Contents

1. Deploying a smart contract to a channel

1.1 Start the network

1.2 Setup Logspout (optional)

1.3 Package the smart contract

1.4 Install the chaincode package

1.5 Approve a chaincode definition

1.6 Committing the chaincode definition to the channel

1.7 Invoking the chaincode

1.8 Upgrading a smart contract

1.9 Clean up

1.10 Next steps

1.11 troubleshooting

1.11.1 Chaincode not agreed to by this org

1.11.2 Invoke failure

1.12 Endorsement policy failure

2. Running a Fabric Application

2.1 Before you begin

2.2 Set up the blockchain network

2.2.1 Launch the blockchain network

2.2.2 Deploy the smart contract

2.2.3 Prepare the sample application

2.3 Run the sample application

2.3.1 First, establish a gRPC connection to the Gateway

2.3.2 Second, create a Gateway connection

2.3.3 Third, access the smart contract to be invoked

2.3.4 Fourth, populate the ledger with sample assets

2.3.5 Fifth, invoke transaction functions to read and write assets

2.4 Clean up

2.5 Summary

3. Using Private Data in Fabric

3.1 Asset transfer private data sample use case

3.2 Build a collection definition JSON file

3.3 Read and Write private data using chaincode APIs

3.3.1 Reading collection data

3.3.2 Writing private data

3.4 Start the network

3.5 Deploy the private data smart contract to the channel

3.6 Register identities

3.7 Create an asset in private data

3.8 Query the private data as an authorized peer

3.9 Query the private data as an unauthorized peer

3.9.1 Switch to a peer in Org2

3.9.2 Query private data Org2 is authorized to

3.9.3 Query private data Org2 is not authorized to

3.10 Transfer the Asset

3.11 Purge Private Data

3.12 Using indexes with private data

3.13 Clean up

4. Secured asset transfer in Fabric

4.1 Scenario requirements

4.2 How privacy is maintained

4.3 How the transfer is implemented

4.3.1 Creating the asset

4.3.2 Agreeing to the transfer

4.3.3 Transferring the asset

4.4 Running the secured asset transfer smart contract

4.5 Deploy the test network

4.6 Deploy the smart contract

4.6.1 Set the environment variables to operate as Org1

4.6.2 Set the environment variables to operate as Org2

4.7 Create an asset

4.7.1 Operate from the Org1 terminal

4.7.2 Operate from the Org2 terminal

4.8 Agree to sell the asset

4.9 Agree to sell as Org1

4.10 Agree to buy as Org2

4.11 Transfer the asset from Org1 to Org2

4.11.1 Transfer the asset as Org1

4.11.2 Update the asset description as Org2

4.12 Clean up

5. Using CouchDB

5.1 Why CouchDB?

5.2 Enable CouchDB in Hyperledger Fabric

5.3 Create an index

5.4 Add the index to your chaincode folder

5.4.1 Start the network

5.5 Deploy the smart contract

5.5.1 Verify index was deployed

5.6 Query the CouchDB State Database

5.6.1 Build the query in chaincode

5.6.2 Run the query using the peer command

5.7 Use best practices for queries and indexes

5.8 Query the CouchDB State Database With Pagination

5.8.1 Range Query Pagination

5.9 Update an Index

5.9.1 Iterating on your index definition

5.10 Delete an Index

5.11 Clean up

6. Create a channel

6.1 Folder structure

6.2 Prerequisites

6.2.1 Use existing ordering service

6.2.2 Deploy a new set of orderers

6.2.3 Define your peer organizations

6.3 Step one: Generate the genesis block of the channel

6.3.1 Set up the configtxgen tool

6.3.2 The configtx.yaml file

6.3.3 Generate the genesis block

6.4 Step two: Use the osnadmin CLI to add the first orderer to the channel

6.5 Step three: Join additional ordering nodes

6.6 Next steps

6.6.1 Join peers to the channel

6.6.2 Add or remove orderers from existing channels

7. Create a channel using the test network

7.1 Before you begin

7.2 Prerequisites

7.2.1 Start the test network

7.2.2 Set up the configtxgen tool

7.2.3 The configtx.yaml file

7.3 Step one: Generate the genesis block of the channel

7.4 Step two: Create the application channel

7.4.1 Consenter vs. Follower

7.4.2 Active vs. onboarding

7.5 Next steps

7.5.1 List channels on an orderer

7.5.2 Join peers to the channel

7.6 Set anchor peer

7.7 Deploy a chaincode to the new channel

7.7.1 Create a channel without the test network

8. Using configtx.yaml to build a channel configuration

8.1 Overview

8.2 Organizations

8.3 Capabilities

8.4 Application

8.5 Orderer

8.6 Channel

8.7 Profiles

8.7.1 TwoOrgsOrdererGenesis

8.7.2 TwoOrgsChannel

8.7.3 SampleAppChannelEtcdRaft

9. Adding an Org to a Channel

9.1 Setup the Environment

9.2 Bring Org3 into the Channel with the Script

9.3 Bring Org3 into the Channel Manually

9.4 Generate the Org3 Crypto Material

9.5 Bring up Org3 components

9.6 Fetch the Configuration

9.7 Convert the Configuration to JSON and Trim It Down

9.8 Add the Org3 Crypto Material

9.9 Sign and Submit the Config Update

9.10 Join Org3 to the Channel

9.11 Configuring Leader Election

9.12 Install, define, and invoke chaincode

9.13 Conclusion

9.14 Updating the Channel Config to include an Org3 Anchor Peer (Optional)

10. Updating a channel configuration

10.1 What is a channel configuration?

10.2 Channel parameters that can be updated

10.2.1 Sample channel configuration

10.2.2 More about these parameters

10.3 Editing a config

10.3.1 Set environment variables for your config update

10.3.2 Step 1: Pull and translate the config

10.3.3 Step 2: Modify the config

10.3.4 Step 3: Re-encode and submit the config

10.4 Get the Necessary Signatures

11. Writing Your First Chaincode

11.1 What is Chaincode?

11.2 Fabric Contract API

11.3 Asset Transfer Chaincode

11.3.1 Choosing a Location for the Code

11.3.2 Housekeeping

11.3.3 Initializing the Chaincode

11.3.4 Pulling it All Together

11.4 Chaincode access control

12. Running chaincode in development mode

12.1 Set up environment

12.2 Start the orderer

12.3 Start the peer in DevMode

12.4 Create channel and join peer

12.5 Build the chaincode

12.6 Start the chaincode

12.7 Approve and commit the chaincode definition

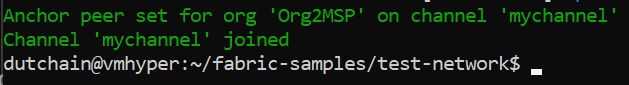
12.8 Next steps

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.

1.1 Start the network



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 test-network directory in the Fabric samples. 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 then start Logspout by running the following command:

./monitordocker.sh fabric\_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

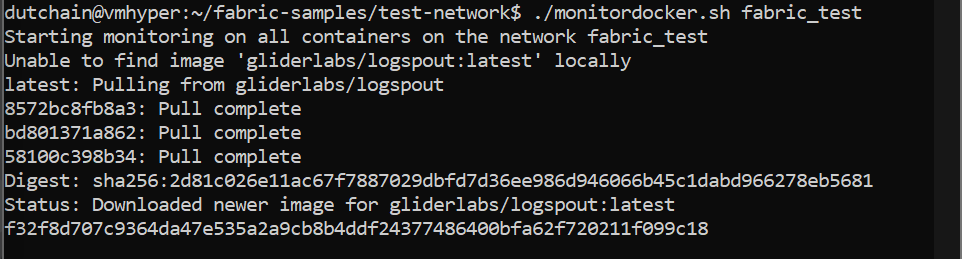
ad60f6b6c009: Pull complete

Digest: sha256:374e06b17b004bddc5445525796b5f7adb8234d64c5c5d663095fccafb6e4c26

Status: Downloaded newer image **for** gliderlabs/logspout:latest

1f99d130f15cf01706eda3e1f040496ec885036d485cb6bcc0da4a567ad84361

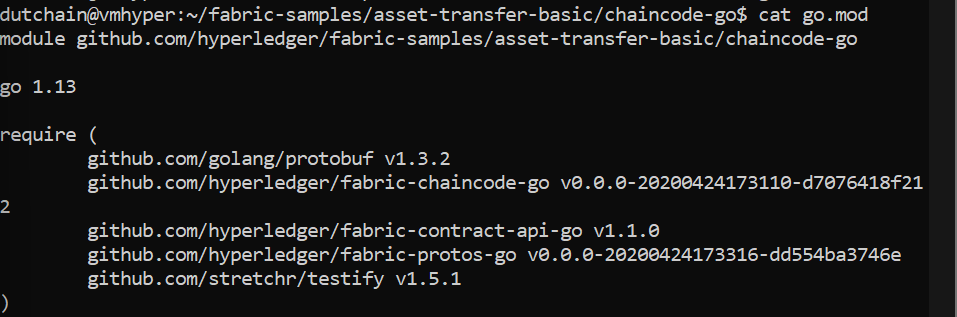
You will not see any logs at first, but this will change when we deploy our chaincode. It can be helpful to make this terminal window wide and the font small.



