writing, so there’s no need to set it explicitly.)

7. Orderers: **Point each OSN to the genesis block.** Edit `General.BootstrapFile` in `orderer.yaml` so that it points to the genesis block created in Step 5 above. While at it, ensure all other keys in that YAML file are set appropriately.

8. Orderers: **Adjust polling intervals and timeouts.** (Optional step.)

- The `Kafka.Retry` section in the `orderer.yaml` file allows you to adjust the frequency of the meta-data/producer/consumer requests, as well as the socket timeouts. (These are all settings you would expect to see in a Kafka producer or consumer.)
- Additionally, when a new channel is created, or when an existing channel is reloaded (in case of a just-restarted orderer), the orderer interacts with the Kafka cluster in the following ways:
 - It creates a Kafka producer (writer) for the Kafka partition that corresponds to the channel. . It uses that producer to post a no-op `CONNECT` message to that partition. . It creates a Kafka consumer (reader) for that partition.

- If any of these steps fail, you can adjust the frequency with which they are repeated. Specifically they will be re-attempted every `Kafka.Retry.ShortInterval` for a total of `Kafka.Retry.ShortTotal`, and then every `Kafka.Retry.LongInterval` for a total of `Kafka.Retry.LongTotal` until they succeed. Note that the orderer will be unable to write to or read from a channel until all of the steps above have been completed successfully.
9. **Set up the OSNs and Kafka cluster so that they communicate over SSL.** (Optional step, but highly recommended.) Refer to [the Confluent guide](#) for the Kafka cluster side of the equation, and set the keys under `Kafka.TLS` in `orderer.yaml` on every OSN accordingly.
 10. **Bring up the nodes in the following order: ZooKeeper ensemble, Kafka cluster, ordering service nodes.**

9.19.4 Additional considerations

1. **Preferred message size.** In Step 4 above (see [Steps](#) section) you can also set the preferred size of blocks by setting the `Orderer.Batchsize.PreferredMaxBytes` key. Kafka offers higher throughput when dealing with relatively small messages; aim for a value no bigger than 1 MiB.
2. **Using environment variables to override settings.** When using the sample Kafka and Zookeeper Docker images provided with Fabric (see `images/kafka` and `images/zookeeper` respectively), you can override a Kafka broker or a ZooKeeper server's settings by using environment variables. Replace the dots of the configuration key with underscores. For example, `KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE=false` will allow you to override the default value of `unclean.leader.election.enable`. The same applies to the OSNs for their *local* configuration, i.e. what can be set in `orderer.yaml`. For example `ORDERER_KAFKA_RETRY_SHORTINTERVAL=1s` allows you to override the default value for `Orderer.Kafka.Retry.ShortInterval`.

9.19.5 Kafka Protocol Version Compatibility

Fabric uses the [sarama client library](#) and vendors a version of it that supports Kafka 0.10 to 1.0, yet is still known to work with older versions.

Using the `Kafka.Version` key in `orderer.yaml`, you can configure which version of the Kafka protocol is used to communicate with the Kafka cluster's brokers. Kafka brokers are backward compatible with older protocol versions. Because of a Kafka broker's backward compatibility with older protocol versions, upgrading your Kafka brokers to a new version does not require an update of the `Kafka.Version` key value, but the Kafka cluster might suffer a [performance penalty](#) while using an older protocol version.

9.19.6 Debugging

Set environment variable `FABRIC_LOGGING_SPEC` to `DEBUG` and set `Kafka.Verbose` to `true` in `orderer.yaml`.

Upgrading to the latest release

If you're familiar with previous releases of Hyperledger Fabric, you're aware that upgrading the nodes and channels to the latest version of Fabric is, at a high level, a four step process.

1. Backup the ledger and MSPs.
2. Upgrade the orderer binaries in a rolling fashion to the latest Fabric version.
3. Upgrade the peer binaries in a rolling fashion to the latest Fabric version.
4. Update the orderer system channel and any application channels to the latest capability levels, where available. Note that some releases will have capabilities in all groups while other releases may have few or even no new capabilities at all.
5. If you want to migrate your network by removing the orderer system channel, check out [Create a channel](#).

For more information about capabilities, check out [Channel capabilities](#).

For a look at how these upgrade processes are accomplished, please consult these tutorials:

1. [Considerations for getting to v2.x](#). This topic discusses the important considerations for getting to the latest release from the previous release as well as from the most recent long term support (LTS) release.
2. [Upgrading your components](#). Components should be upgraded to the latest version before updating any capabilities.
3. [Updating the capability level of a channel](#). Completed after updating the versions of all nodes.
4. [Enabling the new chaincode lifecycle](#). Necessary to add organization specific endorsement policies central to the new chaincode lifecycle for Fabric v2.x.

As the upgrading of nodes and increasing the capability levels of channels is by now considered a standard Fabric process, we will not show the specific commands for upgrading to the newest release. Similarly, there is no script in the `fabric-samples` repo that will upgrade a sample network from the previous release to this one, as there has been for previous releases.

Note: It is a best practice to upgrade your SDK to the latest version as a part of a general upgrade of your network. While the SDK will always be compatible with equivalent releases of Fabric and lower, it might be necessary to

upgrade to the latest SDK to leverage the latest Fabric features. Consult the documentation of the Fabric SDK you are using for information about how to upgrade.

10.1 Considerations for getting to v2.x

In this topic we'll cover recommendations for upgrading to the newest release from the previous release as well as from the most recent long term support (LTS) release.

10.1.1 Upgrading from 2.1 to 2.2

The v2.1 and v2.2 releases of Fabric are stabilization releases, featuring bug fixes and other forms of code hardening. As such there are no particular considerations needed for upgrade, and no new capability levels requiring particular image versions or channel configuration updates.

Upgrading from 2.2 to 2.3

The v2.3 release of Fabric includes two main new features:

1. The ability to take snapshots of the ledgers on peers (and to bootstrap a new peer from a snapshot). For more information, check out [Taking ledger snapshots and using them to join channels](#).
2. Channels can now be created without first creating a system channel. For more information, check out [Creating a channel without a system channel](#).

Neither of these features require channel updates to capability in order to function. However, you will need to upgrade to v2.3 to take advantage of both features and, in the case of the new channel creation process, need to migrate away from the system channel, which is covered in the [Creating a channel without a system channel](#) tutorial.

10.1.2 Upgrading to 2.2 from the 1.4.x long term support release

Before attempting to upgrade from v1.4.x to v2.2, make sure to consider the following:

Chaincode lifecycle

The new chaincode lifecycle that debuted in v2.0 allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. For more information about the new chaincode lifecycle, check out [Fabric chaincode lifecycle](#) concept topic.

It is a best practice to upgrade all of the peers on a channel before enabling the `Channel` and `Application` capabilities that enable the new chaincode lifecycle (the `Channel` capability is not strictly required, but it makes sense to update it at this time). Note that any peers that are not at v2.x will crash after enabling either capability, while any ordering nodes that are not at v2.x will crash after the `Channel` capability has been enabled. This crashing behavior is intentional, as the peer or orderer cannot safely participate in the channel if it does not support the required capabilities.

After the `Application` capability has been updated to `V2_0` on a channel, you must use the v2.x lifecycle procedures to package, install, approve, and commit new chaincodes on the channel. As a result, make sure to be prepared for the new lifecycle before updating the capability.

The new lifecycle defaults to using the endorsement policy configured in the channel config (e.g., a `MAJORITY` of orgs). Therefore this endorsement policy should be added to the channel configuration when enabling capabilities on the channel.

For information about how to edit the relevant channel configurations to enable the new lifecycle by adding an endorsement policy for each organization, check out [Enabling the new chaincode lifecycle](#).

Chaincode shim changes (Go chaincode only)

The `v2.x ccenv` image that is used to build Go chaincodes no longer automatically vendors the Go chaincode shim dependency like the `v1.4 ccenv` image did. The recommended approach is to vendor the shim in your `v1.4` Go chaincode before making upgrades to the peers and channels, since this approach works with both a `v1.4.x` and `v2.x` peer. If you are already using an existing tool such as `govendor` you may continue using it to vendor the chaincode shim. Best practice, however, would be to use Go modules to vendor the chaincode shim, as modules are now the de facto standard for dependency management in the Go ecosystem. Note that since Fabric `v2.0`, chaincode using Go modules without vendored dependencies is also supported. If you do this, you do not need to make any additional changes to your chaincode.

If you did not vendor the shim in your `v1.4` chaincode, the old `v1.4` chaincode images will still technically work after upgrade, but you are in a risky state. If the chaincode image gets deleted from your environment for whatever reason, the next invoke on `v2.x` peer will try to rebuild the chaincode image and you'll get an error that the shim cannot be found.

At this point, you have two options:

1. If the entire channel is ready to upgrade chaincode, you can upgrade the chaincode on all peers and on the channel (using either the old or new lifecycle depending on the `Application` capability level you have enabled). The best practice at this point would be to vendor the new Go chaincode shim using modules.
2. If the entire channel is not yet ready to upgrade the chaincode, you can use peer environment variables to specify the `v1.4` chaincode environment `ccenv` be used to rebuild the chaincode images. This `v1.4 ccenv` should still work with a `v2.x` peer.

Chaincode logger (Go chaincode only)

Support for user chaincodes to utilize the chaincode shim's logger via `NewLogger()` has been removed. Chaincodes that used the shim's `NewLogger()` must now shift to their own preferred logging mechanism.

For more information, check out [Logging control](#).

Peer databases upgrade

For information about how to upgrade peers, check out our documentation on [upgrading components](#). During the process for [upgrading your peers](#), you will need to perform one additional step to upgrade the peer databases. The databases of all peers (which include not just the state database but the history database and other internal databases for the peer) must be rebuilt using the `v2.x` data format as part of the upgrade to `v2.x`. To trigger the rebuild, the databases must be dropped before the peer is started. The instructions below utilize the `peer node upgrade-dbs` command to drop the local databases managed by the peer and prepare them for upgrade, so that they can be rebuilt the first time the `v2.x` peer starts. If you are using CouchDB as the state database, the peer has support to automatically drop this database as of `v2.2`. To leverage the support, you must configure the peer with CouchDB as the state database and start CouchDB before running the `upgrade-dbs` command. In `v2.0` and `v2.1`, the peer does not automatically drop the CouchDB state database; therefore you must drop it yourself.

Follow the commands to upgrade a peer until the `docker run` command to launch the new peer container (you can skip the step where you set an `IMAGE_TAG`, since the `upgrade-dbs` command is for the `v2.x` release of Fabric

only, but you will need to set the `PEER_CONTAINER` and `LEDGERS_BACKUP` environment variables). Instead of the `docker run` command to launch the peer, run this command instead to drop and prepare the local databases managed by the peer (substitute 2.1 for 2.0 in these commands if you are upgrading to that binary version from the 1.4.x LTS):

```
docker run --rm -v /opt/backup/$PEER_CONTAINER/:/var/hyperledger/production/ \
-v /opt/msp/:/etc/hyperledger/fabric/msp/ \
--env-file ./env<name of node>.list \
--name $PEER_CONTAINER \
hyperledger/fabric-peer:2.0 peer node upgrade-dbs
```

In v2.0 and v2.1, if you are using CouchDB as the state database, also drop the CouchDB database. This can be done by removing the CouchDB `/data` volume directory.

Then issue this command to start the peer using the 2.0 tag:

```
docker run -d -v /opt/backup/$PEER_CONTAINER/:/var/hyperledger/production/ \
-v /opt/msp/:/etc/hyperledger/fabric/msp/ \
--env-file ./env<name of node>.list \
--name $PEER_CONTAINER \
hyperledger/fabric-peer:2.0 peer node start
```

The peer will rebuild the databases using the v2.x data format the first time it starts. Because rebuilding the databases can be a lengthy process (several hours, depending on the size of your databases), monitor the peer logs to check the status of the rebuild. Every 1000th block you will see a message like `[lockbasedtxmgr] CommitLostBlock -> INFO 041 Recommitting block [1000] to state database` indicating the rebuild is ongoing.

If the database is not dropped as part of the upgrade process, the peer start will return an error message stating that its databases are in the old format and must be dropped using the `peer node upgrade-dbs` command above (or dropped manually if using CouchDB state database). The node will then need to be restarted again.

Capabilities

The 2.0 release featured three new capabilities.

- **Application V2_0:** enables the new chaincode lifecycle as described in [Fabric chaincode lifecycle](#) concept topic.
- **Channel V2_0:** this capability has no changes, but is used for consistency with the application and orderer capability levels.
- **Orderer V2_0:** controls `UseChannelCreationPolicyAsAdmins`, changing the way that channel creation transactions are validated. When combined with the `-baseProfile` option of `configtxgen`, values which were previously inherited from the orderer system channel may now be overridden.

As with any update of the capability levels, make sure to upgrade your peer binaries before updating the Application and Channel capabilities, and make sure to upgrade your orderer binaries before updating the Orderer and Channel capabilities.

For information about how to set new capabilities, check out [Updating the capability level of a channel](#).

Define ordering node endpoint per org (recommend)

Starting with version v1.4.2, it was recommended to define orderer endpoints in all channels at the organization level by adding a new `OrdererEndpoints` stanza within the channel configuration of an organization, replacing the global `OrdererAddresses` section of channel configuration. If at least one organization has an ordering service endpoint

defined at an organizational level, all orderers and peers will ignore the channel level endpoints when connecting to ordering nodes.

Utilizing organization level orderer endpoints is required when using service discovery with ordering nodes provided by multiple organizations. This allows clients to provide the correct organization TLS certificates.

If your channel configuration does not yet include `OrdererEndpoints` per org, you will need to perform a channel configuration update to add them to the config. First, create a JSON file that includes the new configuration stanza.

In this example, we will create a stanza for a single org called `OrdererOrg`. Note that if you have multiple ordering service organizations, they will all have to be updated to include endpoints. Let's call our JSON file `orglevelEndpoints.json`.

```
{
  "OrdererOrgEndpoint": {
    "Endpoints": {
      "mod_policy": "Admins",
      "value": {
        "addresses": [
          "127.0.0.1:30000"
        ]
      }
    }
  }
}
```

Then, export the following environment variables:

- `CH_NAME`: the name of the channel being updated.
- `CORE_PEER_LOCALMSPID`: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the orderer organizations.
- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP representing your organization.
- `TLS_ROOT_CA`: the absolute path to the root CA certificate of the organization proposing the channel update.
- `ORDERER_CONTAINER`: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.
- `ORGNAME`: The name of the organization you are currently updating. For example, `OrdererOrg`.

Once you have set the environment variables, navigate to [Step 1: Pull and translate the config](#).

Then, add the lifecycle organization policy (as listed in `orglevelEndpoints.json`) to a file called `modified_config.json` using this command:

```
jq -s ".[0] * {\n\"channel_group\\\":{\n\"groups\\\":{\n\"Orderer\\\": {\n\"groups\\\": {\n\"$ORGNAME\\  
→ \": {\n\"values\\\": .[1].${ORGNAME}Endpoint}}}}}" config.json ./orglevelEndpoints.json_  
→> modified_config.json
```

Then, follow the steps at [Step 3: Re-encode and submit the config](#).

If every ordering service organization performs their own channel edit, they can edit the configuration without needing further signatures (by default, the only signature needed to edit parameters within an organization is an admin of that organization). If a different organization proposes the update, then the organization being edited will need to sign the channel update request.

10.2 Upgrading your components

Audience: network administrators, node administrators

For information about special considerations for the latest release of Fabric, check out [Upgrading to the latest release of Fabric](#).

This topic will only cover the process for upgrading components. For information about how to edit a channel to change the capability level of your channels, check out [Updating a channel capability](#).

Note: when we use the term “upgrade” in Hyperledger Fabric, we’re referring to changing the version of a component (for example, going from one version of a binary to the next version). The term “update,” on the other hand, refers not to versions but to configuration changes, such as updating a channel configuration or a deployment script. As there is no data migration, technically speaking, in Fabric, we will not use the term “migration” or “migrate” here.

10.2.1 Overview

At a high level, upgrading the binary level of your nodes is a two step process:

1. Backup the ledger and MSPs.
2. Upgrade binaries to the latest version.

If you own both ordering nodes and peers, it is a best practice to upgrade the ordering nodes first. If a peer falls behind or is temporarily unable to process certain transactions, it can always catch up. If enough ordering nodes go down, by comparison, a network can effectively cease to function.

This topic presumes that these steps will be performed using Docker CLI commands. If you are utilizing a different deployment method (Rancher, Kubernetes, OpenShift, etc) consult their documentation on how to use their CLI.

For native deployments, note that you will also need to update the YAML configuration file for the nodes (for example, the `orderer.yaml` file) with the one from the release artifacts.

To do this, backup the `orderer.yaml` or `core.yaml` file (for the peer) and replace it with the `orderer.yaml` or `core.yaml` file from the release artifacts. Then port any modified variables from the backed up `orderer.yaml` or `core.yaml` to the new one. Using a utility like `diff` may be helpful. Note that updating the YAML file from the release rather than updating your old YAML file **is the recommended way to update your node YAML files**, as it reduces the likelihood of making errors.

This tutorial assumes a Docker deployment where the YAML files will be baked into the images and environment variables will be used to overwrite the defaults in the configuration files.

10.2.2 Environment variables for the binaries

When you deploy a peer or an ordering node, you had to set a number of environment variables relevant to its configuration. A best practice is to create a file for these environment variables, give it a name relevant to the node being deployed, and save it somewhere on your local file system. That way you can be sure that when upgrading the peer or ordering node you are using the same variables you set when creating it.

Here’s a list of some of the **peer** environment variables (with sample values — as you can see from the addresses, these environment variables are for a network deployed locally) that can be set that be listed in the file. Note that you may or may not need to set all of these environment variables:

```
CORE_PEER_TLS_ENABLED=true
CORE_PEER_GOSSIP_USELEADERELECTION=true
CORE_PEER_GOSSIP_ORGLEADER=false
CORE_PEER_PROFILE_ENABLED=true
```

(continues on next page)

(continued from previous page)

```

CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
CORE_PEER_ID=peer0.org1.example.com
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_LISTENADDRESS=0.0.0.0:7051
CORE_PEER_CHAINCODEADDRESS=peer0.org1.example.com:7052
CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:7052
CORE_PEER_GOSSIP_BOOTSTRAP=peer0.org1.example.com:7051
CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
CORE_PEER_LOCALMSPID=Org1MSP

```

Here are some **ordering node** variables (again, these are sample values) that might be listed in the environment variable file for a node. Again, you may or may not need to set all of these environment variables:

```

ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
ORDERER_GENERAL_GENESISMETHOD=file
ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
ORDERER_GENERAL_LOCALMSPID=OrdererMSP
ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
ORDERER_GENERAL_TLS_ENABLED=true
ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
ORDERER_GENERAL_CLUSTER_CLIENTCERTIFICATE=/var/hyperledger/orderer/tls/server.crt
ORDERER_GENERAL_CLUSTER_CLIENTPRIVATEKEY=/var/hyperledger/orderer/tls/server.key
ORDERER_GENERAL_CLUSTER_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]

```

However you choose to set your environment variables, note that they will have to be set for each node you want to upgrade.

10.2.3 Ledger backup and restore

While we will demonstrate the process for backing up ledger data in this tutorial, it is not strictly required to backup the ledger data of a peer or an ordering node (assuming the node is part of a larger group of nodes in an ordering service). This is because, even in the worst case of catastrophic failure of a peer (such as a disk failure), the peer can be brought up with no ledger at all. You can then have the peer re-join the desired channels and as a result, the peer will automatically create a ledger for each of the channels and will start receiving the blocks via regular block transfer mechanism from either the ordering service or the other peers in the channel. As the peer processes blocks, it will also build up its state database.

However, backing up ledger data enables the restoration of a peer without the time and computational costs associated with bootstrapping from the genesis block and reprocessing all transactions, a process that can take hours (depending on the size of the ledger). In addition, ledger data backups may help to expedite the addition of a new peer, which can be achieved by backing up the ledger data from one peer and starting the new peer with the backed up ledger data.

This tutorial presumes that the file path to the ledger data has not been changed from the default value of `/var/hyperledger/production/` (for peers) or `/var/hyperledger/production/orderer` (for ordering nodes). If this location has been changed for your nodes, enter the path to the data on your ledgers in the commands below.

Note that there will be data for both the ledger and chaincodes at this file location. While it is a best practice to backup both, it is possible to skip the `stateLeveldb`, `historyLeveldb`, `chains/index` folders at `/var/hyperledger/production/ledgersData`. While skipping these folders reduces the storage needed for the backup, the peer recovery from the backed up data may take more time as these ledger artifacts will be re-constructed when the peer starts.

If using CouchDB as state database, there will be no `stateLeveldb` directory, as the state database data would be stored within CouchDB instead. But similarly, if peer starts up and finds CouchDB databases are missing or at lower block height (based on using an older CouchDB backup), the state database will be automatically re-constructed to catch up to current block height. Therefore, if you backup peer ledger data and CouchDB data separately, ensure that the CouchDB backup is always older than the peer backup.

10.2.4 Upgrade ordering nodes

Orderer containers should be upgraded in a rolling fashion (one at a time). At a high level, the ordering node upgrade process goes as follows:

1. Stop the ordering node.
2. Back up the ordering node's ledger and MSP.
3. Remove the ordering node container.
4. Launch a new ordering node container using the relevant image tag.

Repeat this process for each node in your ordering service until the entire ordering service has been upgraded.

Set command environment variables

Export the following environment variables before attempting to upgrade your ordering nodes.

- `ORDERER_CONTAINER`: the name of your ordering node container. Note that you will need to export this variable for each node when upgrading it.
- `LEDGERS_BACKUP`: the place in your local filesystem where you want to store the ledger being backed up. As you will see below, each node being backed up will have its own subfolder containing its ledger. You will need to create this folder.
- `IMAGE_TAG`: the Fabric version you are upgrading to. For example, `2.0`.

Note that you will have to set an **image tag** to ensure that the node you are starting using the correct images. The process you use to set the tag will depend on your deployment method.

Upgrade containers

Let's begin the upgrade process by **bringing down the orderer**:

```
docker stop $ORDERER_CONTAINER
```

Once the orderer is down, you'll want to **backup its ledger and MSP**:

```
docker cp $ORDERER_CONTAINER:/var/hyperledger/production/orderer/ ./LEDGERS_BACKUP/  
↪$ORDERER_CONTAINER
```

Then remove the ordering node container itself (since we will be giving our new container the same name as our old one):

```
docker rm -f $ORDERER_CONTAINER
```

Then you can launch the new ordering node container by issuing:


```
docker run -d -v /opt/backup/$ORDERER_CONTAINER:/var/hyperledger/production/orderer/ \
    -v /opt/msp:/etc/hyperledger/fabric/msp/ \
    --env-file ./env<name of node>.list \
    --name $ORDERER_CONTAINER \
    hyperledger/fabric-orderer:$IMAGE_TAG orderer
```

Once all of the ordering nodes have come up, you can move on to upgrading your peers.

10.2.5 Upgrade the peers

Peers should, like the ordering nodes, be upgraded in a rolling fashion (one at a time). As mentioned during the ordering node upgrade, ordering nodes and peers may be upgraded in parallel, but for the purposes of this tutorial we've separated the processes out. At a high level, we will perform the following steps:

1. Stop the peer.
2. Back up the peer's ledger and MSP.
3. Remove chaincode containers and images.
4. Remove the peer container.
5. Launch a new peer container using the relevant image tag.

Set command environment variables

Export the following environment variables before attempting to upgrade your peers.

- **PEER_CONTAINER**: the name of your peer container. Note that you will need to set this variable for each node.
- **LEDGERS_BACKUP**: the place in your local filesystem where you want to store the ledger being backed up. As you will see below, each node being backed up will have its own subfolder containing its ledger. You will need to create this folder.
- **IMAGE_TAG**: the Fabric version you are upgrading to. For example, 2.0.

Note that you will have to set an **image tag** to ensure that the node you are starting is using the correct images. The process you use to set the tag will depend on your deployment method.

Repeat this process for each of your peers until every node has been upgraded.

Upgrade containers

Let's **bring down the first peer** with the following command:

```
docker stop $PEER_CONTAINER
```

We can then **backup the peer's ledger and MSP**:

```
docker cp $PEER_CONTAINER:/var/hyperledger/production ./LEDGERS_BACKUP/$PEER_
CONTAINER
```

With the peer stopped and the ledger backed up, **remove the peer chaincode containers**:

```
CC_CONTAINERS=$(docker ps | grep dev-$PEER_CONTAINER | awk '{print $1}')
if [ -n "$CC_CONTAINERS" ] ; then docker rm -f $CC_CONTAINERS ; fi
```

And the peer chaincode images:

```
CC_IMAGES=$(docker images | grep dev-$PEER | awk '{print $1}')
if [ -n "$CC_IMAGES" ] ; then docker rmi -f $CC_IMAGES ; fi
```

Then remove the peer container itself (since we will be giving our new container the same name as our old one):

```
docker rm -f $PEER_CONTAINER
```

Then you can launch the new peer container by issuing:

```
docker run -d -v /opt/backup/$PEER_CONTAINER:/var/hyperledger/production/ \
-v /opt/msp:/etc/hyperledger/fabric/msp/ \
--env-file ./env<name of node>.list \
--name $PEER_CONTAINER \
hyperledger/fabric-peer:$IMAGE_TAG peer node start
```

You do not need to relaunch the chaincode container. When the peer gets a request for a chaincode, (invoke or query), it first checks if it has a copy of that chaincode running. If so, it uses it. Otherwise, as in this case, the peer launches the chaincode (rebuilding the image if required).

Verify peer upgrade completion

It's a best practice to ensure the upgrade has been completed properly with a chaincode invoke. Note that it should be possible to verify that a single peer has been successfully updated by querying one of the ledgers hosted on the peer. If you want to verify that multiple peers have been upgraded, and are updating your chaincode as part of the upgrade process, you should wait until peers from enough organizations to satisfy the endorsement policy have been upgraded.

Before you attempt this, you may want to upgrade peers from enough organizations to satisfy your endorsement policy. However, this is only mandatory if you are updating your chaincode as part of the upgrade process. If you are not updating your chaincode as part of the upgrade process, it is possible to get endorsements from peers running at different Fabric versions.

10.2.6 Upgrade your CAs

To learn how to upgrade your Fabric CA server, click over to the [CA documentation](#).

10.2.7 Upgrade Node SDK clients

Upgrade Fabric and Fabric CA before upgrading Node SDK clients. Fabric and Fabric CA are tested for backwards compatibility with older SDK clients. While newer SDK clients often work with older Fabric and Fabric CA releases, they may expose features that are not yet available in the older Fabric and Fabric CA releases, and are not tested for full compatibility.

Use NPM to upgrade any Node.js client by executing these commands in the root directory of your application:

```
npm install fabric-client@latest
npm install fabric-ca-client@latest
```

These commands install the new version of both the Fabric client and Fabric-CA client and write the new versions to package.json.

10.2.8 Upgrading CouchDB

If you are using CouchDB as state database, you should upgrade the peer's CouchDB at the same time the peer is being upgraded.

To upgrade CouchDB:

1. Stop CouchDB.
2. Backup CouchDB data directory.
3. Install the latest CouchDB binaries or update deployment scripts to use a new Docker image.
4. Restart CouchDB.

10.2.9 Upgrade Node chaincode shim

To move to the new version of the Node chaincode shim a developer would need to:

1. Change the level of `fabric-shim` in their chaincode `package.json` from their old level to the new one.
2. Repackage this new chaincode package and install it on all the endorsing peers in the channel.
3. Perform an upgrade to this new chaincode. To see how to do this, check out [Peer chaincode commands](#).

10.2.10 Upgrade Chaincodes with vendored shim

For information about upgrading the Go chaincode shim specific to the v2.0 release, check out [Chaincode shim changes](#).

A number of third party tools exist that will allow you to vendor a chaincode shim. If you used one of these tools, use the same one to update your vendored chaincode shim and re-package your chaincode.

If your chaincode vendors the shim, after updating the shim version, you must install it to all peers which already have the chaincode. Install it with the same name, but a newer version. Then you should execute a chaincode upgrade on each channel where this chaincode has been deployed to move to the new version.

10.3 Updating the capability level of a channel

Audience: network administrators, node administrators

Note: this topic describes the process for updating capabilities on a network that has not been bootstrapped with a system channel genesis block. For a version of this topic that includes information about updating system channel capabilities, check out [Updating the capability level of a channel](#).

If you're not familiar with capabilities, check out [Capabilities](#) before proceeding, paying particular attention to the fact that **peers and orderers that belong to the channel must be upgraded before enabling capabilities**.

For information about any new capability levels in the latest release of Fabric, check out [Upgrading your components](#).

Note: when we use the term “upgrade” in Hyperledger Fabric, we're referring to changing the version of a component (for example, going from one version of a binary to the next version). The term “update,” on the other hand, refers not to versions but to configuration changes, such as updating a channel configuration or a deployment script. As there is no data migration, technically speaking, in Fabric, we will not use the term “migration” or “migrate” here.

10.3.1 Prerequisites and considerations

If you haven't already done so, ensure you have all of the dependencies on your machine as described in [Prerequisites](#). This will ensure that you have the latest versions of the tools required to make a channel configuration update.

Although Fabric binaries can and should be upgraded in a rolling fashion, it is important to **finish upgrading binaries before enabling capabilities**. Any binaries which are not upgraded to at least the level of the relevant capabilities will crash to indicate a misconfiguration which could otherwise result in a ledger fork.

Once a capability has been enabled, it becomes part of the permanent record for that channel. This means that even after disabling the capability, old binaries will not be able to participate in the channel because they cannot process beyond the block which enabled the capability to get to the block which disables it. As a result, once a capability has been enabled, disabling it is neither recommended nor supported.

For this reason, think of enabling channel capabilities as a point of no return. Please experiment with the new capabilities in a test setting and be confident before proceeding to enable them in production.

10.3.2 Overview

Whether you will need to update every part of the configuration for all of your channels will depend on the contents of the latest release as well as your own use case. For more information, check out [Upgrading to the latest version of Fabric](#). Note that it may be necessary to update to the newest capability levels before using the features in the latest release, and it is considered a best practice to always be at the latest binary versions and capability levels.

Because updating the capability level of a channel involves the configuration update transaction process, we will be relying on our [Updating a channel configuration](#) topic for many of the commands.

As with any channel configuration update, updating capabilities is, at a high level, a three step process (for each channel):

1. Get the latest channel config
2. Create a modified channel config
3. Create a config update transaction

Recall from the conceptual topic on capabilities that three types of capabilities exist, reflecting the three areas of administration:

- Orderer group
- Channel group
- Application group

While it is possible to edit multiple parts of the configuration of a channel at the same time, in this tutorial we will show how this process is done incrementally. In other words, we will not bundle a change to the `Orderer` group and the `Channel` group into one configuration change. This is because not every release will have both a new `Orderer` group capability and a `Channel` group capability.

Note that in production networks, it will not be possible or desirable for one user to be able to update all of these channels (and parts of configurations) unilaterally. Updating either the `Orderer` or `Channel` groups of a channel configuration, for example, requires the signature of an ordering service organization in addition to peer organizations. Distributed systems require collaborative management.

10.3.3 Create a capabilities config file

Note that this tutorial presumes that a file called `capabilities.json` has been created and includes the capability updates you want to make to the various sections of the config. It also uses `jq` to apply the edits to the modified config

file.

Note that you are not obligated to create a file like `capabilities.json` or to use a tool like `jq`. The modified config can also be edited manually (after it has been pulled, translated, and scoped). Check out this [sample channel configuration](#) for reference.

However, the process described here (using a JSON file and a tool like `jq`) does have the advantage of being scriptable, making it suitable for proposing configuration updates to a large number of channels. This is why it is **the recommended way to update channels**.

In this example, the `capabilities.json` file looks like this (note: if you are updating your channel as part of [Upgrading to the latest version of Fabric](#) you will need to set the capabilities to the levels appropriate to that release):

```
{
  "channel": {
    "mod_policy": "Admins",
    "value": {
      "capabilities": {
        "V2_0": {}
      }
    },
    "version": "0"
  },
  "orderer": {
    "mod_policy": "Admins",
    "value": {
      "capabilities": {
        "V2_0": {}
      }
    },
    "version": "0"
  },
  "application": {
    "mod_policy": "Admins",
    "value": {
      "capabilities": {
        "V2_0": {}
      }
    },
    "version": "0"
  }
}
```

10.3.4 Update capabilities on existing channels

Make sure all of the ordering nodes in your ordering service and peers on the channel have been upgraded to the required binary level before updating capabilities. If a peer or an ordering node is not at the required level, it will be unable to process the config block with the capability and will crash. For more information, check out [Capabilities](#).

Set environment variables

You will need to export the following variables:

- `CH_NAME`: the name of the application channel being updated. You will have to reset this variable for every channel you update.

- `CORE_PEER_LOCALMSPID`: the MSP ID of the organization proposing the channel update. This will be the MSP of your peer organization.
- `TLS_ROOT_CA`: the absolute path to the TLS cert of your peer organization.
- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP representing your organization.
- `ORDERER_CONTAINER`: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.

Orderer group

Navigate to [Step 1: Pull and translate the config](#). Once you have a `modified_config.json`, add the capabilities to the Orderer group of the config (as listed in `capabilities.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"groups":{"Orderer": {"values": {"Capabilities": .[1].
↪orderer}}}}}' config.json ./capabilities.json > modified_config.json
```

Then, follow the steps at [Step 3: Re-encode and submit the config](#).

Note the `mod_policy` for this capability defaults to the MAJORITY of the Admins of the Orderer group (in other words, a majority of the admins of the ordering service). Peer organizations can propose an update to this capability, but their signatures will not satisfy the default policy in this case.

Channel group

Navigate to [Step 1: Pull and translate the config](#). Once you have a `modified_config.json`, add the capabilities to the Channel group of the config (as listed in `capabilities.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"values": {"Capabilities": .[1].channel}}}' config.
↪json ./capabilities.json > modified_config.json
```

Then, follow the steps at [Step 3: Re-encode and submit the config](#).

Note that the `mod_policy` for this capability defaults to requiring signatures from both the MAJORITY of Admins in the Application and Orderer groups. In other words, both a majority of the peer organization admins and ordering service organization admins must sign this request.

Application group

Navigate to [Step 1: Pull and translate the config](#). Once you have a `modified_config.json`, add the capabilities to the Application group of the config (as listed in `capabilities.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"groups":{"Application": {"values": {"Capabilities": .
↪[1].application}}}}}' config.json ./capabilities.json > modified_config.json
```

Then, follow the steps at [Step 3: Re-encode and submit the config](#).

Note that the `mod_policy` for this capability defaults to requiring signatures from the MAJORITY of Admins in the Application group. In other words, a majority of peer organizations will need to approve. Ordering service admins have no say in this capability.

As a result, be very careful to not change this capability to a level that does not exist. Because ordering nodes neither understand nor validate Application capabilities, they will approve a configuration to any level and send the new config block to the peers to be committed to their ledgers. However, the peers will be unable to process the capability and will crash. And even it was possible to drive a corrected configuration change to a valid capability level,

the previous config block with the faulty capability would still exist on the ledger and cause peers to crash when trying to process it.

This is one reason why a file like `capabilities.json` can be useful. It prevents a simple user error — for example, setting the `Application` capability to `V20` when the intent was to set it to `V2_0` — that can cause a channel to be unusable and unrecoverable.

10.3.5 Verify a transaction after capabilities have been enabled

It's a best practice to ensure that capabilities have been enabled successfully with a chaincode invoke on all channels. If any nodes that do not understand new capabilities have not been upgraded to a sufficient binary level, they will crash. You will have to upgrade their binary level before they can be successfully restarted.

10.4 Enabling the new chaincode lifecycle

Users upgrading from `v1.4.x` to `v2.x` will have to edit their channel configurations to enable the new lifecycle features. This process involves a series of [channel configuration updates](#) the relevant users will have to perform.

Note that the `Channel` and `Application` [capabilities](#) of your application channels will have to be updated to `V2_0` for the new chaincode lifecycle to work. Check out [Considerations for getting to 2.0](#) for more information.

Updating a channel configuration is, at a high level, a three step process (for each channel):

1. Get the latest channel config
2. Create a modified channel config
3. Create a config update transaction

We will be performing these channel configuration updates by leveraging a file called `enable_lifecycle.json`, which contains all of the updates we will be making in the channel configurations. Note that in a production setting it is likely that multiple users would be making these channel update requests. However, for the sake of simplicity, we are presenting all of the updates as how they would appear in a single file.

Note: this topic describes a network that does not use a “system channel”, a channel that the ordering service is bootstrapped with and the ordering service exclusively controls. Since the release of `v2.3`, using a system channel is now considered the legacy process as compared to the new process to [Create a channel](#) without a system channel. For a version of this topic that includes information about the system channel, check out [Enabling the new chaincode lifecycle](#) from the `v2.2` documentation.

10.4.1 Create `enable_lifecycle.json`

Note that in addition to using `enable_lifecycle.json`, this tutorial also uses `jq` to apply the edits to the modified config file. The modified config can also be edited manually (after it has been pulled, translated, and scoped). Check out this [sample channel configuration](#) for reference.

However, the process described here (using a JSON file and a tool like `jq`) does have the advantage of being scriptable, making it suitable for proposing configuration updates to a large number of channels, and is the recommended process for editing a channel configuration.

Note that the `enable_lifecycle.json` uses sample values, for example `org1Policies` and the `Org1ExampleCom`, which will be specific to your deployment):

```

{
  "org1Policies": {
    "Endorsement": {
      "mod_policy": "Admins",
      "policy": {
        "type": 1,
        "value": {
          "identities": [
            {
              "principal": {
                "msp_identifier": "Org1ExampleCom",
                "role": "PEER"
              },
              "principal_classification": "ROLE"
            }
          ],
          "rule": {
            "n_out_of": {
              "n": 1,
              "rules": [
                {
                  "signed_by": 0
                }
              ]
            }
          }
        },
        "version": 0
      },
      "version": "0"
    }
  },
  "org2Policies": {
    "Endorsement": {
      "mod_policy": "Admins",
      "policy": {
        "type": 1,
        "value": {
          "identities": [
            {
              "principal": {
                "msp_identifier": "Org2ExampleCom",
                "role": "PEER"
              },
              "principal_classification": "ROLE"
            }
          ],
          "rule": {
            "n_out_of": {
              "n": 1,
              "rules": [
                {
                  "signed_by": 0
                }
              ]
            }
          }
        },
        "version": 0
      },
      "version": "0"
    }
  }
}

```

(continues on next page)

(continued from previous page)

```

        "version": 0
      }
    },
    "version": "0"
  }
},
"appPolicies": {
  "Endorsement": {
    "mod_policy": "Admins",
    "policy": {
      "type": 3,
      "value": {
        "rule": "MAJORITY",
        "sub_policy": "Endorsement"
      }
    }
  },
  "version": "0"
},
"LifecycleEndorsement": {
  "mod_policy": "Admins",
  "policy": {
    "type": 3,
    "value": {
      "rule": "MAJORITY",
      "sub_policy": "Endorsement"
    }
  },
  "version": "0"
},
"acls": {
  "_lifecycle/CheckCommitReadiness": {
    "policy_ref": "/Channel/Application/Writers"
  },
  "_lifecycle/CommitChaincodeDefinition": {
    "policy_ref": "/Channel/Application/Writers"
  },
  "_lifecycle/QueryChaincodeDefinition": {
    "policy_ref": "/Channel/Application/Writers"
  },
  "_lifecycle/QueryChaincodeDefinitions": {
    "policy_ref": "/Channel/Application/Writers"
  }
}
}

```

Note: the “role” field of these new policies should say 'PEER' if [NodeOUs](#) are enabled for the org, and 'MEMBER' if they are not.

10.4.2 Edit the channel configurations

To fully enable the new chaincode lifecycle, you must first edit the configuration of your own organization as it exists in a channel configuration, and then you must update the channel itself to include a default endorsement policy for the channel. You can then optionally update your channel access control list.

Note: this topic leverages the instructions on how to update a channel configuration that are found in the [Updating](#)

a [channel configuration](#) tutorial. The environment variables listed here work in conjunction with those commands to update your channels.

Edit the peer organizations

By default, peer organizations are able to make configuration update requests to their own organization on an application channel without needing the approval of any other peer organizations. However, if you are attempting to make a change to a different organization, that organization will have to approve the change.

You will need to export the following variables:

- `CH_NAME`: the name of the application channel being updated.
- `ORGNAME`: The name of the organization you are currently updating.
- `TLS_ROOT_CA`: the absolute path to the TLS cert of your ordering node.
- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP representing your organization.
- `CORE_PEER_LOCALMSPID`: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the peer organizations.
- `ORDERER_CONTAINER`: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.

Once you have set the environment variables, navigate to [Step 1: Pull and translate the config](#).

Then, add the lifecycle organization policy (as listed in `enable_lifecycle.json`) to a file called `modified_config.json` using this command:

```
jq -s ".[0] * {\n\"channel_group\":{\n\"groups\":{\n\"Application\": {\n\"groups\": {\n\n↪$ORGNAME\": {\n\"policies\": .[1].${ORGNAME}Policies}}}}}\n\" config.json ./enable_\n↪lifecycle.json > modified_config.json"
```

Then, follow the steps at [Step 3: Re-encode and submit the config](#).

Edit the application channels

After all of the application channels have been [updated to include V2_0 capabilities](#), endorsement policies for the new chaincode lifecycle must be added to each channel.

You can set the same environment variables you set when updating the peer organizations. Note that in this case you will not be updating the configuration of an org in the configuration, so the `ORGNAME` variable will not be used.

Once you have set the environment variables, navigate to [Step 1: Pull and translate the config](#).

Then, add the lifecycle organization policy (as listed in `enable_lifecycle.json`) to a file called `modified_config.json` using this command:

```
jq -s '.[0] * {\n\"channel_group\":{\n\"groups\":{\n\"Application\": {\n\"policies\": .[1].\n↪appPolicies}}}}' config.json ./enable_lifecycle.json > modified_config.json"
```

Then, follow the steps at [Step 3: Re-encode and submit the config](#).

For this channel update to be approved, the policy for modifying the Channel/Application section of the configuration must be satisfied. By default, this is a MAJORITY of the peer organizations on the channel.

Edit channel ACLs (optional)

The following [Access Control List \(ACL\)](#) in `enable_lifecycle.json` are the default values for the new lifecycle, though you have the option to change them depending on your use case.

```
"acls": {
  "_lifecycle/CheckCommitReadiness": {
    "policy_ref": "/Channel/Application/Writers"
  },
  "_lifecycle/CommitChaincodeDefinition": {
    "policy_ref": "/Channel/Application/Writers"
  },
  "_lifecycle/QueryChaincodeDefinition": {
    "policy_ref": "/Channel/Application/Readers"
  },
  "_lifecycle/QueryChaincodeDefinitions": {
    "policy_ref": "/Channel/Application/Readers"
  }
}
```

You can leave the same environment in place as when you previously edited application channels.

Once you have the environment variables set, navigate to [Step 1: Pull and translate the config](#).

Then, add the ACLs (as listed in `enable_lifecycle.json`) and create a file called `modified_config.json` using this command:

```
jq -s '[0] * {"channel_group":{"groups":{"Application": {"values": {"ACLs": {"value
↪": {"acls": [1].acls}}}}}}}' config.json ./enable_lifecycle.json > modified_config.
↪json
```

Then, follow the steps at [Step 3: Re-encode and submit the config](#).

For this channel update to be approved, the policy for modifying the `Channel/Application` section of the configuration must be satisfied. By default, this is a MAJORITY of the peer organizations on the channel.

10.4.3 Enable new lifecycle in `core.yaml`

If you follow [the recommended process](#) for using a tool like `diff` to compare the new version of `core.yaml` packaged with the binaries with your old one, you will not need to add `_lifecycle: enable` to the list of enabled system chaincodes because the new `core.yaml` has added it under `chaincode/system`.

However, if you are updating your old node YAML file directly, you will have to add `_lifecycle: enable` to the list of enabled system chaincodes.

For more information about upgrading nodes, check out [Upgrading your components](#).

11.1 peer

11.1.1 Description

The `peer` command has five different subcommands, each of which allows administrators to perform a specific set of tasks related to a peer. For example, you can use the `peer channel` subcommand to join a peer to a channel, or the `peer chaincode` command to deploy a smart contract chaincode to a peer.

11.1.2 Syntax

The `peer` command has five different subcommands within it:

```
peer chaincode [option] [flags]
peer channel   [option] [flags]
peer node      [option] [flags]
peer version   [option] [flags]
```

Each subcommand has different options available, and these are described in their own dedicated topic. For brevity, we often refer to a command (`peer`), a subcommand (`channel`), or subcommand option (`fetch`) simply as a **command**.

If a subcommand is specified without an option, then it will return some high level help text as described in the `--help` flag below.

11.1.3 Flags

Each `peer` subcommand has a specific set of flags associated with it, many of which are designated *global* because they can be used in all subcommand options. These flags are described with the relevant `peer` subcommand.

The top level `peer` command has the following flag:

- `--help`

Use `--help` to get brief help text for any `peer` command. The `--help` flag is very useful – it can be used to get command help, subcommand help, and even option help.

For example

```
peer --help
peer channel --help
peer channel list --help
```

See individual `peer` subcommands for more detail.

11.1.4 Usage

Here is an example using the available flag on the `peer` command.

- Using the `--help` flag on the `peer channel join` command.

```
peer channel join --help

Joins the peer to a channel.

Usage:
  peer channel join [flags]

Flags:
  -b, --blockpath string    Path to file containing genesis block
  -h, --help                help for join

Global Flags:
  --cafile string                Path to file containing PEM-encoded
  trusted certificate(s) for the ordering endpoint
  --certfile string             Path to file containing PEM-encoded
  X509 public key to use for mutual TLS communication with the orderer endpoint
  --clientauth                  Use mutual TLS when communicating
  with the orderer endpoint
  --connTimeout duration        Timeout for client to connect
  (default 3s)
  --keyfile string              Path to file containing PEM-encoded
  private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string          Ordering service endpoint
  --ordererTLSHostnameOverride string The hostname override to use when
  validating the TLS connection to the orderer.
  --tls                          Use TLS when communicating with the
  orderer endpoint
```

This shows brief help syntax for the `peer channel join` command.

11.2 peer chaincode

The `peer chaincode` command allows administrators to perform chaincode related operations on a peer, such as installing, instantiating, invoking, packaging, querying, and upgrading chaincode.

11.2.1 Syntax

The `peer chaincode` command has the following subcommands:

- `install`
- `instantiate`
- `invoke`
- `list`
- `package`
- `query`
- `signpackage`
- `upgrade`

The different subcommand options (`install`, `instantiate`...) relate to the different chaincode operations that are relevant to a peer. For example, use the `peer chaincode install` subcommand option to install a chaincode on a peer, or the `peer chaincode query` subcommand option to query a chaincode for the current value on a peer's ledger.

Some subcommands take flag `--ctor`, of which the value must be a JSON string that has either key 'Args' or 'Function' and 'Args'. These keys are case-insensitive.

If the JSON string only has the Args key, the key value is an array, where the first array element is the target function to call, and the subsequent elements are arguments of the function. If the JSON string has both 'Function' and 'Args', the value of Function is the target function to call, and the value of Args is an array of arguments of the function. For instance, `{"Args": ["GetAllAssets"]}` is equivalent to `{"Function": "GetAllAssets", "Args": []}`.

Each `peer chaincode` subcommand is described together with its options in its own section in this topic.

11.2.2 Flags

Each `peer chaincode` subcommand has both a set of flags specific to an individual subcommand, as well as a set of global flags that relate to all `peer chaincode` subcommands. Not all subcommands would use these flags. For instance, the `query` subcommand does not need the `--orderer` flag.

The individual flags are described with the relevant subcommand. The global flags are

- `--cafile <string>`
Path to file containing PEM-encoded trusted certificate(s) for the ordering endpoint
- `--certfile <string>`
Path to file containing PEM-encoded X509 public key to use for mutual TLS communication with the orderer endpoint
- `--keyfile <string>`
Path to file containing PEM-encoded private key to use for mutual TLS communication with the orderer endpoint
- `-o` or `--orderer <string>`
Ordering service endpoint specified as `<hostname or IP address>:<port>`
- `--ordererTLSHostnameOverride <string>`
The hostname override to use when validating the TLS connection to the orderer

- `--tls`

Use TLS when communicating with the orderer endpoint

- `--transient <string>`

Transient map of arguments in JSON encoding

11.2.3 peer chaincode install

Install a chaincode on a peer. This installs a chaincode deployment spec package (**if** `--provided`) **or** packages the specified chaincode before subsequently installing it.

Usage:

```
peer chaincode install [flags]
```

Flags:

```
--connectionProfile string      Connection profile that provides the necessary
↳ connection information for the network. Note: currently only supported for
↳ providing peer connection information
-c, --ctor string              Constructor message for the chaincode in JSON
↳ format (default "{}")
-h, --help                    help for install
-l, --lang string              Language the chaincode is written in (default
↳ "golang")
-n, --name string              Name of the chaincode
-p, --path string              Path to chaincode
--peerAddresses stringArray     The addresses of the peers to connect to
--tlsRootCertFiles stringArray  If TLS is enabled, the paths to the TLS root
↳ cert files of the peers to connect to. The order and number of certs specified
↳ should match the --peerAddresses flag
-v, --version string           Version of the chaincode specified in install/
↳ instantiate/upgrade commands
```

Global Flags:

```
--cafile string                Path to file containing PEM-encoded
↳ trusted certificate(s) for the ordering endpoint
--certfile string              Path to file containing PEM-encoded X509
↳ public key to use for mutual TLS communication with the orderer endpoint
--clientauth                   Use mutual TLS when communicating with
↳ the orderer endpoint
--connTimeout duration         Timeout for client to connect (default 3s)
--keyfile string               Path to file containing PEM-encoded
↳ private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string            Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↳ validating the TLS connection to the orderer
--tls                          Use TLS when communicating with the
↳ orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↳ certificate expiration checks during TLS handshakes with the orderer endpoint
--transient string             Transient map of arguments in JSON
↳ encoding
```


11.2.4 peer chaincode instantiate

Deploy the specified chaincode to the network.

Usage:

```
peer chaincode instantiate [flags]
```

Flags:

```
-C, --channelID string      The channel on which this command should be
↳executed
--collections-config string The fully qualified path to the collection
↳JSON file including the file name
--connectionProfile string  Connection profile that provides the necessary
↳connection information for the network. Note: currently only supported for
↳providing peer connection information
-C, --ctor string          Constructor message for the chaincode in JSON
↳format (default "{}")
-E, --escc string          The name of the endorsement system chaincode
↳to be used for this chaincode
-h, --help                help for instantiate
-l, --lang string          Language the chaincode is written in (default
↳"golang")
-n, --name string          Name of the chaincode
--peerAddresses stringArray The addresses of the peers to connect to
-P, --policy string        The endorsement policy associated to this
↳chaincode
--tlsRootCertFiles stringArray If TLS is enabled, the paths to the TLS root
↳cert files of the peers to connect to. The order and number of certs specified
↳should match the --peerAddresses flag
-v, --version string       Version of the chaincode specified in install/
↳instantiate/upgrade commands
-V, --vscc string          The name of the verification system chaincode
↳to be used for this chaincode
```

Global Flags:

```
--cafile string            Path to file containing PEM-encoded
↳trusted certificate(s) for the ordering endpoint
--certfile string          Path to file containing PEM-encoded X509
↳public key to use for mutual TLS communication with the orderer endpoint
--clientauth               Use mutual TLS when communicating with
↳the orderer endpoint
--connTimeout duration     Timeout for client to connect (default 3s)
--keyfile string           Path to file containing PEM-encoded
↳private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string       Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↳validating the TLS connection to the orderer
--tls                     Use TLS when communicating with the
↳orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↳certificate expiration checks during TLS handshakes with the orderer endpoint
--transient string         Transient map of arguments in JSON
↳encoding
```

11.2.5 peer chaincode invoke

Invoke the specified chaincode. It will **try** to commit the endorsed transaction to the **network**.

Usage:

```
peer chaincode invoke [flags]
```

Flags:

```
-C, --channelID string      The channel on which this command should be
↳executed
--connectionProfile string  Connection profile that provides the necessary
↳connection information for the network. Note: currently only supported for
↳providing peer connection information
-c, --ctor string          Constructor message for the chaincode in JSON
↳format (default "{}")
-h, --help                 help for invoke
-I, --isInit               Is this invocation for init (useful for
↳supporting legacy chaincodes in the new lifecycle)
-n, --name string          Name of the chaincode
--peerAddresses stringArray The addresses of the peers to connect to
--tlsRootCertFiles stringArray If TLS is enabled, the paths to the TLS root
↳cert files of the peers to connect to. The order and number of certs specified
↳should match the --peerAddresses flag
--waitForEvent             Whether to wait for the event from each peer's
↳deliver filtered service signifying that the 'invoke' transaction has been
↳committed successfully
--waitForEventTimeout duration Time to wait for the event from each peer's
↳deliver filtered service signifying that the 'invoke' transaction has been
↳committed successfully (default 30s)
```

Global Flags:

```
--cafile string            Path to file containing PEM-encoded
↳trusted certificate(s) for the ordering endpoint
--certfile string          Path to file containing PEM-encoded X509
↳public key to use for mutual TLS communication with the orderer endpoint
--clientauth               Use mutual TLS when communicating with
↳the orderer endpoint
--connTimeout duration     Timeout for client to connect (default 3s)
--keyfile string           Path to file containing PEM-encoded
↳private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string        Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↳validating the TLS connection to the orderer
--tls                      Use TLS when communicating with the
↳orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↳certificate expiration checks during TLS handshakes with the orderer endpoint
--transient string         Transient map of arguments in JSON
↳encoding
```

11.2.6 peer chaincode list

Get the instantiated chaincodes **in** the channel **if** specify channel, **or** get installed **chaincodes** on the peer

(continues on next page)

(continued from previous page)

```

Usage:
  peer chaincode list [flags]

Flags:
  -C, --channelID string          The channel on which this command should be
    ↪executed
  --connectionProfile string      Connection profile that provides the necessary
    ↪connection information for the network. Note: currently only supported for
    ↪providing peer connection information
  -h, --help                      help for list
  --installed                     Get the installed chaincodes on a peer
  --instantiated                 Get the instantiated chaincodes on a channel
  --peerAddresses stringArray     The addresses of the peers to connect to
  --tlsRootCertFiles stringArray  If TLS is enabled, the paths to the TLS root
    ↪cert files of the peers to connect to. The order and number of certs specified
    ↪should match the --peerAddresses flag

Global Flags:
  --cafile string                Path to file containing PEM-encoded
    ↪trusted certificate(s) for the ordering endpoint
  --certfile string              Path to file containing PEM-encoded X509
    ↪public key to use for mutual TLS communication with the orderer endpoint
  --clientauth                   Use mutual TLS when communicating with
    ↪the orderer endpoint
  --connTimeout duration         Timeout for client to connect (default 3s)
  --keyfile string               Path to file containing PEM-encoded
    ↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string           Ordering service endpoint
  --ordererTLSHostnameOverride string The hostname override to use when
    ↪validating the TLS connection to the orderer
  --tls                          Use TLS when communicating with the
    ↪orderer endpoint
  --tlsHandshakeTimeShift duration The amount of time to shift backwards for
    ↪certificate expiration checks during TLS handshakes with the orderer endpoint
  --transient string             Transient map of arguments in JSON
    ↪encoding

```

11.2.7 peer chaincode package

Package a chaincode and write the package to a file.

```

Usage:
  peer chaincode package [outputfile] [flags]

Flags:
  -s, --cc-package              create CC deployment spec for owner endorsements
    ↪instead of raw CC deployment spec
  -c, --ctor string             Constructor message for the chaincode in JSON
    ↪format (default "{}")
  -h, --help                    help for package
  -i, --instantiate-policy string instantiation policy for the chaincode
  -l, --lang string             Language the chaincode is written in (default
    ↪"golang")
  -n, --name string             Name of the chaincode

```

(continues on next page)

(continued from previous page)

```

-p, --path string          Path to chaincode
-S, --sign                if creating CC deployment spec package for owner
↳endorsements, also sign it with local MSP
-v, --version string      Version of the chaincode specified in install/
↳instantiate/upgrade commands

Global Flags:
--cafile string          Path to file containing PEM-encoded
↳trusted certificate(s) for the ordering endpoint
--certfile string        Path to file containing PEM-encoded X509
↳public key to use for mutual TLS communication with the orderer endpoint
--clientauth            Use mutual TLS when communicating with
↳the orderer endpoint
--connTimeout duration   Timeout for client to connect (default 3s)
--keyfile string         Path to file containing PEM-encoded
↳private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string     Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↳validating the TLS connection to the orderer
--tls                  Use TLS when communicating with the
↳orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↳certificate expiration checks during TLS handshakes with the orderer endpoint
--transient string      Transient map of arguments in JSON
↳encoding

```

11.2.8 peer chaincode query

Get endorsed result of chaincode function call and print it. It won't generate
↳transaction.

Usage:

```
peer chaincode query [flags]
```

Flags:

```

-C, --channelID string    The channel on which this command should be
↳executed
--connectionProfile string Connection profile that provides the necessary
↳connection information for the network. Note: currently only supported for
↳providing peer connection information
-c, --ctor string         Constructor message for the chaincode in JSON
↳format (default "{}")
-h, --help               help for query
-x, --hex                If true, output the query value byte array in
↳hexadecimal. Incompatible with --raw
-n, --name string        Name of the chaincode
--peerAddresses stringArray The addresses of the peers to connect to
-r, --raw                If true, output the query value as raw bytes,
↳otherwise format as a printable string
--tlsRootCertFiles stringArray If TLS is enabled, the paths to the TLS root
↳cert files of the peers to connect to. The order and number of certs specified
↳should match the --peerAddresses flag

```

Global Flags:

```

--cafile string          Path to file containing PEM-encoded
↳trusted certificate(s) for the ordering endpoint

```

(continues on next page)

(continued from previous page)

```

--certfile string                Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth                     Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration           Timeout for client to connect (default 3s)
--keyfile string                Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string             Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                           Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint
--transient string              Transient map of arguments in JSON
↪encoding

```

11.2.9 peer chaincode signpackage

Sign the specified chaincode package

Usage:

```
peer chaincode signpackage [flags]
```

Flags:

```
-h, --help    help for signpackage
```

Global Flags:

```

--cafile string                Path to file containing PEM-encoded
↪trusted certificate(s) for the ordering endpoint
--certfile string             Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth                  Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration        Timeout for client to connect (default 3s)
--keyfile string              Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string           Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                          Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint
--transient string            Transient map of arguments in JSON
↪encoding

```

11.2.10 peer chaincode upgrade

Upgrade an existing chaincode with the specified one. The new chaincode will
↪immediately replace the existing chaincode upon the transaction committed.

(continues on next page)

(continued from previous page)

```

Usage:
  peer chaincode upgrade [flags]

Flags:
  -C, --channelID string          The channel on which this command should be
    ↪executed
  --collections-config string      The fully qualified path to the collection
    ↪JSON file including the file name
  --connectionProfile string       Connection profile that provides the necessary
    ↪connection information for the network. Note: currently only supported for
    ↪providing peer connection information
  -C, --ctor string               Constructor message for the chaincode in JSON
    ↪format (default "{}")
  -E, --escc string               The name of the endorsement system chaincode
    ↪to be used for this chaincode
  -h, --help                      help for upgrade
  -l, --lang string               Language the chaincode is written in (default
    ↪"golang")
  -n, --name string               Name of the chaincode
  -p, --path string               Path to chaincode
  --peerAddresses stringArray      The addresses of the peers to connect to
  -P, --policy string             The endorsement policy associated to this
    ↪chaincode
  --tlsRootCertFiles stringArray  If TLS is enabled, the paths to the TLS root
    ↪cert files of the peers to connect to. The order and number of certs specified
    ↪should match the --peerAddresses flag
  -v, --version string            Version of the chaincode specified in install/
    ↪instantiate/upgrade commands
  -V, --vscc string               The name of the verification system chaincode
    ↪to be used for this chaincode

Global Flags:
  --cafile string                 Path to file containing PEM-encoded
    ↪trusted certificate(s) for the ordering endpoint
  --certfile string               Path to file containing PEM-encoded X509
    ↪public key to use for mutual TLS communication with the orderer endpoint
  --clientauth                    Use mutual TLS when communicating with
    ↪the orderer endpoint
  --connTimeout duration          Timeout for client to connect (default 3s)
  --keyfile string                Path to file containing PEM-encoded
    ↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string            Ordering service endpoint
  --ordererTLSHostnameOverride string The hostname override to use when
    ↪validating the TLS connection to the orderer
  --tls                           Use TLS when communicating with the
    ↪orderer endpoint
  --tlsHandshakeTimeShift duration The amount of time to shift backwards for
    ↪certificate expiration checks during TLS handshakes with the orderer endpoint
  --transient string              Transient map of arguments in JSON
    ↪encoding

```

11.2.11 Example Usage

peer chaincode instantiate examples

Here are some examples of the `peer chaincode instantiate` command, which instantiates the chaincode named `mycc` at version `1.0` on channel `mychannel`:

- Using the `--tls` and `--cafile` global flags to instantiate the chaincode in a network with TLS enabled:

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem
peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile $ORDERER_CA
↪-C mychannel -n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "AND (
↪'Org1MSP.peer','Org2MSP.peer')"
```

```
2018-02-22 16:33:53.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001
↪Using default escc
2018-02-22 16:33:53.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002
↪Using default vscc
2018-02-22 16:34:08.698 UTC [main] main -> INFO 003 Exiting.....
```

- Using only the command-specific options to instantiate the chaincode in a network with TLS disabled:

```
peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mycc -v 1.
↪0 -c '{"Args":["init","a","100","b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.
↪peer')"
```

```
2018-02-22 16:34:09.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001
↪Using default escc
2018-02-22 16:34:09.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002
↪Using default vscc
2018-02-22 16:34:24.698 UTC [main] main -> INFO 003 Exiting.....
```

peer chaincode invoke example

Here is an example of the `peer chaincode invoke` command:

- Invoke the chaincode named `mycc` at version `1.0` on channel `mychannel` on `peer0.org1.example.com:7051` and `peer0.org2.example.com:9051` (the peers defined by `--peerAddresses`), request to move 10 units from variable `a` to variable `b`:

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n mycc --
↪peerAddresses peer0.org1.example.com:7051 --peerAddresses peer0.org2.example.
↪com:9051 -c '{"Args":["invoke","a","b","10"]}'
```

```
2018-02-22 16:34:27.069 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001
↪Using default escc
2018-02-22 16:34:27.069 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002
↪Using default vscc
.
.
.
2018-02-22 16:34:27.106 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 00a
↪ESCC invoke result: version:1 response:<status:200 message:"OK" > payload:"\n
↪\237mM\376? [\214\002 \332\204\035\275q\227\2132A\n\204&\2106\037W\346
↪#\3413\274\022Y\nE\022\024\n\004lscc\022\014\n\n\n\004mycc\022\002\010\003\022-
↪\n\004mycc\022
↪%\n\007\n\001a\022\002\010\003\n\007\n\001b\022\002\010\003\032\007\n\001\n\033\00290\032\010\n
↪"\013\022\004mycc\032\0031.0" endorsement:<endorser:"\n\007Org1MSP\022\262\006--
↪---BEGIN CERTIFICATE-----\nMIICLjCCAdWgAwIBAgIRAJYomxY2cqHA/fbRnH5a/
↪j0EAwiwczELnMAkGA1UEBhMCVVMxEzARBgNVBAGTCkNhbgGlm3JuaWExFjAUBgNV527TDVNHbiBG\n
↪/7JFDHATJXtLgJhkK5KosDdHuKLYbCqvge\n46u3AC16MZyJRvKBiw6jTTBLMA4GA1UdDwEB/
↪wQEAWIHgDAMBgNVHRMBAf8EAjAA\nMCsGA1UdIwQkMCKAIN7dJR9dimkFtkusOR5pAO1Rz5SA3FB5t8Eaxl9A71kgMAoG\
↪Xj3C81A==\n-----END CERTIFICATE-----\n" signature:"0D\002 \022_
↪\342\350\344\231G6
```

11.2. peer chaincode

```
↪/7JFDHATJXtLgJhkK5KosDdHuKLYbCqvge\n46u3AC16MZyJRvKBiw6jTTBLMA4GA1UdDwEB/
↪wQEAWIHgDAMBgNVHRMBAf8EAjAA\nMCsGA1UdIwQkMCKAIN7dJR9dimkFtkusOR5pAO1Rz5SA3FB5t8Eaxl9A71kgMAoG\
↪Xj3C81A==\n-----END CERTIFICATE-----\n" signature:"0D\002 \022_
↪\342\350\344\231G6
```

(continued from previous page)

```
2018-02-22 16:34:27.107 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00b_
↳Chaincode invoke successful. result: status:200
2018-02-22 16:34:27.107 UTC [main] main -> INFO 00c Exiting.....
```

Here you can see that the invoke was submitted successfully based on the log message:

```
2018-02-22 16:34:27.107 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00b_
↳Chaincode invoke successful. result: status:200
```

A successful response indicates that the transaction was submitted for ordering successfully. The transaction will then be added to a block and, finally, validated or invalidated by each peer on the channel.

Here is an example of how to format the `peer chaincode invoke` command when the chaincode package includes multiple smart contracts.

- If you are using the `contract-api`, the name you pass to `super("MyContract")` can be used as a prefix.

```
peer chaincode invoke -C $CHANNEL_NAME -n $CHAINCODE_NAME -c '{ "Args": [
↳"MyContract:methodName", "{}"] }'

peer chaincode invoke -C $CHANNEL_NAME -n $CHAINCODE_NAME -c '{ "Args": [
↳"MyOtherContract:methodName", "{}"] }'
```

peer chaincode list example

Here are some examples of the `peer chaincode list` command:

- Using the `--installed` flag to list the chaincodes installed on a peer.

```
peer chaincode list --installed

Get installed chaincodes on peer:
Name: mycc, Version: 1.0, Path: github.com/hyperledger/fabric-samples/chaincode/
↳abstore/go, Id: 8cc2730fdafd0b28ef734eac12b29df5fc98ad98bdb1b7e0ef96265c3d893d61
2018-02-22 17:07:13.476 UTC [main] main -> INFO 001 Exiting.....
```

You can see that the peer has installed a chaincode called `mycc` which is at version `1.0`.

- Using the `--instantiated` in combination with the `-C` (channel ID) flag to list the chaincodes instantiated on a channel.

```
peer chaincode list --instantiated -C mychannel

Get instantiated chaincodes on channel mychannel:
Name: mycc, Version: 1.0, Path: github.com/hyperledger/fabric-samples/chaincode/
↳abstore/go, Escc: escc, Vscc: vscc
2018-02-22 17:07:42.969 UTC [main] main -> INFO 001 Exiting.....
```

You can see that chaincode `mycc` at version `1.0` is instantiated on channel `mychannel`.

peer chaincode package example

Here is an example of the `peer chaincode package` command, which packages the chaincode named `mycc` at version `1.1`, creates the chaincode deployment spec, signs the package using the local MSP, and outputs it as `ccpack.out`:


```
peer chaincode package ccpack.out -n mycc -p github.com/hyperledger/fabric-samples/
↳chaincode/abstore/go -v 1.1 -s -S
.
.
.
2018-02-22 17:27:01.404 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003_
↳Using default escc
2018-02-22 17:27:01.405 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004_
↳Using default vscc
.
.
.
2018-02-22 17:27:01.879 UTC [chaincodeCmd] chaincodePackage -> DEBU 011 Packaged_
↳chaincode into deployment spec of size <3426>, with args = [ccpack.out]
2018-02-22 17:27:01.879 UTC [main] main -> INFO 012 Exiting.....
```

peer chaincode query example

Here is an example of the `peer chaincode query` command, which queries the peer ledger for the chaincode named `mycc` at version `1.0` for the value of variable `a`:

- You can see from the output that variable `a` had a value of `90` at the time of the query.

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
2018-02-22 16:34:30.816 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001_
↳Using default escc
2018-02-22 16:34:30.816 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002_
↳Using default vscc
Query Result: 90
```

peer chaincode signpackage example

Here is an example of the `peer chaincode signpackage` command, which accepts an existing signed package and creates a new one with signature of the local MSP appended to it.

```
peer chaincode signpackage ccwith1sig.pak ccwith2sig.pak
Wrote signed package to ccwith2sig.pak successfully
2018-02-24 19:32:47.189 EST [main] main -> INFO 002 Exiting.....
```

peer chaincode upgrade example

Here is an example of the `peer chaincode upgrade` command, which upgrades the chaincode named `mycc` at version `1.1` on channel `mychannel` to version `1.2`, which contains a new variable `c`:

- Using the `--tls` and `--cafile` global flags to upgrade the chaincode in a network with TLS enabled:

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↳ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↳tlsca.example.com-cert.pem
peer chaincode upgrade -o orderer.example.com:7050 --tls --cafile $ORDERER_CA -C_
↳mychannel -n mycc -v 1.2 -c '{"Args":["init","a","100","b","200","c","300"]}' -
↳P "AND ('Org1MSP.peer','Org2MSP.peer')"
```

(continues on next page)

(continued from previous page)

```

.
.
.
2018-02-22 18:26:31.433 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003_
↳Using default escc
2018-02-22 18:26:31.434 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004_
↳Using default vscc
2018-02-22 18:26:31.435 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java_
↳chaincode enabled
2018-02-22 18:26:31.435 UTC [chaincodeCmd] upgrade -> DEBU 006 Get upgrade_
↳proposal for chaincode <name:"mycc" version:"1.1" >
.
.
.
2018-02-22 18:26:46.687 UTC [chaincodeCmd] upgrade -> DEBU 009 endorse upgrade_
↳proposal, get response <status:200 message:"OK" payload:"\n\004mycc\022\0031.
↳1\032\004escc"\004vscc*,
↳\022\014\022\n\010\001\022\002\010\000\022\002\010\001\032\r\022\013\n\007Org1MSP\020\003\032\
↳\261g(^
↳v\021\220\240\332\251\014\204V\210P\310o\231\271\036\301\022\032\205fc[|= \215\372\223\022_
↳\311b\025?
↳\323N\343\325\032\005\365\236\001XKj\004E\351\007\247\265fu\305j\367\331\275\253\307R\032_
↳\014H#\014\272!\#\345\306s\323\371\350\364\006.
↳\000\356\230\353\270\263\215\217\303\256\220i^\277\305\214: \375\200zY\275\203}
↳\375\244\205\035\340\226]!uE\334\273\214\214\020\303\3474\360\014\234-
↳\006\315B\031\022\010\022\006\010\001\022\002\010\000\032\r\022\013\n\007Org1MSP\020\001
↳" >
.
.
.
2018-02-22 18:26:46.693 UTC [chaincodeCmd] upgrade -> DEBU 00c Get Signed envelope
2018-02-22 18:26:46.693 UTC [chaincodeCmd] chaincodeUpgrade -> DEBU 00d Send_
↳signed envelope to orderer
2018-02-22 18:26:46.908 UTC [main] main -> INFO 00e Exiting.....

```

- Using only the command-specific options to upgrade the chaincode in a network with TLS disabled:

```

peer chaincode upgrade -o orderer.example.com:7050 -C mychannel -n mycc -v 1.2 -c
↳ '{"Args":["init","a","100","b","200","c","300"]}' -P "AND ('Org1MSP.peer',
↳ 'Org2MSP.peer') "

```

```

.
.
.
2018-02-22 18:28:31.433 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003_
↳Using default escc
2018-02-22 18:28:31.434 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004_
↳Using default vscc
2018-02-22 18:28:31.435 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java_
↳chaincode enabled
2018-02-22 18:28:31.435 UTC [chaincodeCmd] upgrade -> DEBU 006 Get upgrade_
↳proposal for chaincode <name:"mycc" version:"1.1" >
.
.
.
2018-02-22 18:28:46.687 UTC [chaincodeCmd] upgrade -> DEBU 009 endorse upgrade_
↳proposal, get response <status:200 message:"OK" payload:"\n\004mycc\022\0031.
↳1\032\004escc"\004vscc*,
↳\022\014\022\n\010\001\022\002\010\000\022\002\010\001\032\r\022\013\n\007Org1MSP\020\003\032\
↳\261g(^
↳v\021\220\240\332\251\014\204V\210P\310o\231\271\036\301\022\032\205fc[|= \215\372\223\022_
↳\311b\025?
↳\323N\343\325\032\005\365\236\001XKj\004E\351\007\247\265fu\305j\367\331\275\253\307R\032_
↳\014H#\014\272!\#\345\306s\323\371\350\364\006.
↳\000\356\230\353\270\263\215\217\303\256\220i^\277\305\214: \375\200zY\275\203}
↳\375\244\205\035\340\226]!uE\334\273\214\214\020\303\3474\360\014\234-
↳\006\315B\031\022\010\022\006\010\001\022\002\010\000\032\r\022\013\n\007Org1MSP\020\001
↳" >

```

(continues on next page)

(continued from previous page)

```

.
.
.
2018-02-22 18:28:46.693 UTC [chaincodeCmd] upgrade -> DEBU 00c Get Signed envelope
2018-02-22 18:28:46.693 UTC [chaincodeCmd] chaincodeUpgrade -> DEBU 00d Send_
↪signed envelope to orderer
2018-02-22 18:28:46.908 UTC [main] main -> INFO 00e Exiting.....

```

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.3 peer lifecycle chaincode

The `peer lifecycle chaincode` subcommand allows administrators to use the Fabric chaincode lifecycle to package a chaincode, install it on your peers, approve a chaincode definition for your organization, and then commit the definition to a channel. The chaincode is ready to be used after the definition has been successfully committed to the channel. For more information, visit [Fabric chaincode lifecycle](#).

Note: These instructions use the Fabric chaincode lifecycle introduced in the v2.0 release. If you would like to use the old lifecycle to install and instantiate a chaincode, visit the [peer chaincode](#) command reference.

11.3.1 Syntax

The `peer lifecycle chaincode` command has the following subcommands:

- `package`
- `install`
- `queryinstalled`
- `getinstalledpackage`
- `approveformyorg`
- `queryapproved`
- `checkcommitreadiness`
- `commit`
- `querycommitted`

Each peer lifecycle chaincode subcommand is described together with its options in its own section in this topic.

11.3.2 peer lifecycle

Perform `_lifecycle` operations

Usage:

```
peer lifecycle [command]
```

Available Commands:

```
chaincode    Perform chaincode operations:↪
```

```
↪package|install|queryinstalled|getinstalledpackage|approveformyorg|queryapproved|checkcommitreadiness
```

(continues on next page)

(continued from previous page)

Flags:

`-h, --help` help **for** lifecycleUse `"peer lifecycle [command] --help"` **for** more information about a command.

11.3.3 peer lifecycle chaincode

Perform chaincode operations: `└``↪package|install|queryinstalled|getinstalledpackage|approveformyorg|queryapproved|checkcommitreadiness`

Usage:

`peer lifecycle chaincode [command]`

Available Commands:

<code>approveformyorg</code>	Approve the chaincode definition for my org.
<code>checkcommitreadiness</code>	Check whether a chaincode definition is ready to be committed <code>└</code>
<code>↪on a channel.</code>	
<code>commit</code>	Commit the chaincode definition on the channel.
<code>getinstalledpackage</code>	Get an installed chaincode package from a peer.
<code>install</code>	Install a chaincode.
<code>package</code>	Package a chaincode
<code>queryapproved</code>	Query an org's approved chaincode definition from its peer.
<code>querycommitted</code>	Query the committed chaincode definitions by channel on a peer.
<code>queryinstalled</code>	Query the installed chaincodes on a peer.

Flags:

<code>--cafile string</code>	Path to file containing PEM-encoded <code>└</code>
<code>↪trusted certificate(s) for the ordering endpoint</code>	
<code>--certfile string</code>	Path to file containing PEM-encoded X509 <code>└</code>
<code>↪public key to use for mutual TLS communication with the orderer endpoint</code>	
<code>--clientauth</code>	Use mutual TLS when communicating with <code>└</code>
<code>↪the orderer endpoint</code>	
<code>--connTimeout duration</code>	Timeout for client to connect (default 3s)
<code>-h, --help</code>	help for chaincode
<code>--keyfile string</code>	Path to file containing PEM-encoded <code>└</code>
<code>↪private key to use for mutual TLS communication with the orderer endpoint</code>	
<code>-o, --orderer string</code>	Ordering service endpoint
<code>--ordererTLSHostnameOverride string</code>	The hostname override to use when <code>└</code>
<code>↪validating the TLS connection to the orderer</code>	
<code>--tls</code>	Use TLS when communicating with the <code>└</code>
<code>↪orderer endpoint</code>	
<code>--tlsHandshakeTimeShift duration</code>	The amount of time to shift backwards for <code>└</code>
<code>↪certificate expiration checks during TLS handshakes with the orderer endpoint</code>	

Use `"peer lifecycle chaincode [command] --help"` **for** more information about a command.

11.3.4 peer lifecycle chaincode package

Package a chaincode **and** write the package to a file.

Usage:

`peer lifecycle chaincode package [outputfile] [flags]`

(continues on next page)

(continued from previous page)

```

Flags:
  --connectionProfile string      The fully qualified path to the connection_
  ↪profile that provides the necessary connection information for the network. Note:_
  ↪currently only supported for providing peer connection information
  -h, --help                      help for package
  --label string                  The package label contains a human-readable_
  ↪description of the package
  -l, --lang string               Language the chaincode is written in (default
  ↪"golang")
  -p, --path string               Path to the chaincode
  --peerAddresses stringArray     The addresses of the peers to connect to
  --tlsRootCertFiles stringArray  If TLS is enabled, the paths to the TLS root_
  ↪cert files of the peers to connect to. The order and number of certs specified_
  ↪should match the --peerAddresses flag

Global Flags:
  --cafile string                 Path to file containing PEM-encoded_
  ↪trusted certificate(s) for the ordering endpoint
  --certfile string               Path to file containing PEM-encoded X509_
  ↪public key to use for mutual TLS communication with the orderer endpoint
  --clientauth                    Use mutual TLS when communicating with_
  ↪the orderer endpoint
  --connTimeout duration          Timeout for client to connect (default 3s)
  --keyfile string                Path to file containing PEM-encoded_
  ↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string            Ordering service endpoint
  --ordererTLSHostnameOverride string  The hostname override to use when_
  ↪validating the TLS connection to the orderer
  --tls                           Use TLS when communicating with the_
  ↪orderer endpoint
  --tlsHandshakeTimeShift duration  The amount of time to shift backwards for_
  ↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.3.5 peer lifecycle chaincode install

Install a chaincode on a peer.