1.3 Package the smart contract

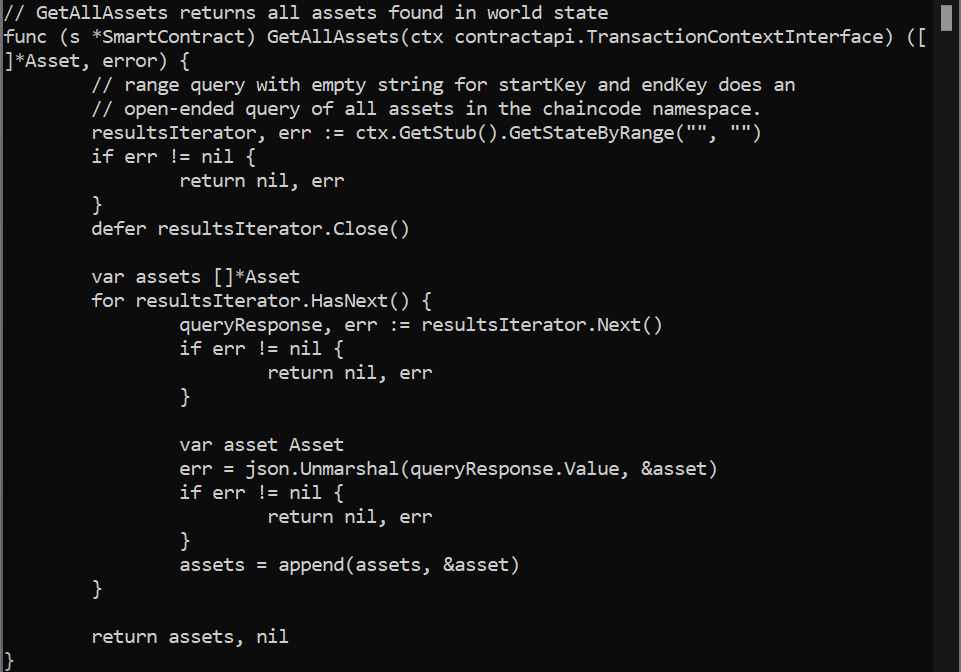
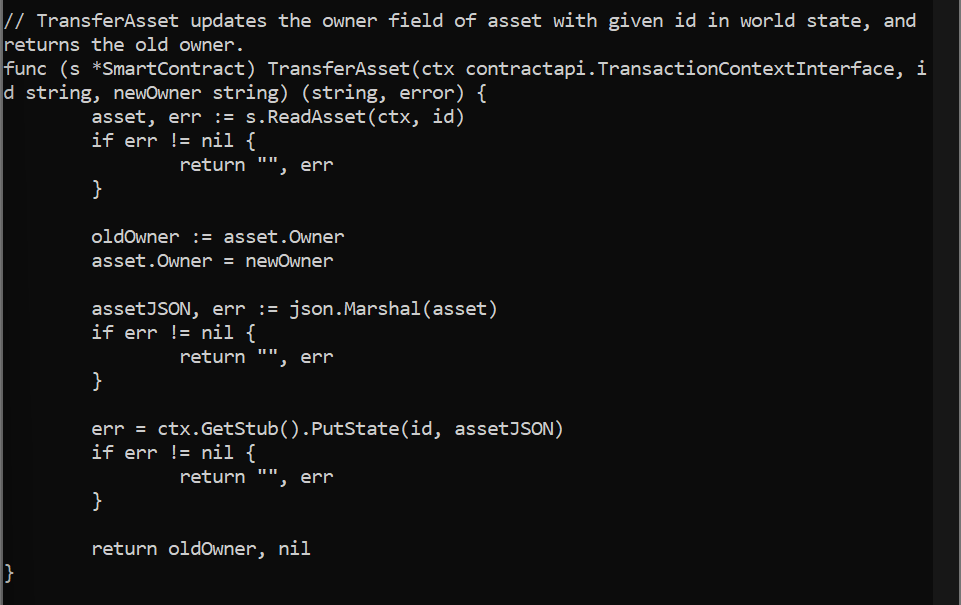
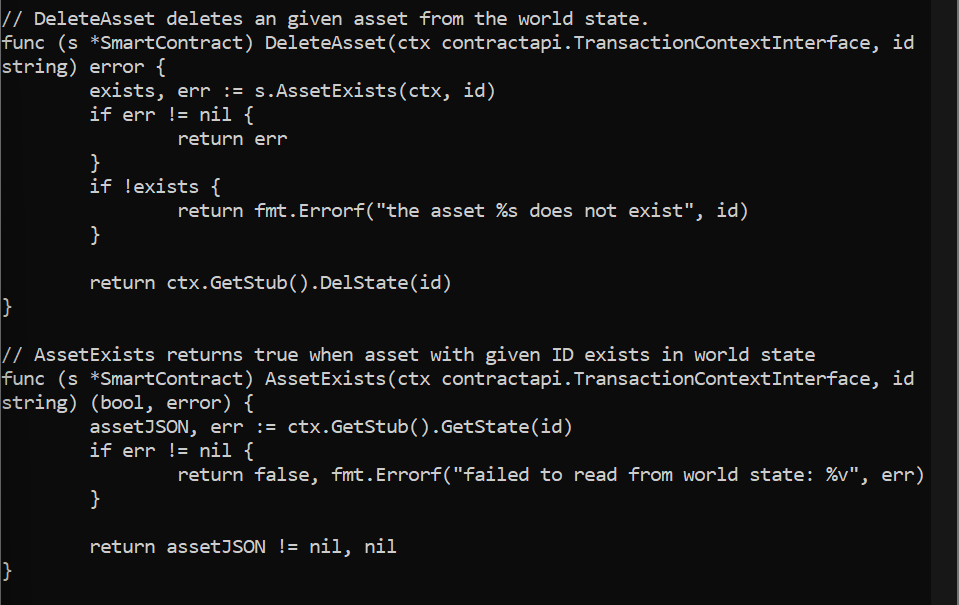
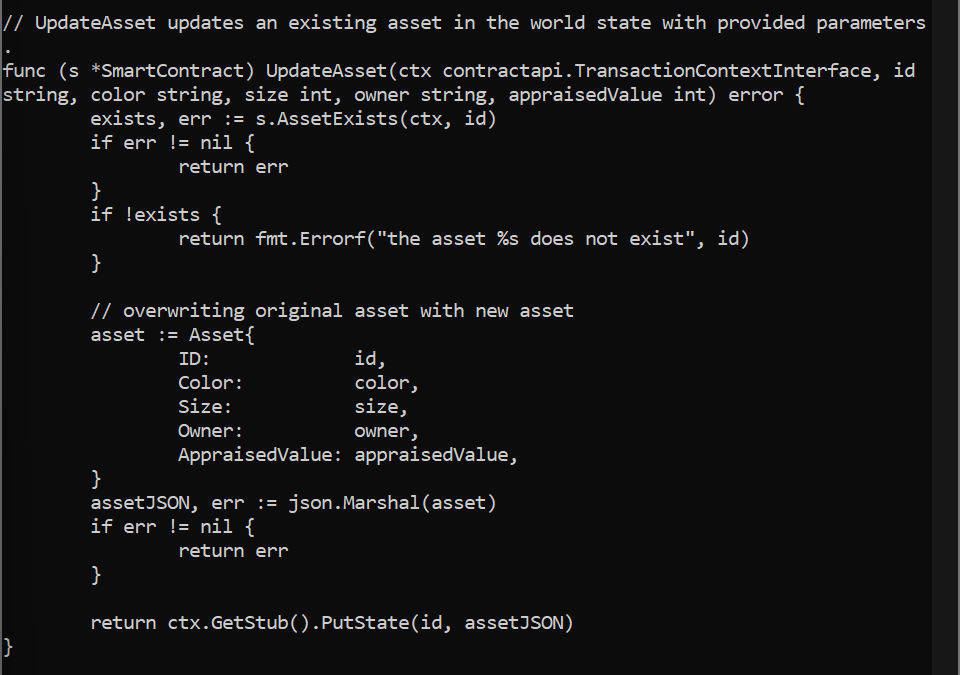
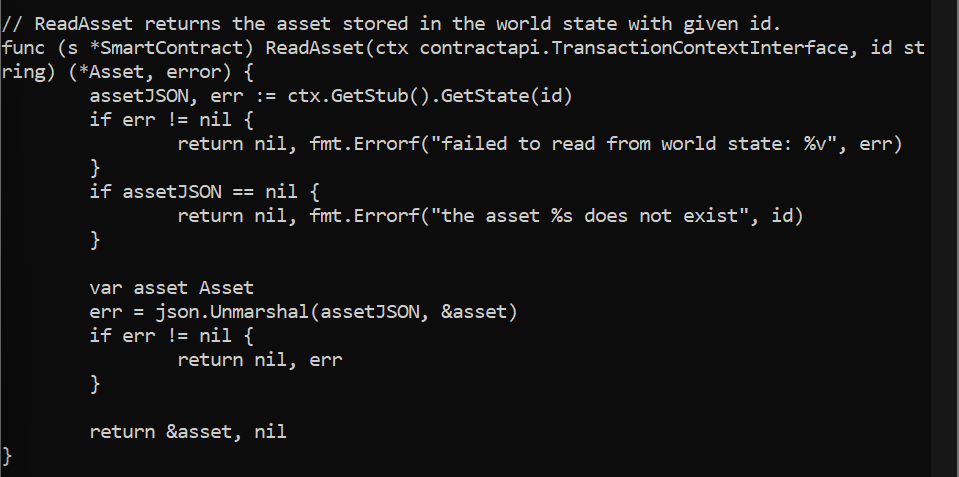
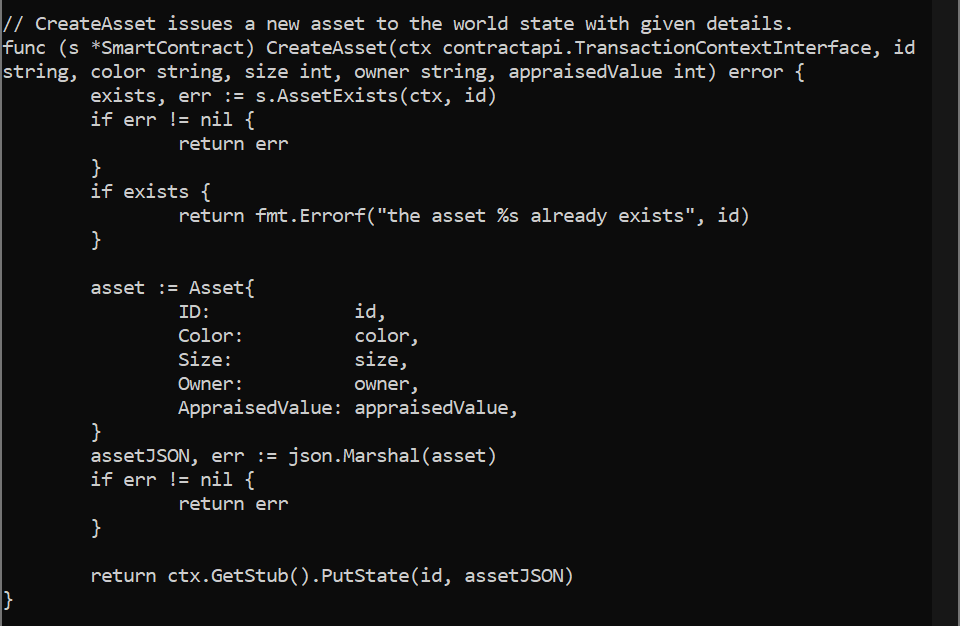