Usage:

```
peer lifecycle chaincode install [flags]
```

Flags:

```

  --connectionProfile string      The fully qualified path to the connection_
  ↪profile that provides the necessary connection information for the network. Note:_
  ↪currently only supported for providing peer connection information
  -h, --help                      help for install
  --peerAddresses stringArray     The addresses of the peers to connect to
  --targetPeer string             When using a connection profile, the name of_
  ↪the peer to target for this action
  --tlsRootCertFiles stringArray  If TLS is enabled, the paths to the TLS root_
  ↪cert files of the peers to connect to. The order and number of certs specified_
  ↪should match the --peerAddresses flag

```

Global Flags:

```

  --cafile string                 Path to file containing PEM-encoded_
  ↪trusted certificate(s) for the ordering endpoint

```

(continues on next page)

(continued from previous page)

```

--certfile string                Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth                    Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration          Timeout for client to connect (default 3s)
--keyfile string                Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string            Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                          Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.3.6 peer lifecycle chaincode queryinstalled

Query the installed chaincodes on a peer.

Usage:

```
peer lifecycle chaincode queryinstalled [flags]
```

Flags:

```

--connectionProfile string      The fully qualified path to the connection
↪profile that provides the necessary connection information for the network. Note:
↪currently only supported for providing peer connection information
-h, --help                    help for queryinstalled
-O, --output string            The output format for query results. Default
↪is human-readable plain-text. json is currently the only supported format.
--peerAddresses stringArray    The addresses of the peers to connect to
--targetPeer string            When using a connection profile, the name of
↪the peer to target for this action
--tlsRootCertFiles stringArray If TLS is enabled, the paths to the TLS root
↪cert files of the peers to connect to. The order and number of certs specified
↪should match the --peerAddresses flag

```

Global Flags:

```

--cafile string                Path to file containing PEM-encoded
↪trusted certificate(s) for the ordering endpoint
--certfile string              Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth                   Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration          Timeout for client to connect (default 3s)
--keyfile string                Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string            Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                          Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.3.7 peer lifecycle chaincode getinstalledpackage

Get an installed chaincode package **from a** peer.

Usage:

```
peer lifecycle chaincode getinstalledpackage [outputfile] [flags]
```

Flags:

```
--connectionProfile string      The fully qualified path to the connection_
↪profile that provides the necessary connection information for the network. Note:_
↪currently only supported for providing peer connection information
-h, --help                      help for getinstalledpackage
--output-directory string       The output directory to use when writing a_
↪chaincode install package to disk. Default is the current working directory.
--package-id string            The identifier of the chaincode install package
--peerAddresses stringArray     The addresses of the peers to connect to
--targetPeer string             When using a connection profile, the name of_
↪the peer to target for this action
--tlsRootCertFiles stringArray If TLS is enabled, the paths to the TLS root_
↪cert files of the peers to connect to. The order and number of certs specified_
↪should match the --peerAddresses flag
```

Global Flags:

```
--cafile string                Path to file containing PEM-encoded_
↪trusted certificate(s) for the ordering endpoint
--certfile string              Path to file containing PEM-encoded X509_
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth                   Use mutual TLS when communicating with_
↪the orderer endpoint
--connTimeout duration         Timeout for client to connect (default 3s)
--keyfile string               Path to file containing PEM-encoded_
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string           Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when_
↪validating the TLS connection to the orderer
--tls                          Use TLS when communicating with the_
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for_
↪certificate expiration checks during TLS handshakes with the orderer endpoint
```

11.3.8 peer lifecycle chaincode approveformyorg

Approve the chaincode definition **for** my organization.

Usage:

```
peer lifecycle chaincode approveformyorg [flags]
```

Flags:

```
--channel-config-policy string The endorsement policy associated to this_
↪chaincode specified as a channel config policy reference
-C, --channelID string         The channel on which this command should be_
↪executed
--collections-config string     The fully qualified path to the collection_
↪JSON file including the file name
--connectionProfile string      The fully qualified path to the connection_
↪profile that provides the necessary connection information for the network. Note:_
↪currently only supported for providing peer connection information
```

(continues on next page)

(continued from previous page)

```

-E, --endorsement-plugin string      The name of the endorsement plugin to be used
↳for this chaincode
-h, --help                          help for approveformyorg
--init-required                      Whether the chaincode requires invoking 'init'
-n, --name string                    Name of the chaincode
--package-id string                  The identifier of the chaincode install package
--peerAddresses stringArray          The addresses of the peers to connect to
--sequence int                       The sequence number of the chaincode
↳definition for the channel
--signature-policy string            The endorsement policy associated to this
↳chaincode specified as a signature policy
--tlsRootCertFiles stringArray       If TLS is enabled, the paths to the TLS root
↳cert files of the peers to connect to. The order and number of certs specified
↳should match the --peerAddresses flag
-V, --validation-plugin string        The name of the validation plugin to be used
↳for this chaincode
-v, --version string                 Version of the chaincode
--waitForEvent                       Whether to wait for the event from each peer's
↳deliver filtered service signifying that the transaction has been committed
↳successfully (default true)
--waitForEventTimeout duration        Time to wait for the event from each peer's
↳deliver filtered service signifying that the 'invoke' transaction has been
↳committed successfully (default 30s)

Global Flags:
--cafile string                      Path to file containing PEM-encoded
↳trusted certificate(s) for the ordering endpoint
--certfile string                    Path to file containing PEM-encoded X509
↳public key to use for mutual TLS communication with the orderer endpoint
--clientauth                          Use mutual TLS when communicating with
↳the orderer endpoint
--connTimeout duration               Timeout for client to connect (default 3s)
--keyfile string                     Path to file containing PEM-encoded
↳private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string                  Ordering service endpoint
--ordererTLSHostnameOverride string   The hostname override to use when
↳validating the TLS connection to the orderer
--tls                                Use TLS when communicating with the
↳orderer endpoint
--tlsHandshakeTimeShift duration      The amount of time to shift backwards for
↳certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.3.9 peer lifecycle chaincode queryapproved

Query an organization's approved chaincode definition from its peer.

Usage:

```
peer lifecycle chaincode queryapproved [flags]
```

Flags:

```

-C, --channelID string                The channel on which this command should be
↳executed
--connectionProfile string             The fully qualified path to the connection
↳profile that provides the necessary connection information for the network. Note:
↳currently only supported for providing peer connection information

```

(continues on next page)

(continued from previous page)

```

-h, --help                help for queryapproved
-n, --name string         Name of the chaincode
-O, --output string       The output format for query results. Default
↳ is human-readable plain-text. json is currently the only supported format.
    --peerAddresses stringArray The addresses of the peers to connect to
    --sequence int        The sequence number of the chaincode
↳ definition for the channel
    --tlsRootCertFiles stringArray If TLS is enabled, the paths to the TLS root
↳ cert files of the peers to connect to. The order and number of certs specified
↳ should match the --peerAddresses flag

Global Flags:
    --cafile string        Path to file containing PEM-encoded
↳ trusted certificate(s) for the ordering endpoint
    --certfile string      Path to file containing PEM-encoded X509
↳ public key to use for mutual TLS communication with the orderer endpoint
    --clientauth           Use mutual TLS when communicating with
↳ the orderer endpoint
    --connTimeout duration Timeout for client to connect (default 3s)
    --keyfile string       Path to file containing PEM-encoded
↳ private key to use for mutual TLS communication with the orderer endpoint
    -o, --orderer string   Ordering service endpoint
    --ordererTLSHostnameOverride string The hostname override to use when
↳ validating the TLS connection to the orderer
    --tls                 Use TLS when communicating with the
↳ orderer endpoint
    --tlsHandshakeTimeShift duration The amount of time to shift backwards for
↳ certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.3.10 peer lifecycle chaincode checkcommitreadiness

Check whether a chaincode definition **is** ready to be committed on a channel.

Usage:

```
peer lifecycle chaincode checkcommitreadiness [flags]
```

Flags:

```

--channel-config-policy string The endorsement policy associated to this
↳ chaincode specified as a channel config policy reference
-C, --channelID string        The channel on which this command should be
↳ executed
--collections-config string    The fully qualified path to the collection
↳ JSON file including the file name
--connectionProfile string     The fully qualified path to the connection
↳ profile that provides the necessary connection information for the network. Note:
↳ currently only supported for providing peer connection information
-E, --endorsement-plugin string The name of the endorsement plugin to be used
↳ for this chaincode
-h, --help                    help for checkcommitreadiness
--init-required               Whether the chaincode requires invoking 'init'
-n, --name string             Name of the chaincode
-O, --output string           The output format for query results. Default
↳ is human-readable plain-text. json is currently the only supported format.
    --peerAddresses stringArray The addresses of the peers to connect to
    --sequence int            The sequence number of the chaincode
↳ definition for the channel

```

(continues on next page)

(continued from previous page)

```

--signature-policy string      The endorsement policy associated to this
↪chaincode specified as a signature policy
--tlsRootCertFiles stringArray If TLS is enabled, the paths to the TLS root
↪cert files of the peers to connect to. The order and number of certs specified
↪should match the --peerAddresses flag
-V, --validation-plugin string The name of the validation plugin to be used
↪for this chaincode
-v, --version string           Version of the chaincode

Global Flags:
--cafile string                Path to file containing PEM-encoded
↪trusted certificate(s) for the ordering endpoint
--certfile string              Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth                   Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration         Timeout for client to connect (default 3s)
--keyfile string               Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string           Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                           Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.3.11 peer lifecycle chaincode commit

Commit the chaincode definition on the channel.

Usage:

```
peer lifecycle chaincode commit [flags]
```

Flags:

```

--channel-config-policy string The endorsement policy associated to this
↪chaincode specified as a channel config policy reference
-C, --channelID string         The channel on which this command should be
↪executed
--collections-config string     The fully qualified path to the collection
↪JSON file including the file name
--connectionProfile string      The fully qualified path to the connection
↪profile that provides the necessary connection information for the network. Note:
↪currently only supported for providing peer connection information
-E, --endorsement-plugin string The name of the endorsement plugin to be used
↪for this chaincode
-h, --help                     help for commit
--init-required                 Whether the chaincode requires invoking 'init'
-n, --name string              Name of the chaincode
--peerAddresses stringArray     The addresses of the peers to connect to
--sequence int                 The sequence number of the chaincode
↪definition for the channel
--signature-policy string       The endorsement policy associated to this
↪chaincode specified as a signature policy
--tlsRootCertFiles stringArray If TLS is enabled, the paths to the TLS root
↪cert files of the peers to connect to. The order and number of certs specified
↪should match the --peerAddresses flag

```

(continues on next page)

(continued from previous page)

```

-V, --validation-plugin string      The name of the validation plugin to be used
→for this chaincode
-v, --version string                Version of the chaincode
--waitForEvent                      Whether to wait for the event from each peer's
→deliver filtered service signifying that the transaction has been committed
→successfully (default true)
--waitForEventTimeout duration      Time to wait for the event from each peer's
→deliver filtered service signifying that the 'invoke' transaction has been
→committed successfully (default 30s)

Global Flags:
--cafile string                     Path to file containing PEM-encoded
→trusted certificate(s) for the ordering endpoint
--certfile string                   Path to file containing PEM-encoded X509
→public key to use for mutual TLS communication with the orderer endpoint
--clientauth                        Use mutual TLS when communicating with
→the orderer endpoint
--connTimeout duration              Timeout for client to connect (default 3s)
--keyfile string                    Path to file containing PEM-encoded
→private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string                 Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
→validating the TLS connection to the orderer
--tls                               Use TLS when communicating with the
→orderer endpoint
--tlsHandshakeTimeShift duration     The amount of time to shift backwards for
→certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.3.12 peer lifecycle chaincode querycommitted

Query the committed chaincode definitions by channel on a peer. Optional: provide a chaincode name to query a specific definition.

Usage:

```
peer lifecycle chaincode querycommitted [flags]
```

Flags:

```

-C, --channelID string              The channel on which this command should be
→executed
--connectionProfile string           The fully qualified path to the connection
→profile that provides the necessary connection information for the network. Note:
→currently only supported for providing peer connection information
-h, --help                           help for querycommitted
-n, --name string                    Name of the chaincode
-O, --output string                  The output format for query results. Default
→is human-readable plain-text. json is currently the only supported format.
--peerAddresses stringArray          The addresses of the peers to connect to
--tlsRootCertFiles stringArray       If TLS is enabled, the paths to the TLS root
→cert files of the peers to connect to. The order and number of certs specified
→should match the --peerAddresses flag

```

Global Flags:

```

--cafile string                     Path to file containing PEM-encoded
→trusted certificate(s) for the ordering endpoint
--certfile string                   Path to file containing PEM-encoded X509
→public key to use for mutual TLS communication with the orderer endpoint

```

(continues on next page)

(continued from previous page)

```

--clientauth                Use mutual TLS when communicating with
↳the orderer endpoint
--connTimeout duration      Timeout for client to connect (default 3s)
--keyfile string            Path to file containing PEM-encoded
↳private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string        Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↳validating the TLS connection to the orderer
--tls                      Use TLS when communicating with the
↳orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↳certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.3.13 Example Usage

peer lifecycle chaincode package example

A chaincode needs to be packaged before it can be installed on your peers. This example uses the `peer lifecycle chaincode package` command to package a Go chaincode.

- Use the `--path` flag to indicate the location of the chaincode. The path must be a fully qualified path or a path relative to your present working directory.
- Use the `--label` flag to provide a chaincode package label of `myccv1` that your organization will use to identify the package.

```

peer lifecycle chaincode package mycc.tar.gz --path $CHAINCODE_DIR --lang golang -
↳-label myccv1

```

peer lifecycle chaincode install example

After the chaincode is packaged, you can use the `peer chaincode install` command to install the chaincode on your peers.

- Install the `mycc.tar.gz` package on `peer0.org1.example.com:7051` (the peer defined by `--peerAddresses`).

```

peer lifecycle chaincode install mycc.tar.gz --peerAddresses peer0.org1.example.
↳com:7051

```

If successful, the command will return the package identifier. The package ID is the package label combined with a hash of the chaincode package taken by the peer.

```

2019-03-13 13:48:53.691 UTC [cli.lifecycle.chaincode] submitInstallProposal ->
↳INFO 001 Installed remotely: response:<status:200 payload:
↳"\nEmycc:ebd89878c2bbccf62f68c36072626359376aa83c36435a058d453e8dbfd894cc" >
2019-03-13 13:48:53.691 UTC [cli.lifecycle.chaincode] submitInstallProposal ->
↳INFO 002 Chaincode code package identifier:
↳mycc:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9

```

peer lifecycle chaincode queryinstalled example

You need to use the chaincode package identifier to approve a chaincode definition for your organization. You can find the package ID for the chaincodes you have installed by using the `peer lifecycle chaincode queryinstalled` command:

```
peer lifecycle chaincode queryinstalled --peerAddresses peer0.org1.example.com:7051
```

A successful command will return the package ID associated with the package label.

Get installed chaincodes on peer:

```
Package ID: myccv1:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9,
↳Label: myccv1
```

- You can also use the `--output` flag to have the CLI format the output as JSON.

```
peer lifecycle chaincode queryinstalled --peerAddresses peer0.org1.example.
↳com:7051 --output json
```

If successful, the command will return the chaincodes you have installed as JSON.

```
{
  "installed_chaincodes": [
    {
      "package_id": "mycc_
↳1:aab9981fa5649cfe25369fce7bb5086a69672a631e4f95c4af1b5198fe9f845b",
      "label": "mycc_1",
      "references": {
        "mychannel": {
          "chaincodes": [
            {
              "name": "mycc",
              "version": "1"
            }
          ]
        }
      }
    }
  ]
}
```

peer lifecycle chaincode getinstalledpackage example

You can retrieve an installed chaincode package from a peer using the `peer lifecycle chaincode getinstalledpackage` command. Use the package identifier returned by `queryinstalled`.

- Use the `--package-id` flag to pass in the chaincode package identifier. Use the `--output-directory` flag to specify where to write the chaincode package. If the output directory is not specified, the chaincode package will be written in the current directory.

```
peer lifecycle chaincode getinstalledpackage --package-id_
↳myccv1:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9 --output-
↳directory /tmp --peerAddresses peer0.org1.example.com:7051
```

peer lifecycle chaincode approveformyorg example

Once the chaincode package has been installed on your peers, you can approve a chaincode definition for your organization. The chaincode definition includes the important parameters of chaincode governance, including the chaincode name, version and the endorsement policy.

Here is an example of the `peer lifecycle chaincode approveformyorg` command, which approves the definition of a chaincode named `mycc` at version `1.0` on channel `mychannel`.

- Use the `--package-id` flag to pass in the chaincode package identifier. Use the `--signature-policy` flag to define an endorsement policy for the chaincode. Use the `init-required` flag to request the execution of the `Init` function to initialize the chaincode.

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode approveformyorg -o orderer.example.com:7050 --tls --
↪cafile $ORDERER_CA --channelID mychannel --name mycc --version 1.0 --init-
↪required --package-id_
↪myccv1:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9 --
↪sequence 1 --signature-policy "AND ('Org1MSP.peer','Org2MSP.peer')"
```

```
2019-03-18 16:04:09.046 UTC [cli.lifecycle.chaincode] InitCmdFactory -> INFO 001_
↪Retrieved channel (mychannel) orderer endpoint: orderer.example.com:7050
2019-03-18 16:04:11.253 UTC [chaincodeCmd] ClientWait -> INFO 002 txid_
↪[efba188ca77889cc1c328fc98e0bb12d3ad0abcda3f84da3714471c7c1e6c13c] committed_
↪with status (VALID) at peer0.org1.example.com:7051
```

- You can also use the `--channel-config-policy` flag use a policy inside the channel configuration as the chaincode endorsement policy. The default endorsement policy is `Channel/Application/Endorsement`

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode approveformyorg -o orderer.example.com:7050 --tls --
↪cafile $ORDERER_CA --channelID mychannel --name mycc --version 1.0 --init-
↪required --package-id_
↪myccv1:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9 --
↪sequence 1 --channel-config-policy Channel/Application/Admins
```

```
2019-03-18 16:04:09.046 UTC [cli.lifecycle.chaincode] InitCmdFactory -> INFO 001_
↪Retrieved channel (mychannel) orderer endpoint: orderer.example.com:7050
2019-03-18 16:04:11.253 UTC [chaincodeCmd] ClientWait -> INFO 002 txid_
↪[efba188ca77889cc1c328fc98e0bb12d3ad0abcda3f84da3714471c7c1e6c13c] committed_
↪with status (VALID) at peer0.org1.example.com:7051
```

peer lifecycle chaincode queryapproved example

You can query an organization's approved chaincode definition by using the `peer lifecycle chaincode queryapproved` command. You can use this command to see the details (including package ID) of approved chaincode definitions.

- Here is an example of the `peer lifecycle chaincode queryapproved` command, which queries the approved definition of a chaincode named `mycc` at sequence number `1` on channel `mychannel`.

```
peer lifecycle chaincode queryapproved -C mychannel -n mycc --sequence 1
```

Approved chaincode definition **for** chaincode 'mycc' on channel 'mychannel':
sequence: 1, version: 1, init-required: true, package-id: mycc_
↪1:d02f72000e7c0f715840f51cb8d72d70bc1ba230552f8445dded0ec8b6e0b830, endorsement_
↪plugin: escc, validation plugin: vscc

If NO package is specified for the approved definition, this command will display an empty package ID.

- You can also use this command without specifying the sequence number in order to query the latest approved definition (latest: the newer of the currently defined sequence number and the next sequence number).

```
peer lifecycle chaincode queryapproved -C mychannel -n mycc
```

Approved chaincode definition **for** chaincode 'mycc' on channel 'mychannel':
sequence: 3, version: 3, init-required: false, package-id: mycc_
↪1:d02f72000e7c0f715840f51cb8d72d70bc1ba230552f8445dded0ec8b6e0b830, endorsement_
↪plugin: escc, validation plugin: vscc

- You can also use the `--output` flag to have the CLI format the output as JSON.
 - When querying an approved chaincode definition for which package is specified

```
peer lifecycle chaincode queryapproved -C mychannel -n mycc --sequence 1 --  
↪output json
```

If successful, the command will return a JSON that has the approved chaincode definition for chaincode mycc at sequence number 1 on channel mychannel.

```
{
  "sequence": 1,
  "version": "1",
  "endorsement_plugin": "escc",
  "validation_plugin": "vscc",
  "validation_parameter": "EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA==",
  "collections": {},
  "init_required": true,
  "source": {
    "Type": {
      "LocalPackage": {
        "package_id": "mycc_  

↪1:d02f72000e7c0f715840f51cb8d72d70bc1ba230552f8445dded0ec8b6e0b830"
      }
    }
  }
}
```

- When querying an approved chaincode definition for which package is NOT specified

```
peer lifecycle chaincode queryapproved -C mychannel -n mycc --sequence 2 --  
↪output json
```

If successful, the command will return a JSON that has the approved chaincode definition for chaincode mycc at sequence number 2 on channel mychannel.

```
{
  "sequence": 2,
  "version": "2",
```

(continues on next page)

(continued from previous page)

```

"endorsement_plugin": "escc",
"validation_plugin": "vscc",
"validation_parameter": "EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA==",
"collections": {},
"source": {
  "Type": {
    "Unavailable": {}
  }
}
}

```

peer lifecycle chaincode checkcommitreadiness example

You can check whether a chaincode definition is ready to be committed using the `peer lifecycle chaincode checkcommitreadiness` command, which will return successfully if a subsequent commit of the definition is expected to succeed. It also outputs which organizations have approved the chaincode definition. If an organization has approved the chaincode definition specified in the command, the command will return a value of `true`. You can use this command to learn whether enough channel members have approved a chaincode definition to meet the / Channel/Application/Endorsement policy (a majority by default) before the definition can be committed to a channel.

- Here is an example of the `peer lifecycle chaincode checkcommitreadiness` command, which checks a chaincode named `mycc` at version `1.0` on channel `mychannel`.

```

export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode checkcommitreadiness -o orderer.example.com:7050 --
↪channelID mychannel --tls --cafile $ORDERER_CA --name mycc --version 1.0 --init-
↪required --sequence 1

```

If successful, the command will return the organizations that have approved the chaincode definition.

```

Chaincode definition for chaincode 'mycc', version '1.0', sequence '1' on channel
'mychannel' approval status by org:
Org1MSP: true
Org2MSP: true

```

- You can also use the `--output` flag to have the CLI format the output as JSON.

```

export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode checkcommitreadiness -o orderer.example.com:7050 --
↪channelID mychannel --tls --cafile $ORDERER_CA --name mycc --version 1.0 --init-
↪required --sequence 1 --output json

```

If successful, the command will return a JSON map that shows if an organization has approved the chaincode definition.

```

{
  "Approvals": {
    "Org1MSP": true,

```

(continues on next page)

(continued from previous page)

```

    "Org2MSP": true
  }
}

```

peer lifecycle chaincode commit example

Once a sufficient number of organizations approve a chaincode definition for their organizations (a majority by default), one organization can commit the definition the channel using the `peer lifecycle chaincode commit` command:

- This command needs to target the peers of other organizations on the channel to collect their organization endorsement for the definition.

```

export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID mychannel_
↪--name mycc --version 1.0 --sequence 1 --init-required --tls --cafile $ORDERER_
↪CA --peerAddresses peer0.org1.example.com:7051 --peerAddresses peer0.org2.
↪example.com:9051

2019-03-18 16:14:27.258 UTC [chaincodeCmd] ClientWait -> INFO 001 txid_
↪[b6f657a14689b27d69a50f39590b3949906b5a426f9d7f0dcee557f775e17882] committed_
↪with status (VALID) at peer0.org2.example.com:9051
2019-03-18 16:14:27.321 UTC [chaincodeCmd] ClientWait -> INFO 002 txid_
↪[b6f657a14689b27d69a50f39590b3949906b5a426f9d7f0dcee557f775e17882] committed_
↪with status (VALID) at peer0.org1.example.com:7051

```

peer lifecycle chaincode querycommitted example

You can query the chaincode definitions that have been committed to a channel by using the `peer lifecycle chaincode querycommitted` command. You can use this command to query the current definition sequence number before upgrading a chaincode.

- You need to supply the chaincode name and channel name in order to query a specific chaincode definition and the organizations that have approved it.

```

export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode querycommitted -o orderer.example.com:7050 --channelID_
↪mychannel --name mycc --tls --cafile $ORDERER_CA --peerAddresses peer0.org1.
↪example.com:7051

Committed chaincode definition for chaincode 'mycc' on channel 'mychannel':
Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc
Approvals: [Org1MSP: true, Org2MSP: true]

```

- You can also specify just the channel name in order to query all chaincode definitions on that channel.

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlsacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode querycommitted -o orderer.example.com:7050 --channelID_
↪mychannel --tls --cafile $ORDERER_CA --peerAddresses peer0.org1.example.com:7051

Committed chaincode definitions on channel 'mychannel':
Name: mycc, Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin:_
↪vsccl
Name: yourcc, Version: 2, Sequence: 3, Endorsement Plugin: escc, Validation_
↪Plugin: vsccl
```

- You can also use the `--output` flag to have the CLI format the output as JSON.
 - For querying a specific chaincode definition

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlsacerts/tlsca.example.com-cert.pem

peer lifecycle chaincode querycommitted -o orderer.example.com:7050 --
↪channelID mychannel --name mycc --tls --cafile $ORDERER_CA --peerAddresses_
↪peer0.org1.example.com:7051 --output json
```

If successful, the command will return a JSON that has committed chaincode definition for chaincode 'mycc' on channel 'mychannel'.

```
{
  "sequence": 1,
  "version": "1",
  "endorsement_plugin": "escc",
  "validation_plugin": "vsccl",
  "validation_parameter": "EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA==",
  "collections": {},
  "init_required": true,
  "approvals": {
    "Org1MSP": true,
    "Org2MSP": true
  }
}
```

The `validation_parameter` is base64 encoded. An example of the command to decode it is as follows.

```
echo EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA== | base64 -d

/Channel/Application/Endorsement
```

- For querying all chaincode definitions on that channel

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlsacerts/tlsca.example.com-cert.pem

peer lifecycle chaincode querycommitted -o orderer.example.com:7050 --
↪channelID mychannel --tls --cafile $ORDERER_CA --peerAddresses peer0.org1.
↪example.com:7051 --output json
```

If successful, the command will return a JSON that has committed chaincode definitions on channel ‘my-channel’.

```
{
  "chaincode_definitions": [
    {
      "name": "mycc",
      "sequence": 1,
      "version": "1",
      "endorsement_plugin": "escc",
      "validation_plugin": "vsc",
      "validation_parameter":
      ↪ "EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA==",
      "collections": {},
      "init_required": true
    },
    {
      "name": "yourcc",
      "sequence": 3,
      "version": "2",
      "endorsement_plugin": "escc",
      "validation_plugin": "vsc",
      "validation_parameter":
      ↪ "EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA==",
      "collections": {}
    }
  ]
}
```

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.4 peer channel

The `peer channel` command allows administrators to perform channel related operations on a peer, such as joining a channel or listing the channels to which a peer is joined.

11.4.1 Syntax

The `peer channel` command has the following subcommands:

- create
- fetch
- getinfo
- join
- joinbysnapshot
- joinbysnapshotstatus
- list
- signconfigtx
- update

11.4.2 peer channel

Operate a channel: `create|fetch|join|joinbysnapshot|joinbysnapshotstatus|list|update|signconfigtx|getinfo`.

Usage:

```
peer channel [command]
```

Available Commands:

<code>create</code>	Create a channel
<code>fetch</code>	Fetch a block
<code>getinfo</code>	get blockchain information of a specified channel.
<code>join</code>	Joins the peer to a channel.
<code>joinbysnapshot</code>	Joins the peer to a channel by the specified snapshot
<code>joinbysnapshotstatus</code>	Query if joinbysnapshot is running for any channel
<code>list</code>	List of channels peer has joined.
<code>signconfigtx</code>	Signs a configtx update.
<code>update</code>	Send a configtx update.

Flags:

<code>--cafile string</code>	Path to file containing PEM-encoded
<code>trusted certificate(s) for the ordering endpoint</code>	
<code>--certfile string</code>	Path to file containing PEM-encoded X509
<code>public key to use for mutual TLS communication with the orderer endpoint</code>	
<code>--clientauth</code>	Use mutual TLS when communicating with
<code>the orderer endpoint</code>	
<code>--connTimeout duration</code>	Timeout for client to connect (default 3s)
<code>-h, --help</code>	help for channel
<code>--keyfile string</code>	Path to file containing PEM-encoded
<code>private key to use for mutual TLS communication with the orderer endpoint</code>	
<code>-o, --orderer string</code>	Ordering service endpoint
<code>--ordererTLSHostnameOverride string</code>	The hostname override to use when
<code>validating the TLS connection to the orderer</code>	
<code>--tls</code>	Use TLS when communicating with the
<code>orderer endpoint</code>	
<code>--tlsHandshakeTimeShift duration</code>	The amount of time to shift backwards for
<code>certificate expiration checks during TLS handshakes with the orderer endpoint</code>	

Use "`peer channel [command] --help`" **for** more information about a command.

11.4.3 peer channel create

Create a channel **and** write the genesis block to a file.

Usage:

```
peer channel create [flags]
```

Flags:

<code>-c, --channelID string</code>	In case of a newChain command, the channel ID to create.
<code>It must be all lower case, less than 250 characters long and match the regular</code>	
<code>expression: [a-z][a-z0-9.-]*</code>	
<code>-f, --file string</code>	Configuration transaction file generated by a tool such
<code>as configtxgen for submitting to orderer</code>	
<code>-h, --help</code>	help for create

(continues on next page)

(continued from previous page)

```

--outputBlock string    The path to write the genesis block for the channel.
↳(default ./<channelID>.block)
-t, --timeout duration    Channel creation timeout (default 10s)

Global Flags:
--cafile string                Path to file containing PEM-encoded
↳trusted certificate(s) for the ordering endpoint
--certfile string                Path to file containing PEM-encoded X509
↳public key to use for mutual TLS communication with the orderer endpoint
--clientauth                    Use mutual TLS when communicating with
↳the orderer endpoint
--connTimeout duration          Timeout for client to connect (default 3s)
--keyfile string                Path to file containing PEM-encoded
↳private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string            Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↳validating the TLS connection to the orderer
--tls                            Use TLS when communicating with the
↳orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↳certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.4.4 peer channel fetch

Fetch a specified block, writing it to a file.

Usage:

```
peer channel fetch <newest|oldest|config|(number)> [outputfile] [flags]
```

Flags:

```

--bestEffort                Whether fetch requests should ignore errors and return
↳blocks on a best effort basis
-c, --channelID string        In case of a newChain command, the channel ID to create.
↳It must be all lower case, less than 250 characters long and match the regular
↳expression: [a-z][a-z0-9.-]*
-h, --help                    help for fetch

```

Global Flags:

```

--cafile string                Path to file containing PEM-encoded
↳trusted certificate(s) for the ordering endpoint
--certfile string                Path to file containing PEM-encoded X509
↳public key to use for mutual TLS communication with the orderer endpoint
--clientauth                    Use mutual TLS when communicating with
↳the orderer endpoint
--connTimeout duration          Timeout for client to connect (default 3s)
--keyfile string                Path to file containing PEM-encoded
↳private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string            Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↳validating the TLS connection to the orderer
--tls                            Use TLS when communicating with the
↳orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↳certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.4.5 peer channel getinfo

get blockchain information of a specified channel. Requires '-c'.

Usage:

```
peer channel getinfo [flags]
```

Flags:

```
-c, --channelID string    In case of a newChain command, the channel ID to create.
↳ It must be all lower case, less than 250 characters long and match the regular
↳ expression: [a-z][a-z0-9.-]*
-h, --help                help for getinfo
```

Global Flags:

```
--cafile string          Path to file containing PEM-encoded
↳ trusted certificate(s) for the ordering endpoint
--certfile string        Path to file containing PEM-encoded X509
↳ public key to use for mutual TLS communication with the orderer endpoint
--clientauth             Use mutual TLS when communicating with
↳ the orderer endpoint
--connTimeout duration   Timeout for client to connect (default 3s)
--keyfile string         Path to file containing PEM-encoded
↳ private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string      Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↳ validating the TLS connection to the orderer
--tls                   Use TLS when communicating with the
↳ orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↳ certificate expiration checks during TLS handshakes with the orderer endpoint
```

11.4.6 peer channel join

Joins the peer to a channel.

Usage:

```
peer channel join [flags]
```

Flags:

```
-b, --blockpath string    Path to file containing genesis block
-h, --help                help for join
```

Global Flags:

```
--cafile string          Path to file containing PEM-encoded
↳ trusted certificate(s) for the ordering endpoint
--certfile string        Path to file containing PEM-encoded X509
↳ public key to use for mutual TLS communication with the orderer endpoint
--clientauth             Use mutual TLS when communicating with
↳ the orderer endpoint
--connTimeout duration   Timeout for client to connect (default 3s)
--keyfile string         Path to file containing PEM-encoded
↳ private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string      Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↳ validating the TLS connection to the orderer
```

(continues on next page)

(continued from previous page)

```

--tls                                Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration      The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.4.7 peer channel joinbysnapshot

Joins the peer to a channel by the specified snapshot

Usage:

```
peer channel joinbysnapshot [flags]
```

Flags:

```

-h, --help                help for joinbysnapshot
--snapshotpath string      Path to the snapshot directory

```

Global Flags:

```

--cafile string            Path to file containing PEM-encoded
↪trusted certificate(s) for the ordering endpoint
--certfile string          Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth               Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration      Timeout for client to connect (default 3s)
--keyfile string           Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string        Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                      Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.4.8 peer channel joinbysnapshotstatus

Query if joinbysnapshot is running for any channel

Usage:

```
peer channel joinbysnapshotstatus [flags]
```

Flags:

```
-h, --help    help for joinbysnapshotstatus
```

Global Flags:

```

--cafile string            Path to file containing PEM-encoded
↪trusted certificate(s) for the ordering endpoint
--certfile string          Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth               Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration      Timeout for client to connect (default 3s)

```

(continues on next page)

(continued from previous page)

```

--keyfile string                Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string            Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                          Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.4.9 peer channel list

List of channels peer has joined.

Usage:

```
peer channel list [flags]
```

Flags:

```
-h, --help    help for list
```

Global Flags:

```

--cafile string                Path to file containing PEM-encoded
↪trusted certificate(s) for the ordering endpoint
--certfile string              Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth                   Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration         Timeout for client to connect (default 3s)
--keyfile string                Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string            Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                          Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.4.10 peer channel signconfigtx

Signs the supplied configtx update file in place on the filesystem. Requires '-f'.

Usage:

```
peer channel signconfigtx [flags]
```

Flags:

```

-f, --file string    Configuration transaction file generated by a tool such as
↪configtxgen for submitting to orderer
-h, --help           help for signconfigtx

```

Global Flags:

```

--cafile string                Path to file containing PEM-encoded
↪trusted certificate(s) for the ordering endpoint

```

(continues on next page)

(continued from previous page)

```

--certfile string                Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth                     Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration          Timeout for client to connect (default 3s)
--keyfile string                Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string            Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                           Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.4.11 peer channel update

Signs and sends the supplied configtx update file to the channel. Requires '-f', '-o',
↪ '-c'.

Usage:

```
peer channel update [flags]
```

Flags:

```

-c, --channelID string          In case of a newChain command, the channel ID to create.
↪It must be all lower case, less than 250 characters long and match the regular
↪expression: [a-z][a-z0-9.-]*
-f, --file string              Configuration transaction file generated by a tool such as
↪configtxgen for submitting to orderer
-h, --help                     help for update

```

Global Flags:

```

--cafile string                Path to file containing PEM-encoded
↪trusted certificate(s) for the ordering endpoint
--certfile string              Path to file containing PEM-encoded X509
↪public key to use for mutual TLS communication with the orderer endpoint
--clientauth                   Use mutual TLS when communicating with
↪the orderer endpoint
--connTimeout duration          Timeout for client to connect (default 3s)
--keyfile string                Path to file containing PEM-encoded
↪private key to use for mutual TLS communication with the orderer endpoint
-o, --orderer string            Ordering service endpoint
--ordererTLSHostnameOverride string The hostname override to use when
↪validating the TLS connection to the orderer
--tls                           Use TLS when communicating with the
↪orderer endpoint
--tlsHandshakeTimeShift duration The amount of time to shift backwards for
↪certificate expiration checks during TLS handshakes with the orderer endpoint

```

11.4.12 Example Usage

peer channel create examples

Here's an example that uses the `--orderer` global flag on the `peer channel create` command.

- Create a sample channel `mychannel` defined by the configuration transaction contained in file `./createchannel.tx`. Use the orderer at `orderer.example.com:7050`.

```
peer channel create -c mychannel -f ./createchannel.tx --orderer orderer.example.
↪com:7050

2018-02-25 08:23:57.548 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and_
↪orderer connections initialized
2018-02-25 08:23:57.626 UTC [channelCmd] InitCmdFactory -> INFO 019 Endorser and_
↪orderer connections initialized
2018-02-25 08:23:57.834 UTC [channelCmd] readBlock -> INFO 020 Received block: 0
2018-02-25 08:23:57.835 UTC [main] main -> INFO 021 Exiting.....
```

Block 0 is returned indicating that the channel has been successfully created.

Here's an example of the `peer channel create` command option.

- Create a new channel `mychannel` for the network, using the orderer at ip address `orderer.example.com:7050`. The configuration update transaction required to create this channel is defined the file `./createchannel.tx`. Wait 30 seconds for the channel to be created.

```
peer channel create -c mychannel --orderer orderer.example.com:7050 -f ./
↪createchannel.tx -t 30s

2018-02-23 06:31:58.568 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser_
↪and orderer connections initialized
2018-02-23 06:31:58.669 UTC [channelCmd] InitCmdFactory -> INFO 019 Endorser_
↪and orderer connections initialized
2018-02-23 06:31:58.877 UTC [channelCmd] readBlock -> INFO 020 Received block: 0
2018-02-23 06:31:58.878 UTC [main] main -> INFO 021 Exiting.....

ls -l

-rw-r--r-- 1 root root 11982 Feb 25 12:24 mychannel.block
```

You can see that channel `mychannel` has been successfully created, as indicated in the output where block 0 (zero) is added to the blockchain for this channel and returned to the peer, where it is stored in the local directory as `mychannel.block`.

Block zero is often called the *genesis block* as it provides the starting configuration for the channel. All subsequent updates to the channel will be captured as configuration blocks on the channel's blockchain, each of which supersedes the previous configuration.

peer channel fetch example

Here's some examples of the `peer channel fetch` command.

- Using the newest option to retrieve the most recent channel block, and store it in the file `mychannel.block`.

```
peer channel fetch newest mychannel.block -c mychannel --orderer orderer.example.
↪com:7050

2018-02-25 13:10:16.137 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and_
↪orderer connections initialized
```

(continues on next page)

(continued from previous page)

```

2018-02-25 13:10:16.144 UTC [channelCmd] readBlock -> INFO 00a Received block: 32
2018-02-25 13:10:16.145 UTC [main] main -> INFO 00b Exiting.....

ls -l

-rw-r--r-- 1 root root 11982 Feb 25 13:10 mychannel.block

```

You can see that the retrieved block is number 32, and that the information has been written to the file `mychannel.block`.

- Using the `(block number)` option to retrieve a specific block – in this case, block number 16 – and store it in the default block file.

```

peer channel fetch 16 -c mychannel --orderer orderer.example.com:7050

2018-02-25 13:46:50.296 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and_
↳orderer connections initialized
2018-02-25 13:46:50.302 UTC [channelCmd] readBlock -> INFO 00a Received block: 16
2018-02-25 13:46:50.302 UTC [main] main -> INFO 00b Exiting.....

ls -l

-rw-r--r-- 1 root root 11982 Feb 25 13:10 mychannel.block
-rw-r--r-- 1 root root 4783 Feb 25 13:46 mychannel_16.block

```

You can see that the retrieved block is number 16, and that the information has been written to the default file `mychannel_16.block`.

For configuration blocks, the block file can be decoded using the `configtxlator` command. See this command for an example of decoded output. User transaction blocks can also be decoded, but a user program must be written to do this.

peer channel getinfo example

Here's an example of the `peer channel getinfo` command.

- Get information about the local peer for channel `mychannel`.

```

peer channel getinfo -c mychannel

2018-02-25 15:15:44.135 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and_
↳orderer connections initialized
Blockchain info: {"height":5,"currentBlockHash":"JgK9lcaPUNmFb5Mp1qe1SVMsx3o/
↳22Ct4+n5tejcXCw=", "previousBlockHash":
↳"f81ZXoAn3gF86zrFq7L1DzW2aKuabH9Ow6SIE5Y04a4="}
2018-02-25 15:15:44.139 UTC [main] main -> INFO 006 Exiting.....

```

You can see that the latest block for channel `mychannel` is block 5. You can also see the cryptographic hashes for the most recent blocks in the channel's blockchain.

peer channel join example

Here's an example of the `peer channel join` command.

- Join a peer to the channel defined in the genesis block identified by the file `./mychannel.genesis.block`. In this example, the channel block was previously retrieved by the `peer channel fetch` command.

```
peer channel join -b ./mychannel.genesis.block

2018-02-25 12:25:26.511 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and_
↳orderer connections initialized
2018-02-25 12:25:26.571 UTC [channelCmd] executeJoin -> INFO 006 Successfully_
↳submitted proposal to join channel
2018-02-25 12:25:26.571 UTC [main] main -> INFO 007 Exiting.....
```

You can see that the peer has successfully made a request to join the channel.

peer channel joinbysnapshot example

Here's an example of the `peer channel joinbysnapshot` command.

- Join a peer to the channel from a snapshot identified by the directory `/snapshots/completed/testchannel/1000`. The snapshot was previously created on a different peer.

```
peer channel joinbysnapshot --snapshotpath /snapshots/completed/testchannel/1000

2020-10-12 11:41:45.442 EDT [channelCmd] InitCmdFactory -> INFO 001 Endorser and_
↳orderer connections initialized
2020-10-12 11:41:45.444 EDT [channelCmd] executeJoin -> INFO 002 Successfully_
↳submitted proposal to join channel
2020-10-12 11:41:45.444 EDT [channelCmd] joinBySnapshot -> INFO 003 The_
↳joinbysnapshot operation is in progress. Use "peer channel joinbysnapshotstatus
↳" to check the status.
```

You can see that the peer has successfully made a request to join the channel from the specified snapshot. When a `joinbysnapshot` operation is in progress, you cannot run another `peer channel join` or `peer channel joinbysnapshot` simultaneously. To know whether or not a `joinbysnapshot` operation is in progress, you can call the `peer channel joinbysnapshotstatus` command.

peer channel joinbysnapshotstatus example

Here are some examples of the `peer channel joinbysnapshotstatus` command.

- Query if `joinbysnapshot` is in progress for any channel. If yes, it returns a message indicating a `joinbysnapshot` operation is in progress.

```
peer channel joinbysnapshotstatus

2020-10-12 11:41:45.952 EDT [channelCmd] InitCmdFactory -> INFO 001 Endorser and_
↳orderer connections initialized
A joinbysnapshot operation is in progress for snapshot at /snapshots/completed/
↳testchannel/1000
```

- If no `joinbysnapshot` operation is in progress, it returns a message indicating no `joinbysnapshot` operation is in progress.

```
peer channel joinbysnapshotstatus

2020-10-12 11:41:47.922 EDT [channelCmd] InitCmdFactory -> INFO 001 Endorser and_
↳orderer connections initialized
No joinbysnapshot operation is in progress
```

peer channel list example

Here's an example of the `peer channel list` command.

- List the channels to which a peer is joined.

```
peer channel list

2018-02-25 14:21:20.361 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and_
↪orderer connections initialized
Channels peers has joined:
mychannel
2018-02-25 14:21:20.372 UTC [main] main -> INFO 006 Exiting.....
```

You can see that the peer is joined to channel `mychannel`.

peer channel signconfigtx example

Here's an example of the `peer channel signconfigtx` command.

- Sign the channel update transaction defined in the file `./updatechannel.tx`. The example lists the configuration transaction file before and after the command.

```
ls -l

-rw-r--r--  1 anthonydowd  staff   284 25 Feb 18:16 updatechannel.tx

peer channel signconfigtx -f updatechannel.tx

2018-02-25 18:16:44.456 GMT [channelCmd] InitCmdFactory -> INFO 001 Endorser and_
↪orderer connections initialized
2018-02-25 18:16:44.459 GMT [main] main -> INFO 002 Exiting.....

ls -l

-rw-r--r--  1 anthonydowd  staff  2180 25 Feb 18:16 updatechannel.tx
```

You can see that the peer has successfully signed the configuration transaction by the increase in the size of the file `updatechannel.tx` from 284 bytes to 2180 bytes.

peer channel update example

Here's an example of the `peer channel update` command.

- Update the channel `mychannel` using the configuration transaction defined in the file `./updatechannel.tx`. Use the orderer at ip address `orderer.example.com:7050` to send the configuration transaction to all peers in the channel to update their copy of the channel configuration.

```
peer channel update -c mychannel -f ./updatechannel.tx -o orderer.example.com:7050

2018-02-23 06:32:11.569 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and_
↪orderer connections initialized
2018-02-23 06:32:11.626 UTC [main] main -> INFO 010 Exiting.....
```

At this point, the channel `mychannel` has been successfully updated.

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.5 peer snapshot

The `peer snapshot` command allows administrators to perform snapshot related operations on a peer, such as submit a snapshot request, cancel a snapshot request and list pending requests. Once a snapshot request is submitted for a specified block number, the snapshot will be automatically generated when the block number is committed on the channel.

11.5.1 Syntax

The `peer snapshot` command has the following subcommands:

- `cancelrequest`
- `listpending`
- `submitrequest`

11.5.2 peer snapshot cancelrequest

```
Cancel a request for a snapshot at the specified block.

Usage:
  peer snapshot cancelrequest [flags]

Flags:
  -b, --blockNumber uint           The block number for which a snapshot will be_
    generated
  -c, --channelID string           The channel on which this command should be executed
  -h, --help                       help for cancelrequest
      --peerAddress string         The address of the peer to connect to
      --tlsRootCertFile string    The path to the TLS root cert file of the peer to_
    connect to, required if TLS is enabled and ignored if TLS is disabled.
```

11.5.3 peer snapshot listpending

```
List pending requests for snapshots.

Usage:
  peer snapshot listpending [flags]

Flags:
  -c, --channelID string           The channel on which this command should be executed
  -h, --help                       help for listpending
      --peerAddress string         The address of the peer to connect to
      --tlsRootCertFile string    The path to the TLS root cert file of the peer to_
    connect to, required if TLS is enabled and ignored if TLS is disabled.
```

11.5.4 peer snapshot submitrequest

Submit a request **for** a snapshot at the specified block. When the `blockNumber` parameter **is** set to 0 **or not** provided, it will submit a request **for** the last committed block.

Usage:

```
peer snapshot submitrequest [flags]
```

Flags:

```
-b, --blockNumber uint      The block number for which a snapshot will be
                             generated
-c, --channelID string      The channel on which this command should be executed
-h, --help                  help for submitrequest
                             --peerAddress string    The address of the peer to connect to
                             --tlsRootCertFile string The path to the TLS root cert file of the peer to
                             connect to, required if TLS is enabled and ignored if TLS is disabled.
```

11.5.5 Example Usage

peer snapshot cancelrequest example

Here is an example of the `peer snapshot cancelrequest` command.

- Cancel a snapshot request for block number 1000 on channel `mychannel` for `peer0.org1.example.com:7051`:

```
peer snapshot cancelrequest -c mychannel -b 1000 --peerAddress peer0.org1.example.com:7051

Snapshot request cancelled successfully
```

A successful response indicates that the snapshot request for block number 1000 has been removed from `mychannel`. If there is no pending snapshot request for block number 1000, the command will return an error.

- Use the `--tlsRootCertFile` flag in a network with TLS enabled

peer snapshot listpending example

Here is an example of the `peer snapshot listpending` command. If a snapshot is already generated, the corresponding block number will be removed from the pending list.

- List pending snapshot requests on channel `mychannel` for `peer0.org1.example.com:7051`:

```
peer snapshot listpending -c mychannel --peerAddress peer0.org1.example.com:7051

Successfully got pending snapshot requests: [1000 5000]
```

You can see that the command returns a list of block numbers for the pending snapshot requests.

- Use the `--tlsRootCertFile` flag in a network with TLS enabled

peer snapshot submitrequest example

Here is an example of the `peer snapshot submitrequest` command.

- Submit a snapshot request for block number 1000 on channel `mychannel` for `peer0.org1.example.com:7051`:

```
peer snapshot submitrequest -c mychannel -b 1000 --peerAddress peer0.org1.example.com:7051

Snapshot request submitted successfully
```

You can see that the snapshot request is successfully submitted on channel `mychannel`. A snapshot will be automatically generated after the block number 1000 is committed on the channel.

The specified block number must be at or above the last block number on the channel. Otherwise, the command will return an error.

- Use the `--tlsRootCertFile` flag in a network with TLS enabled

11.6 peer version

The `peer version` command displays the version information of the peer. It displays version, Commit SHA, Go version, OS/architecture, and chaincode information. For example:

```
peer:
  Version: 2.1.0
  Commit SHA: b78d79b
  Go version: go1.14.1
  OS/Arch: linux/amd64
  Chaincode:
    Base Docker Label: org.hyperledger.fabric
    Docker Namespace: hyperledger
```

11.6.1 Syntax

The `peer version` command takes no arguments.

11.6.2 peer version

```
Print current version of the fabric peer server.

Usage:
  peer version [flags]

Flags:
  -h, --help    help for version
```

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.7 peer node

The `peer node` command allows an administrator to start a peer node, pause and resume a channel, rebuild databases, reset all channels in a peer to the genesis block, rollback a channel to a given block number, and upgrade the database format.

11.7.1 Syntax

The `peer node` command has the following subcommands:

- `pause`
- `rebuild-dbs`
- `reset`
- `resume`
- `rollback`
- `start`
- `upgrade-dbs`

11.7.2 peer node pause

Pauses a channel on the peer. When the command **is** executed, the peer must be offline.↵
↵When the peer starts after pause, it will **not** receive blocks **for** the paused channel.

Usage:

```
peer node pause [flags]
```

Flags:

```
-c, --channelID string  Channel to pause.
-h, --help              help for pause
```

11.7.3 peer node rebuild-dbs

Drops the databases **for all** the channels **and** rebuilds them upon peer restart. When↵
↵the command **is** executed, the peer must be offline. The command **is not** supported **if**↵
↵the peer contains **any** channel that was bootstrapped **from a** snapshot.

Usage:

```
peer node rebuild-dbs [flags]
```

Flags:

```
-h, --help  help for rebuild-dbs
```

11.7.4 peer node reset

Resets **all** channels to the genesis block. When the command **is** executed, the peer must↵
↵be offline. When the peer starts after the reset, it will receive blocks starting↵
↵**with** block number one **from an** orderer **or** another peer to rebuild the block store↵
↵**and** state database. The command **is not** supported **if** the peer contains **any** channel↵
↵that was bootstrapped **from a** snapshot.

Usage:

```
peer node reset [flags]
```

Flags:

```
-h, --help  help for reset
```

11.7.5 peer node resume

Resumes a channel on the peer. When the command **is** executed, the peer must be offline.
→ When the peer starts after resume, it will receive blocks **for** the resumed channel.

Usage:

```
peer node resume [flags]
```

Flags:

```
-c, --channelID string  Channel to resume.
-h, --help              help for resume
```

11.7.6 peer node rollback

Rolls back a channel to a specified block number. When the command **is** executed, the
→ peer must be offline. When the peer starts after the rollback, it will receive
→ blocks, which got removed during the rollback, **from an** orderer **or** another peer to
→ rebuild the block store **and** state database. The command **is not** supported **if** the
→ peer contains **any** channel that was bootstrapped **from a** snapshot.

Usage:

```
peer node rollback [flags]
```

Flags:

```
-b, --blockNumber uint  Block number to which the channel needs to be rolled back,
→ to.
-c, --channelID string  Channel to rollback.
-h, --help              help for rollback
```

11.7.7 peer node start

Starts a node that interacts **with** the network.

Usage:

```
peer node start [flags]
```

Flags:

```
-h, --help              help for start
--peer-chaincodedev    start peer in chaincode development mode
```

11.7.8 peer node upgrade-dbs

Upgrades databases by directly updating the database **format** **or** dropping the databases.
→ Dropped databases will be rebuilt **with** new **format** upon peer restart. When the
→ command **is** executed, the peer must be offline.

Usage:

```
peer node upgrade-dbs [flags]
```

Flags:

```
-h, --help              help for upgrade-dbs
```

11.7.9 Example Usage

peer node pause example

The following command:

```
peer node pause -c ch1
```

pauses a channel on the peer. When the peer starts after pause, the paused channel will not be started and the peer will not receive blocks for the paused channel.

peer node rebuild-dbs example

The following command:

```
peer node rebuild-dbs
```

drops the databases for all the channels. When the peer is started after running this command, the peer will retrieve the blocks stored on the peer and rebuild the dropped databases for all the channels.

peer node reset example

The following command:

```
peer node reset
```

resets all channels in the peer to the genesis block, i.e., the first block in the channel. The command also records the pre-reset height of each channel in the file system. Note that the peer process should be stopped while executing this command. If the peer process is running, this command detects that and returns an error instead of performing the reset. When the peer is started after performing the reset, the peer will fetch the blocks for each channel which were removed by the reset command (either from other peers or orderers) and commit the blocks up to the pre-reset height. Until all channels reach the pre-reset height, the peer will not endorse any transactions.

peer node resume example

The following command:

```
peer node resume -c ch1
```

resumes a channel on the peer. When the peer starts after resume, the resumed channel will be started and the peer will receive blocks for the resumed channel.

peer node rollback example

The following command:

```
peer node rollback -c ch1 -b 150
```

rolls back the channel ch1 to block number 150. The command also records the pre-rolled back height of channel ch1 in the file system. Note that the peer should be stopped while executing this command. If the peer process is running, this command detects that and returns an error instead of performing the rollback. When the peer is started after performing the rollback, the peer will fetch the blocks for channel ch1 which were removed by the rollback command

(either from other peers or orderers) and commit the blocks up to the pre-rolled back height. Until the channel `chl` reaches the pre-rolled back height, the peer will not endorse any transaction for any channel.

peer node start example

The following command:

```
peer node start --peer-chaincodedev
```

starts a peer node in chaincode development mode. Normally chaincode containers are started and maintained by peer. However in chaincode development mode, chaincode is built and started by the user. This mode is useful during chaincode development phase for iterative development.

peer node upgrade-dbs example

The following command:

```
peer node upgrade-dbs
```

checks the data format in the databases for all the channels and drops databases if data format is in the previous version. The command will return an error if the data format is already up to date. When the peer is started after running this command, the peer will retrieve the blocks stored on the peer and rebuild the dropped databases in the new format.

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.8 osnadmin channel

The `osnadmin channel` command allows administrators to perform channel-related operations on an orderer, such as joining a channel, listing the channels an orderer has joined, and removing a channel. The channel participation API must be enabled and the Admin endpoint must be configured in the `orderer.yaml` for each orderer.

*Note: For a network using a system channel, `list` (for all channels) and `remove` (for the system channel) are the only supported operations. Any other attempted operation will return an error.

11.8.1 Syntax

The `osnadmin channel` command has the following subcommands:

- `join`
- `list`
- `remove`

11.8.2 osnadmin channel

```
usage: osnadmin channel <command> [<args> ...]
```

Channel actions

Flags:

`--help`

Show context-sensitive help (also **try**

(continues on next page)

(continued from previous page)

```

--help-long and --help-man).
-o, --orderer-address=ORDERER-ADDRESS
    Admin endpoint of the OSN
--ca-file=CA-FILE
    Path to file containing PEM-encoded TLS CA
    certificate(s) for the OSN
--client-cert=CLIENT-CERT
    Path to file containing PEM-encoded X509 public
    key to use for mutual TLS communication with
    the OSN
--client-key=CLIENT-KEY
    Path to file containing PEM-encoded private key
    to use for mutual TLS communication with the
    OSN

```

Subcommands:

```

channel join --channelID=CHANNELID --config-block=CONFIG-BLOCK
    Join an Ordering Service Node (OSN) to a channel. If the channel does not
    yet exist, it will be created.

channel list [<flags>]
    List channel information for an Ordering Service Node (OSN). If the
    channelID flag is set, more detailed information will be provided for that
    channel.

channel remove --channelID=CHANNELID
    Remove an Ordering Service Node (OSN) from a channel.

```

11.8.3 osnadmin channel join

```
usage: osnadmin channel join --channelID=CHANNELID --config-block=CONFIG-BLOCK
```

Join an Ordering Service Node (OSN) to a channel. If the channel does not yet exist, it will be created.

Flags:

```

--help
    Show context-sensitive help (also try
    --help-long and --help-man).
-o, --orderer-address=ORDERER-ADDRESS
    Admin endpoint of the OSN
--ca-file=CA-FILE
    Path to file containing PEM-encoded TLS CA
    certificate(s) for the OSN
--client-cert=CLIENT-CERT
    Path to file containing PEM-encoded X509 public
    key to use for mutual TLS communication with
    the OSN
--client-key=CLIENT-KEY
    Path to file containing PEM-encoded private key
    to use for mutual TLS communication with the
    OSN
-c, --channelID=CHANNELID
    Channel ID
-b, --config-block=CONFIG-BLOCK
    Path to the file containing an up-to-date
    config block for the channel

```

11.8.4 osnadmin channel list

```
usage: osnadmin channel list [<flags>]
```

List channel information **for** an Ordering Service Node (OSN). If the channelID flag **is set**, more detailed information will be provided **for** that channel.

Flags:

<code>--help</code>	Show context-sensitive help (also try <code>--help-long</code> and <code>--help-man</code>).
<code>-o, --orderer-address=ORDERER-ADDRESS</code>	Admin endpoint of the OSN
<code>--ca-file=CA-FILE</code>	Path to file containing PEM-encoded TLS CA certificate(s) for the OSN
<code>--client-cert=CLIENT-CERT</code>	Path to file containing PEM-encoded X509 public key to use for mutual TLS communication with the OSN
<code>--client-key=CLIENT-KEY</code>	Path to file containing PEM-encoded private key to use for mutual TLS communication with the OSN
<code>-c, --channelID=CHANNELID</code>	Channel ID

11.8.5 osnadmin channel remove

```
usage: osnadmin channel remove --channelID=CHANNELID
```

Remove an Ordering Service Node (OSN) **from a** channel.

Flags:

<code>--help</code>	Show context-sensitive help (also try <code>--help-long</code> and <code>--help-man</code>).
<code>-o, --orderer-address=ORDERER-ADDRESS</code>	Admin endpoint of the OSN
<code>--ca-file=CA-FILE</code>	Path to file containing PEM-encoded TLS CA certificate(s) for the OSN
<code>--client-cert=CLIENT-CERT</code>	Path to file containing PEM-encoded X509 public key to use for mutual TLS communication with the OSN
<code>--client-key=CLIENT-KEY</code>	Path to file containing PEM-encoded private key to use for mutual TLS communication with the OSN
<code>-c, --channelID=CHANNELID</code>	Channel ID

11.8.6 Example Usage

osnadmin channel join examples

Here's an example of the `osnadmin channel join` command.

- Create and join a sample channel `mychannel` defined by the application channel genesis block contained in file `mychannel-genesis-block.pb`. Use the orderer admin endpoint at `orderer.example.com:9443`.

```
osnadmin channel join -o orderer.example.com:9443 --ca-file $CA_FILE --client-
↪cert $CLIENT_CERT --client-key $CLIENT_KEY --channelID mychannel (continues on next page)
↪mychannel-genesis-block.pb
```

(continued from previous page)

```
Status: 201
{
  "name": "mychannel",
  "url": "/participation/v1/channels/mychannel",
  "consensusRelation": "consenter",
  "status": "active",
  "height": 1
}
```

Status 201 and the channel details are returned indicating that the channel has been successfully created and joined.

osnadmin channel list example

Here are some examples of the `osnadmin channel list` command.

- Listing all the channels that the orderer has joined. This includes the system channel (if one exists) and all of the application channels.

```
osnadmin channel list -o orderer.example.com:9443 --ca-file $CA_FILE --client-
↪cert $CLIENT_CERT --client-key $CLIENT_KEY

Status: 200
{
  "systemChannel": null,
  "channels": [
    {
      "name": "mychannel",
      "url": "/participation/v1/channels/mychannel"
    }
  ]
}
```

Status 200 and the list of channels are returned.

- Using the `--channelID` flag to list more details for mychannel.

```
osnadmin channel list -o orderer.example.com:9443 --ca-file $CA_FILE --client-
↪cert $CLIENT_CERT --client-key $CLIENT_KEY --channelID mychannel

Status: 200
{
  "name": "mychannel",
  "url": "/participation/v1/channels/mychannel",
  "consensusRelation": "consenter",
  "status": "active",
  "height": 3
}
```

Status 200 and the details of the channels are returned.

osnadmin channel remove example

Here's an example of the `osnadmin channel remove` command.

- Removing channel mychannel from the orderer at orderer.example.com:9443.

```
osnadmin channel remove -o orderer.example.com:9443 --ca-file $CA_FILE --client-  
↪cert $CLIENT_CERT --client-key $CLIENT_KEY --channelID mychannel  
  
Status: 204
```

Status 204 is returned upon successful removal of a channel.

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.9 configtxgen

The configtxgen command allows users to create and inspect channel config related artifacts. The content of the generated artifacts is dictated by the contents of configtx.yaml.

11.9.1 Syntax

The configtxgen tool has no sub-commands, but supports flags which can be set to accomplish a number of tasks.

11.9.2 configtxgen

```
Usage of configtxgen:  
-asOrg string  
    Performs the config generation as a particular organization (by name), only_  
↪including values in the write set that org (likely) has privilege to set  
-channelCreateTxBaseProfile string  
    Specifies a profile to consider as the orderer system channel current state_  
↪to allow modification of non-application parameters during channel create tx_  
↪generation. Only valid in conjunction with 'outputCreateChannelTx'.  
-channelID string  
    The channel ID to use in the configtx  
-configPath string  
    The path containing the configuration to use (if set)  
-inspectBlock string  
    Prints the configuration contained in the block at the specified path  
-inspectChannelCreateTx string  
    Prints the configuration contained in the transaction at the specified path  
-outputAnchorPeersUpdate string  
    [DEPRECATED] Creates a config update to update an anchor peer (works only_  
↪with the default channel creation, and only for the first update)  
-outputBlock string  
    The path to write the genesis block to (if set)  
-outputCreateChannelTx string  
    The path to write a channel creation configtx to (if set)  
-printOrg string  
    Prints the definition of an organization as JSON. (useful for adding an org_  
↪to a channel manually)  
-profile string  
    The profile from configtx.yaml to use for generation.  
-version  
    Show version information
```


11.9.3 Usage

Output a genesis block

Write a genesis block to `genesis_block.pb` for channel `orderer-system-channel` for profile `SampleSingleMSPRaftV1_1`.

```
configtxgen -outputBlock genesis_block.pb -profile SampleSingleMSPRaftV1_1 -channelID_
↪orderer-system-channel
```

Output a channel creation tx

Write a channel creation transaction to `create_chan_tx.pb` for profile `SampleSingleMSPChannelV1_1`.

```
configtxgen -outputCreateChannelTx create_chan_tx.pb -profile_
↪SampleSingleMSPChannelV1_1 -channelID application-channel-1
```

Inspect a genesis block

Print the contents of a genesis block named `genesis_block.pb` to the screen as JSON.

```
configtxgen -inspectBlock genesis_block.pb
```

Inspect a channel creation tx

Print the contents of a channel creation tx named `create_chan_tx.pb` to the screen as JSON.

```
configtxgen -inspectChannelCreateTx create_chan_tx.pb
```

Print an organization definition

Construct an organization definition based on the parameters such as `MSPDir` from `configtx.yaml` and print it as JSON to the screen. (This output is useful for channel reconfiguration workflows, such as adding a member).

```
configtxgen -printOrg Org1
```

Output anchor peer tx (deprecated)

Output a channel configuration update transaction `anchor_peer_tx.pb` based on the anchor peers defined for `Org1` and channel profile `SampleSingleMSPChannelV1_1` in `configtx.yaml`. Transaction will set anchor peers for `Org1` if no anchor peers have been set on the channel.

```
configtxgen -outputAnchorPeersUpdate anchor_peer_tx.pb -profile_
↪SampleSingleMSPChannelV1_1 -asOrg Org1
```

The `-outputAnchorPeersUpdate` output flag has been deprecated. To set anchor peers on the channel, use `configtxlator` to update the channel configuration.

11.9.4 Configuration

The `configtxgen` tool's output is largely controlled by the content of `configtx.yaml`. This file is searched for at `FABRIC_CFG_PATH` and must be present for `configtxgen` to operate.

Refer to the sample `configtx.yaml` shipped with Fabric for all possible configuration options. You may find this file in the `config` directory of the release artifacts tar, or you may find it under the `sampleconfig` folder if you are building from source.

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.10 configtxlator

The `configtxlator` command allows users to translate between protobuf and JSON versions of fabric data structures and create config updates. The command may either start a REST server to expose its functions over HTTP or may be utilized directly as a command line tool.

11.10.1 Syntax

The `configtxlator` tool has five sub-commands, as follows:

- `start`
- `proto_encode`
- `proto_decode`
- `compute_update`
- `version`

11.10.2 configtxlator start

```
usage: configtxlator start [<flags>]

Start the configtxlator REST server

Flags:
  --help                Show context-sensitive help (also try --help-long and
                        --help-man).
  --hostname="0.0.0.0"  The hostname or IP on which the REST server will listen
  --port=7059           The port on which the REST server will listen
  --CORS=CORS ...       Allowable CORS domains, e.g. '*' or 'www.example.com'
                        (may be repeated).
```

11.10.3 configtxlator proto_encode

```
usage: configtxlator proto_encode --type=TYPE [<flags>]

Converts a JSON document to protobuf.

Flags:
  --help                Show context-sensitive help (also try --help-long and
```

(continues on next page)

(continued from previous page)

```

--type=TYPE          --help-man).
                    The type of protobuf structure to encode to. For
--input=/dev/stdin    example, 'common.Config'.
                    A file containing the JSON document.
--output=/dev/stdout  A file to write the output to.

```

11.10.4 configtxlator proto_decode

```
usage: configtxlator proto_decode --type=TYPE [<flags>]
```

Converts a proto message to JSON.

Flags:

```

--help              Show context-sensitive help (also try --help-long and
--help-man).
--type=TYPE         The type of protobuf structure to decode from. For
                    example, 'common.Config'.
--input=/dev/stdin  A file containing the proto message.
--output=/dev/stdout A file to write the JSON document to.

```

11.10.5 configtxlator compute_update

```
usage: configtxlator compute_update --channel_id=CHANNEL_ID [<flags>]
```

Takes two marshaled `common.Config` messages `and` computes the config update which transitions between the two.

Flags:

```

--help              Show context-sensitive help (also try --help-long and
--help-man).
--original=ORIGINAL The original config message.
--updated=UPDATED    The updated config message.
--channel_id=CHANNEL_ID The name of the channel for this update.
--output=/dev/stdout A file to write the JSON document to.

```

11.10.6 configtxlator version

```
usage: configtxlator version
```

Show version information

Flags:

```
--help Show context-sensitive help (also try --help-long and --help-man).
```

11.10.7 Examples

Decoding

Decode a block named `fabric_block.pb` to JSON and print to stdout.

```
configtxlator proto_decode --input fabric_block.pb --type common.Block
```

Alternatively, after starting the REST server, the following curl command performs the same operation through the REST API.

```
curl -X POST --data-binary @fabric_block.pb "${CONFIGTXLATOR_URL}/protolator/decode/  
↪common.Block"
```

Encoding

Convert a JSON document for a policy from stdin to a file named `policy.pb`.

```
configtxlator proto_encode --type common.Policy --output policy.pb
```

Alternatively, after starting the REST server, the following curl command performs the same operation through the REST API.

```
curl -X POST --data-binary /dev/stdin "${CONFIGTXLATOR_URL}/protolator/encode/common.  
↪Policy" > policy.pb
```

Pipelines

Compute a config update from `original_config.pb` and `modified_config.pb` and decode it to JSON to stdout.

```
configtxlator compute_update --channel_id testchan --original original_config.pb --  
↪updated modified_config.pb | configtxlator proto_decode --type common.ConfigUpdate
```

Alternatively, after starting the REST server, the following curl commands perform the same operations through the REST API.

```
curl -X POST -F channel=testchan -F "original=@original_config.pb" -F  
↪"updated=@modified_config.pb" "${CONFIGTXLATOR_URL}/configtxlator/compute/update-  
↪from-configs" | curl -X POST --data-binary /dev/stdin "${CONFIGTXLATOR_URL}/  
↪protolator/decode/common.ConfigUpdate"
```

11.10.8 Additional Notes

The tool name is a portmanteau of *configtx* and *translator* and is intended to convey that the tool simply converts between different equivalent data representations. It does not generate configuration. It does not submit or retrieve configuration. It does not modify configuration itself, it simply provides some bijective operations between different views of the configtx format.

There is no configuration file `configtxlator` nor any authentication or authorization facilities included for the REST server. Because `configtxlator` does not have any access to data, key material, or other information which might be considered sensitive, there is no risk to the owner of the server in exposing it to other clients. However, because the data sent by a user to the REST server might be confidential, the user should either trust the administrator of the server, run a local instance, or operate via the CLI.

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.11 cryptogen

`cryptogen` is an utility for generating Hyperledger Fabric key material. It is provided as a means of preconfiguring a network for testing purposes. It would normally not be used in the operation of a production network.

11.11.1 Syntax

The `cryptogen` command has five subcommands, as follows:

- `help`
- `generate`
- `showtemplate`
- `extend`
- `version`

11.11.2 cryptogen help

```
usage: cryptogen [<flags>] <command> [<args> ...]

Utility for generating Hyperledger Fabric key material

Flags:
  --help  Show context-sensitive help (also try --help-long and --help-man).

Commands:
  help [<command>...]
    Show help.

  generate [<flags>]
    Generate key material

  showtemplate
    Show the default configuration template

  version
    Show version information

  extend [<flags>]
    Extend existing network
```

11.11.3 cryptogen generate

```
usage: cryptogen generate [<flags>]

Generate key material

Flags:
  --help          Show context-sensitive help (also try --help-long
                  and --help-man).
```

(continues on next page)

(continued from previous page)

```
--output="crypto-config"  The output directory in which to place artifacts
--config=CONFIG           The configuration template to use
```

11.11.4 cryptogen showtemplate

```
usage: cryptogen showtemplate
```

Show the default configuration template

Flags:

```
--help  Show context-sensitive help (also try --help-long and --help-man).
```

11.11.5 cryptogen extend

```
usage: cryptogen extend [<flags>]
```

Extend existing network

Flags:

```
--help                Show context-sensitive help (also try --help-long and
--help-man) .
--input="crypto-config"  The input directory in which existing network place
--config=CONFIG          The configuration template to use
```

11.11.6 cryptogen version

```
usage: cryptogen version
```

Show version information

Flags:

```
--help  Show context-sensitive help (also try --help-long and --help-man).
```

11.11.7 Usage

Here's an example using the different available flags on the `cryptogen extend` command.

```
cryptogen extend --input="crypto-config" --config=config.yaml
org3.example.com
```

Where `config.yaml` adds a new peer organization called `org3.example.com`

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.12 Service Discovery CLI

The discovery service has its own Command Line Interface (CLI) which uses a YAML configuration file to persist properties such as certificate and private key paths, as well as MSP ID.

The `discover` command has the following subcommands:

- `saveConfig`
- `peers`
- `config`
- `endorsers`

And the usage of the command is shown below:

```
usage: discover [<flags>] <command> [<args> ...]

Command line client for fabric discovery service

Flags:
  --help                Show context-sensitive help (also try --help-long and --
↪help-man) .
  --configFile=CONFIGFILE  Specifies the config file to load the configuration from
  --peerTLSCA=PEERTLSCA   Sets the TLS CA certificate file path that verifies the
↪TLS peer's certificate
  --tlsCert=TLSCERT        (Optional) Sets the client TLS certificate file path that
↪is used when the peer enforces client authentication
  --tlsKey=TLSKEY          (Optional) Sets the client TLS key file path that is used
↪when the peer enforces client authentication
  --userKey=USERKEY        Sets the user's key file path that is used to sign
↪messages sent to the peer
  --userCert=USERCERT      Sets the user's certificate file path that is used to
↪authenticate the messages sent to the peer
  --MSP=MSP                Sets the MSP ID of the user, which represents the CA(s)
↪that issued its user certificate

Commands:
  help [<command>...]
    Show help.

  peers [<flags>]
    Discover peers

  config [<flags>]
    Discover channel config

  endorsers [<flags>]
    Discover chaincode endorsers

  saveConfig
    Save the config passed by flags into the file specified by --configFile
```

11.12.1 Configuring external endpoints

Currently, to see peers in service discovery they need to have `EXTERNAL_ENDPOINT` to be configured for them. Otherwise, Fabric assumes the peer should not be disclosed.

To define these endpoints, you need to specify them in the `core.yaml` of the peer, replacing the sample endpoint below with the ones of your peer.