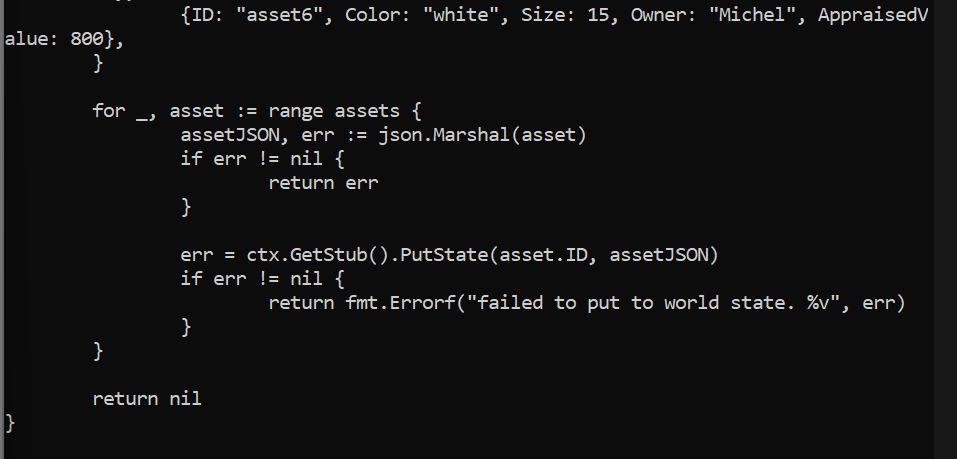
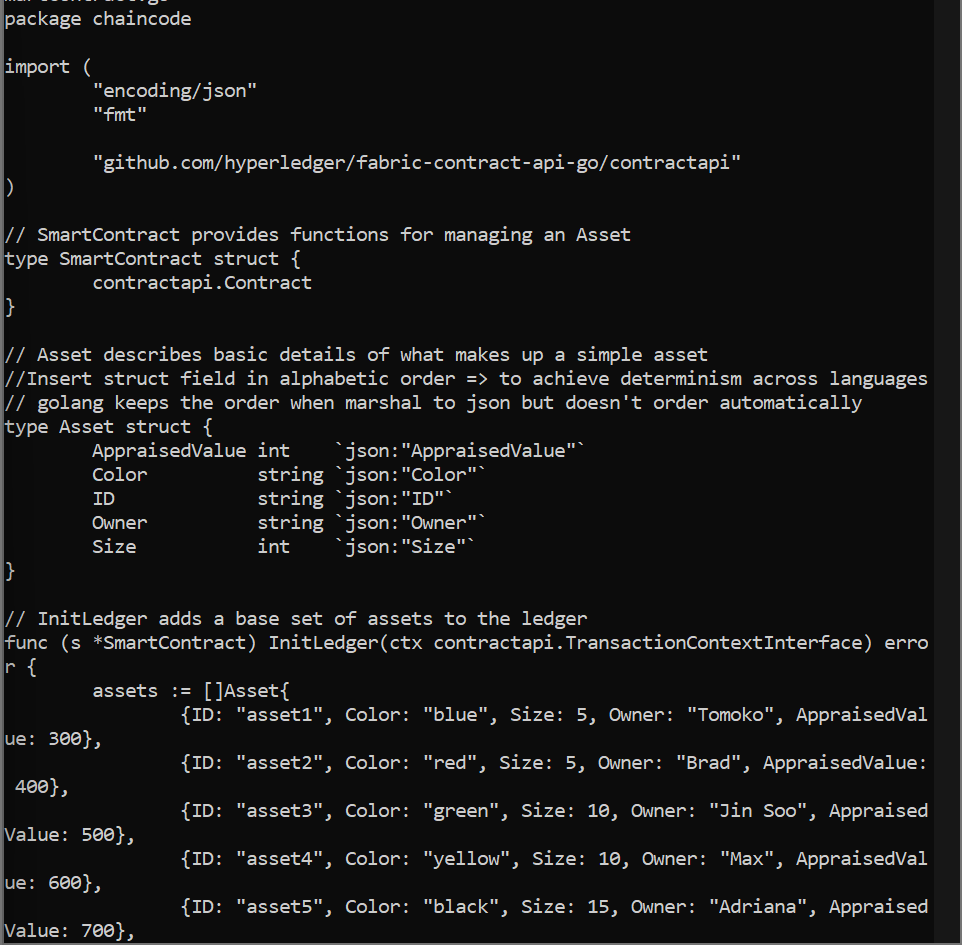
cd fabric-samples/asset-transfer-basic/chaincode-go

$ cat go.mod



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:

cat smartcontract.go



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.

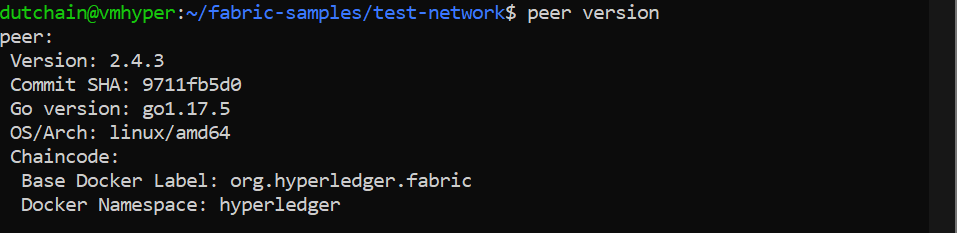


cd ../../test-network

export PATH=${PWD}/../bin:$PATH

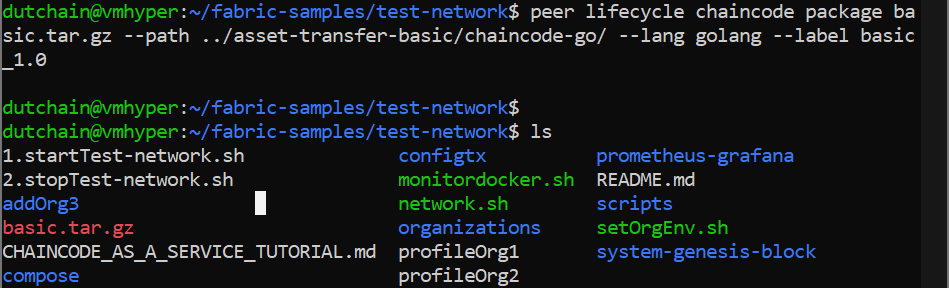
export FABRIC\_CFG\_PATH=$PWD/../config/

peer version



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



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.

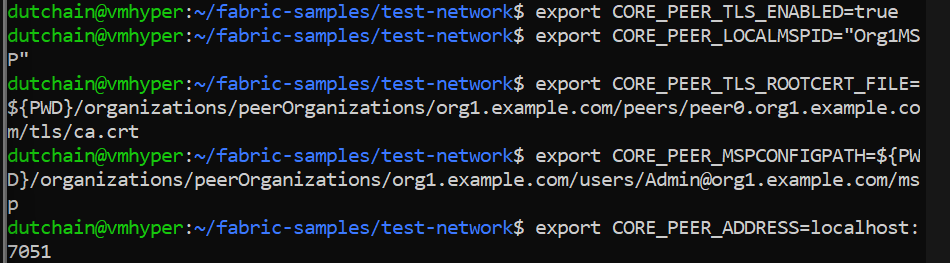
export CORE\_PEER\_TLS\_ENABLED=true

export CORE\_PEER\_LOCALMSPID="Org1MSP"

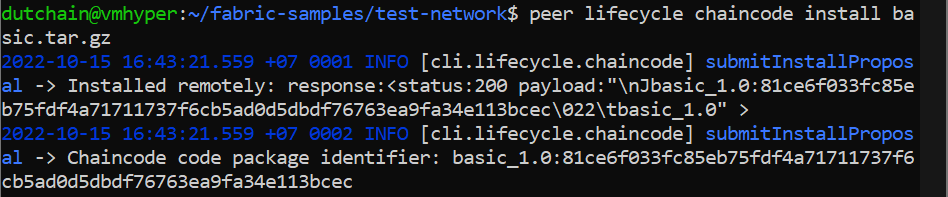
export CORE\_PEER\_TLS\_ROOTCERT\_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt

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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#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:"**\n**Jbasic\_1.0:e2db7f693d4aa6156e652741d5606e9c5f0de9ebb88c5721cb8248c3aead8123**\022\t**basic\_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.

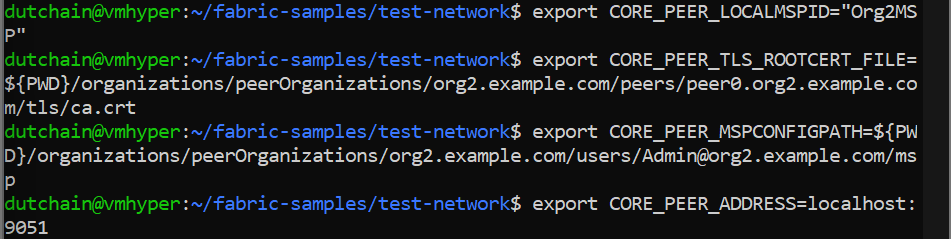
export CORE\_PEER\_LOCALMSPID="Org2MSP"

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..

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 /Channel/Application/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.

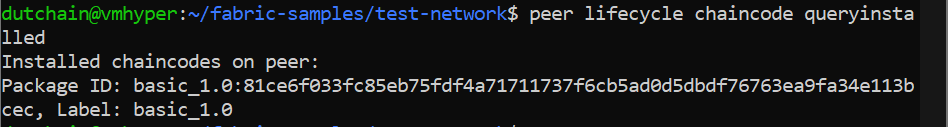
If an organization has installed the chaincode on their peer, they need to include the packageID in the chaincode definition approved by their organization. The package ID is used to associate the chaincode installed on a peer with an approved chaincode definition, and allows an organization to use the chaincode to endorse transactions. You can find the package ID of a chaincode by using the [peer lifecycle chaincode queryinstalled](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#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

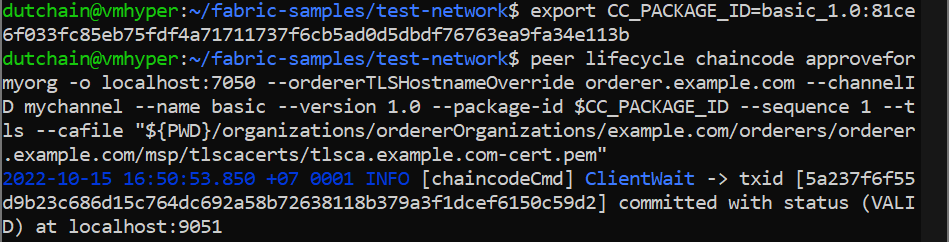


We are going to use the package ID when we approve the chaincode, so let’s go ahead and save it as an environment variable. Paste the package ID returned by peer lifecycle chaincode queryinstalled into the command below. **Note:** The package ID will not be the same for all users, so you need to complete 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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#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](https://hyperledger-fabric.readthedocs.io/en/latest/endorsement-policies.html) 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"

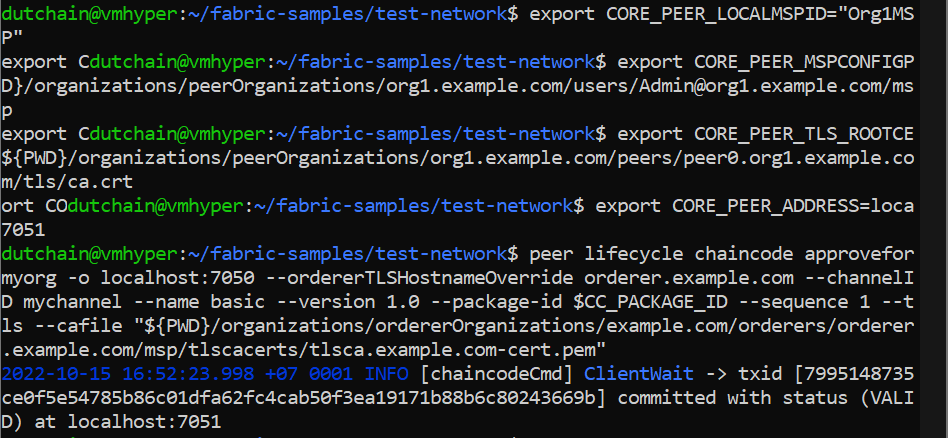
export CORE\_PEER\_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp

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.