```
CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer1.org1.example.com:8051
```

11.12.2 Persisting configuration

To persist the configuration, a config file name should be supplied via the flag `--configFile`, along with the command `saveConfig`:

```
discover --configFile conf.yaml --peerTLSCA tls/ca.crt --userKey msp/keystore/
↪ea4f6a38ac7057b6fa9502c2f5f39f182e320f71f667749100fe7dd94c23ce43_sk --userCert msp/
↪signcerts/User1\@org1.example.com-cert.pem --MSP Org1MSP saveConfig
```

By executing the above command, configuration file would be created:

```
$ cat conf.yaml
version: 0
tlsconfig:
  certpath: ""
  keypath: ""
  peercacertpath: /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/users/User1@org1.example.com/tls/ca.crt
  timeout: 0s
signerconfig:
  mspid: Org1MSP
  identitypath: /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/users/User1@org1.example.com/msp/signcerts/
↪User1@org1.example.com-cert.pem
  keypath: /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/users/User1@org1.example.com/msp/keystore/
↪ea4f6a38ac7057b6fa9502c2f5f39f182e320f71f667749100fe7dd94c23ce43_sk
```

When the peer runs with TLS enabled, the discovery service on the peer requires the client to connect to it with mutual TLS, even if the peer has not set `tls.clientAuthRequired` to `true`.

When `tls.clientAuthRequired` is set to `false`, the peer will still request (and verify if given, but not require) client TLS certificates. Therefore if the client does not pass a TLS certificate, TLS connections can be established to the peer but will be rejected in the peer's discovery layer. To that end, the discovery CLI provides a TLS certificate on its own if the user doesn't explicitly set one.

More concretely, when the discovery CLI's config file has a certificate path for `peercacertpath`, but the `certpath` and `keypath` aren't configured as in the above - the discovery CLI generates a self-signed TLS certificate and uses this to connect to the peer.

When the `peercacertpath` isn't configured, the discovery CLI connects without TLS, and this is highly not recommended, as the information is sent over plaintext, un-encrypted.

11.12.3 Querying the discovery service

The `discoverCLI` acts as a discovery client, and it needs to be executed against a peer. This is done via specifying the `--server` flag. In addition, the queries are channel-scoped, so the `--channel` flag must be used.

The only query that doesn't require a channel is the local membership peer query, which by default can only be used by administrators of the peer being queried.

The `discover` CLI supports all server-side queries:

- Peer membership query
- Configuration query
- Endorsers query

Let's go over them and see how they should be invoked and parsed:

11.12.4 Peer membership query:

```
$ discover --configFile conf.yaml peers --channel mychannel --server peer0.org1.
↪example.com:7051
[
  {
    "MSPID": "Org2MSP",
    "LedgerHeight": 5,
    "Endpoint": "peer0.org2.example.com:9051",
    "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKTCCAC+gAwIBAgIRANK4WBck5gKuzTxVQIwhYMUwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkI
↪ecJNvdAV2zmSx5Sf2qospVAH1MYCHyudDEvkiRuBPgmCdOdWJsE0g+h\nnz0nZdKq6/
↪X+jTTBLMA4GA1UdDwEB/
↪wQEAWIHgDAMBGNVHRMBAf8EAjAAMCsGA1Ud\nIwQkMCKAIFZMuZfUtY6n2iyxaVr3rl+x5lU0CdG9x7KAeYydQGTMMaOGCCqGSM
↪LJ7j3I9NEPQ/B1BpnJP+UNPnGO2peVrM/
↪mJlnVgIgS1ZA\nA1tsxuDYllaQuHx2P+P9NDFdjXx5T08lZhXuWYM=\n-----END CERTIFICATE-----\n
↪",
    "Chaincodes": [
      "mycc"
    ]
  },
  {
    "MSPID": "Org2MSP",
    "LedgerHeight": 5,
    "Endpoint": "peer1.org2.example.com:10051",
    "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKDCCAC+gAwIBAgIRALnNJzplCrYy4Y8CjZtqL7AwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkI
↪YMnlhS6sM+bFDgkJKaIG7s9Hg3URF0aGpy51R\nu+4F9Muo+Xa jTTBLMA4GA1UdDwEB/
↪wQEAWIHgDAMBGNVHRMBAf8EAjAAMCsGA1Ud\nIwQkMCKAIFZMuZfUtY6n2iyxaVr3rl+x5lU0CdG9x7KAeYydQGTMMaOGCCqGSM
↪ExunQ==\n-----END CERTIFICATE-----\n",
    "Chaincodes": [
      "mycc"
    ]
  },
  {
    "MSPID": "Org1MSP",
    "LedgerHeight": 5,
    "Endpoint": "peer0.org1.example.com:7051",
    "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKDCCAC6gAwIBAgIQP18LeXtEXGoN8ptQzXTHZTAKBggqhkJOPQQDAjBzMQsw\nnCQYDVQQGEwJVUzETMBEGA1UECBMKQ2I
↪lRg/ynSk\nnNItaMlaCDZOaQvxJEL6o3fqx1PVFlfXE4NarY3OO1N3YZI4lhWwoXksSwJu/
↪35S\nm7wMEzw+3KNNMESwDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwKwYDVR0j\nBCQwIoAgceCTOxTes6rfgyxHH6KIW7hsRAw2bhp9ikCHkvtv/
↪RcwCgYIKoZIzj0E\nAwIDAQAARQIhAKiJEv79XBmr8gGY6kHrGL0L3sq95E7IsCYzYdAQHj+DAiBPcBTg\nRuA0/
↪Kq+3aHJ2T0KpKHqD3FfhZZolKDkcrkwQ==\n-----END CERTIFICATE-----\n",
    "Chaincodes": [
      "mycc"
    ]
  }
],
```

(continues on next page)

(continued from previous page)

```

{
  "MSPID": "Org1MSP",
  "LedgerHeight": 5,
  "Endpoint": "peer1.org1.example.com:8051",
  "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICJzCCAc6gAwIBAgIQO7zMEHlMfRhnp6Xt65jwtdAKBggqhkJOPQQDAjBzMQsw\nnCQYDVQQGEwJVUzETMBEGA1UECBMKQ2l
↪Q2g\nnRHw5rk3SYw+OMFw9jNbsJJyC5ttJRvc12Dn7lQ8ZR9hW1vLQ3NtqO/
↪couccDJcHg\nt47iHBNadaNNMEswDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwKwYDVR0j\nBCQwIoAgcecTOxTes6rfgyxHH6KIW7hsRAw2bhP9ikCHkvtv/
↪RcwCgYIKoZIzj0E\nAwIDRwAwRAIgGHGtRVxcFVeMqr9yRlebs23OXEECNo6hNqd/
↪4ChLwwoCIBFKFd6t\nlL5BVzVMGQyXWcZGrjFgl4+fDrwjmMe+jAfa\n-----END CERTIFICATE-----\n
↪",
  "Chaincodes": null
}
]

```

As seen, this command outputs a JSON containing membership information about all the peers in the channel that the peer queried possesses.

The Identity that is returned is the enrollment certificate of the peer, and it can be parsed with a combination of jq and openssl:

```

$ discover --configFile conf.yaml peers --channel mychannel --server peer0.org1.
↪example.com:7051 | jq .[0].Identity | sed "s/\\n/\\n/g" | sed "s/\\/\\/g" | openssl
↪x509 -text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      55:e9:3f:97:94:d5:74:db:e2:d6:99:3c:01:24:be:bf
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: C=US, ST=California, L=San Francisco, O=org2.example.com, CN=ca.org2.
↪example.com
    Validity
      Not Before: Jun  9 11:58:28 2018 GMT
      Not After : Jun  6 11:58:28 2028 GMT
    Subject: C=US, ST=California, L=San Francisco, OU=peer, CN=peer0.org2.example.
↪com
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
      Public-Key: (256 bit)
      pub:
        04:f5:69:7a:11:65:d9:85:96:65:b7:b7:1b:08:77:
        43:de:cb:ad:3a:79:ec:cc:2a:bc:d7:93:68:ae:92:
        1c:4b:d8:32:47:d6:3d:72:32:f1:f1:fb:26:e4:69:
        c2:eb:c9:45:69:99:78:d7:68:a9:77:09:88:c6:53:
        01:2a:c1:f8:c0
      ASN1 OID: prime256v1
      NIST CURVE: P-256
    X509v3 extensions:
      X509v3 Key Usage: critical
        Digital Signature
      X509v3 Basic Constraints: critical
        CA:FALSE
      X509v3 Authority Key Identifier:
↪keyid:8E:58:82:C9:0A:11:10:A9:0B:93:03:EE:A0:54:42:F4:A3:EF:11:4C:82:B6:F9:CE:10:A2:1E:24:AB:13:82

```

(continues on next page)

(continued from previous page)

```
Signature Algorithm: ecdsa-with-SHA256
30:44:02:20:29:3f:55:2b:9f:7b:99:b2:cb:06:ca:15:3f:93:
a1:3d:65:5c:7b:79:a1:7a:d1:94:50:f0:cd:db:ea:61:81:7a:
02:20:3b:40:5b:60:51:3c:f8:0f:9b:fc:ae:fc:21:fd:c8:36:
a3:18:39:58:20:72:3d:1a:43:74:30:f3:56:01:aa:26
```

11.12.5 Configuration query:

The configuration query returns a mapping from MSP IDs to orderer endpoints, as well as the FabricMSPConfig which can be used to verify all peer and orderer nodes by the SDK:

```
$ discover --configFile conf.yaml config --channel mychannel --server peer0.org1.
↪example.com:7051
{
  "msps": {
    "OrdererOrg": {
      "name": "OrdererMSP",
      "root_certs": [
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tCk1JSUNMekNDQWRhZ0F3SUJBZ01lSQU1pWkxUb3RmMHR6VTRzNUdIdkQ0UjR3Q0
↪"
        ],
        "admins": [
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tCk1JSUNDVENDQWJDZ0F3SUJBZ01RR2wzTjhaSzMDeDkRRQmZqYVpwMVY5VEFLQ0
↪"
        ],
        "crypto_config": {
          "signature_hash_family": "SHA2",
          "identity_identifier_hash_function": "SHA256"
        },
        "tls_root_certs": [
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tCk1JSUNORENDQWR1Z0F3SUJBZ01RZDdodzFiaHNZTXI2a25ETWJrZThTakFLQ0
↪"
        ]
      },
      "Org1MSP": {
        "name": "Org1MSP",
        "root_certs": [
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ01lSQU1nN2VETnhwS0t0ZG10TDVNDVNDZ01RZDdodzFiaHNZTXI2a25ETWJrZThTakFLQ0
↪"
        ],
        "admins": [
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tCk1JSUNLakNDQWRDZ0F3SUJBZ01RRTRFRk0tqSHgwdTlZRSsxZUgrLldOakFLQ0
↪"
        ],
        "crypto_config": {
          "signature_hash_family": "SHA2",
          "identity_identifier_hash_function": "SHA256"
        },
        "tls_root_certs": [
```

(continues on next page)

(continued from previous page)

```

↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNTVEVENDQWUxZ0F3SUJBZ01RZlRWTE9iTENVUjdxVEY3Z283UXgvakFLQr
↪ "
    ],
    "fabric_node_ous": {
        "enable": true,
        "client_ou_identifier": {
            "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ01SQUlnN2VETnhws0t0ZG10TDRVNDRZMU13Q2
↪ ",
            "organizational_unit_identifier": "client"
        },
        "peer_ou_identifier": {
            "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ01SQUlnN2VETnhws0t0ZG10TDRVNDRZMU13Q2
↪ ",
            "organizational_unit_identifier": "peer"
        }
    },
    "Org2MSP": {
        "name": "Org2MSP",
        "root_certs": [
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ01SQUx2SWV2KzE4Vm9LZFR2V1RLNCTaZ2d3Q2
↪ "
        ],
        "admins": [
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNLVENDQWRDZ0F3SUJBZ01RU11peE1vdmpoM1N2c25WbmFmFUOX11REFLQr
↪ "
        ],
        "crypto_config": {
            "signature_hash_family": "SHA2",
            "identity_identifier_hash_function": "SHA256"
        },
        "tls_root_certs": [
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNTakNDQWZDZ0F3SUJBZ01SQUtoUFFxUGZSYnVpSktqL0JRanQ3RXN3Q2
↪ "
        ],
        "fabric_node_ous": {
            "enable": true,
            "client_ou_identifier": {
                "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ01SQUx2SWV2KzE4Vm9LZFR2V1RLNCTaZ2d3Q2
↪ ",
                "organizational_unit_identifier": "client"
            },
            "peer_ou_identifier": {
                "certificate":
↪ "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ01SQUx2SWV2KzE4Vm9LZFR2V1RLNCTaZ2d3Q2
↪ ",
                "organizational_unit_identifier": "peer"
            }
        }
    },
    ],

```

(continues on next page)

(continued from previous page)

```

    "Org3MSP": {
      "name": "Org3MSP",
      "root_certs": [
        "CgJPVQoEUm9sZQoMRW5yb2xsbWVudE1EChBSZXZvY2F0aW9uSGFuZGx1EkQKIKoEXcq/
→psdYnMKCiT79N+dS1hM8k+SuzU1blOgTuN++EiBe2m3E+FjWLuQGMNRGRrEVTMqTvC4A/
→5jvCLv2jaIsZxpECiDBbIOkwetxAwFzHwb1hi8T1kGW3OofvuVzfFt9VlewcRigyvsxG5/
→THdWYKJTdNx8Gle2hoCbVF0Y1/
→DQESBjGOGciRAog25fMyWps+FL0jzjlvIsGUyO457ri3YMvmUcycIH2FvQSICTtzaFvSPUiDtNtAVz+uetuB9kfmjUdUSQxjyXU
→uaJMuVph7Dy/
→icgnAtVYHShET4100Eh3Q5BIgy5q9VMQrch9VW5yajaY8dH1uA593gKd5kBqGdLfiXzAiRAogAnUYq/
→kwKzFfmIm/
→W4nZxi1kjG2C8NRjsYYBkeAOQ6wSIGyX5GGmwgvxgXXehNWBfijyNIJALGRVhO8YtBqr+vnrKogBCiDHRlXQsDbpcBoZFJ09V9
→wwC+tg3oBIgSWT/
→peiO2BI0DecypKfgMpVR8DWXl8ZHSrPIssL3Mc8aINem9+BOezLwFKCbtVH1KAHIRLyYiNP+TkIKW6x9RkThIiAbIJCYU6002E
→1lHxV0vtWdIsKCTLx2EZmDJECiCPXeyUyFzPS3iFv8CQUOLCPZxf6buZS5JlM6EE/
→gCRaxIgmF9GKPLLMeoA77+AU3J8Iwnu9pBxnaHtUlyf/F9p30c6RAogG7ENKWlOZ4aF0HprqXAjl++Iao7/
→iE8xeVcKRlmg1ASIGtmmavDAVS2bw3zClQd4ZBD2DrqCBO9NPocLNB0IWeIQiCjxTdbmcuBNINZYWe+5fWyI1oY9LavKzDVkd
→PRAmBaeTQLXdbMxIthxM2gw+Zkc5+IJEWX"
      ],
      "intermediate_certs": [
        "CtgCCkQKIP0UVivtH8NlnRNrZuuu6jpaj2ZbEB4/
→secGS57MfbINEiDSJweLUMIQSW12jugBQG811IQflJWvi7vi925u+PU/
→+xJECiDgOGdNbAiGSoHmTjKhT22fqUqYLIVh+JBHetm4kf4skhIg9XTWRkUqtsfYKENzPgm7ZUSmCHNF8xH7Vnhuc1EpAUgaIN
→cIiCnlRj+mfNVAJGKthLgQBB/
→JKM14NbUeutyJtTgrmDDiCogme25qGvxJfgQNNzldMMicVyiI6YMfnoThAUyqsTzyXkqIAAAAAAAAAAAAAAAAAAAAAAAAAA
→El+BA2fOV0wyJxXKIjpgx+ehOEBTKqxPPJeSogAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAESIFYUenRvjbmEh+37
→gwzULTJbCAoVg9XfCiROs4cU5oSv4Q80iYWtonAnvsSIE6mYFdzisBU21rhxjfYE7kk3Xjih9A1idJp7TSjfmorGiBwIEbnxUK
→yIgBVTjvNOIwpBC7qZJKX6yn4tMvoCCGpiz4BKBEUqtBjsaZzBlajBwZ4WXYOttkhsNA2r94gBfLUdx/
→4VhW4hwUImcztlau1T14U1NzJolCNkdiLc9CqsCMQD6OBkgDWGq9UlhK9dJBzU+RElcZdSfVV1hDbbqt+lFRWOzzEkZ+BXCRI
→"
      ],
      "admins": [
        "LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUhZd0VBWUhlb1pJemowQ0FRWUZLNEVFQUNJRFlnQUVUYk13SEZteEpEMWR3S
→"
      ]
    },
    "orderers": {
      "OrdererOrg": {
        "endpoint": [
          {
            "host": "orderer.example.com",
            "port": 7050
          }
        ]
      }
    }
  }
}

```

It's important to note that the certificates here are base64 encoded, and thus should be decoded in a manner similar to the following:

```

$ discover --configFile conf.yaml config --channel mychannel --server peer0.org1.
→example.com:7051 | jq .msps.OrdererOrg.root_certs[0] | sed "s/\\/\\/g" | base64 --
→decode | openssl x509 -text -noout

```

(continues on next page)

(continued from previous page)

```

Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      c8:99:2d:3a:2d:7f:4b:73:53:8b:39:18:7b:c3:e1:1e
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: C=US, ST=California, L=San Francisco, O=example.com, CN=ca.example.com
    Validity
      Not Before: Jun  9 11:58:28 2018 GMT
      Not After : Jun  6 11:58:28 2028 GMT
    Subject: C=US, ST=California, L=San Francisco, O=example.com, CN=ca.example.
↪com

  Subject Public Key Info:
    Public Key Algorithm: id-ecPublicKey
    Public-Key: (256 bit)
    pub:
      04:28:ac:9e:51:8d:a4:80:15:0a:ff:ae:c9:61:d6:
      08:67:b0:15:c3:c7:99:46:61:63:0a:10:a6:42:6a:
      b0:af:14:0c:c0:e2:5b:b4:a1:c3:f0:07:7e:5b:7c:
      c4:b2:95:13:95:81:4b:6a:b9:e3:87:a4:f3:2c:7c:
      ae:00:91:9e:32
    ASN1 OID: prime256v1
    NIST CURVE: P-256
  X509v3 extensions:
    X509v3 Key Usage: critical
      Digital Signature, Key Encipherment, Certificate Sign, CRL Sign
    X509v3 Extended Key Usage:
      Any Extended Key Usage
    X509v3 Basic Constraints: critical
      CA:TRUE
    X509v3 Subject Key Identifier:
      ↪
↪60:9D:F2:30:26:CE:8F:65:81:41:AD:96:15:0E:24:8D:A0:9D:C5:79:C1:17:BF:FE:E5:1B:FB:75:50:10:A6:4C
  Signature Algorithm: ecdsa-with-SHA256
    30:44:02:20:3d:e1:a7:6c:99:3f:87:2a:36:44:51:98:37:11:
    d8:a0:47:7a:33:ff:30:c1:09:a6:05:ec:b0:53:53:39:c1:0e:
    02:20:6b:f4:1d:48:e0:72:e4:c2:ef:b0:84:79:d4:2e:c2:c5:
    1b:6f:e4:2f:56:35:51:18:7d:93:51:86:05:84:ce:1f

```

11.12.6 Endorsers query:

To query for the endorsers of a chaincode call, additional flags need to be supplied:

- The `--chaincode` flag is mandatory and it provides the chaincode name(s). To query for a chaincode-to-chaincode invocation, one needs to repeat the `--chaincode` flag with all the chaincodes.
- The `--collection` is used to specify private data collections that are expected to be used by the chaincode(s). To map from the chaincodes passed via `--chaincode` to the collections, the following syntax should be used: `collection=CC:Collection1,Collection2,...`
- The `--noPrivateReads` is used to indicate that the transaction is not expected to read private data for a certain chaincode. This is useful for private data “blind writes”, among other things.

For example, to query for a chaincode invocation that results in both `cc1` and `cc2` to be invoked, as well as writes to private data collection `coll` by `cc2`, one needs to specify: `--chaincode=cc1 --chaincode=cc2 --collection=cc2:coll`

If chaincode cc2 is not expected to read from collection coll then --noPrivateReads=cc2 should be used.

Below is the output of an endorsers query for chaincode **mycc** when the endorsement policy is AND ('Org1.peer' , 'Org2.peer'):

```
$ discover --configFile conf.yaml endorsers --channel mychannel --server peer0.org1.
example.com:7051 --chaincode mycc
[
  {
    "Chaincode": "mycc",
    "EndorsersByGroups": {
      "G0": [
        {
          "MSPID": "Org1MSP",
          "LedgerHeight": 5,
          "Endpoint": "peer0.org1.example.com:7051",
          "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKDCCAcgAwIBAgIRANTiKfUVHVGrYVzEy1ZSKIwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAGTCk1
↪k\n/CtORCDPQ02jTTBLMA4GA1UdDwEB/
↪wQEAWIHgDAMBgNVHRMBAf8EAjAAMCsGA1Ud\nIiwQkMCKAIOBdQLF+cMWA6elp2CpOEx7SHUinzVvd55hLm7w6v72oMAoGCCqGSI
↪zwD08t7hJxNe8MwgP8/48fAiBiC0cr\nu99oLsRNCFB7R3egyKg1YYao0KWTrr1T+rK9Bg==\n-----END
↪CERTIFICATE-----\n"
        },
        {
          "MSPID": "Org2MSP",
          "LedgerHeight": 5,
          "Endpoint": "peer1.org2.example.com:10051",
          "Identity": "-----BEGIN CERTIFICATE-----\nMIICKDCCAcgAwIBAgIRAIIs6fFxx4Y5cJxSwTjyJ9A8wCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAGTCk1
↪cQ\n0cGrMKR93vKjTTBLMA4GA1UdDwEB/
↪wQEAWIHgDAMBgNVHRMBAf8EAjAAMCsGA1Ud\nIiwQkMCKAII5YgskKERCPc5MD7qBUQvSj7xFMgrb5zhCiHiSrE4KgMAoGCCqGSI
↪OidQ2SBR7OZyMAZgXc5nAabWZpdkuQ==\n-----END CERTIFICATE-----\n"
        },
        {
          "MSPID": "Org2MSP",
          "LedgerHeight": 5,
          "Endpoint": "peer0.org2.example.com:9051",
          "Identity": "-----BEGIN CERTIFICATE-----\nMIICJzCCAc6gAwIBAgIQVek/
↪15TVdNvilpk8ASS+vzAKBgqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMQ2FsaWZvcm5pYTEwMBQGA1UEBxMN
↪BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwKwYDVR0j\nBCQwIoAgjliCyQoREKkLkwPuoFRC9KPvEUyCtvnOEKIEJKsTgqAwCgYIKoZIzj0E\nAwIDRwAwRAIgKT9
↪yu/CH9yDajGD1YIH19Gkn0MPNWAaom\n-----END CERTIFICATE-----\n"
        }
      ],
      "Layouts": [
        {
          "quantities_by_group": {
            "G0": 1,
            "G1": 1
          }
        }
      ]
    }
  }
]
```

11.12.7 Not using a configuration file

It is possible to execute the discovery CLI without having a configuration file, and just passing all needed configuration as commandline flags. The following is an example of a local peer membership query which loads administrator credentials:

```
$ discover --peerTLSCA tls/ca.crt --userKey msp/keystore/
↪ cf31339d09e8311ac9ca5ed4e27a104a7f82f1e5904b3296a170ba4725ffde0d_sk --userCert msp/
↪ signcerts/Admin\@org1.example.com-cert.pem --MSP Org1MSP --tlsCert tls/client.crt --
↪ tlsKey tls/client.key peers --server peer0.org1.example.com:7051
[
  {
    "MSPID": "Org1MSP",
    "Endpoint": "peer1.org1.example.com:8051",
    "Identity": "-----BEGIN CERTIFICATE-----
↪ \nMIICJzCCAc6gAwIBAgIQO7zMEH1MfRhnP6Xt65jwtDAKBggqhkJOPQQDAjBzMQsw\nnCQYDVQQGEwJVUzETMBEGA1UECBMKQ2l
↪ Q2g\nnRHw5rk3SYw+OMFw9jNbsJJyC5ttJRvc12Dn7lQ8ZR9hW1vLQ3NtqO/
↪ couccDJChg\nt47iHBNadaNNMESwDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪ wQCMAAwKwYDVR0j\nBCQwIoAgcectOxTes6rfgyxHH6KIW7hsRAw2bhp9ikCHkvtv/
↪ RcwCgYIKoZIzj0E\nAwIDRwAwRAIgGHGTRVxcFVeMQR9yRlebs23OXEECNo6hNqd/
↪ 4ChLwwoCIBFKFd6t\nlL5BVzVMGQyXWcZGrjFgl4+fDrwjmMe+jAfa\n-----END CERTIFICATE-----\n
↪ ",
    },
  },
  {
    "MSPID": "Org1MSP",
    "Endpoint": "peer0.org1.example.com:7051",
    "Identity": "-----BEGIN CERTIFICATE-----
↪ \nMIICKDCCAc6gAwIBAgIQP18LeXtEXGoN8pTqzXTHZTAKBggqhkJOPQQDAjBzMQsw\nnCQYDVQQGEwJVUzETMBEGA1UECBMKQ2l
↪ 1Rg\ynSk\nnNNItaMlaCDZoAQvxJEl6o3fqxlPVFlfXE4NarY3OO1N3YZI4lhWwoXksSwJu/
↪ 35S\nm7wMEzw+3KNNMESwDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪ wQCMAAwKwYDVR0j\nBCQwIoAgcectOxTes6rfgyxHH6KIW7hsRAw2bhp9ikCHkvtv/
↪ RcwCgYIKoZIzj0E\nAwIDSAAARQIhAKiJEv79XBmr8gGY6kHrGL0L3sq95E7IsCYzYdAQHj+DAiBPcBTg\nRuA0/
↪ /Kq+3aHJ2T0KpKHqD3FfhZz0lKDkcrkwQ==\n-----END CERTIFICATE-----\n",
    },
  },
  {
    "MSPID": "Org2MSP",
    "Endpoint": "peer0.org2.example.com:9051",
    "Identity": "-----BEGIN CERTIFICATE-----
↪ \nMIICKTCCAc+gAwIBAgIRANK4WBck5gKuzTxVQIwhYMUwCgYIKoZIzj0EAwIwcZEL\nnMAkGA1UEBhMCVVMxEzARBgNVBAgTCKI
↪ ecJNvdAV2zmSx5Sf2qospVAH1MYCHyudDEvkiRuBPgmCdOdwJsE0g+h\nnz0nZdKq6/
↪ X+jTBLMA4GA1UdDwEB/
↪ wQEAwIHgDAMBGNVHRMBAf8EAjAAMCsGA1Ud\nnIwQkMCKAIFZMuZfUtY6n2iyxaVr3rl+x5lU0CdG9x7KAeYydQGTMMaoGCCqGS
↪ LJ7j3I9NEPQ/B1BpnJP+UNPnGO2peVrM/
↪ mJlnVgIgS1ZA\nAltsxuDYllaQuHx2P+P9NDFdjXx5T08lZhXuWYM=\n-----END CERTIFICATE-----\n
↪ ",
    },
  },
  {
    "MSPID": "Org2MSP",
    "Endpoint": "peer1.org2.example.com:10051",
    "Identity": "-----BEGIN CERTIFICATE-----
↪ \nMIICKDCCAc+gAwIBAgIRALnNJzplCrYy4Y8CjZtqL7AwCgYIKoZIzj0EAwIwcZEL\nnMAkGA1UEBhMCVVMxEzARBgNVBAgTCKI
↪ YMn1hS6sM+bFDgkJKa1G7s9Hg3URF0aGpy51R\nnU+4F9Muo+XajTBLMA4GA1UdDwEB/
↪ wQEAwIHgDAMBGNVHRMBAf8EAjAAMCsGA1Ud\nnIwQkMCKAIFZMuZfUtY6n2iyxaVr3rl+x5lU0CdG9x7KAeYydQGTMMaoGCCqGS
↪ ExunQ==\n-----END CERTIFICATE-----\n",
    },
  }
]
```


11.13 Fabric-CA Commands

The Hyperledger Fabric CA is a Certificate Authority (CA) for Hyperledger Fabric. The commands available for the fabric-ca client and fabric-ca server are described in the links below.

11.13.1 Fabric-CA Client

The fabric-ca-client command allows you to manage identities (including attribute management) and certificates (including renewal and revocation).

More information on `fabric-ca-client` commands can be found [here](#).

11.13.2 Fabric-CA Server

The fabric-ca-server command allows you to initialize and start a server process which may host one or more certificate authorities.

More information on `fabric-ca-server` commands can be found [here](#).

12.1 Hyperledger Fabric SDKs

Hyperledger Fabric offers a number of SDKs for a wide variety of programming languages. The first three delivered are the Node.js, Java, and Go SDKs. We hope to provide Python, and REST SDKs in a subsequent release.

- [Hyperledger Fabric Node SDK documentation.](#)
- [Hyperledger Fabric Java SDK documentation.](#)
- [Hyperledger Fabric Go SDK documentation.](#)

12.2 Transaction Flow

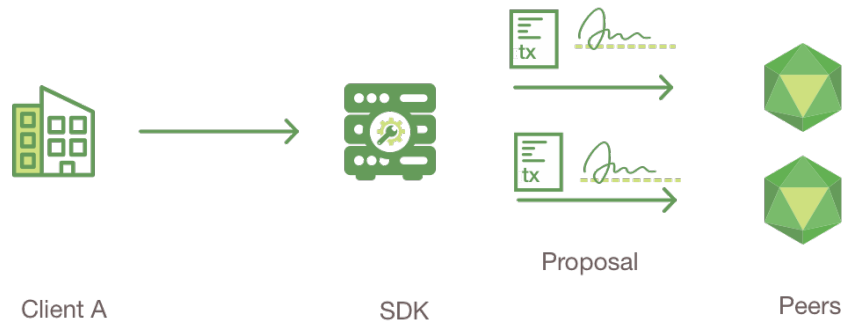
This document outlines the transactional mechanics that take place during a standard asset exchange. The scenario includes two clients, A and B, who are buying and selling radishes. They each have a peer on the network through which they send their transactions and interact with the ledger.



Assumptions

This flow assumes that a channel is set up and running. The application user has registered and enrolled with the organization's Certificate Authority (CA) and received back necessary cryptographic material, which is used to authenticate to the network.

The chaincode (containing a set of key value pairs representing the initial state of the radish market) is installed on the peers and deployed to the channel. The chaincode contains logic defining a set of transaction instructions and the agreed upon price for a radish. An endorsement policy has also been set for this chaincode, stating that both `peerA` and `peerB` must endorse any transaction.

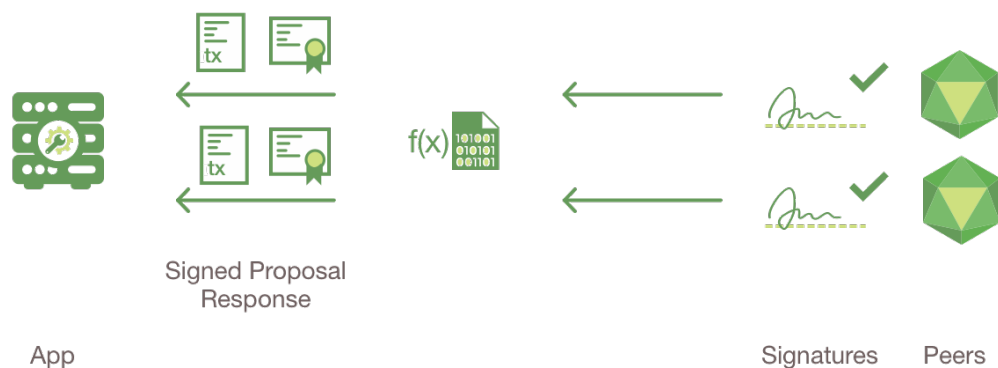


1. Client A initiates a transaction

What's happening? Client A is sending a request to purchase radishes. This request targets `peerA` and `peerB`, who are respectively representative of Client A and Client B. The endorsement policy states that both peers must endorse any transaction, therefore the request goes to `peerA` and `peerB`.

Next, the transaction proposal is constructed. An application leveraging a supported SDK (Node, Java, Python) utilizes one of the available API's to generate a transaction proposal. The proposal is a request to invoke a chaincode function with certain input parameters, with the intent of reading and/or updating the ledger.

The SDK serves as a shim to package the transaction proposal into the properly architected format (protocol buffer over gRPC) and takes the user's cryptographic credentials to produce a unique signature for this transaction proposal.



2. Endorsing peers verify signature & execute the transaction

The endorsing peers verify (1) that the transaction proposal is well formed, (2) it has not been submitted already in the past (replay-attack protection), (3) the signature is valid (using the MSP), and (4) that the submitter (Client A, in the example) is properly authorized to perform the proposed operation on that channel (namely, each endorsing peer ensures that the submitter satisfies the channel's *Writers* policy). The endorsing peers take the transaction proposal inputs as arguments to the invoked chaincode's function. The chaincode is then executed against the current state database to produce transaction results including a response value, read set, and write set (i.e. key/value pairs representing an asset to create or update). No updates are made to the ledger at this point. The set of these values, along with the endorsing peer's signature is passed back as a "proposal response" to the SDK which parses the payload for

the application to consume.

Note: The MSP is a peer component that allows peers to verify transaction requests arriving from clients and to sign transaction results (endorsements). The writing policy is defined at channel creation time and determines which users are entitled to submit a transaction to that channel. For more information about membership, check out our [Membership Service Provider \(MSP\)](#) documentation.



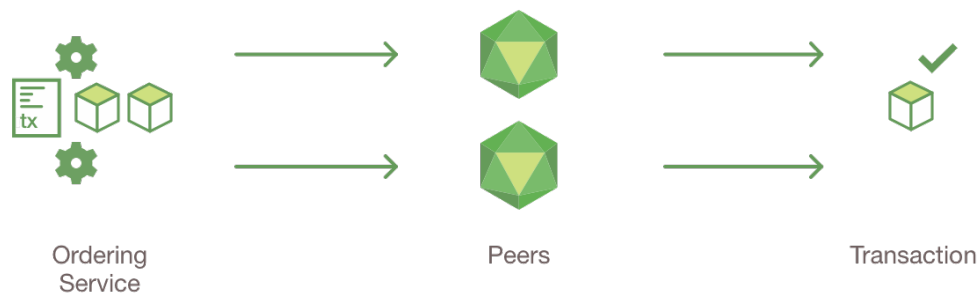
3. Proposal responses are inspected

The application verifies the endorsing peer signatures and compares the proposal responses to determine if the proposal responses are the same. If the chaincode is only querying the ledger, the application would only inspect the query response and would typically not submit the transaction to the ordering service. If the client application intends to submit the transaction to the ordering service to update the ledger, the application determines if the specified endorsement policy has been fulfilled before submitting (i.e. did peerA and peerB both endorse). The architecture is such that even if an application chooses not to inspect responses or otherwise forwards an unendorsed transaction, the endorsement policy will still be enforced by peers and upheld at the commit validation phase.



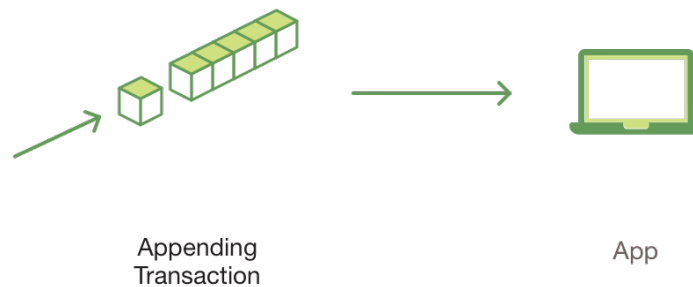
4. Client assembles endorsements into a transaction

The application “broadcasts” the transaction proposal and response within a “transaction message” to the ordering service. The transaction will contain the read/write sets, the endorsing peers signatures and the Channel ID. The ordering service does not need to inspect the entire content of a transaction in order to perform its operation, it simply receives transactions from all channels in the network, orders them chronologically by channel, and creates blocks of transactions per channel.



5. Transaction is validated and committed

The blocks of transactions are “delivered” to all peers on the channel. The transactions within the block are validated to ensure endorsement policy is fulfilled and to ensure that there have been no changes to ledger state for read set variables since the read set was generated by the transaction execution. Transactions in the block are tagged as being valid or invalid.

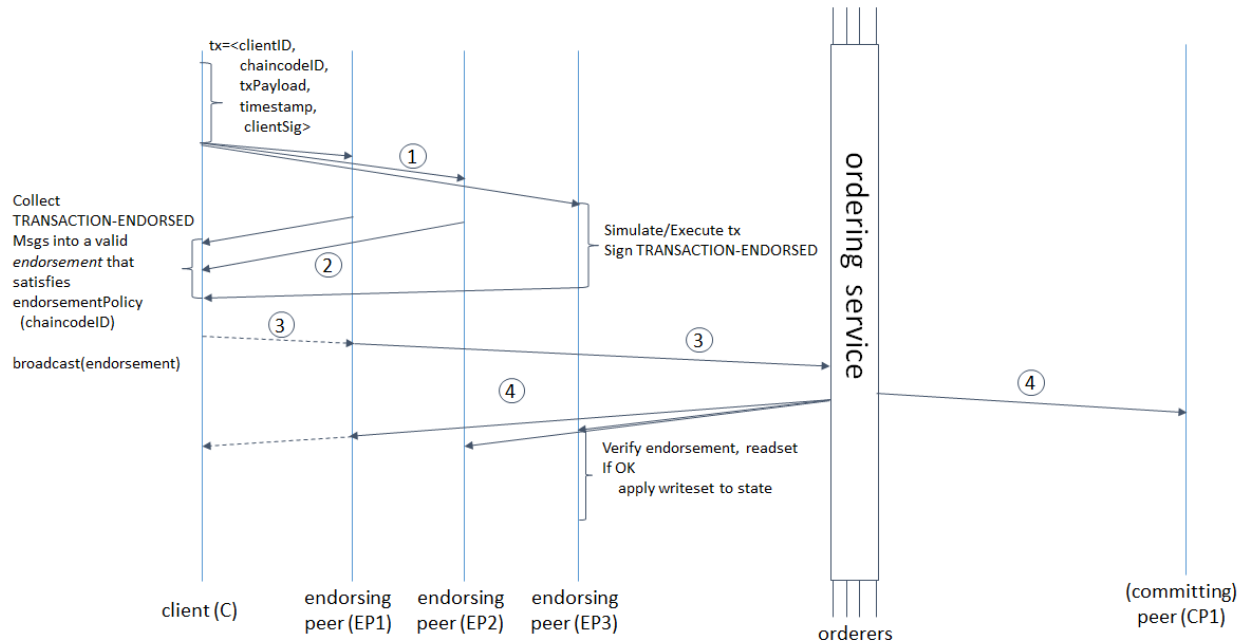


6. Ledger updated

Each peer appends the block to the channel’s chain, and for each valid transaction the write sets are committed to current state database. An event is emitted by each peer to notify the client application that the transaction (invocation) has been immutably appended to the chain, as well as notification of whether the transaction was validated or invalidated.

Note: Applications should listen for the transaction event after submitting a transaction, for example by using the `submitTransaction` API, which automatically listen for transaction events. Without listening for transaction events, you will not know whether your transaction has actually been ordered, validated, and committed to the ledger.

You can also use the swimlane sequence diagram below to examine the transaction flow in more detail.



12.3 Service Discovery

12.3.1 Why do we need service discovery?

In order to execute chaincode on peers, submit transactions to orderers, and to be updated about the status of transactions, applications connect to an API exposed by an SDK.

However, the SDK needs a lot of information in order to allow applications to connect to the relevant network nodes. In addition to the CA and TLS certificates of the orderers and peers on the channel – as well as their IP addresses and port numbers – it must know the relevant endorsement policies as well as which peers have the chaincode installed (so the application knows which peers to send chaincode proposals to).

Prior to v1.2, this information was statically encoded. However, this implementation is not dynamically reactive to network changes (such as the addition of peers who have installed the relevant chaincode, or peers that are temporarily offline). Static configurations also do not allow applications to react to changes of the endorsement policy itself (as might happen when a new organization joins a channel).

In addition, the client application has no way of knowing which peers have updated ledgers and which do not. As a result, the application might submit proposals to peers whose ledger data is not in sync with the rest of the network, resulting in transaction being invalidated upon commit and wasting resources as a consequence.

The **discovery service** improves this process by having the peers compute the needed information dynamically and present it to the SDK in a consumable manner.

12.3.2 How service discovery works in Fabric

The application is bootstrapped knowing about a group of peers which are trusted by the application developer/administrator to provide authentic responses to discovery queries. A good candidate peer to be used by the client application is one that is in the same organization. Note that in order for peers to be known to the discovery service, they must have an `EXTERNAL_ENDPOINT` defined. To see how to do this, check out our [Service Discovery CLI](#) documentation.

The application issues a configuration query to the discovery service and obtains all the static information it would have otherwise needed to communicate with the rest of the nodes of the network. This information can be refreshed at any point by sending a subsequent query to the discovery service of a peer.

The service runs on peers – not on the application – and uses the network metadata information maintained by the gossip communication layer to find out which peers are online. It also fetches information, such as any relevant endorsement policies, from the peer’s state database.

With service discovery, applications no longer need to specify which peers they need endorsements from. The SDK can simply send a query to the discovery service asking which peers are needed given a channel and a chaincode ID. The discovery service will then compute a descriptor comprised of two objects:

1. **Layouts:** a list of groups of peers and a corresponding amount of peers from each group which should be selected.
2. **Group to peer mapping:** from the groups in the layouts to the peers of the channel. In practice, each group would most likely be peers that represent individual organizations, but because the service API is generic and ignorant of organizations this is just a “group”.

The following is an example of a descriptor from the evaluation of a policy of `AND (Org1, Org2)` where there are two peers in each of the organizations.