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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#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:

{

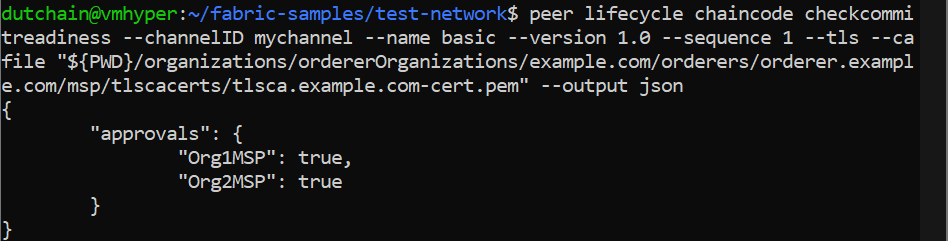
**"Approvals"**: {

**"Org1MSP"**: **true**,

**"Org2MSP"**: **true**

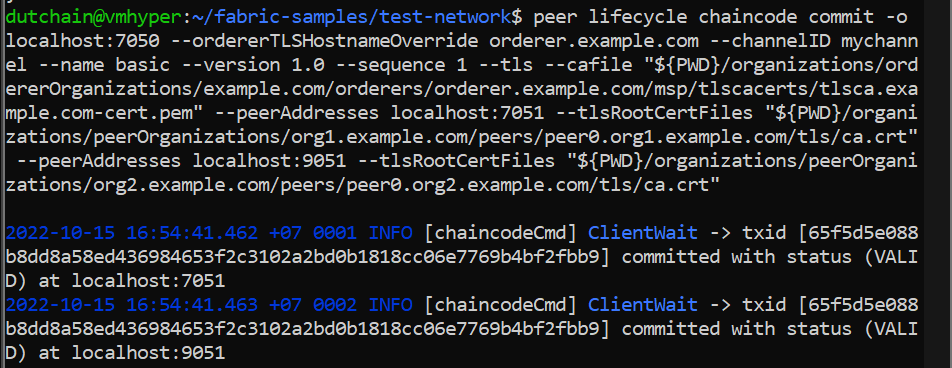
}

}



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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#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.

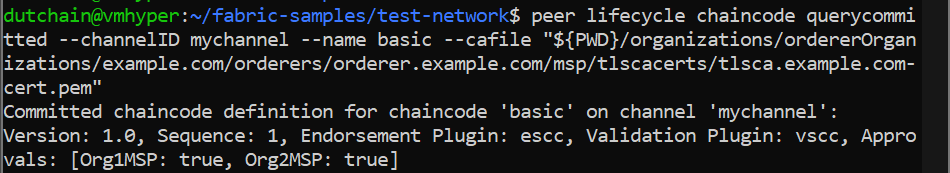
You can use the [peer lifecycle chaincode querycommitted](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#peer-lifecycle-chaincode-querycommitted) command to confirm that the chaincode definition has been committed to the channel.

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]



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 to create an initial set of assets on the ledger. Note that the invoke command needs to target a sufficient number of peers to meet the chaincode endorsement policy. (Note the CLI does not access the Fabric Gateway peer, so each endorsing peer must be specified.)

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 see 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}}]

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](https://hyperledger-fabric.readthedocs.io/en/latest/deploy_chaincode.html#go) or [TypeScript](https://hyperledger-fabric.readthedocs.io/en/latest/deploy_chaincode.html#typescript).

Issue the following commands from the test-network directory to install the chaincode dependencies.

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"

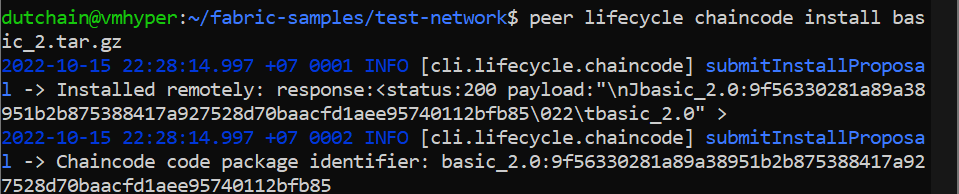
export CORE\_PEER\_TLS\_ROOTCERT\_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt

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.

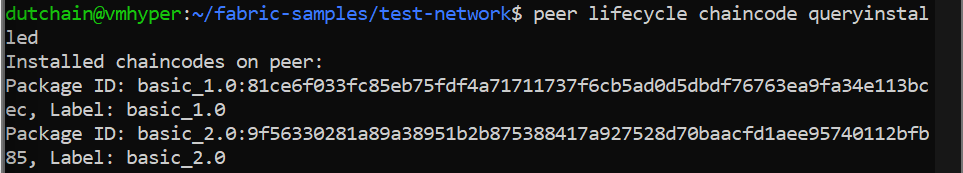
peer lifecycle chaincode queryinstalled

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

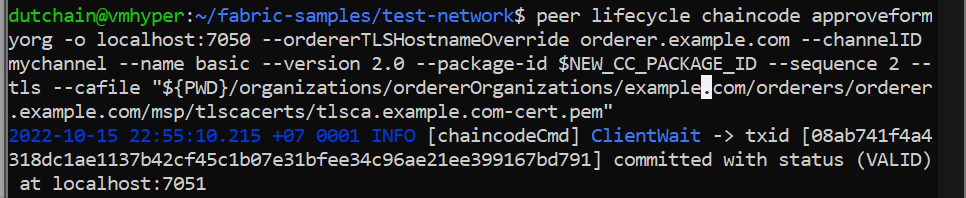


You can use the package label to find the package ID of the new chaincode and save it as a new environment variable. This output is for example only – your package ID will be different, so DO NOT COPY AND PASTE!

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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#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\_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp

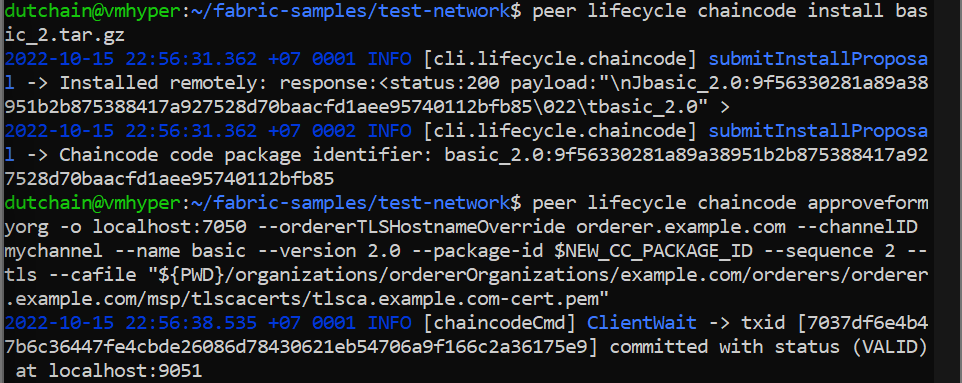
export CORE\_PEER\_ADDRESS=localhost:9051

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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#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

The chaincode is ready to be upgraded if the command returns the following JSON:

{

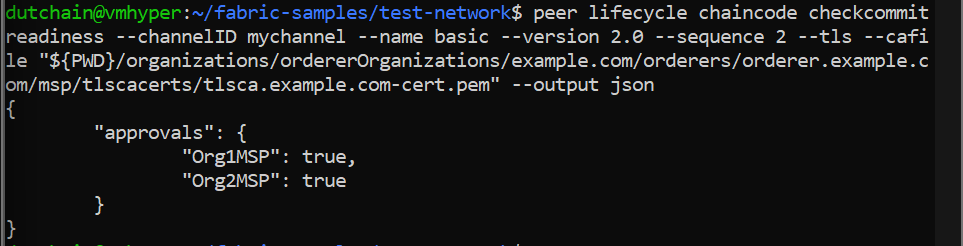
**"Approvals"**: {

**"Org1MSP"**: **true**,

**"Org2MSP"**: **true**

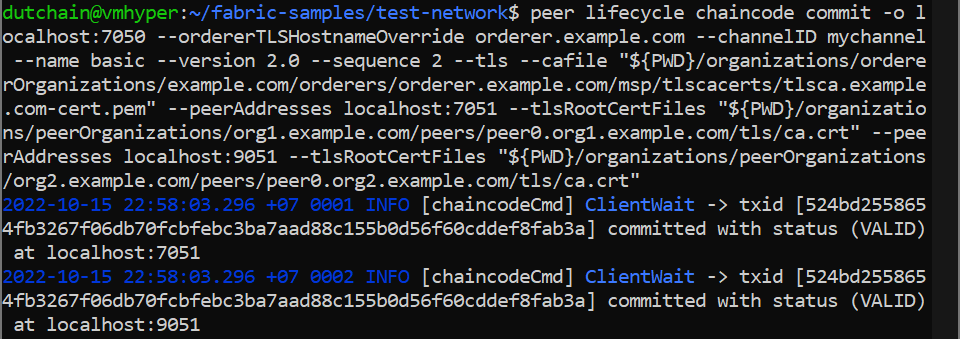
}

}



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

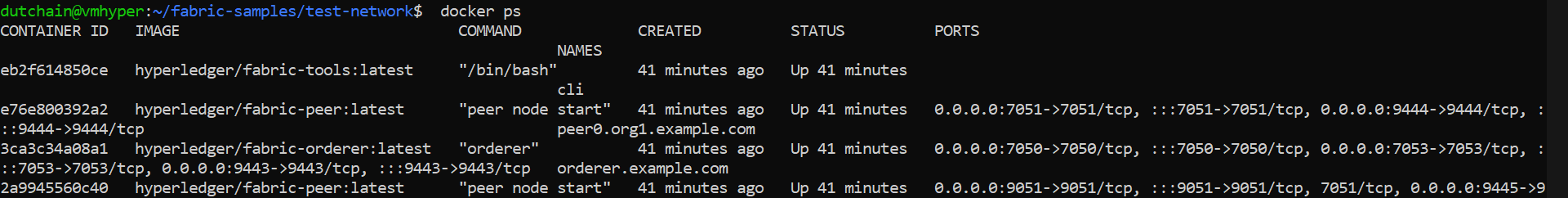
7bf2f1bf792b dev-peer0.org1.example.com-basic\_2.0-572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c-69c9e3e44ed18cafd1e58de37a70e2ec54cd49c7da0cd461fbd5e333de32879b "docker-entrypoint.s…" 2 minutes ago Up 2 minutes dev-peer0.org1.example.com-basic\_2.0-572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c

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}}]

2. Running a Fabric Application

Note

If you’re not yet familiar with the fundamental architecture of a Fabric blockchain network, you may want to visit the Key Concepts section prior to continuing.

You should also be familiar with the Fabric Gateway service and how it relates to the application transaction flow, documented in the Fabric Gateway section.

This tutorial provides an introduction to how Fabric applications interact with deployed blockchain networks. The tutorial uses sample programs built using the Fabric Gateway client API to invoke a smart contract, which queries and updates the ledger with the smart contract API – described in detail in Deploying a smart contract to a channel.

About Asset Transfer

The Asset Transfer (basic) sample demonstrates how to create, update, and query assets. It involves the following two components:

1. Sample application: which makes calls to the blockchain network, invoking transactions implemented in the smart contract. The application is located in the following fabric-samples directory:

asset-transfer-basic/application-gateway-typescript

2. Smart contract: which implements the transactions that interact with the ledger. The smart contract is located in the following fabric-samples directory:

asset-transfer-basic/chaincode-(typescript, go, java)

For this example, we will be using the TypeScript smart contract.

This tutorial consists of two principle parts:

1. Set up a blockchain network. Our application needs a blockchain network to interact with, so we will launch a basic network and deploy a smart contract for our application.

Diagram

Description automatically generated

2. Run the sample application to interact with the smart contract. Our application will use the assetTransfer smart contract to create, query, and update assets on the ledger. We will step through the code of the application and the transactions it invokes, including creating some initial 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.

2.1 Before you begin

Before you can run the sample application, you need to install Fabric Samples in your environment. Follow the instructions on Getting Started - Install to install the required software.

The sample application in this tutorial uses the Fabric Gateway client API for Node. See the documentation for a up to date list of supported programming language runtimes and dependencies.

· Ensure you have a suitable version of Node installed. Instructions for installing Node can be found in the Node.js documentation.

First, install nvm (node version manager) to install required node version(12). Follow these instructions below:

curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.2/install.sh | bash

echo “$SHELL” # to check what shell is running

Then, insert these line to bashrc file to add env

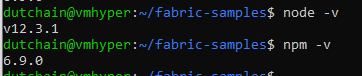
export NVM\_DIR="$([ -z "${XDG\_CONFIG\_HOME-}" ] && printf %s "${HOME}/.nvm" || printf %s "${XDG\_CONFIG\_HOME}/nvm")"

[ -s "$NVM\_DIR/nvm.sh" ] && \. "$NVM\_DIR/nvm.sh" # This loads nvm

After that, install nodejs 12 with this command

nvm install 12

Check node version and npm(node package manager) version



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, to ensure you start with an empty ledger.

2.2.1 Launch the blockchain network

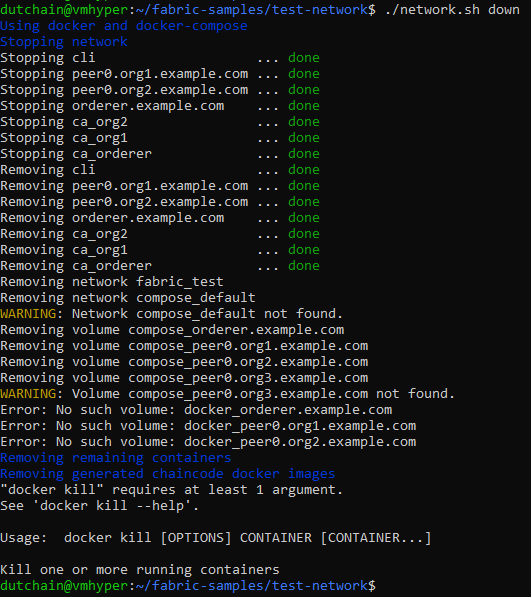
Navigate to the test-network subdirectory within your local clone of the fabric-samples repository.

cd fabric-samples/test-network

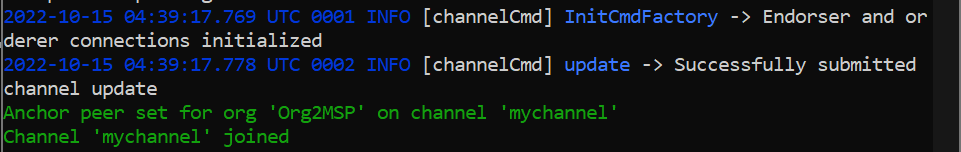
If you already have a test network running, bring it down to ensure the environment is clean.

./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.

2.2.2 Deploy the smart contract

Next, let’s deploy the chaincode package containing the smart contract by calling the ./network.sh script with the chaincode name and language options.

./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-typescript/ -ccl typescript

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 package 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

2.2.3 Prepare the sample application

Now, let’s prepare the sample Asset Transfer TypeScript application that will be used to interact with the deployed smart contract.

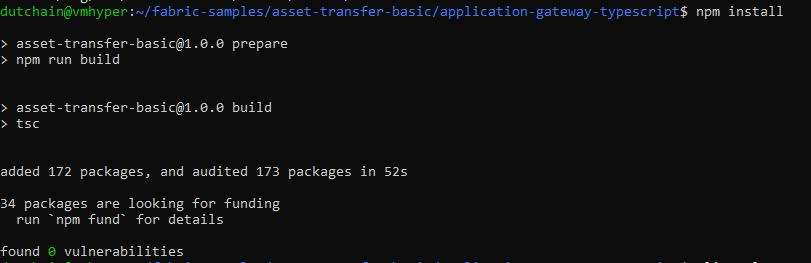
Open a new terminal, and navigate to the application-gateway-typescript directory.

cd asset-transfer-basic/application-gateway-typescript

This directory contains a sample application developed using the Fabric Gateway client API for Node.

Run the following command to install the dependencies and build the application. It may take some time to complete:

npm install



This process installs the application dependencies defined in the application’s package.json. The most important of which is the @hyperledger/fabric-gateway Node.js package; this provides the Fabric Gateway client API used to connect a Fabric Gateway and, using a specific client identity, to submit and evaluate transactions, and receive events.

Once npm install completes, everything is in place to run the application.