```
Layouts: [  
  QuantitiesByGroup: {  
    "Org1": 1,  
    "Org2": 1,  
  }  
],  
EndorsersByGroups: {  
  "Org1": [peer0.org1, peer1.org1],  
  "Org2": [peer0.org2, peer1.org2]  
}
```

In other words, the endorsement policy requires a signature from one peer in Org1 and one peer in Org2. And it provides the names of available peers in those orgs who can endorse (`peer0` and `peer1` in both Org1 and in Org2).

The SDK then selects a random layout from the list. In the example above, the endorsement policy is Org1 AND Org2. If instead it was an OR policy, the SDK would randomly select either Org1 or Org2, since a signature from a peer from either Org would satisfy the policy.

After the SDK has selected a layout, it selects from the peers in the layout based on a criteria specified on the client side (the SDK can do this because it has access to metadata like ledger height). For example, it can prefer peers with higher ledger heights over others – or to exclude peers that the application has discovered to be offline – according to the number of peers from each group in the layout. If no single peer is preferable based on the criteria, the SDK will randomly select from the peers that best meet the criteria.

Capabilities of the discovery service

The discovery service can respond to the following queries:

- **Configuration query:** Returns the `MSPConfig` of all organizations in the channel along with the orderer endpoints of the channel.
- **Peer membership query:** Returns the peers that have joined the channel.
- **Endorsement query:** Returns an endorsement descriptor for given chaincode(s) in a channel.
- **Local peer membership query:** Returns the local membership information of the peer that responds to the query. By default the client needs to be an administrator for the peer to respond to this query.

Special requirements

When the peer is running with TLS enabled the client must provide a TLS certificate when connecting to the peer. If the peer isn't configured to verify client certificates (`clientAuthRequired` is false), this TLS certificate can be self-signed.

12.4 Defining capability requirements

Note: This topic describes a network that does not use a “system channel”, a channel that the ordering service is bootstrapped with and the ordering service exclusively controls. Since the release of v2.3, using a system channel is now considered the legacy process as compared to the process to create `channel_participation`. For a version of this topic that includes information about the system channel, check out [Capability requirements](#) from the v2.2 documentation.

As discussed in [Channel capabilities](#), capability requirements are defined per channel in the channel configuration (found in the channel's most recent configuration block). The channel configuration contains three locations, each of which defines a capability of a different type.

Capability Type	Canonical Path	JSON Path
Channel	/Channel/Capabilities	.channel_group.values.Capabilities
Orderer	/Channel/Orderer/Capabilities	.channel_group.groups.Orderer.values.Capabilities
Application	/Channel/Application/Capabilities	.channel_group.groups.Application.values.Capabilities

12.4.1 Setting Capabilities

Capabilities are set as part of the channel configuration (either as part of the initial configuration – which we'll talk about in a moment – or as part of a reconfiguration).

Note: For more information about how to update a channel configuration, check out [Updating a channel configuration](#).

Capabilities in an Initial Configuration

In the `configtx.yaml` file distributed in the `config` directory of the release artifacts, there is a `Capabilities` section which enumerates the possible capabilities for each capability type (Channel, Orderer, and Application).

Note that there is a `Capabilities` section defined at the root level (for the channel capabilities), and at the Orderer level (for orderer capabilities).

12.5 Channels

A Hyperledger Fabric `channel` is a private “subnet” of communication between two or more specific network members, for the purpose of conducting private and confidential transactions. A channel is defined by members (organizations), anchor peers per member, the shared ledger, chaincode application(s) and the ordering service node(s). Each transaction on the network is executed on a channel, where each party must be authenticated and authorized to transact on that channel. Each peer that joins a channel, has its own identity given by a membership services provider (MSP), which authenticates each peer to its channel peers and services.

To create a new channel, the client SDK calls configuration system chaincode and references properties such as `anchor_peers`, and `members` (organizations). This request creates a `genesis_block` for the channel ledger, which stores configuration information about the channel policies, members and anchor peers. When adding a new member to an existing channel, either this genesis block, or if applicable, a more recent reconfiguration block, is shared with the new member.

Note: See the *Channel Configuration (configtx)* section for more details on the properties and proto structures of config transactions.

The election of a `leading_peer` for each member on a channel determines which peer communicates with the ordering service on behalf of the member. If no leader is identified, an algorithm can be used to identify the leader. The consensus service orders transactions and delivers them, in a block, to each leading peer, which then distributes the block to its member peers, and across the channel, using the `gossip` protocol.

Although any one anchor peer can belong to multiple channels, and therefore maintain multiple ledgers, no ledger data can pass from one channel to another. This separation of ledgers, by channel, is defined and implemented by configuration chaincode, the identity membership service and the gossip data dissemination protocol. The dissemination of data, which includes information on transactions, ledger state and channel membership, is restricted to peers with verifiable membership on the channel. This isolation of peers and ledger data, by channel, allows network members that require private and confidential transactions to coexist with business competitors and other restricted members, on the same blockchain network.

12.6 CouchDB as the State Database

12.6.1 State Database options

The current options for the peer state database are LevelDB and CouchDB. LevelDB is the default key-value state database embedded in the peer process. CouchDB is an alternative external state database. Like the LevelDB key-value store, CouchDB can store any binary data that is modeled in chaincode (CouchDB attachments are used internally for non-JSON data). As a document object store, CouchDB allows you to store data in JSON format, issue JSON queries against your data, and use indexes to support your queries.

Both LevelDB and CouchDB support core chaincode operations such as getting and setting a key (asset), and querying based on keys. Keys can be queried by range, and composite keys can be modeled to enable equivalence queries against multiple parameters. For example a composite key of `owner, asset_id` can be used to query all assets owned by a certain entity. These key-based queries can be used for read-only queries against the ledger, as well as in transactions that update the ledger.

Modeling your data in JSON allows you to issue JSON queries against the values of your data, instead of only being able to query the keys. This makes it easier for your applications and chaincode to read the data stored on the blockchain ledger. Using CouchDB can help you meet auditing and reporting requirements for many use cases that are not supported by LevelDB. If you use CouchDB and model your data in JSON, you can also deploy indexes with your chaincode. Using indexes makes queries more flexible and efficient and enables you to query large datasets from chaincode.

CouchDB runs as a separate database process alongside the peer, therefore there are additional considerations in terms of setup, management, and operations. It is a good practice to model asset data as JSON, so that you have the option to perform complex JSON queries if needed in the future.

Note: The key for a CouchDB JSON document can only contain valid UTF-8 strings and cannot begin with an underscore (“_”). Whether you are using CouchDB or LevelDB, you should avoid using U+0000 (nil byte) in keys.

JSON documents in CouchDB cannot use the following values as top level field names. These values are reserved for internal use.

- Any field beginning with an underscore, `"_"`
- `~version`

Because of these data incompatibilities between LevelDB and CouchDB, the database choice must be finalized prior to deploying a production peer. The database cannot be converted at a later time.

12.6.2 Using CouchDB from Chaincode

Reading and writing JSON data

When writing JSON data values to CouchDB (e.g. using `PutState`) and reading JSON back in later chaincode requests (e.g. using `GetState`), the format of the JSON and the order of the JSON fields are not guaranteed, based on the JSON specification. Your chaincode should therefore unmarshal the JSON before working with the data. Similarly, when marshaling JSON, utilize a library that guarantees deterministic results, so that proposed chaincode writes and responses to clients will be identical across endorsing peers (note that `Go json.Marshal()` does in fact sort keys deterministically, but in other languages you may need to utilize a canonical JSON library).

Chaincode queries

Most of the [chaincode shim APIs](#) can be utilized with either LevelDB or CouchDB state database, e.g. `GetState`, `PutState`, `GetStateByRange`, `GetStateByPartialCompositeKey`. Additionally when you utilize CouchDB as the state database and model assets as JSON in chaincode, you can perform JSON queries against the data in the state database by using the `GetQueryResult` API and passing a CouchDB query string. The query string follows the [CouchDB JSON query syntax](#).

The [asset transfer Fabric sample](#) demonstrates use of CouchDB queries from chaincode. It includes a `queryAssetsByOwner()` function that demonstrates parameterized queries by passing an owner id into chaincode. It then queries the state data for JSON documents matching the `docType` of “asset” and the owner id using the JSON query syntax:

```
{ "selector": { "docType": "asset", "owner": <OWNER_ID> } }
```

The responses to JSON queries are useful for understanding the data on the ledger. However, there is no guarantee that the result set for a JSON query will be stable between the chaincode execution and commit time. As a result, you should not use a JSON query and update the channel ledger in a single transaction. For example, if you perform a JSON query for all assets owned by Alice and transfer them to Bob, a new asset may be assigned to Alice by another transaction between chaincode execution time and commit time.

CouchDB pagination

Fabric supports paging of query results for JSON queries and key range based queries. APIs supporting pagination allow the use of page size and bookmarks to be used for both key range and JSON queries. To support efficient pagination, the Fabric pagination APIs must be used. Specifically, the CouchDB `limit` keyword will not be honored in CouchDB queries since Fabric itself manages the pagination of query results and implicitly sets the `pageSize` limit that is passed to CouchDB.

If a `pageSize` is specified using the paginated query APIs (`GetStateByRangeWithPagination()`, `GetStateByPartialCompositeKeyWithPagination()`, and `GetQueryResultWithPagination()`), a set of results (bound by the `pageSize`) will be returned to the chaincode along with a bookmark. The bookmark can

be returned from chaincode to invoking clients, which can use the bookmark in a follow on query to receive the next “page” of results.

The pagination APIs are for use in read-only transactions only, the query results are intended to support client paging requirements. For transactions that need to read and write, use the non-paginated chaincode query APIs. Within chaincode you can iterate through result sets to your desired depth.

Regardless of whether the pagination APIs are utilized, all chaincode queries are bound by `totalQueryLimit` (default 100000) from `core.yaml`. This is the maximum number of results that chaincode will iterate through and return to the client, in order to avoid accidental or malicious long-running queries.

Note: Regardless of whether chaincode uses paginated queries or not, the peer will query CouchDB in batches based on `internalQueryLimit` (default 1000) from `core.yaml`. This behavior ensures reasonably sized result sets are passed between the peer and CouchDB when executing chaincode, and is transparent to chaincode and the calling client.

An example using pagination is included in the [Using CouchDB](#) tutorial.

CouchDB indexes

Indexes in CouchDB are required in order to make JSON queries efficient and are required for any JSON query with a sort. Indexes enable you to query data from chaincode when you have a large amount of data on your ledger. Indexes can be packaged alongside chaincode in a `/META-INF/statedb/couchdb/indexes` directory. Each index must be defined in its own text file with extension `*.json` with the index definition formatted in JSON following the [CouchDB index JSON syntax](#). For example, to support the above marble query, a sample index on the `docType` and `owner` fields is provided:

```
{ "index": { "fields": [ "docType", "owner" ] }, "ddoc": "indexOwnerDoc", "name": "indexOwner",  
  ↪ "type": "json" }
```

The sample index can be found [here](#).

Any index in the chaincode’s `META-INF/statedb/couchdb/indexes` directory will be packaged up with the chaincode for deployment. The index will be deployed to a peers channel and chaincode specific database when the chaincode package is installed on the peer and the chaincode definition is committed to the channel. If you install the chaincode first and then commit the chaincode definition to the channel, the index will be deployed at commit time. If the chaincode has already been defined on the channel and the chaincode package subsequently installed on a peer joined to the channel, the index will be deployed at chaincode **installation** time.

Upon deployment, the index will automatically be utilized by chaincode queries. CouchDB can automatically determine which index to use based on the fields being used in a query. Alternatively, in the selector query the index can be specified using the `use_index` keyword.

The same index may exist in subsequent versions of the chaincode that gets installed. To change the index, use the same index name but alter the index definition. Upon installation/instantiation, the index definition will get re-deployed to the peer’s state database.

If you have a large volume of data already, and later install the chaincode, the index creation upon installation may take some time. Similarly, if you have a large volume of data already and commit the definition of a subsequent chaincode version, the index creation may take some time. Avoid calling chaincode functions that query the state database at these times as the chaincode query may time out while the index is getting initialized. During transaction processing, the indexes will automatically get refreshed as blocks are committed to the ledger. If the peer crashes during chaincode installation, the couchdb indexes may not get created. If this occurs, you need to reinstall the chaincode to create the indexes.

12.6.3 CouchDB Configuration

CouchDB is enabled as the state database by changing the `stateDatabase` configuration option from `goleveldb` to `CouchDB`. Additionally, the `couchDBAddress` needs to be configured to point to the CouchDB to be used by the peer. The username and password properties should be populated with an admin username and password. Additional options are provided in the `couchDBConfig` section and are documented in place. Changes to the `core.yaml` will be effective immediately after restarting the peer.

You can also pass in docker environment variables to override `core.yaml` values, for example `CORE_LEDGER_STATE_STATEDATABASE` and `CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS`.

Below is the `stateDatabase` section from `core.yaml`:

```
state:
  # stateDatabase - options are "goleveldb", "CouchDB"
  # goleveldb - default state database stored in goleveldb.
  # CouchDB - store state database in CouchDB
  stateDatabase: goleveldb
  # Limit on the number of records to return per query
  totalQueryLimit: 10000
  couchDBConfig:
    # It is recommended to run CouchDB on the same server as the peer, and
    # not map the CouchDB container port to a server port in docker-compose.
    # Otherwise proper security must be provided on the connection between
    # CouchDB client (on the peer) and server.
    couchDBAddress: couchdb:5984
    # This username must have read and write authority on CouchDB
    username:
      # The password is recommended to pass as an environment variable
      # during start up (e.g. LEDGER_COUCHDBCONFIG_PASSWORD).
      # If it is stored here, the file must be access control protected
      # to prevent unintended users from discovering the password.
    password:
      # Number of retries for CouchDB errors
    maxRetries: 3
      # Number of retries for CouchDB errors during peer startup
    maxRetriesOnStartup: 10
      # CouchDB request timeout (unit: duration, e.g. 20s)
    requestTimeout: 35s
      # Limit on the number of records per each CouchDB query
      # Note that chaincode queries are only bound by totalQueryLimit.
      # Internally the chaincode may execute multiple CouchDB queries,
      # each of size internalQueryLimit.
    internalQueryLimit: 1000
      # Limit on the number of records per CouchDB bulk update batch
    maxBatchUpdateSize: 1000
      # Warm indexes after every N blocks.
      # This option warms any indexes that have been
      # deployed to CouchDB after every N blocks.
      # A value of 1 will warm indexes after every block commit,
      # to ensure fast selector queries.
      # Increasing the value may improve write efficiency of peer and CouchDB,
      # but may degrade query response time.
    warmIndexesAfterNBlocks: 1
```

CouchDB hosted in docker containers supplied with Hyperledger Fabric have the capability of setting the CouchDB username and password with environment variables passed in with the `COUCHDB_USER` and `COUCHDB_PASSWORD` environment variables using Docker Compose scripting.

For CouchDB installations outside of the docker images supplied with Fabric, the `local.ini` file of that installation must be edited to set the admin username and password.

Docker compose scripts only set the username and password at the creation of the container. The `local.ini` file must be edited if the username or password is to be changed after creation of the container.

If you choose to map the `fabric-couchdb` container port to a host port, make sure you are aware of the security implications. Mapping the CouchDB container port in a development environment exposes the CouchDB REST API and allows you to visualize the database via the CouchDB web interface (Fauxton). In a production environment you should refrain from mapping the host port to restrict access to the CouchDB container. Only the peer will be able to access the CouchDB container.

Note: CouchDB peer options are read on each peer startup.

12.6.4 Good practices for queries

Avoid using chaincode for queries that will result in a scan of the entire CouchDB database. Full length database scans will result in long response times and will degrade the performance of your network. You can take some of the following steps to avoid long queries:

- When using JSON queries:
 - Be sure to create indexes in the chaincode package.
 - Avoid query operators such as `$or`, `$in` and `$regex`, which lead to full database scans.
- For range queries, composite key queries, and JSON queries:
 - Utilize paging support instead of one large result set.
- If you want to build a dashboard or collect aggregate data as part of your application, you can query an off-chain database that replicates the data from your blockchain network. This will allow you to query and analyze the blockchain data in a data store optimized for your needs, without degrading the performance of your network or disrupting transactions. To achieve this, applications may use block or chaincode events to write transaction data to an off-chain database or analytics engine. For each block received, the block listener application would iterate through the block transactions and build a data store using the key/value writes from each valid transaction's `rwset`. The *Peer channel-based event services* provide replayable events to ensure the integrity of downstream data stores.

12.7 Peer channel-based event services

12.7.1 General overview

In previous versions of Fabric, the peer event service was known as the event hub. This service sent events any time a new block was added to the peer's ledger, regardless of the channel to which that block pertained, and it was only accessible to members of the organization running the eventing peer (i.e., the one being connected to for events).

Starting with v1.1, there are new services which provide events. These services use an entirely different design to provide events on a per-channel basis. This means that registration for events occurs at the level of the channel instead of the peer, allowing for fine-grained control over access to the peer's data. Requests to receive events are accepted from identities outside of the peer's organization (as defined by the channel configuration). This also provides greater reliability and a way to receive events that may have been missed (whether due to a connectivity issue or because the peer is joining a network that has already been running).

12.7.2 Available services

- `Deliver`

This service sends entire blocks that have been committed to the ledger. If any events were set by a chaincode, these can be found within the `ChaincodeActionPayload` of the block.

- `DeliverWithPrivateData`

This service sends the same data as the `Deliver` service, and additionally includes any private data from collections that the client's organization is authorized to access.

- `DeliverFiltered`

This service sends “filtered” blocks, minimal sets of information about blocks that have been committed to the ledger. It is intended to be used in a network where owners of the peers wish for external clients to primarily receive information about their transactions and the status of those transactions. If any events were set by a chaincode, these can be found within the `FilteredChaincodeAction` of the filtered block.

Note: The payload of chaincode events will not be included in filtered blocks.

12.7.3 How to register for events

Registration for events is done by sending an envelope containing a deliver seek info message to the peer that contains the desired start and stop positions, the seek behavior (block until ready or fail if not ready). There are helper variables `SeekOldest` and `SeekNewest` that can be used to indicate the oldest (i.e. first) block or the newest (i.e. last) block on the ledger. To have the services send events indefinitely, the `SeekInfo` message should include a stop position of `MAXINT64`.

Note: If mutual TLS is enabled on the peer, the TLS certificate hash must be set in the envelope's channel header.

By default, the event services use the Channel Readers policy to determine whether to authorize requesting clients for events.

12.7.4 Overview of deliver response messages

The event services send back `DeliverResponse` messages.

Each message contains one of the following:

- `status` – HTTP status code. Each of the services will return the appropriate failure code if any failure occurs; otherwise, it will return `200 - SUCCESS` once the service has completed sending all information requested by the `SeekInfo` message.
- `block` – returned only by the `Deliver` service.
- `block and private data` – returned only by the `DeliverWithPrivateData` service.
- `filtered block` – returned only by the `DeliverFiltered` service.

A filtered block contains:

- `channel ID`.
- `number` (i.e. the block number).
- `array of filtered transactions`.

- transaction ID.
 - type (e.g. `ENDORSER_TRANSACTION`, `CONFIG`).
 - transaction validation code.
- **filtered transaction actions.**
 - **array of filtered chaincode actions.**
 - * chaincode event for the transaction (with the payload nilled out).

12.7.5 SDK event documentation

For further details on using the event services, refer to the [SDK documentation](#).

12.8 Private Data

Note: This topic assumes an understanding of the conceptual material in the [documentation on private data](#).

12.8.1 Private data collection definition

A collection definition contains one or more collections, each having a policy definition listing the organizations in the collection, as well as properties used to control dissemination of private data at endorsement time and, optionally, whether the data will be purged.

Beginning with the Fabric chaincode lifecycle introduced with Fabric v2.0, the collection definition is part of the chaincode definition. The chaincode including collection definition must be approved by the required channel members, and then becomes effective when the chaincode definition is committed to the channel. The collection definition that is approved must be identical for each of the required channel members. When using the peer CLI to approve and commit the chaincode definition, use the `--collections-config` flag to specify the path to the collection definition file.

Collection definitions are composed of the following properties:

- `name`: Name of the collection.
- `policy`: The private data collection distribution policy defines which organizations' peers are allowed to retrieve and persist the collection data expressed using the `Signature` policy syntax, with each member being included in an `OR` signature policy list. To support read/write transactions, the private data distribution policy must define a broader set of organizations than the endorsement policy, as peers must have the private data in order to endorse proposed transactions. For example, in a channel with ten organizations, five of the organizations might be included in a private data collection distribution policy, but the endorsement policy might call for any three of the organizations in the channel to endorse a read/write transaction. For write-only transactions, organizations that are not members of the collection distribution policy but are included in the chaincode level endorsement policy may endorse transactions that write to the private data collection. If this is not desirable, utilize a collection level `endorsementPolicy` to restrict the set of allowed endorsers to the private data distribution policy members.
- `requiredPeerCount`: Minimum number of peers (across authorized organizations) that each endorsing peer must successfully disseminate private data to before the peer signs the endorsement and returns the proposal response back to the client. Requiring dissemination as a condition of endorsement will ensure that private data is available in the network even if the endorsing peer(s) become unavailable. When `requiredPeerCount` is 0, it means that no distribution is **required**, but there may be some distribution if `maxPeerCount` is greater

than zero. A `requiredPeerCount` of 0 would typically not be recommended, as it could lead to loss of private data in the network if the endorsing peer(s) becomes unavailable. Typically you would want to require at least some distribution of the private data at endorsement time to ensure redundancy of the private data on multiple peers in the network.

- `maxPeerCount`: For data redundancy purposes, the maximum number of other peers (across authorized organizations) that each endorsing peer will attempt to distribute the private data to. If an endorsing peer becomes unavailable between endorsement time and commit time, other peers that are collection members but who did not yet receive the private data at endorsement time, will be able to pull the private data from peers the private data was disseminated to. If this value is set to 0, the private data is not disseminated at endorsement time, forcing private data pulls against endorsing peers on all authorized peers at commit time.
- `blockToLive`: Represents how long the data should live on the private database in terms of blocks. The data will live for this specified number of blocks on the private database and after that it will get purged, making this data obsolete from the network so that it cannot be queried from chaincode, and cannot be made available to requesting peers. To keep private data indefinitely, that is, to never purge private data, set the `blockToLive` property to 0.
- `memberOnlyRead`: a value of `true` indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed read access to private data. If a client from a non-member org attempts to execute a chaincode function that performs a read of a private data key, the chaincode invocation is terminated with an error. Utilize a value of `false` if you would like to encode more granular access control within individual chaincode functions.
- `memberOnlyWrite`: a value of `true` indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed to write private data from chaincode. If a client from a non-member org attempts to execute a chaincode function that performs a write on a private data key, the chaincode invocation is terminated with an error. Utilize a value of `false` if you would like to encode more granular access control within individual chaincode functions, for example you may want certain clients from non-member organization to be able to create private data in a certain collection.
- `endorsementPolicy`: An optional endorsement policy to utilize for the collection that overrides the chaincode level endorsement policy. A collection level endorsement policy may be specified in the form of a `signaturePolicy` or may be a `channelConfigPolicy` reference to an existing policy from the channel configuration. The `endorsementPolicy` may be the same as the collection distribution policy, or may require fewer or additional organization peers.

Here is a sample collection definition JSON file, containing an array of two collection definitions:

```
[
  {
    "name": "collectionMarbles",
    "policy": "OR('Org1MSP.member', 'Org2MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive": 1000000,
    "memberOnlyRead": true,
    "memberOnlyWrite": true
  },
  {
    "name": "collectionMarblePrivateDetails",
    "policy": "OR('Org1MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive": 3,
    "memberOnlyRead": true,
    "memberOnlyWrite": true,
    "endorsementPolicy": {
```

(continues on next page)

(continued from previous page)

```
    "signaturePolicy": "OR('Org1MSP.member') "  
  }  
}  
]
```

This example uses the organizations from the Fabric test network, `Org1` and `Org2`. The policy in the `collectionMarbles` definition authorizes both organizations to the private data. This is a typical configuration when the chaincode data needs to remain private from the ordering service nodes. However, the policy in the `collectionMarblePrivateDetails` definition restricts access to a subset of organizations in the channel (in this case `Org1`). Additionally, writing to this collection requires endorsement from an `Org1` peer, even though the chaincode level endorsement policy may require endorsement from `Org1` or `Org2`. And since “`memberOnlyWrite`” is true, only clients from `Org1` may invoke chaincode that writes to the private data collection. In this way you can control which organizations are entrusted to write to certain private data collections.

12.8.2 Implicit private data collections

In addition to explicitly defined private data collections, every chaincode has an implicit private data namespace reserved for organization-specific private data. These implicit organization-specific private data collections can be used to store an individual organization’s private data, and do not need to be defined explicitly.

The private data dissemination policy and endorsement policy for implicit organization-specific collections is the respective organization itself. The implication is that if data exists in an implicit private data collection, it was endorsed by the respective organization. Implicit private data collections can therefore be used by an organization to record their agreement or vote for some fact, which is a useful pattern to leverage in multi-party business processes implemented in chaincode since other organizations can check the on-chain hash to verify the organization’s record. Private data can also be shared or transferred to an implicit collection of another organization, making implicit collections a useful pattern to leverage in chaincode applications, without the need to explicitly manage collection definitions.

Since implicit private data collections are not explicitly defined, it is not possible to set the additional collection properties. Specifically, `memberOnlyRead` and `memberOnlyWrite` are not available, meaning that access control for clients reading data from or writing data to an implicit private data collection must be encoded in the chaincode on the organization’s peer. Furthermore, `blockToLive` is not available, meaning that private data is never automatically purged.

The properties `requiredPeerCount` and `maxPeerCount` can however be set in the peer’s `core.yaml` (`peer.gossip.pvtData.implicitCollectionDisseminationPolicy.requiredPeerCount` and `peer.gossip.pvtData.implicitCollectionDisseminationPolicy.maxPeerCount`). An organization can set these properties based on the number of peers that they deploy, as described in the next section.

Note: Since implicit private data collections are not explicitly defined, it is not possible to associate CouchDB indexes with them. Utilize key-based queries and key-range queries rather than JSON queries.

12.8.3 Private data dissemination

Since private data is not included in the transactions that get submitted to the ordering service, and therefore not included in the blocks that get distributed to all peers in a channel, the endorsing peer plays an important role in disseminating private data to other peers of authorized organizations. This ensures the availability of private data in the channel’s collection, even if endorsing peers become unavailable after their endorsement. To assist with this dissemination, the `maxPeerCount` and `requiredPeerCount` properties control the degree of dissemination at endorsement time.

If the endorsing peer cannot successfully disseminate the private data to at least the `requiredPeerCount`, it will return an error back to the client. The endorsing peer will attempt to disseminate the private data to peers of different organizations, in an effort to ensure that each authorized organization has a copy of the private data. Since transactions are not committed at chaincode execution time, the endorsing peer and recipient peers store a copy of the private data in a local `transient` store alongside their blockchain until the transaction is committed.

When authorized peers do not have a copy of the private data in their transient data store at commit time (either because they were not an endorsing peer or because they did not receive the private data via dissemination at endorsement time), they will attempt to pull the private data from another authorized peer, *for a configurable amount of time* based on the peer property `peer.gossip.pvtData.pullRetryThreshold` in the peer configuration `core.yaml` file.

Note: The peers being asked for private data will only return the private data if the requesting peer is a member of the collection as defined by the private data dissemination policy.

Considerations when using `pullRetryThreshold`:

- If the requesting peer is able to retrieve the private data within the `pullRetryThreshold`, it will commit the transaction to its ledger (including the private data hash), and store the private data in its state database, logically separated from other channel state data.
- If the requesting peer is not able to retrieve the private data within the `pullRetryThreshold`, it will commit the transaction to its blockchain (including the private data hash), without the private data.
- If the peer was entitled to the private data but it is missing, then that peer will not be able to endorse future transactions that reference the missing private data - a chaincode query for a key that is missing will be detected (based on the presence of the key's hash in the state database), and the chaincode will receive an error.

Therefore, it is important to set the `requiredPeerCount` and `maxPeerCount` properties large enough to ensure the availability of private data in your channel. For example, if each of the endorsing peers become unavailable before the transaction commits, the `requiredPeerCount` and `maxPeerCount` properties will have ensured the private data is available on other peers.

Note: For collections to work, it is important to have cross organizational gossip configured correctly. Refer to our documentation on *Gossip data dissemination protocol*, paying particular attention to the “anchor peers” and “external endpoint” configuration.

12.8.4 Referencing collections from chaincode

A set of [shim APIs](#) are available for setting and retrieving private data.

The same chaincode data operations can be applied to channel state data and private data, but in the case of private data, a collection name is specified along with the data in the chaincode APIs, for example `PutPrivateData(collection, key, value)` and `GetPrivateData(collection, key)`.

A single chaincode can reference multiple collections.

12.8.5 Referencing implicit collections from chaincode

Starting in v2.0, an implicit private data collection can be used for each organization in a channel, so that you don't have to define collections if you'd like to utilize per-organization collections. Each org-specific implicit collection has a distribution policy and endorsement policy of the matching organization. You can therefore utilize implicit collections for use cases where you'd like to ensure that a specific organization has written to a collection key namespace. The v2.0 chaincode lifecycle uses implicit collections to track which organizations have approved a chaincode definition.

Similarly, you can use implicit collections in application chaincode to track which organizations have approved or voted for some change in state.

To write and read an implicit private data collection key, in the `PutPrivateData` and `GetPrivateData` chaincode APIs, specify the collection parameter as `"_implicit_org_<MSPID>"`, for example `"_implicit_org_Org1MSP"`.

Note: Application defined collection names are not allowed to start with an underscore, therefore there is no chance for an implicit collection name to collide with an application defined collection name.

How to pass private data in a chaincode proposal

Since the chaincode proposal gets stored on the blockchain, it is also important not to include private data in the main part of the chaincode proposal. A special field in the chaincode proposal called the `transient` field can be used to pass private data from the client (or data that chaincode will use to generate private data), to chaincode invocation on the peer. The chaincode can retrieve the `transient` field by calling the `GetTransient()` API. This `transient` field gets excluded from the channel transaction.

Protecting private data content

If the private data is relatively simple and predictable (e.g. transaction dollar amount), channel members who are not authorized to the private data collection could try to guess the content of the private data via brute force hashing of the domain space, in hopes of finding a match with the private data hash on the chain. Private data that is predictable should therefore include a random “salt” that is concatenated with the private data key and included in the private data value, so that a matching hash cannot realistically be found via brute force. The random “salt” can be generated at the client side (e.g. by sampling a secure pseudo-random source) and then passed along with the private data in the transient field at the time of chaincode invocation.

Access control for private data

Until version 1.3, access control to private data based on collection membership was enforced for peers only. Access control based on the organization of the chaincode proposal submitter was required to be encoded in chaincode logic. Collection configuration options `memberOnlyRead` (since version v1.4) and `memberOnlyWrite` (since version v2.0) can automatically enforce that the chaincode proposal submitter must be from a collection member in order to read or write private data keys. For more information about collection configuration definitions and how to set them, refer back to the *Private data collection definition* section of this topic.

Note: If you would like more granular access control, you can set `memberOnlyRead` and `memberOnlyWrite` to false (implicit collections always behave as if `memberOnlyRead` and `memberOnlyWrite` are false). You can then apply your own access control logic in chaincode, for example by calling the `GetCreator()` chaincode API or using the client identity [chaincode library](#).

Querying Private Data

Private data collection can be queried just like normal channel data, using shim APIs:

- `GetPrivateDataByRange(collection, startKey, endKey string)`
- `GetPrivateDataByPartialCompositeKey(collection, objectType string, keys []string)`

And if using explicit private data collections and CouchDB state database, JSON content queries can be passed using the shim API:

- `GetPrivateDataQueryResult(collection, query string)`

Limitations:

- Clients that call chaincode that executes key range queries or JSON queries should be aware that they may receive a subset of the result set, if the peer they query has missing private data, based on the explanation in Private Data Dissemination section above. Clients can query multiple peers and compare the results to determine if a peer may be missing some of the result set.
- Chaincode that executes key range queries or JSON queries and updates data in a single transaction is not supported, as the query results cannot be validated on the peers that don't have access to the private data, or on peers that are missing the private data that they have access to. If a chaincode invocation both queries and updates private data, the proposal request will return an error. If your application can tolerate result set changes between chaincode execution and validation/commit time, then you could call one chaincode function to perform the query, and then call a second chaincode function to make the updates. Note that calls to `GetPrivateData()` to retrieve individual keys can be made in the same transaction as `PutPrivateData()` calls, since all peers can validate key reads based on the hashed key version.
- Since implicit private data collections are not explicitly defined, it is not possible to associate CouchDB indexes with them. It is therefore not recommended to utilize JSON queries with implicit private data collections.

Using Indexes with collections

The topic *CouchDB as the State Database* describes indexes that can be applied to the channel's state database to enable JSON content queries, by packaging indexes in a `META-INF/statedb/couchdb/indexes` directory at chaincode installation time. Similarly, indexes can also be applied to private data collections that are explicitly defined, by packaging indexes in a `META-INF/statedb/couchdb/collections/<collection_name>/indexes` directory. An example index is available [here](#).

12.8.6 Considerations when using private data

Private data purging

Private data in explicitly defined private data collections can be periodically purged from peers. For more details, see the `blockToLive` collection definition property above.

Additionally, recall that prior to commit, peers store private data in a local transient data store. This data automatically gets purged when the transaction commits. But if a transaction was never submitted to the channel and therefore never committed, the private data would remain in each peer's transient store. This data is purged from the transient store after a configurable number blocks by using the peer's `peer.gossip.pvtData.transientstoreMaxBlockRetention` property in the `peer core.yaml` file.

Updating a collection definition

To update a collection definition or add a new collection, you can update the chaincode definition and pass the new collection configuration in the chaincode approve and commit transactions, for example using the `--collections-config` flag if using the CLI. If a collection configuration is specified when updating the chaincode definition, a definition for each of the existing collections must be included.

When updating a chaincode definition, you can add new private data collections, and update existing private data collections, for example to add new members to an existing collection or change one of the collection definition prop-

erties. Note that you cannot update the collection name or the `blockToLive` property, since a consistent `blockToLive` is required regardless of a peer's block height.

Collection updates becomes effective when a peer commits the block with the updated chaincode definition. Note that collections cannot be deleted, as there may be prior private data hashes on the channel's blockchain that cannot be removed.

Private data reconciliation

Starting in v1.4, peers of organizations that are added to an existing collection will automatically fetch private data that was committed to the collection before they joined the collection.

This private data “reconciliation” also applies to peers that were entitled to receive private data but did not yet receive it — because of a network failure, for example — by keeping track of private data that was “missing” at the time of block commit.

Private data reconciliation occurs periodically based on the `peer.gossip.pvtData.reconciliationEnabled` and `peer.gossip.pvtData.reconcileSleepInterval` properties in `core.yaml`. The peer will periodically attempt to fetch the private data from other collection member peers that are expected to have it.

Note that this private data reconciliation feature only works on peers running v1.4 or later of Fabric.

12.9 Read-Write set semantics

This document discusses the details of the current implementation about the semantics of read-write sets.

12.9.1 Transaction simulation and read-write set

During simulation of a transaction at an `endorser`, a read-write set is prepared for the transaction. The `read set` contains a list of unique keys and their committed version numbers (but not values) that the transaction reads during simulation. The `write set` contains a list of unique keys (though there can be overlap with the keys present in the read set) and their new values that the transaction writes. A delete marker is set (in the place of new value) for the key if the update performed by the transaction is to delete the key.

Further, if the transaction writes a value multiple times for a key, only the last written value is retained. Also, if a transaction reads a value for a key, the value in the committed state is returned even if the transaction has updated the value for the key before issuing the read. In another words, Read-your-writes semantics are not supported.

As noted earlier, the versions of the keys are recorded only in the read set; the write set just contains the list of unique keys and their latest values set by the transaction.

There could be various schemes for implementing versions. The minimal requirement for a versioning scheme is to produce non-repeating identifiers for a given key. For instance, using monotonically increasing numbers for versions can be one such scheme. In the current implementation, we use a blockchain height based versioning scheme in which the height of the committing transaction is used as the latest version for all the keys modified by the transaction. In this scheme, the height of a transaction is represented by a tuple (`txNumber` is the height of the transaction within the block). This scheme has many advantages over the incremental number scheme - primarily, it enables other components such as `statedb`, transaction simulation and validation to make efficient design choices.

Following is an illustration of an example read-write set prepared by simulation of a hypothetical transaction. For the sake of simplicity, in the illustrations, we use the incremental numbers for representing the versions.

```

<TxReadWriteSet>
  <NsReadWriteSet name="chaincode1">
    <read-set>
      <read key="K1", version="1">
      <read key="K2", version="1">
    </read-set>
    <write-set>
      <write key="K1", value="V1">
      <write key="K3", value="V2">
      <write key="K4", isDelete="true">
    </write-set>
  </NsReadWriteSet>
</TxReadWriteSet>

```

Additionally, if the transaction performs a range query during simulation, the range query as well as its results will be added to the read-write set as `query-info`.

12.9.2 Transaction validation and updating world state using read-write set

A `committer` uses the read set portion of the read-write set for checking the validity of a transaction and the write set portion of the read-write set for updating the versions and the values of the affected keys.

In the validation phase, a transaction is considered `valid` if the version of each key present in the read set of the transaction (from time of simulation) matches the current version for the same key, taking into consideration `valid` transactions that have been committed to state from new blocks since the transaction was simulated, as well as `valid` preceding transactions in the same block. An additional validation is performed if the read-write set also contains one or more `query-info`.

This additional validation should ensure that no key has been inserted/deleted/updated in the super range (i.e., union of the ranges) of the results captured in the `query-info(s)`. In other words, if we re-execute any of the range queries (that the transaction performed during simulation) during validation on the committed-state, it should yield the same results that were observed by the transaction at the time of simulation. This check ensures that if a transaction observes phantom items during commit, the transaction should be marked as invalid. Note that the this phantom protection is limited to range queries (i.e., `GetStateByRange` function in the chaincode) and not yet implemented for other queries (i.e., `GetQueryResult` function in the chaincode). Other queries are at risk of phantoms, and should therefore only be used in read-only transactions that are not submitted to ordering, unless the application can guarantee the stability of the result set between simulation and validation/commit time.

If a transaction passes the validity check, the committer uses the write set for updating the world state. In the update phase, for each key present in the write set, the value in the world state for the same key is set to the value as specified in the write set. Further, the version of the key in the world state is changed to reflect the latest version.

12.9.3 Example simulation and validation

This section helps with understanding the semantics through an example scenario. For the purpose of this example, the presence of a key, `k`, in the world state is represented by a tuple `(k, ver, val)` where `ver` is the latest version of the key `k` having `val` as its value.

Now, consider a set of five transactions `T1`, `T2`, `T3`, `T4`, and `T5`, all simulated on the same snapshot of the world state. The following snippet shows the snapshot of the world state against which the transactions are simulated and the sequence of read and write activities performed by each of these transactions.