Let’s take a look at the sample TypeScript 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:



dist

node\_modules

package-lock.json

package.json

src

tsconfig.json

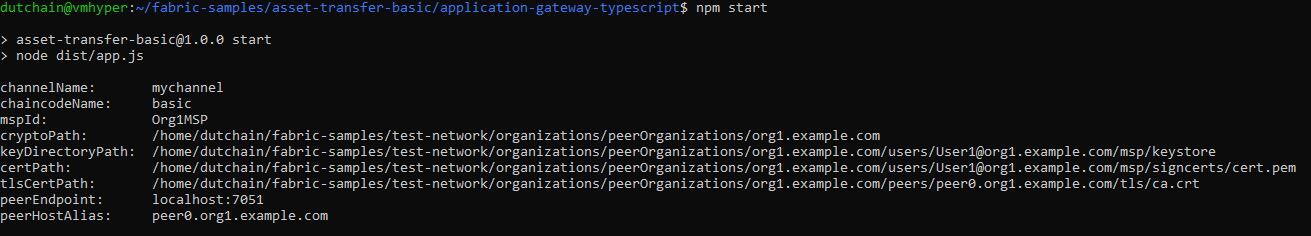
The src directory contains the client application source code. The JavaScript output generated from this source code during the install process is located in the dist directory, and can be ignored.

2.3 Run the sample application

When we started the Fabric test network earlier in this tutorial, several identities were created using the Certificate Authorities. These include a user identity for each of the organizations. The application will use the credentials of one of these user identities to transact with the blockchain network.

Let’s run the application and then step through each of the interactions with the smart contract functions. From the asset-transfer-basic/application-gateway-typescript directory, run the following command:

npm start



2.3.1 First, establish a gRPC connection to the Gateway

The client application establishes a gRPC connection to the Fabric Gateway service that it will use to transact with the blockchain network. To do this, it only requires the Fabric Gateway’s endpoint address and, if it is configured to use TLS, appropriate TLS certificates. In this sample, the gateway endpoint address is the address of a peer, which provides the Fabric Gateway service.

Note

There is significant overhead associated with establishing gRPC connections, so this connection should be retained by the application and used for all interactions with the Fabric Gateway.

Warning

In order to maintain security of any private data used in transactions, the application should connect to a Fabric Gateway belonging to the same organization as the client identity. If the client identity’s organization does not host any gateways, then a trusted gateway in another organization should be used.

The TypeScript application creates a gRPC connection using the TLS certificate of the signing certificate authority so that the authenticity of the gateway’s TLS certificate can be verified.

For a TLS connection to be successfully established, the endpoint address used by the client must match the address in the gateway’s TLS certificate. Since the client accesses the gateway’s Docker container at a localhost address, a gRPC option is specified to force this endpoint address to be interpreted as the gateway’s configured hostname.

const peerEndpoint = 'localhost:7051';

async function newGrpcConnection(): Promise<grpc.Client> {

const tlsRootCert = await fs.readFile(tlsCertPath);

const tlsCredentials = grpc.credentials.createSsl(tlsRootCert);

return new grpc.Client(peerEndpoint, tlsCredentials, {

'grpc.ssl\_target\_name\_override': 'peer0.org1.example.com',

});

}

2.3.2 Second, create a Gateway connection

The application then creates a Gateway connection, which it uses to access any of the Networks (analogous to channels) accessible to the Fabric Gateway, and subsequently smart Contracts deployed to those networks. A Gateway connection has three requirements:

1. gRPC connection to the Fabric Gateway.

2. Client identity used to transact with the network.

3. Signing implementation used to generate digital signatures for the client identity.

The sample application uses the Org1 user’s X.509 certificate as the client identity, and a signing implementation based on that user’s private key.

const client = await newGrpcConnection();

const gateway = connect({

client,

identity: await newIdentity(),

signer: await newSigner(),

});

async function newIdentity(): Promise<Identity> {

const credentials = await fs.readFile(certPath);

return { mspId: 'Org1MSP', credentials };

}

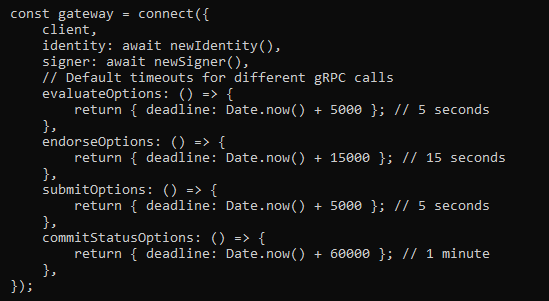
async function newSigner(): Promise<Signer> {

const privateKeyPem = await fs.readFile(keyPath);

const privateKey = crypto.createPrivateKey(privateKeyPem);

return signers.newPrivateKeySigner(privateKey);

}

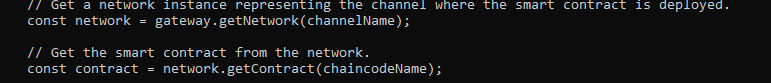


2.3.3 Third, access the smart contract to be invoked

The sample application uses the Gateway connection to get a reference to the Network and then the default Contract within a chaincode deployed on that network.

const network = gateway.getNetwork(channelName);

const contract = network.getContract(chaincodeName);



When a chaincode package includes multiple smart contracts, you can provide both the name of the chaincode and the name of a specific smart contract as arguments to the getContract() call. For example:

const contract = network.getContract(chaincodeName, smartContractName);

2.3.4 Fourth, populate the ledger with sample assets

Immediately after initial deployment of the chaincode package, the ledger is empty. The application uses submitTransaction() to invoke the InitLedger transaction function, which populates the ledger with some sample assets. submitTransaction() will use the Fabric Gateway to:

1. Endorse the transaction proposal.

2. Submit the endorsed transaction to the ordering service.

3. Wait for the transaction to be committed, updating ledger state.

Sample application InitLedger call:

await contract.submitTransaction('InitLedger');

2.3.5 Fifth, invoke transaction functions to read and write assets

Now the application is ready to execute business logic that queries, creates additional assets, and modifies assets on the ledger by invoking transactions functions on the smart contract.

2.3.5.1 Query all assets

The application uses evaluateTransaction() to query the ledger by performing a read-only transaction invocation. evaluateTransaction() will use the Fabric Gateway to invoke the transaction function and return its result. The transaction is not sent to the ordering service and no ledger update occurs.

Below, the sample application is just getting all the assets created in the previous step when we populated the ledger.

Sample application GetAllAssets call:

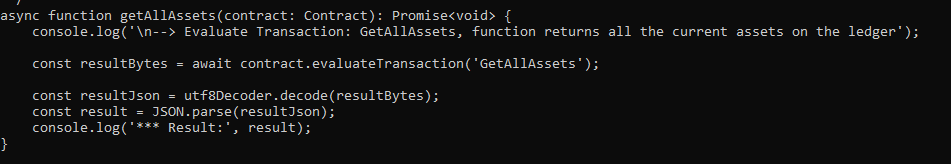
const resultBytes = await contract.evaluateTransaction('GetAllAssets');

const resultJson = utf8Decoder.decode(resultBytes);

const result = JSON.parse(resultJson);

console.log('\*\*\* Result:', result);

Note



Transaction function results are always returned as bytes since transaction functions can return any type of data. Often transaction functions return strings; or, as in the case above, a UTF-8 string of JSON data. The application is responsible for correctly interpreting the result bytes.

The terminal output should look like this:

\*\*\* Result: [

{

AppraisedValue: 300,

Color: 'blue',

ID: 'asset1',

Owner: 'Tomoko',

Size: 5,

docType: 'asset'

},

{

AppraisedValue: 400,

Color: 'red',

ID: 'asset2',

Owner: 'Brad',

Size: 5,

docType: 'asset'

},

{

AppraisedValue: 500,

Color: 'green',

ID: 'asset3',

Owner: 'Jin Soo',

Size: 10,

docType: 'asset'

},

{

AppraisedValue: 600,

Color: 'yellow',

ID: 'asset4',

Owner: 'Max',

Size: 10,

docType: 'asset'

},

{

AppraisedValue: 700,

Color: 'black',

ID: 'asset5',

Owner: 'Adriana',

Size: 15,

docType: 'asset'

},

{

AppraisedValue: 800,

Color: 'white',

ID: 'asset6',

Owner: 'Michel',

Size: 15,

docType: 'asset'

}

]

2.3.5.2 Create a new asset

The sample application submits a transaction to create a new asset.