```

World state: (k1,1,v1), (k2,1,v2), (k3,1,v3), (k4,1,v4), (k5,1,v5)
T1 -> Write(k1, v1'), Write(k2, v2')

```

(continues on next page)

(continued from previous page)

```

T2 -> Read(k1), Write(k3, v3')
T3 -> Write(k2, v2'')
T4 -> Write(k2, v2''), read(k2)
T5 -> Write(k6, v6'), read(k5)

```

Now, assume that these transactions are ordered in the sequence of T1,...,T5 (could be contained in a single block or different blocks)

1. T1 passes validation because it does not perform any read. Further, the tuple of keys k1 and k2 in the world state are updated to (k1, 2, v1'), (k2, 2, v2')
2. T2 fails validation because it reads a key, k1, which was modified by a preceding transaction - T1
3. T3 passes the validation because it does not perform a read. Further the tuple of the key, k2, in the world state is updated to (k2, 3, v2'')
4. T4 fails the validation because it reads a key, k2, which was modified by a preceding transaction T1
5. T5 passes validation because it reads a key, k5, which was not modified by any of the preceding transactions

Note: Transactions with multiple read-write sets are not yet supported.

12.10 Gossip data dissemination protocol

Hyperledger Fabric optimizes blockchain network performance, security, and scalability by dividing workload across transaction execution (endorsing and committing) peers and transaction ordering nodes. This decoupling of network operations requires a secure, reliable and scalable data dissemination protocol to ensure data integrity and consistency. To meet these requirements, Fabric implements a **gossip data dissemination protocol**.

12.10.1 Gossip protocol

Peers leverage gossip to broadcast ledger and channel data in a scalable fashion. Gossip messaging is continuous, and each peer on a channel is constantly receiving current and consistent ledger data from multiple peers. Each gossiped message is signed, thereby allowing Byzantine participants sending faked messages to be easily identified and the distribution of the message(s) to unwanted targets to be prevented. Peers affected by delays, network partitions, or other causes resulting in missed blocks will eventually be synced up to the current ledger state by contacting peers in possession of these missing blocks.

The gossip-based data dissemination protocol performs three primary functions on a Fabric network:

1. Manages peer discovery and channel membership, by continually identifying available member peers, and eventually detecting peers that have gone offline.
2. Disseminates ledger data across all peers on a channel. Any peer with data that is out of sync with the rest of the channel identifies the missing blocks and syncs itself by copying the correct data.
3. Bring newly connected peers up to speed by allowing peer-to-peer state transfer update of ledger data.

Gossip-based broadcasting operates by peers receiving messages from other peers on the channel, and then forwarding these messages to a number of randomly selected peers on the channel, where this number is a configurable constant. Peers can also exercise a pull mechanism rather than waiting for delivery of a message. This cycle repeats, with the result of channel membership, ledger and state information continually being kept current and in sync. For dissemination of new blocks, the **leader** peer on the channel pulls the data from the ordering service and initiates gossip dissemination to peers in its own organization.

12.10.2 Leader election

The leader election mechanism is used to **elect** one peer per organization which will maintain connection with the ordering service and initiate distribution of newly arrived blocks across the peers of its own organization. Leveraging leader election provides the system with the ability to efficiently utilize the bandwidth of the ordering service. There are two possible modes of operation for a leader election module:

1. **Static** — a system administrator manually configures a peer in an organization to be the leader.
2. **Dynamic** — peers execute a leader election procedure to select one peer in an organization to become leader.

Static leader election

Static leader election allows you to manually define one or more peers within an organization as leader peers. Please note, however, that having too many peers connect to the ordering service may result in inefficient use of bandwidth. To enable static leader election mode, configure the following parameters within the section of `core.yaml`:

```
peer:
  # Gossip related configuration
  gossip:
    useLeaderElection: false
    orgLeader: true
```

Alternatively these parameters could be configured and overridden with environmental variables:

```
export CORE_PEER_GOSSIP_USELEADERELECTION=false
export CORE_PEER_GOSSIP_ORGLEADER=true
```

Note: The following configuration will keep peer in **stand-by** mode, i.e. peer will not try to become a leader:

```
export CORE_PEER_GOSSIP_USELEADERELECTION=false
export CORE_PEER_GOSSIP_ORGLEADER=false
```

2. Setting `CORE_PEER_GOSSIP_USELEADERELECTION` and `CORE_PEER_GOSSIP_ORGLEADER` with `true` value is ambiguous and will lead to an error.
3. In static configuration organization admin is responsible to provide high availability of the leader node in case for failure or crashes.

Dynamic leader election

Dynamic leader election enables organization peers to **elect** one peer which will connect to the ordering service and pull out new blocks. This leader is elected for an organization's peers independently.

A dynamically elected leader sends **heartbeat** messages to the rest of the peers as an evidence of liveness. If one or more peers don't receive **heartbeats** updates during a set period of time, they will elect a new leader.

In the worst case scenario of a network partition, there will be more than one active leader for organization to guarantee resiliency and availability to allow an organization's peers to continue making progress. After the network partition has been healed, one of the leaders will relinquish its leadership. In a steady state with no network partitions, there will be **only** one active leader connecting to the ordering service.

Following configuration controls frequency of the leader **heartbeat** messages:

```
peer:
  # Gossip related configuration
  gossip:
    election:
      leaderAliveThreshold: 10s
```

In order to enable dynamic leader election, the following parameters need to be configured within `core.yaml`:

```
peer:
  # Gossip related configuration
  gossip:
    useLeaderElection: true
    orgLeader: false
```

Alternatively these parameters could be configured and overridden with environment variables:

```
export CORE_PEER_GOSSIP_USELEADERELECTION=true
export CORE_PEER_GOSSIP_ORGLEADER=false
```

12.10.3 Anchor peers

Anchor peers are used by gossip to make sure peers in different organizations know about each other.

When a configuration block that contains an update to the anchor peers is committed, peers reach out to the anchor peers and learn from them about all of the peers known to the anchor peer(s). Once at least one peer from each organization has contacted an anchor peer, the anchor peer learns about every peer in the channel. Since gossip communication is constant, and because peers always ask to be told about the existence of any peer they don't know about, a common view of membership can be established for a channel.

For example, let's assume we have three organizations—*A*, *B*, *C*— in the channel and a single anchor peer—*peer0.orgC*— defined for organization *C*. When *peer1.orgA* (from organization *A*) contacts *peer0.orgC*, it will tell it about *peer0.orgA*. And when at a later time *peer1.orgB* contacts *peer0.orgC*, the latter would tell the former about *peer0.orgA*. From that point forward, organizations *A* and *B* would start exchanging membership information directly without any assistance from *peer0.orgC*.

As communication across organizations depends on gossip in order to work, there must be at least one anchor peer defined in the channel configuration. It is strongly recommended that every organization provides its own set of anchor peers for high availability and redundancy. Note that the anchor peer does not need to be the same peer as the leader peer.

External and internal endpoints

In order for gossip to work effectively, peers need to be able to obtain the endpoint information of peers in their own organization as well as from peers in other organizations.

When a peer is bootstrapped it will use `peer.gossip.bootstrap` in its `core.yaml` to advertise itself and exchange membership information, building a view of all available peers within its own organization.

The `peer.gossip.bootstrap` property in the `core.yaml` of the peer is used to bootstrap gossip **within an organization**. If you are using gossip, you will typically configure all the peers in your organization to point to an initial set of bootstrap peers (you can specify a space-separated list of peers). The internal endpoint is usually auto-computed by the peer itself or just passed explicitly via `core.peer.address` in `core.yaml`. If you need to overwrite this value, you can export `CORE_PEER_GOSSIP_ENDPOINT` as an environment variable.

Bootstrap information is similarly required to establish communication **across organizations**. The initial cross-organization bootstrap information is provided via the “anchor peers” setting described above. If you want

to make other peers in your organization known to other organizations, you need to set the `peer.gossip.externalendpoint` in the `core.yaml` of your peer. If this is not set, the endpoint information of the peer will not be broadcast to peers in other organizations.

To set these properties, issue:

```
export CORE_PEER_GOSSIP_BOOTSTRAP=<a list of peer endpoints within the peer's org>
export CORE_PEER_GOSSIP_EXTERNALENDPOINT=<the peer endpoint, as known outside the org>
```

12.10.4 Gossip messaging

Online peers indicate their availability by continually broadcasting “alive” messages, with each containing the **public key infrastructure (PKI)** ID and the signature of the sender over the message. Peers maintain channel membership by collecting these alive messages; if no peer receives an alive message from a specific peer, this “dead” peer is eventually purged from channel membership. Because “alive” messages are cryptographically signed, malicious peers can never impersonate other peers, as they lack a signing key authorized by a root certificate authority (CA).

In addition to the automatic forwarding of received messages, a state reconciliation process synchronizes **world state** across peers on each channel. Each peer continually pulls blocks from other peers on the channel, in order to repair its own state if discrepancies are identified. Because fixed connectivity is not required to maintain gossip-based data dissemination, the process reliably provides data consistency and integrity to the shared ledger, including tolerance for node crashes.

Because channels are segregated, peers on one channel cannot message or share information on any other channel. Though any peer can belong to multiple channels, partitioned messaging prevents blocks from being disseminated to peers that are not in the channel by applying message routing policies based on a peers’ channel subscriptions.

Note: 1. Security of point-to-point messages are handled by the peer TLS layer, and do not require signatures. Peers are authenticated by their certificates, which are assigned by a CA. Although TLS certs are also used, it is the peer certificates that are authenticated in the gossip layer. Ledger blocks are signed by the ordering service, and then delivered to the leader peers on a channel.

2. Authentication is governed by the membership service provider for the peer. When the peer connects to the channel for the first time, the TLS session binds with the membership identity. This essentially authenticates each peer to the connecting peer, with respect to membership in the network and channel.

Frequently Asked Questions

13.1 Endorsement

Endorsement architecture:

Question How many peers in the network need to endorse a transaction?

Answer The number of peers required to endorse a transaction is driven by the endorsement policy that is specified in the chaincode definition.

Question Does an application client need to connect to all peers?

Answer Clients only need to connect to as many peers as are required by the endorsement policy for the chaincode.

13.2 Security & Access Control

Question How do I ensure data privacy?

Answer There are various aspects to data privacy. First, you can segregate your network into channels, where each channel represents a subset of participants that are authorized to see the data for the chaincodes that are deployed to that channel.

Second, you can use [private-data](#) to keep ledger data private from other organizations on the channel. A private data collection allows a defined subset of organizations on a channel the ability to endorse, commit, or query private data without having to create a separate channel. Other participants on the channel receive only a hash of the data. For more information refer to the [Using Private Data in Fabric](#) tutorial. Note that the key concepts topic also explains [when to use private data instead of a channel](#).

Third, as an alternative to Fabric hashing the data using private data, the client application can hash or encrypt the data before calling chaincode. If you hash the data then you will need to provide a means to share the source data. If you encrypt the data then you will need to provide a means to share the decryption keys.

Fourth, you can restrict data access to certain roles in your organization, by building access control into the chaincode logic.

Fifth, ledger data at rest can be encrypted via file system encryption on the peer, and data in-transit is encrypted via TLS.

Question Do the orderers see the transaction data?

Answer Orderers receive endorsed transactions that are submitted from application clients. The endorsed payload contains the chaincode execution results including the ReadSet and WriteSet information. The orderers only validate the submitter's identity and order transactions, they do not open the endorsed transactions.

If you do not want the data to go through the orderers at all, then utilize the private data feature of Fabric. Alternatively, you can hash or encrypt the data in the client application before calling chaincode. If you encrypt the data then you will need to provide a means to share the decryption keys.

13.3 Application-side Programming Model

Question How do application clients know the outcome of a transaction?

Answer The transaction simulation results are returned to the client by the endorser in the proposal response. If there are multiple endorsers, the client can check that the responses are all the same, and submit the results and endorsements for ordering and commitment. Ultimately the committing peers will validate or invalidate the transaction, and the client becomes aware of the outcome via an event, that the SDK makes available to the application client.

Question How do I query the ledger data?

Answer Within chaincode you can query based on keys. Keys can be queried by range, and composite keys can be modeled to enable equivalence queries against multiple parameters. For example a composite key of (owner,asset_id) can be used to query all assets owned by a certain entity. These key-based queries can be used for read-only queries against the ledger, as well as in transactions that update the ledger.

If you model asset data as JSON in chaincode and use CouchDB as the state database, you can also perform complex rich queries against the chaincode data values, using the CouchDB JSON query language within chaincode. The application client can perform read-only queries, but these responses are not typically submitted as part of transactions to the ordering service.

Question How do I query the historical data to understand data provenance?

Answer The chaincode API `GetHistoryForKey()` will return history of values for a key.

Question How to guarantee the query result is correct, especially when the peer being queried may be recovering and catching up on block processing?

Answer The client can query multiple peers, compare their block heights, compare their query results, and favor the peers at the higher block heights.

13.4 Chaincode (Smart Contracts and Digital Assets)

Question Does Hyperledger Fabric support smart contract logic?

Answer Yes. We call this feature *Chaincode*. It is our interpretation of the smart contract method/algorithm, with additional features.

A chaincode is programmatic code deployed on the network, where it is executed and validated by chain validators together during the consensus process. Developers can use chaincodes to develop business contracts, asset definitions, and collectively-managed decentralized applications.

Question How do I create a business contract?

Answer There are generally two ways to develop business contracts: the first way is to code individual contracts into standalone instances of chaincode; the second way, and probably the more efficient way, is to use chaincode to create decentralized applications that manage the life cycle of one or multiple types of business contracts, and let end users instantiate instances of contracts within these applications.

Question How do I create assets?

Answer Users can use chaincode (for business rules) and membership service (for digital tokens) to design assets, as well as the logic that manages them.

There are two popular approaches to defining assets in most blockchain solutions: the stateless UTXO model, where account balances are encoded into past transaction records; and the account model, where account balances are kept in state storage space on the ledger.

Each approach carries its own benefits and drawbacks. This blockchain technology does not advocate either one over the other. Instead, one of our first requirements was to ensure that both approaches can be easily implemented.

Question Which languages are supported for writing chaincode?

Answer Chaincode can be written in any programming language and executed in containers. Currently, Go, Node.js and Java chaincode are supported.

Question Does the Hyperledger Fabric have native currency?

Answer No. However, if you really need a native currency for your chain network, you can develop your own native currency with chaincode. One common attribute of native currency is that some amount will get transacted (the chaincode defining that currency will get called) every time a transaction is processed on its chain.

13.5 Differences in Most Recent Releases

Question Where can I find what are the highlighted differences between releases?

Answer The differences between any subsequent releases are provided together with the [Releases](#).

Question Where to get help for the technical questions not answered above?

Answer Please use [StackOverflow](#).

13.6 Ordering Service

Question I have an ordering service up and running and want to switch consensus algorithms. How do I do that?

Answer This is explicitly not supported.

Question Can I have an organization act both in an ordering and application role?

Answer Although this is possible, it is a highly discouraged configuration. By default the `/Channel/Orderer/BlockValidation` policy allows any valid certificate of the ordering organizations

to sign blocks. If an organization is acting both in an ordering and application role, then this policy should be updated to restrict block signers to the subset of certificates authorized for ordering.

Question I want to write a consensus implementation for Fabric. Where do I begin?

Answer A consensus plugin needs to implement the `Consenter` and `Chain` interfaces defined in the [consensus package](#). There is a plugin built against [raft](#). You can study it to learn more for your own implementation. The ordering service code can be found under the [orderer package](#).

Question I want to change my ordering service configurations, e.g. batch timeout, after I start the network, what should I do?

Answer This falls under reconfiguring the network. Please consult the topic on [configtxlator](#).

13.6.1 BFT

Question When is a BFT version of the ordering service going to be available?

Answer No date has been set. We are working towards a release during the 2.x cycle, i.e. it will come with a minor version upgrade in Fabric.

Contributions Welcome!

We welcome contributions to Hyperledger in many forms, and there's always plenty to do!

First things first, please review the Hyperledger [Code of Conduct](#) before participating. It is important that we keep things civil.

Note: If you want to contribute to this documentation, please check out the [Style guide for contributors](#).

14.1 Ways to contribute

There are many ways you can contribute to Hyperledger Fabric, both as a user and as a developer.

As a user:

- [Making Feature/Enhancement Proposals](#)
- [Reporting bugs](#)

As a writer or information developer:

- Update the documentation using your experience of Fabric and this documentation to improve existing topics and create new ones. A documentation change is an easy way to get started as a contributor, makes it easier for other users to understand and use Fabric, and grows your open source commit history.
- Participate in a language translation to keep the Fabric documentation current in your chosen language. The Fabric documentation is available in a number of languages – English, Chinese, Malayalam and Brazilian Portuguese – so why not join a team that keeps your favorite documentation up-to-date? You'll find a friendly community of users, writers and developers to collaborate with.
- Start a new language translation if the Fabric documentation isn't available in your language. The Chinese, Malayalam and Portuguese Brazilian teams got started this way, and you can too! It's more work, as you'll have to form a community of writers, and organize contributions; but it's really fulfilling to see the Fabric documentation available in your chosen language.

Jump to *Contributing documentation* to get started on your journey.

As a developer:

- If you only have a little time, consider picking up a “good first issue” task, see *Fixing issues and working stories*.
- If you can commit to full-time development, either propose a new feature (see *Making Feature/Enhancement Proposals*) and bring a team to implement it, or join one of the teams working on an existing Epic. If you see an Epic that interests you on the [GitHub epic backlog](#), contact the Epic assignee via the GitHub issue.

14.2 Getting a Linux Foundation account

In order to participate in the development of the Hyperledger Fabric project, you will need a Linux Foundation account. Once you have a LF ID you will be able to access all the Hyperledger community tools, including [RocketChat](#), and the [Wiki](#) (for editing, only).

Follow the steps below to create a Linux Foundation account if you don’t already have one.

1. Go to the [Linux Foundation ID website](#).
2. Select the option I need to create a Linux Foundation ID, and fill out the form that appears.
3. Wait a few minutes, then look for an email message with the subject line: “Validate your Linux Foundation ID email”.
4. Open the received URL to validate your email address.
5. Verify that your browser displays the message `You have successfully validated your e-mail address`.
6. Access [RocketChat](#) to confirm access.

14.3 Contributing documentation

It’s a good idea to make your first change a documentation change. It’s quick and easy to do, ensures that you have a correctly configured machine, (including the required pre-requisite software), and gets you familiar with the contribution process. Use the following topics to help you get started:

14.3.1 Connect with other writers

Audience: Writers who would like to contribute to the Fabric documentation.

This topic gives you general advice on how to contribute to one of the many language translations provided by Fabric.

In this topic, we’re going to cover:

- *How to get started*
- *Using Rocket chat*
- *Documentation workgroup call*
- *Join a language translation workgroup*
- *Other ways to connect*

Getting started

Before you make a documentation change, you might like to connect with other people working on the Fabric documentation. Your [Linux Foundation account](#) will give you access to many different resources to help you connect with other contributors to the documentation.

Once you have a Linux Foundation account, use any or all of the following mechanisms to connect with others.

Rocket chat

Hyperledger Fabric uses [Rocket chat](#) for interactive discussions on a variety of project topics.

- [Documentation channel](#)

You'll find beginners and experts sharing information on the Fabric documentation. Read the conversation or ask a question on how to get started.

- [International languages channel](#)

A dedicated channel for general questions on international languages.

- [Japanese channel](#)

Read, discuss and share ideas related to the Japanese translation.

Documentation workgroup call

A great place to meet people working on documentation is the Documentation workgroup call. These are held twice every Friday at a time convenient for both Eastern and Western hemispheres. The agenda is published in advance, and there are minutes and recordings of each session. Find out more on the [Documentation wiki](#).

Join a language translation workgroup

Each of the international languages has a welcoming workgroup that you are encouraged to join. View the [list of international workgroups](#). See what your favorite workgroup is doing, and get connected with them. Each workgroup has a list of members and their contact information.

Other ways to connect

Hyperledger Fabric has many other collaboration mechanisms such as mailing lists, contributor meetings and maintainer meetings. Find out about these and more [here](#).

Good luck getting started and thanks for your contribution.

14.3.2 Contributing documentation

Audience: Anyone who would like to contribute to the Fabric documentation.

This short guide describes how the Fabric documentation is structured, built and published, as well as a few conventions that help writers make changes to the Fabric documentation.

In this topic, we're going to cover:

- *An introduction to the documentation*
- *Repository folder structure*

- *International language folder structure*
- *Making documentation changes*
- *Building the documentation on your local machine*
- *Building the documentation on GitHub with ReadTheDocs*
- *Getting your change approved*
- *Making a change to Commands Reference*
- *Adding a new CLI command*

Introduction

The Fabric documentation is written in a combination of [Markdown](#) and [reStructuredText](#) source files. As a new author you can use either format. We recommend that you use Markdown as an easy and powerful way to get started. If you have a background in Python, you may prefer to use rST.

During the documentation build process, the documentation source files are converted to HTML using [Sphinx](#). The generated HTML files are subsequently published on the [public documentation website](#). Users can select both different languages and different versions of the Fabric documentation.

For example:

- [Latest version of US English](#)
- [Latest version of Chinese](#)
- [Version 2.2 of US English](#)
- [Version 1.4 of US English](#)

For historical reasons, the US English source files live in the main [Fabric repository](#), whereas all international language source files live in a single [Fabric i18n repository](#). Different versions of the documentation are held within the appropriate GitHub release branch.

Repository folders

Both the US English and international language repositories have essentially the same structure, so let's start by examining the US English source files.

All files relating to documentation reside within the `fabric/docs/` folder:

```
fabric/docs
├── custom_theme
├── source
│   ├── _static
│   ├── _templates
│   ├── commands
│   ├── create_channel
│   ├── dev-setup
│   ├── developapps
│   ├── diagrams
│   ├── ...
│   ├── orderer
│   ├── peers
│   ├── policies
│   └── private-data
```

(continues on next page)

(continued from previous page)

```

├── smartcontract
├── style-guides
├── tutorial
└── wrappers

```

The most important folders is `source/` because it holds the source language files. The documentation build process uses the `make` command to convert these source files to HTML, which are stored in the dynamically created `build/html/` folder:

```

fabric/docs
├── build
│   └── html
├── custom_theme
├── source
│   ├── _static
│   ├── _templates
│   ├── commands
│   ├── create_channel
│   ├── dev-setup
│   ├── developapps
│   └── diagrams
└── ...

```

Spend a little time navigating the `docs` folder in the Hyperledger Fabric repository. Click on the following links to see how different source files map to their corresponding published topics.

- `/docs/source/index.rst` maps to [Hyperledger Fabric title page](#)
- `/docs/source/developapps/developing-applications.rst` maps to [Developing applications](#)
- `/docs/source/peers/peers.md` maps to [Peers](#)

We'll see how to make changes to these files a little later.

International folders

The international language repository, `fabric-docs-i18n`, follows almost exactly the same structure as the `fabric` repository which holds the US English files. The difference is that each language is located within its own folder within `docs/locale/`:

```

fabric-docs-i18n/docs
├── locale
│   ├── ja_JP
│   ├── ml_IN
│   ├── pt_BR
│   └── zh_CN

```

Examining any one of these folders shows a familiar structure:

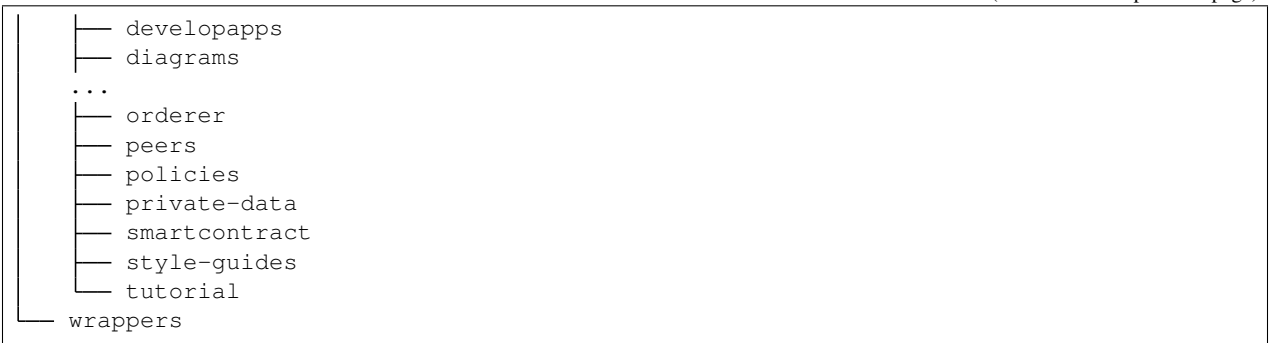
```

locale/ml_IN
├── custom_theme
├── source
│   ├── _static
│   ├── _templates
│   ├── commands
│   └── dev-setup

```

(continues on next page)

(continued from previous page)



As we'll soon see, the similarity of the international language and US English folder structures means that the same instructions and commands can be used to manage different language translations.

Again, spend some time examining the [international language repository](#).

Making changes

To update the documentation, you simply change one or more language source files in a local git feature branch, build the changes locally to check they're OK, and submit a Pull request (PR) to merge your branch with the appropriate Fabric repository branch. Once your PR has been reviewed and approved by the appropriate maintainers for the language, it will be merged into the repository and become part of the published documentation. It really is that easy!

As well as being polite, it's a really good idea to test any documentation changes before you request to include it in a repository. The following sections show you how to:

- Build and review a documentation change on your own machine.
- Push these changes to your GitHub repository fork where they can populate your personal [ReadTheDocs](#) publication website for collaborators to review.
- Submit your documentation PR for inclusion in the `fabric` or `fabric-docs-i18n` repository.

Building locally

Use these simple steps to build the documentation.

1. Create a fork of the appropriate `fabric` or `fabric-i18n` repository to your GitHub account.
2. Install the following prerequisites; you may need to adjust depending on your OS:
 - [Docker](#)
3. For US English:

```
git clone git@github.com:hyperledger/fabric.git
cd fabric
make docs
```

For International Languages (Malayalam as an example):

```
git clone git@github.com:hyperledger/fabric-docs-i18n.git
cd fabric-docs-i18n
make docs-lang-ml_IN
```

The `make` command generates documentation html files in the `docs/build/html/` folder which you can now view locally; simply navigate your browser to the `docs/build/html/index.html` file. For International Languages, you need to read `docs/build/html/` as `docs/locale/${LANG_CODE}/_build/html/` (e.g., `docs/locale/ml_IN/_build/html/`).

4. Now make a small change to a file, and rebuild the documentation to verify that your change was as expected. Every time you make a change to the documentation you will of course need to rerun `make docs`.
5. If you'd like, you may also run a local web server with the following commands (or equivalent depending on your OS):

```
sudo apt-get install apache2
cd docs/build/html
sudo cp -r * /var/www/html/
```

You can then access the html files at `http://localhost/index.html`.

6. You can learn how to make a PR [here](#). Moreover, if you are new to git or GitHub, you will find the [Git book](#) invaluable.

Building on GitHub

It is often helpful to use your fork of the Fabric repository to build the Fabric documentation so that others can review your changes before you submit them for approval. The following instructions show you how to use ReadTheDocs to do this.

1. Go to <http://readthedocs.org> and sign up for an account.
2. Create a project. Your username will preface the URL and you may want to append `-fabric` to ensure that you can distinguish between this and other docs that you need to create for other projects. So for example: `YOURGITHUBID-fabric.readthedocs.io/en/latest`.
3. Click Admin, click Integrations, click Add integration, choose GitHub incoming webhook, then click Add integration.
4. Fork the `fabric` repository.
5. From your fork, go to Settings in the upper right portion of the screen.
6. Click Webhooks.
7. Click Add webhook.
8. Add the ReadTheDocs's URL into Payload URL.
9. Choose Let me select individual events: Pushes Branch or tag creation Branch or tag deletion.
10. Click Add webhook.

Use `fabric-docs-il18n` instead of `fabric` in the above instructions if you're building an international language translation.

Now, anytime you modify or add documentation content to your fork, this URL will automatically get updated with your changes!

Making a PR

You can submit your PR for inclusion using the following [instructions](#).

Pay special attention to signing your commit with the `-s` option:

```
git commit -s -m "My Doc change"
```

This states that your changes conform to the [Developer Certificate of Origin](#).

Before your PR can be included in the appropriate `fabric` or `fabric-docs-118n` repository, it must be approved by an appropriate maintainer. For example, a Japanese translation must be approved by a Japanese maintainer, and so on. You can find the maintainers listed in the following `CODEOWNERS` files:

- US English [CODEOWNERS](#) and their [maintainer GitHub IDs](#)
- International language [CODEOWNERS](#) and their [maintainer GitHub IDs](#)

Both language repositories have a GitHub webhook defined so that, once approved, your newly merged content in the `docs/` folder will trigger an automatic build and publication of the updated documentation.

Note: Documentation maintainers are not able to merge documentation PRs by clicking the `Merge pull request` button. Instead, if you are a documentation maintainer and have approved the PR, simply add the label `doc-merge` to the PR and a Mergify bot that runs every minute will merge the PR.

Commands Reference updates

Updating content in the [Commands Reference](#) topic requires additional steps. Because the information in the `Commands Reference` topic is generated content, you cannot simply update the associated markdown files.

- Instead you need to update the `_preamble.md` or `_postscript.md` files under `src/github.com/hyperledger/fabric/docs/wrappers` for the command.
- To update the command help text, you need to edit the associated `.go` file for the command that is located under `/fabric/internal/peer`.
- Then, from the `fabric` folder, you need to run the command `make help-docs` which generates the updated markdown files under `docs/source/commands`.

Remember that when you push the changes to GitHub, you need to include the `_preamble.md`, `_postscript.md` or `_.go` file that was modified as well as the generated markdown file.

This process only applies to English language translations. Command Reference translation is currently not possible in international languages.

Adding a new CLI command

To add a new CLI command, perform the following steps:

- Create a new folder under `/fabric/internal/peer` for the new command and the associated help text. See `internal/peer/version` for a simple example to get started.
- Add a section for your CLI command in `src/github.com/hyperledger/fabric/scripts/generateHelpDoc.sh`.
- Create two new files under `/src/github.com/hyperledger/fabric/docs/wrappers` with the associated content:
 - `<command>_preamble.md` (Command name and syntax)
 - `<command>_postscript.md` (Example usage)
- Run `make help-docs` to generate the markdown content and push all of the changed files to GitHub.

This process only applies to English language translations. CLI command translation is currently not possible in international languages.

14.3.3 Creating a new translation

Audience: Writers who would like to create a new Fabric translation.

If the Hyperledger Fabric documentation is not available in your chosen language then why not start a new language translation? It's relatively easy to get started, and creating a new language translation can be a very satisfying activity for you and other Fabric users.

In this topic, we're going to cover:

- *An introduction to international language support*
- *How to create a new language workgroup*
- *How to create a new language translation*

Introduction

Hyperledger Fabric documentation is being translated into many different languages. For example:

- Chinese
- Malayalam
- Brazilian Portuguese
- Japanese

If your chosen language is not available, then the first thing to do is to create a new language workgroup.

Create a new workgroup

It's much easier to translate, maintain, and manage a language repository if you collaborate with other translators. Start this process by adding a new workgroup to the [list of international workgroups](#), using one of the existing workgroup pages as an exemplar.

Document how your workgroup will collaborate; meetings, chat and mailing lists can all be very effective. Making these mechanisms clear on your workgroup page can help build a community of translators.

Then use [Rocket chat channels](#) to let other people know you've started a translation, and invite them to join the workgroup.

Create a new translation

Follow these instructions to create your own language repository. Our sample instructions will show you how to create a new language translation for Spanish as spoken in Mexico:

1. Fork the `fabric-docs-il8n` repository to your GitHub account.
2. Clone your repository fork to your local machine:

```
git clone git@github.com:YOURGITHUBID/fabric-docs-il8n.git
```

3. Select the Fabric version you are going to use as a baseline. We recommend that you start with Fabric 2.2 as this is an LTS release. You can add other releases later.

```
cd fabric-docs-il8n
git fetch origin
git checkout release-2.2
```

4. Create a local feature branch:

```
git checkout -b newtranslation
```

5. Identify the appropriate **two or four letter language code**. Mexican Spanish has the language code `es_MX`.

6. Update the fabric `CODEOWNERS` file in the root directory. Add the following line:

```
/docs/locale/ex_EX/ @hyperledger/fabric-core-doc-maintainers @hyperledger/fabric-  
↪es_MX-doc-maintainers
```

7. Create a new language folder under `docs/locale/` for your language.

```
cd docs/locale  
mkdir es_MX
```

8. Copy the language files from another language folder, for example

```
cp -R pt_BR/ es_MX/
```

Alternatively, you could copy the `docs/` folder from the `fabric` repository.

9. Customize the `README.md` file for your new language using [this example](#).

10. Commit your changes locally:

```
git add .  
git commit -s -m "First commit for Mexican Spanish"
```

11. Push your `newtranslation` local feature branch to the `release-2.2` branch of your forked `fabric-docs-il8n` repository:

```
git push origin release-2.2:newtranslation  
  
Total 0 (delta 0), reused 0 (delta 0)  
remote:  
remote: Create a pull request for 'newtranslation' on GitHub by visiting:  
remote:      https://github.com/YOURGITHUBID/fabric-docs-il8n/pull/new/  
↪newtranslation  
remote:  
To github.com:ODOWDAIBM/fabric-docs-il8n.git  
* [new branch]      release-2.2 -> newtranslation
```

12. Connect your repository fork to ReadTheDocs using these [instructions](#). Verify that your documentation builds correctly.

13. Create a pull request (PR) for `newtranslation` on GitHub by visiting:

<https://github.com/YOURGITHUBID/fabric-docs-il8n/pull/new/newtranslation>

Your PR needs to be approved by one of the [documentation maintainers](#). They will be automatically informed of your PR by email, and you can contact them via Rocket chat.

14. On the `il8n` [rocket channel](#) request the creation of the new group of maintainers for your language, `@hyperledger/fabric-es_MX-doc-maintainers`. Provide your GitHubID for addition to this group.

Once you've been added to this list, you can add others translators from your workgroup.

Congratulations! A community of translators can now translate your newly-created language in the `fabric-docs-il8n` repository.

First topics

Before your new language can be published to the documentation website, you must translate the following topics. These topics help users and translators of your new language get started.

- [Fabric front page](#)

This is your advert! Thanks to you, users can now see that the documentation is available in their language. It might not be complete yet, but its clear you and your team are trying to achieve. Translating this page will help you recruit other translators.

- [Introduction](#)

This short topic gives a high level overview of Fabric, and because it's probably one of the first topics a new user will look at, it's important that it's translated.

- [Contributions Welcome!](#)

This topic is vital – it helps contributors understand **what**, **why** and **how** of contributing to Fabric. You need to translate this topic so that others can help you collaborate in your translation.

- [Glossary](#)

Translating this topic provides the essential reference material that helps other language translators make progress; in short, it allows your workgroup to scale.

Once this set of topics have been translated, and you've created a language workgroup, your translation can be published on the documentation website. For example, the Chinese language docs are available [here](#).

You can now request, via the `i18n rocket channel`, that your translation is included on the documentation website.

Translation tools

When translating topics from US English to your international language, it's often helpful to use an online tool to generate a first pass of the translation, which you then correct in a second pass review.

Language workgroups have found the following tools helpful:

- [DocTranslator](#)
- [TexTra](#)

Suggested next topics

Once you have published the mandatory initial set of topics on the documentation website, you are encouraged to translate these topics, in order. If you choose to adopt another order, that's fine; you still will find it helpful to agree an order of translation in your workgroup.

- [Key concepts](#)

For solution architects, application architects, systems architects, developers, academics and students alike, this topic provides a comprehensive conceptual understanding of Fabric.

- [Getting started](#)

For developers who want to get hands-on with Fabric, this topic provides key instructions to help install, build a sample network and get hands-on with Fabric.

- [Developing applications](#)

This topic helps developers write smart contracts and applications; these are the core elements of any solution that uses Fabric.

- [Tutorials](#)

A set of hands-on tutorials to help developers and administrators try out specific Fabric features and capabilities.

- [What's new in Hyperledger Fabric v2.x](#)

This topic covers the latest features in Hyperledger Fabric.

Finally, we wish you good luck, and thank you for your contribution to Hyperledger Fabric.

14.3.4 Style guide for contributors

Audience: documentation writers and editors

While this style guide will also refer to best practices using ReStructured Text (also known as RST), in general we advise writing documentation in Markdown, as it's a more universally accepted documentation standard. Both formats are usable, however, and if you decide to write a topic in RST (or are editing an RST topic), be sure to refer to this style guide.

When in doubt, use the docs themselves for guidance on how to format things.

- [For RST formatting.](#)
- [For Markdown formatting.](#)

If you just want to look at how things are formatted, you can navigate to the Fabric repo to look at the raw file by clicking on [Edit on Github](#) link in the upper right hand corner of the page. Then click the [Raw](#) tab. This will show you the formatting of the doc. **Do not attempt to edit the file on Github.** If you want to make a change, clone the repo and follow the instructions in [Contributing](#) for creating pull requests.

Word choices

Avoid the use of the words “whitelist”, “blacklist”, “master”, or “slave”.

Unless the use of these words is absolutely necessary (for example, when quoting a section of code that uses them), do not use these words. Either be more explicit (for example, describing what “whitelisting” actually does) or find alternate words such as “allowlist” or “blocklist”.

Tutorials should have a list of steps at the top.

A list of steps (with links to the corresponding sections) at the beginning of a tutorial helps users find particular steps they're interested in. For an example, check out [Use private data in Fabric](#).

“Fabric”, “Hyperledger Fabric” or “HLF”?

The first usage should be “Hyperledger Fabric” and afterwards only “Fabric”. Don't use “HLF” or “Hyperledger” by itself.

Chaincode vs. Chaincodes?

One chaincode is a “chaincode”. If you're talking about several chaincodes, use “chaincodes”.

Smart contracts?

Colloquially, smart contracts are considered equivalent to chaincode, though at a technical level, it is more correct to say that a “smart contract” is the business logic inside of a chaincode, which encompasses the larger packaging and implementation.

JSON vs .json?

Use “JSON”. The same applies for any file format (for example, YAML).

curl vs cURL.

The tool is called “cURL”. The commands themselves are “curl” commands.

Fabric CA.

Do not call it “fabric-CA”, “fabricCA”, or FabricCA. It is the Fabric CA. The Fabric CA client binary can, however, be referred to as the `fabric-ca-client`.

Raft and RAFT.

“Raft” is not an acronym. Do not call it a “RAFT ordering service”.

Referring to the reader.

It’s perfectly fine to use the “you” or “we”. Avoid using “I”.

Ampersands (&).

Not a substitute for the word “and”. Avoid them unless you have a reason to use it (such as in a code snippet that includes it).

Acronyms.

The first usage of an acronym should be spelled out, unless it’s an acronym that’s in such wide usage this is unneeded. For example, “Software Development Kit (SDK)” on first usage. Then use “SDK” afterward.

Try to avoid using the same words too often.

If you can avoid using a word twice in one sentence, please do so. Not using it more than twice in a single paragraph is better. Of course sometimes it might not be possible to avoid this — a doc about the state database being used is likely to be replete with uses of the word “database” or “ledger”. But excessive usage of any particular word has a tendency to have a numbing effect on the reader.

How should files be named?

By using underscores between words. Also, tutorials should be named as such. For example, `identity_use_case_tutorial.md`. While not all files use this standard, new files should adhere to it.

Formatting and punctuation**Line lengths.**

If you look at the raw versions of the documentation, you will see that many topics conform to a line length of roughly 70 characters. This restriction is no longer necessary, so you are free to make lines as long as you want.

When to bold?

Not too often. The best use of them is either as a summary or as a way of drawing attention to concepts you want to talk about. “A blockchain network contains a ledger, at least one chaincode, and peers”, especially if you’re going to be talking about those things in that paragraph. Avoid using them simply to emphasize a single word, as in something like “Blockchain networks **must** use property security protocols”.

When to surround something in back ticks `nnn`?

This is useful to draw attention to words that either don’t make sense in plain English or when referencing parts of the code (especially if you’ve put code snippets in your doc). So for example, when talking about the `fabric-samples` directory, surround `fabric-samples` with back ticks. Same with a code function like `hf.Revoker`. It might also make sense to put back ticks around words that do make sense in plain English that are part of the code if you’re referencing them in a code context. For example, when referencing an `attribute` as part of an Access Control List.

Is it ever appropriate to use a dash?

Dashes can be incredibly useful but they're not necessarily as technically appropriate as using separate declarative sentences. Let's consider this example sentence:

```
This leaves us with a trimmed down JSON object --- config.json, located in the fabric-  
→samples folder inside first-network --- which will serve as the baseline for our_  
→config update.
```

There are a number of ways to present this same information, but in this case the dashes break up the information while keeping it as part of the same thought. If you use a dash, make sure to use the “em” dash, which is three times longer than a hyphen. These dashes should have a space before and after them.

When to use hyphens?

Hyphens are mostly commonly used as part of a “compound adjective”. For example, “jet-powered car”. Note that the compound adjective must immediately precede the noun being modified. In other words, “jet powered” does not by itself need a hyphen. When in doubt, use Google, as compound adjectives are tricky and are a popular discussion on grammar discussion boards.

How many spaces after a period?

One.

How should numbers be rendered?

Number zero through nine are spelled out. One, two, three, four, etc. Numbers after 10 are rendered as numbers.

Exceptions to this would be usages from code. In that case, use whatever's in the code. And also examples like Org1. Don't write it as OrgOne.

Capitalization rules for doc titles.

The standard rules for capitalization in sentences should be followed. In other words, unless a word is the first word in the title or a proper noun, do not capitalize its first letter. For example, “Understanding Identities in Fabric” should be “Understanding identities in Fabric”. While not every doc follows this standard yet, it is the standard we're moving to and should be followed for new topics.

Headings inside of topics should follow the same standard.

Use the Oxford comma?

Yes, it's better.

The classic example is, “I'd like to thank my parents, Ayn Rand and God”, as compared to: “I'd like to thank my parents, Ayn Rand, and God.”

Captions.

These should be in italics, and it's the only real valid use for italics in our docs.

Commands.

In general, put each command in its own snippet. It reads better, especially when commands are long. An exception to this rule is when suggesting the export of a number of environment variables.

Code snippets.

In Markdown, if you want to post sample code, use three back tics to set off the snippet. For example:

```
Code goes here.  
  
Even more code goes here.  
  
And still more.
```

In RST, you will need to set off the code snippet using formatting similar to this:

```
.. code:: bash

    Code goes here.
```

You can substitute `bash` for a language like Java or Go, where appropriate.

Enumerated lists in markdown.

Note that in Markdown, enumerated lists will not work if you separate the numbers with a space. Markdown sees this as the start of a new list, not a continuation of the old one (every number will be 1.). If you need an enumerated list, you will have to use RST. Bulleted lists are a good substitute in Markdown, and are the recommended alternative.

Linking.

When linking to another doc, use relative links, not direct links. When naming a link, do not just call it “link”. Use a more creative and descriptive name. For accessibility reasons, the link name should also make it clear that it is a link.

All docs have to end with a license statement.

In RST, it’s this:

```
.. Licensed under Creative Commons Attribution 4.0 International License
   https://creativecommons.org/licenses/by/4.0/
```

In markdown:

```
<!--- Licensed under Creative Commons Attribution 4.0 International License
https://creativecommons.org/licenses/by/4.0/ -->
```

How many spaces for indentation?

This will depend on the use case. Frequently it’s necessary, especially in RST, to indent two spaces, especially in a code block. In a `.. note::` box in RST, you have to indent to the space after the colon after `note`, like this:

```
.. note:: Some words and stuff etc etc etc (line continues until the 70 character_
↪limit line)
        the line directly below has to start at the same space as the one above.
```

When to use which type of heading.

In RST, use this:

```
Chapter 1 Title
=====

Section 1.1 Title
-----

Subsection 1.1.1 Title
~~~~~~~~~~~~~~~~~~~~~

Section 1.2 Title
-----
```

Note that the length of what’s under the title has to be the same as the length of the title itself. This isn’t a problem in Atom, which gives each character the same width by default (this is called “monospacing”, if you’re ever on Jeopardy! and need that information).

In markdown, it’s somewhat simpler. You go:

```
# The Name of the Doc (this will get pulled for the TOC).  
  
## First subsection  
  
## Second subsection
```

Both file formats don't like when these things are done out of order. For example, you might want a ##### to be the first thing after your # Title. Markdown won't allow it. Similarly, RST will default to whatever order you give to the title formats (as they appear in the first sections of your doc).

Relative links should be used whenever possible.

For RST, the preferred syntax is:

```
:doc:`anchor text <relativepath>`
```

Do not put the .rst suffix at the end of the filepath.

For Markdown, the preferred syntax is:

```
[anchor text](<relativepath>)
```

For other files, such as text or YAML files, use a direct link to the file in github for example:

<https://github.com/hyperledger/fabric/blob/master/docs/README.md>

Relative links are unfortunately not working on github when browsing through a RST file.

14.4 Project Governance

Hyperledger Fabric is managed under an open governance model as described in our [charter](#). Projects and sub-projects are lead by a set of maintainers. New sub-projects can designate an initial set of maintainers that will be approved by the top-level project's existing maintainers when the project is first approved.

14.4.1 Maintainers

The Fabric project is lead by the project's top level [maintainers](#). The maintainers are responsible for reviewing and merging all patches submitted for review, and they guide the overall technical direction of the project within the guidelines established by the Hyperledger Technical Steering Committee (TSC).

14.4.2 Becoming a maintainer

The project's maintainers will, from time-to-time, consider adding a maintainer, based on the following criteria:

- Demonstrated track record of PR reviews (both quality and quantity of reviews)
- Demonstrated thought leadership in the project
- Demonstrated shepherding of project work and contributors

An existing maintainer can submit a pull request to the [maintainers](#) file. A nominated Contributor may become a Maintainer by a majority approval of the proposal by the existing Maintainers. Once approved, the change set is then merged and the individual is added to the maintainers group.

Maintainers may be removed by explicit resignation, for prolonged inactivity (e.g. 3 or more months with no review comments), or for some infraction of the [code of conduct](#) or by consistently demonstrating poor judgement. A proposed removal also requires a majority approval. A maintainer removed for inactivity should be restored following a sustained resumption of contributions and reviews (a month or more) demonstrating a renewed commitment to the project.

14.4.3 Releases

Fabric provides a release approximately once every four months with new features and improvements. New feature work is merged to the Fabric main branch on [GitHub](#). Releases branches are created prior to each release so that the code can stabilize while new features continue to get merged to the master branch. Important fixes will also be backported to the most recent LTS (long-term support) release branch, and to the prior LTS release branch during periods of LTS release overlap.

See [releases](#) for more details.

14.4.4 Making Feature/Enhancement Proposals

Minor improvements can be implemented and reviewed via the normal [GitHub pull request workflow](#) but for changes that are more substantial Fabric follows the RFC (request for comments) process.

This process is intended to provide a consistent and controlled path for major changes to Fabric and other official project components, so that all stakeholders can be confident about the direction in which Fabric is evolving.

To propose a new feature, first, check the [GitHub issues backlog](#) and the [Fabric RFC repository](#) to be sure that there isn't already an open (or recently closed) proposal for the same functionality. If there isn't, follow [the RFC process](#) to make a proposal.

14.4.5 Contributor meeting

The maintainers hold regular contributors meetings. The purpose of the contributors meeting is to plan for and review the progress of releases and contributions, and to discuss the technical and operational direction of the project and sub-projects.

Please see the [wiki](#) for maintainer meeting details.

New feature/enhancement proposals as described above should be presented to a maintainers meeting for consideration, feedback and acceptance.

14.4.6 Release roadmap

The Fabric release roadmap is managed as a list of [GitHub issues with Epic label](#).

14.4.7 Communications

We use [RocketChat](#) for communication and Google Hangouts™ for screen sharing between developers. Our development planning and prioritization is done using a [GitHub Issues ZenHub board](#), and we take longer running discussions/decisions to the [Fabric contributor meeting or mailing list](#).

14.5 Contribution guide

14.5.1 Install prerequisites

Before we begin, if you haven't already done so, you may wish to check that you have all the *prerequisites* installed on the platform(s) on which you'll be developing blockchain applications and/or operating Hyperledger Fabric.

14.5.2 Getting help

If you are looking for something to work on, or need some expert assistance in debugging a problem or working out a fix to an issue, our *community* is always eager to help. We hang out on [Chat](#), IRC (#hyperledger on freenode.net) and the [mailing lists](#). Most of us don't bite :grin: and will be glad to help. The only silly question is the one you don't ask. Questions are in fact a great way to help improve the project as they highlight where our documentation could be clearer.

14.5.3 Reporting bugs

If you are a user and you have found a bug, please submit an issue using [GitHub Issues](#). Before you create a new GitHub issue, please try to search the existing issues to be sure no one else has previously reported it. If it has been previously reported, then you might add a comment that you also are interested in seeing the defect fixed.

Note: If the defect is security-related, please follow the Hyperledger [security bug reporting process](#).

If it has not been previously reported, you may either submit a PR with a well documented commit message describing the defect and the fix, or you may create a new GitHub issue. Please try to provide sufficient information for someone else to reproduce the issue. One of the project's maintainers should respond to your issue within 24 hours. If not, please bump the issue with a comment and request that it be reviewed. You can also post to the relevant Hyperledger Fabric channel in [Hyperledger Chat](#). For example, a doc bug should be broadcast to #fabric-documentation, a database bug to #fabric-ledger, and so on...

14.5.4 Submitting your fix

If you just submitted a GitHub issue for a bug you've discovered, and would like to provide a fix, we would welcome that gladly! Please assign the GitHub issue to yourself, then submit a pull request (PR). Please refer to [github/github](#) for a detailed workflow.

14.5.5 Fixing issues and working stories

Fabric issues and bugs are managed in [GitHub issues](#). Review the list of issues and find something that interests you. You could also check the "good first issue" list. It is wise to start with something relatively straight forward and achievable, and that no one is already assigned. If no one is assigned, then assign the issue to yourself. Please be considerate and rescind the assignment if you cannot finish in a reasonable time, or add a comment saying that you are still actively working the issue if you need a little more time.

While GitHub issues tracks a backlog of known issues that could be worked in the future, if you intend to immediately work on a change that does not yet have a corresponding issue, you can submit a pull request to [Github](#) without linking to an existing issue.

14.5.6 Reviewing submitted Pull Requests (PRs)

Another way to contribute and learn about Hyperledger Fabric is to help the maintainers with the review of the PRs that are open. Indeed maintainers have the difficult role of having to review all the PRs that are being submitted and evaluate whether they should be merged or not. You can review the code and/or documentation changes, test the changes, and tell the submitters and maintainers what you think. Once your review and/or test is complete just reply to the PR with your findings, by adding comments and/or voting. A comment saying something like “I tried it on system X and it works” or possibly “I got an error on system X: xxx ” will help the maintainers in their evaluation. As a result, maintainers will be able to process PRs faster and everybody will gain from it.

Just browse through [the open PRs on GitHub](#) to get started.

14.5.7 PR Aging

As the Fabric project has grown, so too has the backlog of open PRs. One problem that nearly all projects face is effectively managing that backlog and Fabric is no exception. In an effort to keep the backlog of Fabric and related project PRs manageable, we are introducing an aging policy which will be enforced by bots. This is consistent with how other large projects manage their PR backlog.

14.5.8 PR Aging Policy

The Fabric project maintainers will automatically monitor all PR activity for delinquency. If a PR has not been updated in 2 weeks, a reminder comment will be added requesting that the PR either be updated to address any outstanding comments or abandoned if it is to be withdrawn. If a delinquent PR goes another 2 weeks without an update, it will be automatically abandoned. If a PR has aged more than 2 months since it was originally submitted, even if it has activity, it will be flagged for maintainer review.

If a submitted PR has passed all validation but has not been reviewed in 72 hours (3 days), it will be flagged to the #fabric-pr-review channel daily until it receives a review comment(s).

This policy applies to all official Fabric projects (fabric, fabric-ca, fabric-samples, fabric-test, fabric-sdk-node, fabric-sdk-java, fabric-sdk-go, fabric-gateway-java, fabric-chaincode-node, fabric-chaincode-java, fabric-chaincode-evm, fabric-baseimage, and fabric-amcl).

14.5.9 Setting up development environment

Next, try *building the project* in your local development environment to ensure that everything is set up correctly.

14.5.10 What makes a good pull request?

- One change at a time. Not five, not three, not ten. One and only one. Why? Because it limits the blast area of the change. If we have a regression, it is much easier to identify the culprit commit than if we have some composite change that impacts more of the code.
- If there is a corresponding GitHub issue, include a link to the GitHub issue in the PR summary and commit message. Why? Because there will often be additional discussion around a proposed change or bug in the GitHub issue. Additionally, if you use syntax like “Resolves #<GitHub issue number>” in the PR summary and commit message, the GitHub issue will automatically be closed when the PR is merged.
- Include unit and integration tests (or changes to existing tests) with every change. This does not mean just happy path testing, either. It also means negative testing of any defensive code that it correctly catches input errors. When you write code, you are responsible to test it and provide the tests that demonstrate that your change does what it claims. Why? Because without this we have no clue whether our current code base actually works.

- Unit tests should have NO external dependencies. You should be able to run unit tests in place with `go test` or equivalent for the language. Any test that requires some external dependency (e.g. needs to be scripted to run another component) needs appropriate mocking. Anything else is not unit testing, it is integration testing by definition. Why? Because many open source developers do Test Driven Development. They place a watch on the directory that invokes the tests automatically as the code is changed. This is far more efficient than having to run a whole build between code changes. See [this definition](#) of unit testing for a good set of criteria to keep in mind for writing effective unit tests.
- Minimize the lines of code per PR. Why? Maintainers have day jobs, too. If you send a 1,000 or 2,000 LOC change, how long do you think it takes to review all of that code? Keep your changes to < 200-300 LOC, if possible. If you have a larger change, decompose it into multiple independent changes. If you are adding a bunch of new functions to fulfill the requirements of a new capability, add them separately with their tests, and then write the code that uses them to deliver the capability. Of course, there are always exceptions. If you add a small change and then add 300 LOC of tests, you will be forgiven;-) If you need to make a change that has broad impact or a bunch of generated code (protobufs, etc.). Again, there can be exceptions.

Note: Large pull requests, e.g. those with more than 300 LOC are more than likely not going to receive an approval, and you'll be asked to refactor the change to conform with this guidance.

- Write a meaningful commit message. Include a meaningful 55 (or less) character title, followed by a blank line, followed by a more comprehensive description of the change.

Note: Example commit message:

```
[FAB-1234] fix foobar() panic

Fix [FAB-1234] added a check to ensure that when foobar(foo string)
is called, that there is a non-empty string argument.
```

Finally, be responsive. Don't let a pull request fester with review comments such that it gets to a point that it requires a rebase. It only further delays getting it merged and adds more work for you - to remediate the merge conflicts.

14.6 Legal stuff

Note: Each source file must include a license header for the Apache Software License 2.0. See the template of the [license header](#).

We have tried to make it as easy as possible to make contributions. This applies to how we handle the legal aspects of contribution. We use the same approach—the [Developer's Certificate of Origin 1.1 \(DCO\)](#)—that the Linux® Kernel community uses to manage code contributions.

We simply ask that when submitting a patch for review, the developer must include a sign-off statement in the commit message.

Here is an example Signed-off-by line, which indicates that the submitter accepts the DCO:

```
Signed-off-by: John Doe <john.doe@example.com>
```

You can include this automatically when you commit a change to your local git repository using `git commit -s`.

14.7 Related Topics

14.7.1 Setting up the development environment

Prerequisites

- [Git client](#)
- [Go version 1.15.x](#)
- [Docker version 18.03 or later](#)
- (macOS) [Xcode Command Line Tools](#)
- [SoftHSM](#)
- [jq](#)

Steps

Install the Prerequisites

For macOS, we recommend using [Homebrew](#) to manage the development prereqs. The Xcode command line tools will be installed as part of the Homebrew installation.

Once Homebrew is ready, installing the necessary prerequisites is very easy:

```
brew install git go jq softhsm
brew cask install --appdir="/Applications" docker
```

Docker Desktop must be launched to complete the installation so be sure to open the application after installing it:

```
open /Applications/Docker.app
```

Developing on Windows

On Windows 10 you should use the native Docker distribution and you may use the Windows PowerShell. However, for the `binaries` command to succeed you will still need to have the `uname` command available. You can get it as part of Git but beware that only the 64bit version is supported.

Before running any `git clone` commands, run the following commands:

```
git config --global core.autocrlf false
git config --global core.longpaths true
```

You can check the setting of these parameters with the following commands:

```
git config --get core.autocrlf
git config --get core.longpaths
```

These need to be `false` and `true` respectively.

The `curl` command that comes with Git and Docker Toolbox is old and does not handle properly the redirect used in [Getting Started](#). Make sure you have and use a newer version which can be downloaded from the [cURL downloads](#) page

Clone the Hyperledger Fabric source

First navigate to <https://github.com/hyperledger/fabric> and fork the fabric repository using the fork button in the top-right corner. After forking, clone the repository.

```
mkdir -p github.com/<your_github_userid>
cd github.com/<your_github_userid>
git clone https://github.com/<your_github_userid>/fabric
```

Note: If you are running Windows, before cloning the repository, run the following command:

```
git config --get core.autocrlf
```

If `core.autocrlf` is set to `true`, you must set it to `false` by running:

```
git config --global core.autocrlf false
```

Configure SoftHSM

A PKCS #11 cryptographic token implementation is required to run the unit tests. The PKCS #11 API is used by the `bccsp` component of Fabric to interact with hardware security modules (HSMs) that store cryptographic information and perform cryptographic computations. For test environments, SoftHSM can be used to satisfy this requirement.

SoftHSM generally requires additional configuration before it can be used. For example, the default configuration will attempt to store token data in a system directory that unprivileged users are unable to write to.

SoftHSM configuration typically involves copying `/etc/softhsm/softhsm2.conf` to `$HOME/.config/softhsm2/softhsm2.conf` and changing `directories.token_dir` to an appropriate location. Please see the man page for `softhsm2.conf` for details.

After SoftHSM has been configured, the following command can be used to initialize the token required by the unit tests:

```
softhsm2-util --init-token --slot 0 --label "ForFabric" --so-pin 1234 --pin 98765432
```

If tests are unable to locate the `libsofthsm2.so` library in your environment, specify the library path, the PIN, and the label of your token in the appropriate environment variables. For example, on macOS:

```
export PKCS11_LIB="/usr/local/Cellar/softhsm/2.6.1/lib/softhsm/libsofthsm2.so"
export PKCS11_PIN=98765432
export PKCS11_LABEL="ForFabric"
```

Install the development tools

Once the repository is cloned, you can use `make` to install some of the tools used in the development environment. By default, these tools will be installed into `$HOME/go/bin`. Please be sure your `PATH` includes that directory.

```
make gotools
```

After installing the tools, the build environment can be verified by running a few commands.

```
make basic-checks integration-test-prereqs
ginkgo -r ./integration/nwo
```

If those commands completely successfully, you're ready to Go!

If you plan to use the Hyperledger Fabric application SDKs then be sure to check out their prerequisites in the Node.js SDK [README](#), Java SDK [README](#), and Go SDK [README](#).

14.7.2 Building Hyperledger Fabric

The following instructions assume that you have already set up your *development environment*.

To build Hyperledger Fabric:

```
make dist-clean all
```

Building the documentation

If you are contributing to the documentation, you can build the Fabric documentation on your local machine. This allows you to check the formatting of your changes using your web browser before you open a pull request.

You need to download the following prerequisites before you can build the documentation:

- [Docker](#)

After you make your updates to the documentation source files, you can generate a build that includes your changes by running the following commands:

```
cd fabric
make docs
```

This will generate all the html files in the `docs/build/html` folder. You can open any file to start browsing the updated documentation using your browser. If you want to make additional edits to the documentation, you can rerun `make html` to incorporate the changes.

Running the unit tests

Use the following command to run all unit tests:

```
make unit-test
```

To run a subset of tests, set the `TEST_PKGS` environment variable. Specify a list of packages (separated by space), for example:

```
export TEST_PKGS="github.com/hyperledger/fabric/core/ledger/..."
make unit-test
```

To run a specific test use the `-run RE` flag where `RE` is a regular expression that matches the test case name. To run tests with verbose output use the `-v` flag. For example, to run the `TestGetFoo` test case, change to the directory containing the `foo_test.go` and call/execute

```
go test -v -run=TestGetFoo
```

Running Node.js Client SDK Unit Tests

You must also run the Node.js unit tests to ensure that the Node.js client SDK is not broken by your changes. To run the Node.js unit tests, follow the instructions [here](#).

14.7.3 Configuration

Configuration utilizes the [viper](#) and [cobra](#) libraries.

There is a **core.yaml** file that contains the configuration for the peer process. Many of the configuration settings can be overridden on the command line by setting ENV variables that match the configuration setting, but by prefixing with `'CORE_'`. For example, setting `peer.networkId` can be accomplished with:

```
CORE_PEER_NETWORKID=custom-network-id peer
```

14.7.4 Coding guidelines

Coding in Go

We code in Go™ and try to follow the best practices and style outlined in [Effective Go](#) and the supplemental rules from the [Go Code Review Comments](#) wiki.

We also recommend new contributors review the following before submitting pull requests:

- [Practical Go](#)
- [Go Proverbs](#)

The following tools are executed against all pull requests. Any errors flagged by these tools must be addressed before the code will be merged:

- `gofmt -s`
- `goimports`
- `go vet`

Testing

Unit tests are expected to accompany all production code changes. These tests should be fast, provide very good coverage for new and modified code, and support parallel execution.

Two matching libraries are commonly used in our tests. When modifying code, please use the matching library that has already been chosen for the package.

- [gomega](#)
- [testify/require](#)

Any fixtures or data required by tests should be generated or placed under version control. When fixtures are generated, they must be placed in a temporary directory created by `ioutil.TempDir` and cleaned up when the test terminates. When fixtures are placed under version control, they should be created inside a `testdata` folder; documentation that describes how to regenerate the fixtures should be provided in the tests or a `README.txt`. Sharing fixtures across packages is strongly discouraged.

When fakes or mocks are needed, they must be generated. Bespoke, hand-coded mocks are a maintenance burden and tend to include simulations that inevitably diverge from reality. Within Fabric, we use `go generate` directives to manage the generation with the following tools:

- `counterfeiter`
- `mockery`

API Documentation

The API documentation for Hyperledger Fabric's Go APIs is available in [GoDoc](#).

14.7.5 Adding or updating Go packages

Hyperledger Fabric uses `go` modules to manage and vendor its dependencies. This means that all of the external packages required to build our binaries reside in the `vendor` folder at the top of the repository. Go uses the packages in this folder instead of the module cache when `go` commands are executed.

If a code change results in a new or updated dependency, please be sure to run `go mod tidy` and `go mod vendor` to keep the `vendor` folder and dependency metadata up to date.

See the [Go Modules Wiki](#) for additional information.

Terminology is important, so that all Hyperledger Fabric users and developers agree on what we mean by each specific term. What is a smart contract for example. The documentation will reference the glossary as needed, but feel free to read the entire thing in one sitting if you like; it's pretty enlightening!

15.1 Anchor Peer

Used by gossip to make sure peers in different organizations know about each other.

When a configuration block that contains an update to the anchor peers is committed, peers reach out to the anchor peers and learn from them about all of the peers known to the anchor peer(s). Once at least one peer from each organization has contacted an anchor peer, the anchor peer learns about every peer in the channel. Since gossip communication is constant, and because peers always ask to be told about the existence of any peer they don't know about, a common view of membership can be established for a channel.

For example, let's assume we have three organizations — A, B, C — in the channel and a single anchor peer — `peer0.orgC` — defined for organization C. When `peer1.orgA` (from organization A) contacts `peer0.orgC`, it will tell `peer0.orgC` about `peer0.orgA`. And when at a later time `peer1.orgB` contacts `peer0.orgC`, the latter would tell the former about `peer0.orgA`. From that point forward, organizations A and B would start exchanging membership information directly without any assistance from `peer0.orgC`.

As communication across organizations depends on gossip in order to work, there must be at least one anchor peer defined in the channel configuration. It is strongly recommended that every organization provides its own set of anchor peers for high availability and redundancy.

15.2 ACL

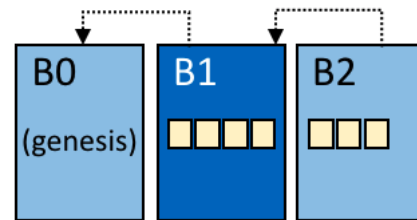
An ACL, or Access Control List, associates access to specific peer resources (such as system chaincode APIs or event services) to a *Policy* (which specifies how many and what types of organizations or roles are required). The ACL is part of a channel's configuration. It is therefore persisted in the channel's configuration blocks, and can be updated using the standard configuration update mechanism.

An ACL is formatted as a list of key-value pairs, where the key identifies the resource whose access we wish to control, and the value identifies the channel policy (group) that is allowed to access it. For example `lscc/GetDeploymentSpec: /Channel/Application/Readers` defines that the access to the life cycle chaincode `GetDeploymentSpec` API (the resource) is accessible by identities which satisfy the `/Channel/Application/Readers` policy.

A set of default ACLs is provided in the `configtx.yaml` file which is used by `configtxgen` to build channel configurations. The defaults can be set in the top level “Application” section of `configtx.yaml` or overridden on a per profile basis in the “Profiles” section.

15.3 Block

A block contains an ordered set of transactions. It is cryptographically linked to the preceding block, and in turn it is linked to be subsequent blocks. The first block in such a chain of blocks is called the **genesis block**. Blocks are created by the ordering service, and then validated and committed by peers.



15.4 Chain

The ledger’s chain is a transaction log structured as hash-linked blocks of transactions. Peers receive blocks of transactions from the ordering service, mark the block’s transactions as valid or invalid based on endorsement policies and concurrency violations, and append the block to the hash chain on the peer’s file system.

Fig. 1: Block B1 is linked to block B0. Block B2 is linked to block B1.

15.5 Chaincode

See *Smart-Contract*.

15.6 Channel

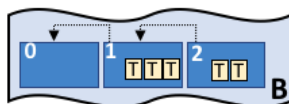


Fig. 2: Blockchain B contains blocks 0, 1, 2.

A channel is a private blockchain overlay which allows for data isolation and confidentiality. A channel-specific ledger is shared across the peers in the channel, and transacting parties must be authenticated to a channel in order to interact with it. Channels are defined by a *Configuration-Block*.

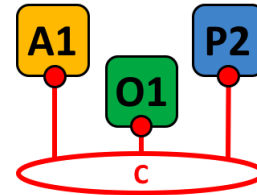


Fig. 3: Channel C connects application A1, peer P2 and ordering service O1.

15.7 Commit

Each *Peer* on a channel validates ordered blocks of transactions and then commits (writes/appends) the blocks to its replica of the channel *Ledger*. Peers also mark each transaction in each block as valid or invalid.

15.8 Concurrency Control Version Check

Concurrency Control Version Check is a method of keeping ledger state in sync across peers on a channel. Peers execute transactions in parallel, and before committing to the ledger, peers check whether the state read at the time the transaction was executed has been modified in a new block that was in-flight at time of execution or in a prior transaction in the same block. If the data read for the transaction has changed between execution time and commit time, then a Concurrency Control Version Check violation has occurred, and the transaction is marked as invalid on the ledger and values are not updated in the state database.

15.9 Configuration Block

Contains the configuration data defining members and policies for a system chain (ordering service) or channel. Any configuration modifications to a channel or overall network (e.g. a member leaving or joining) will result in a new configuration block being appended to the appropriate chain. This block will contain the contents of the genesis block, plus the delta.

15.10 Consensus

A broader term overarching the entire transactional flow, which serves to generate an agreement on the order and to confirm the correctness of the set of transactions constituting a block.

15.11 Consenter set

In a Raft ordering service, these are the ordering nodes actively participating in the consensus mechanism on a channel. If other ordering nodes exist on the system channel, but are not a part of a channel, they are not part of that channel's consenter set.

15.12 Consortium

A consortium is a collection of non-orderer organizations on the blockchain network. These are the organizations that form and join channels and that own peers. While a blockchain network can have multiple consortia, most blockchain networks have a single consortium. At channel creation time, all organizations added to the channel must be part of a consortium. However, an organization that is not defined in a consortium may be added to an existing channel.

15.13 Chaincode definition

A chaincode definition is used by organizations to agree on the parameters of a chaincode before it can be used on a channel. Each channel member that wants to use the chaincode to endorse transactions or query the ledger needs to approve a chaincode definition for their organization. Once enough channel members have approved a chaincode definition to meet the Lifecycle Endorsement policy (which is set to a majority of organizations in the channel by default), the chaincode definition can be committed to the channel. After the definition is committed, the first invoke of the chaincode (or, if requested, the execution of the Init function) will start the chaincode on the channel.

15.14 Dynamic Membership

Hyperledger Fabric supports the addition/removal of members, peers, and ordering service nodes, without compromising the operability of the overall network. Dynamic membership is critical when business relationships adjust and entities need to be added/removed for various reasons.

15.15 Endorsement

Refers to the process where specific peer nodes execute a chaincode transaction and return a proposal response to the client application. The proposal response includes the chaincode execution response message, results (read set and write set), and events, as well as a signature to serve as proof of the peer's chaincode execution. Chaincode applications have corresponding endorsement policies, in which the endorsing peers are specified.

15.16 Endorsement policy

Defines the peer nodes on a channel that must execute transactions attached to a specific chaincode application, and the required combination of responses (endorsements). A policy could require that a transaction be endorsed by a minimum number of endorsing peers, a minimum percentage of endorsing peers, or by all endorsing peers that are assigned to a specific chaincode application. Policies can be curated based on the application and the desired level of resilience against misbehavior (deliberate or not) by the endorsing peers. A transaction that is submitted must satisfy the endorsement policy before being marked as valid by committing peers.

15.17 Follower

In a leader based consensus protocol, such as Raft, these are the nodes which replicate log entries produced by the leader. In Raft, the followers also receive “heartbeat” messages from the leader. In the event that the leader stops sending those message for a configurable amount of time, the followers will initiate a leader election and one of them will be elected leader.

15.18 Genesis Block

The configuration block that initializes the ordering service, or serves as the first block on a chain.

15.19 Gossip Protocol

The gossip data dissemination protocol performs three functions: 1) manages peer discovery and channel membership; 2) disseminates ledger data across all peers on the channel; 3) syncs ledger state across all peers on the channel. Refer to the *Gossip* topic for more details.

15.20 Hyperledger Fabric CA

Hyperledger Fabric CA is the default Certificate Authority component, which issues PKI-based certificates to network member organizations and their users. The CA issues one root certificate (rootCert) to each member and one enrollment certificate (ECert) to each authorized user.

15.21 Init

A method to initialize a chaincode application. All chaincodes need to have an `Init` function. By default, this function is never executed. However you can use the chaincode definition to request the execution of the `Init` function in order to initialize the chaincode.

15.22 Install

The process of placing a chaincode on a peer's file system.

15.23 Instantiate

The process of starting and initializing a chaincode application on a specific channel. After instantiation, peers that have the chaincode installed can accept chaincode invocations.

NOTE: *This method i.e. Instantiate was used in the 1.4.x and older versions of the chaincode lifecycle. For the current procedure used to start a chaincode on a channel with the new Fabric chaincode lifecycle introduced as part of Fabric v2.0, see Chaincode-definition_.*

15.24 Invoke

Used to call chaincode functions. A client application invokes chaincode by sending a transaction proposal to a peer. The peer will execute the chaincode and return an endorsed proposal response to the client application. The client application will gather enough proposal responses to satisfy an endorsement policy, and will then submit the transaction results for ordering, validation, and commit. The client application may choose not to submit the transaction results. For example if the invoke only queried the ledger, the client application typically would not submit the read-only transaction, unless there is desire to log the read on the ledger for audit purpose. The invoke includes a channel identifier, the chaincode function to invoke, and an array of arguments.

15.25 Leader

In a leader based consensus protocol, like Raft, the leader is responsible for ingesting new log entries, replicating them to follower ordering nodes, and managing when an entry is considered committed. This is not a special **type** of orderer. It is only a role that an orderer may have at certain times, and then not others, as circumstances determine.

15.26 Leading Peer

Each *Organization* can own multiple peers on each channel that they subscribe to. One or more of these peers should serve as the leading peer for the channel, in order to communicate with the network ordering service on behalf of the organization. The ordering service delivers blocks to the leading peer(s) on a channel, who then distribute them to other peers within the same organization.

15.27 Ledger

A ledger consists of two distinct, though related, parts – a “blockchain” and the “state database”, also known as “world state”. Unlike other ledgers, blockchains are **immutable** – that is, once a block has been added to the chain, it cannot be changed. In contrast, the “world state” is a database containing the current value of the set of key-value pairs that have been added, modified or deleted by the set of validated and committed transactions in the blockchain.

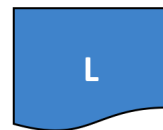


Fig. 4: A Ledger, ‘L’

It’s helpful to think of there being one **logical** ledger for each channel in the network. In reality, each peer in a channel maintains its own copy of the ledger – which is kept consistent with every other peer’s copy through a process called **consensus**. The term **Distributed Ledger Technology (DLT)** is often associated with this kind of ledger – one that is logically singular, but has many identical copies distributed across a set of network nodes (peers and the ordering service).

15.28 Log entry

The primary unit of work in a Raft ordering service, log entries are distributed from the leader orderer to the followers. The full sequence of such entries known as the “log”. The log is considered to be consistent if all members agree on the entries and their order.

15.29 Member

See *Organization*.

15.30 Membership Service Provider

The Membership Service Provider (MSP) refers to an abstract component of the system that provides credentials to clients, and peers for them to participate in a Hyperledger Fabric network. Clients use these credentials to authenticate their transactions, and peers use these credentials to authenticate transaction processing results (endorsements). While strongly connected to the transaction processing components of the



Chapter 15. Glossary

systems, this interface aims to have membership services components defined, in such a way that alternate implementations of this can be smoothly plugged in without modifying the core of transaction processing components of the system.

15.31 Membership Services

Membership Services authenticates, authorizes, and manages identities on a permissioned blockchain network. The membership services code that runs in peers and orderers both authenticates and authorizes blockchain operations. It is a PKI-based implementation of the Membership Services Provider (MSP) abstraction.

15.32 Ordering Service

Also known as **orderer**. A defined collective of nodes that orders transactions into a block and then distributes blocks to connected peers for validation and commit. The ordering service exists independent of the peer processes and orders transactions on a first-come-first-serve basis for all channels on the network. It is designed to support pluggable implementations beyond the out-of-the-box Kafka and Raft varieties. It is a common binding for the overall network; it contains the cryptographic identity material tied to each *Member*.

15.33 Organization

Also known as “members”, organizations are invited to join the blockchain network by a blockchain network provider. An organization is joined to a network by adding its Membership Service Provider (*MSP*) to the network. The MSP defines how other members of the network may verify that signatures (such as those over transactions) were generated by a valid identity, issued by that organization. The particular access rights of identities within an MSP are governed by policies which are also agreed upon when the organization is joined to the network. An organization can be as large as a multi-national corporation or as small as an individual. The transaction endpoint of an organization is a *Peer*. A collection of organizations form a *Consortium*. While all of the organizations on a network are members, not every organization will be part of a consortium.



Fig. 6: An organization, ‘ORG’

15.34 Peer

A network entity that maintains a ledger and runs chaincode containers in order to perform read/write operations to the ledger. Peers are owned and maintained by members.

15.35 Policy

Policies are expressions composed of properties of digital identities, for example: `OR('Org1.peer', 'Org2.peer')`. They are used to restrict access to resources on a blockchain network. For instance, they dictate who can read from or write to a channel, or



Fig. 7: A peer, ‘P’

who can use a specific chaincode API via an [ACL](#). Policies may be defined in `configtx.yaml` prior to bootstrapping an ordering service or creating a channel, or they can be specified when instantiating chaincode on a channel. A default set of policies ship in the sample `configtx.yaml` which will be appropriate for most networks.

15.36 Private Data

Confidential data that is stored in a private database on each authorized peer, logically separate from the channel ledger data. Access to this data is restricted to one or more organizations on a channel via a private data collection definition. Unauthorized organizations will have a hash of the private data on the channel ledger as evidence of the transaction data. Also, for further privacy, hashes of the private data go through the [Ordering-Service](#), not the private data itself, so this keeps private data confidential from Orderer.

15.37 Private Data Collection (Collection)

Used to manage confidential data that two or more organizations on a channel want to keep private from other organizations on that channel. The collection definition describes a subset of organizations on a channel entitled to store a set of private data, which by extension implies that only these organizations can transact with the private data.

15.38 Proposal

A request for endorsement that is aimed at specific peers on a channel. Each proposal is either an Init or an Invoke (read/write) request.

15.39 Query

A query is a chaincode invocation which reads the ledger current state but does not write to the ledger. The chaincode function may query certain keys on the ledger, or may query for a set of keys on the ledger. Since queries do not change ledger state, the client application will typically not submit these read-only transactions for ordering, validation, and commit. Although not typical, the client application can choose to submit the read-only transaction for ordering, validation, and commit, for example if the client wants auditable proof on the ledger chain that it had knowledge of specific ledger state at a certain point in time.

15.40 Quorum

This describes the minimum number of members of the cluster that need to affirm a proposal so that transactions can be ordered. For every consent set, this is a **majority** of nodes. In a cluster with five nodes, three must be available for there to be a quorum. If a quorum of nodes is unavailable for any reason, the cluster becomes unavailable for both read and write operations and no new logs can be committed.

15.41 Raft

New for v1.4.1, Raft is a crash fault tolerant (CFT) ordering service implementation based on the [etcd library](#) of the [Raft protocol](#). Raft follows a “leader and follower” model, where a leader node is elected (per channel) and its

decisions are replicated by the followers. Raft ordering services should be easier to set up and manage than Kafka-based ordering services, and their design allows organizations to contribute nodes to a distributed ordering service.

15.42 Software Development Kit (SDK)

The Hyperledger Fabric client SDK provides a structured environment of libraries for developers to write and test chaincode applications. The SDK is fully configurable and extensible through a standard interface. Components, including cryptographic algorithms for signatures, logging frameworks and state stores, are easily swapped in and out of the SDK. The SDK provides APIs for transaction processing, membership services, node traversal and event handling.

Currently, there are three officially supported SDKs – for Node.js, Java, and Go. While the Python SDK is not yet official but can still be downloaded and tested.

15.43 Smart Contract

A smart contract is code – invoked by a client application external to the blockchain network – that manages access and modifications to a set of key-value pairs in the *World State* via *Transaction*. In Hyperledger Fabric, smart contracts are packaged as chaincode. Chaincode is installed on peers and then defined and used on one or more channels.

15.44 State Database

World state data is stored in a state database for efficient reads and queries from chaincode. Supported databases include levelDB and couchDB.

15.45 System Chain

Contains a configuration block defining the network at a system level. The system chain lives within the ordering service, and similar to a channel, has an initial configuration containing information such as: MSP information, policies, and configuration details. Any change to the overall network (e.g. a new org joining or a new ordering node being added) will result in a new configuration block being added to the system chain.

The system chain can be thought of as the common binding for a channel or group of channels. For instance, a collection of financial institutions may form a consortium (represented through the system chain), and then proceed to create channels relative to their aligned and varying business agendas.

15.46 Transaction

Transactions are created when a chaincode is invoked from a client application to read or write data from the ledger. Fabric application clients submit transaction proposals to endorsing peers for execution and endorsement, gather the signed (endorsed) responses from those endorsing peers, and then package the results and endorsements into a transaction that is submitted to the ordering service. The ordering service orders and places transactions in a block that is broadcast to the peers which validate and commit the transactions to the ledger and update world state.

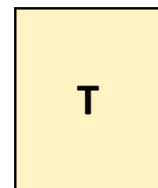


Fig. 8: A transaction, “T”

15.47 World State

Also known as the “current state”, the world state is a component of the HyperLedger Fabric *Ledger*. The world state represents the latest values for all keys included in the chain transaction log. Chaincode executes transaction proposals against world state data because the world state provides direct access to the latest value of these keys rather than having to calculate them by traversing the entire transaction log. The world state will change every time the value of a key changes (for example, when the ownership of a car – the “key” – is transferred from one owner to another – the “value”) or when a new key is added (a car is created). As a result, the world state is critical to a transaction flow, since the current state of a key-value pair must be known before it can be changed. Peers commit the latest values to the ledger world state for each valid transaction included in a processed block.

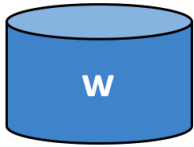


Fig. 9: The World State, ‘W’

CHAPTER 16

Releases

Hyperledger Fabric releases are documented on the [Fabric Github page](#).

CHAPTER 17

Still Have Questions?

We try to maintain a comprehensive set of documentation for various audiences. However, we realize that often there are questions that remain unanswered. For any technical questions relating to Hyperledger Fabric not answered here, please use [StackOverflow](#). Another approach to getting your questions answered is to send an email to the [mailing list](mailto:fabric@lists.hyperledger.org) (fabric@lists.hyperledger.org), or ask your questions on [RocketChat](#) (an alternative to Slack) on the [#fabric](#) or [#fabric-questions](#) channel.

Note: Please, when asking about problems you are facing tell us about the environment in which you are experiencing those problems including the OS, which version of Docker you are using, etc.

CHAPTER 18

Status

Hyperledger Fabric is in the *Active* state. For more information on the history of this project see our [wiki page](#). Information on what *Active* entails can be found in the Hyperledger [Project Lifecycle document](#).

Note: If you have questions not addressed by this documentation, or run into issues with any of the tutorials, please visit the [Still Have Questions?](#) page for some tips on where to find additional help.