Sample application CreateAsset call:

const assetId = `asset${Date.now()}`;

await contract.submitTransaction(

'CreateAsset',

assetId,

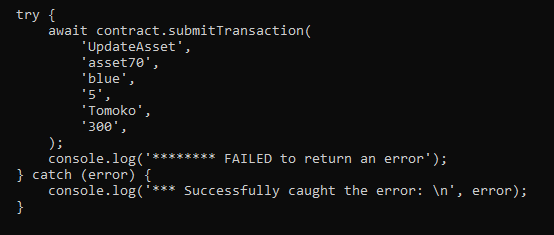
'yellow',

'5',

'Tom',

'1300',

);



Note

In the application snippets above, it is important to note that the CreateAsset transaction is submitted with the same type and number of arguments the chaincode is expecting, and in the correct sequence. In this case the correctly sequenced arguments are:

assetId, "yellow", "5", "Tom", "1300"

The corresponding smart contract’s CreateAsset transaction function is expecting the following sequence of arguments that define the asset object:

ID, Color, Size, Owner, AppraisedValue

2.3.5.3 Update an asset

The sample application submits a transaction to transfer ownership of the newly created asset. This time the transaction is invoked using submitAsync(), which returns after successfully submitting the endorsed transaction to the ordering service instead of waiting until the transaction is committed to the ledger. This allows the application to perform work using the transaction result while waiting for it to be committed.

Sample application TransferAsset call:

const commit = await contract.submitAsync('TransferAsset', {

arguments: [assetId, 'Saptha'],

});

const oldOwner = utf8Decoder.decode(commit.getResult());

console.log(`\*\*\* Successfully submitted transaction to transfer ownership from ${oldOwner} to Saptha`);

console.log('\*\*\* Waiting for transaction commit');

const status = await commit.getStatus();

if (!status.successful) {

throw new Error(`Transaction ${status.transactionId} failed to commit with status code ${status.code}`);

}

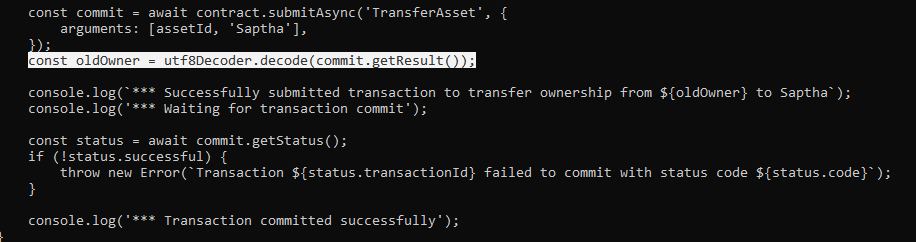
console.log('\*\*\* Transaction committed successfully');

Terminal output:

\*\*\* Successfully submitted transaction to transfer ownership from Tom to Saptha

\*\*\* Waiting for transaction commit

\*\*\* Transaction committed successfully



2.3.5.4 Query the updated asset

The sample application then evaluates a query for the transferred asset, showing that it was both created with the properties described, and then subsequently transferred to a new owner.

Sample application ReadAsset call:

const resultBytes = await contract.evaluateTransaction('ReadAsset', assetId);

const resultJson = utf8Decoder.decode(resultBytes);

const result = JSON.parse(resultJson);

console.log('\*\*\* Result:', result);

Terminal output:

\*\*\* Result: {

AppraisedValue: 1300,

Color: 'yellow',

ID: 'asset1639084597466',

Owner: 'Saptha',

Size: 5

}

2.3.5.5 Handle transaction errors

The final part of the sequence demonstrates an error submitting a transaction. In this example, the application attempts to submit an UpdateAsset transaction but specifies an asset ID that does not exist. The transaction function returns an error response, and the submitTransaction() call fails.

A submitTransaction() failure may generate several different types of error, indicating the point in the submit flow that the error occurred, and containing additional information to enable the application to respond appropriately. Consult the API documentation for details of the different error types that may be generated.

Sample application failing UpdateAsset call:

try {

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);

}

Terminal Output (with stack traces removed for clarity):

\*\*\* Successfully caught the error:

EndorseError: 10 ABORTED: failed to endorse transaction, see attached details for more info

at ... {

code: 10,

details: [

{

address: 'peer0.org1.example.com:7051',

message: 'error in simulation: transaction returned with failure: Error: The asset asset70 does not exist',

mspId: 'Org1MSP'

}

],

cause: Error: 10 ABORTED: failed to endorse transaction, see attached details for more info

at ... {

code: 10,

details: 'failed to endorse transaction, see attached details for more info',

metadata: Metadata { internalRepr: [Map], options: {} }

},

transactionId: 'a92980d41eef1d6492d63acd5fbb6ef1db0f53252330ad28e548fedfdb9167fe'

}

The EndorseError type indicates that failure occurred during endorsement, and the gRPC status code of ABORTED indicates that the application successfully invoked the Fabric Gateway but that a failure occurred during the endorsement process. A gRPC status code of UNAVAILABLE or DEADLINE\_EXCEEDED would suggest that the Fabric Gateway was not reachable or a timely response was not received so retrying the operation might be appropriate.

2.4 Clean up

When you are finished using the asset-transfer sample, you can bring down the test network using the network.sh script.

./network.sh down

This command will bring down the certificate authorities, peers, and ordering nodes of the blockchain 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.

2.5 Summary

You have now seen how to set up a blockchain network by launching the test network and deploying a smart contract. You have then run a client application, and examined the application code to understand how it uses the Fabric Gateway client API to query and update the ledger by connecting to a Fabric Gateway and invoking transaction functions on the deployed smart contract.

# **3.** **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](https://hyperledger-fabric.readthedocs.io/en/latest/private-data/private-data.html).

**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](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private_data_tutorial.html).

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](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-use-case)

2. [Build a collection definition JSON file](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-build-json)

3. [Read and Write private data using chaincode APIs](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-read-write-private-data)

4. [Deploy the private data smart contract to the channel](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-install-define-cc)

5. [Register identities](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-register-identities)

6. [Create an asset in private data](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-store-private-data)

7. [Query the private data as an authorized peer](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-query-authorized)

8. [Query the private data as an unauthorized peer](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-query-unauthorized)

9. [Transfer the Asset](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-transfer-asset)

10. [Purge Private Data](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-purge)

11. [Using indexes with private data](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-indexes)

12. [Additional resources](https://hyperledger-fabric.readthedocs.io/en/latest/private_data_tutorial.html#pd-ref-material)

This tutorial will deploy the [asset transfer private data sample](https://github.com/hyperledger/fabric-samples/tree/main/asset-transfer-private-data/chaincode-go) 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 Fabric and Fabric Samples](https://hyperledger-fabric.readthedocs.io/en/latest/install.html).

## **3.1** **Asset transfer private data sample use case**

This sample demonstrates the use of three private data collections, assetCollection, Org1MSPPrivateCollection & Org2MSPPrivateCollection 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 Org1MSPPrivateCollection.

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 Org2MSPPrivateCollection 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.

## **3.2** **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](https://hyperledger-fabric.readthedocs.io/en/latest/endorsement-policies.html) 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](https://hyperledger-fabric.readthedocs.io/en/latest/secured_asset_transfer/secured_private_asset_transfer_tutorial.html).

The asset transfer private data example contains a collections\_config.json file that defines three private data collection definitions: assetCollection, Org1MSPPrivateCollection, and Org2MSPPrivateCollection.

// collections\_config.json

[

{

"name": "assetCollection",

"policy": "OR('Org1MSP.member', 'Org2MSP.member')",

"requiredPeerCount": 1,

"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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#peer-lifecycle-chaincode-commit). More details on this process are provided in Section 3 below.

## **3.3** **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*

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 Org1MSPPrivateCollection or Org2MSPPrivateCollection , 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](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#ChaincodeStub). 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.

### ***3.3.1*** ***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 Org1MSPPrivateCollection allows only peers of Org1 to have their private data in a side database and Org2MSPPrivateCollection 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](https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-private-data/chaincode-go/chaincode/asset_queries.go):

· **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.

### ***3.3.2*** ***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 Org1MSPPrivateCollection.

If we were acting as Org2, we would replace Org1MSPPrivateCollection with Org2MSPPrivateCollection.

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** {

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 collection: %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)

}

*// 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 Org1MSPPrivateCollection and only peers in Org2 can store and transact with the appraisedValue key data in the Org2 collection Org2MSPPrivateCollection.

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.

## **3.4** **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.

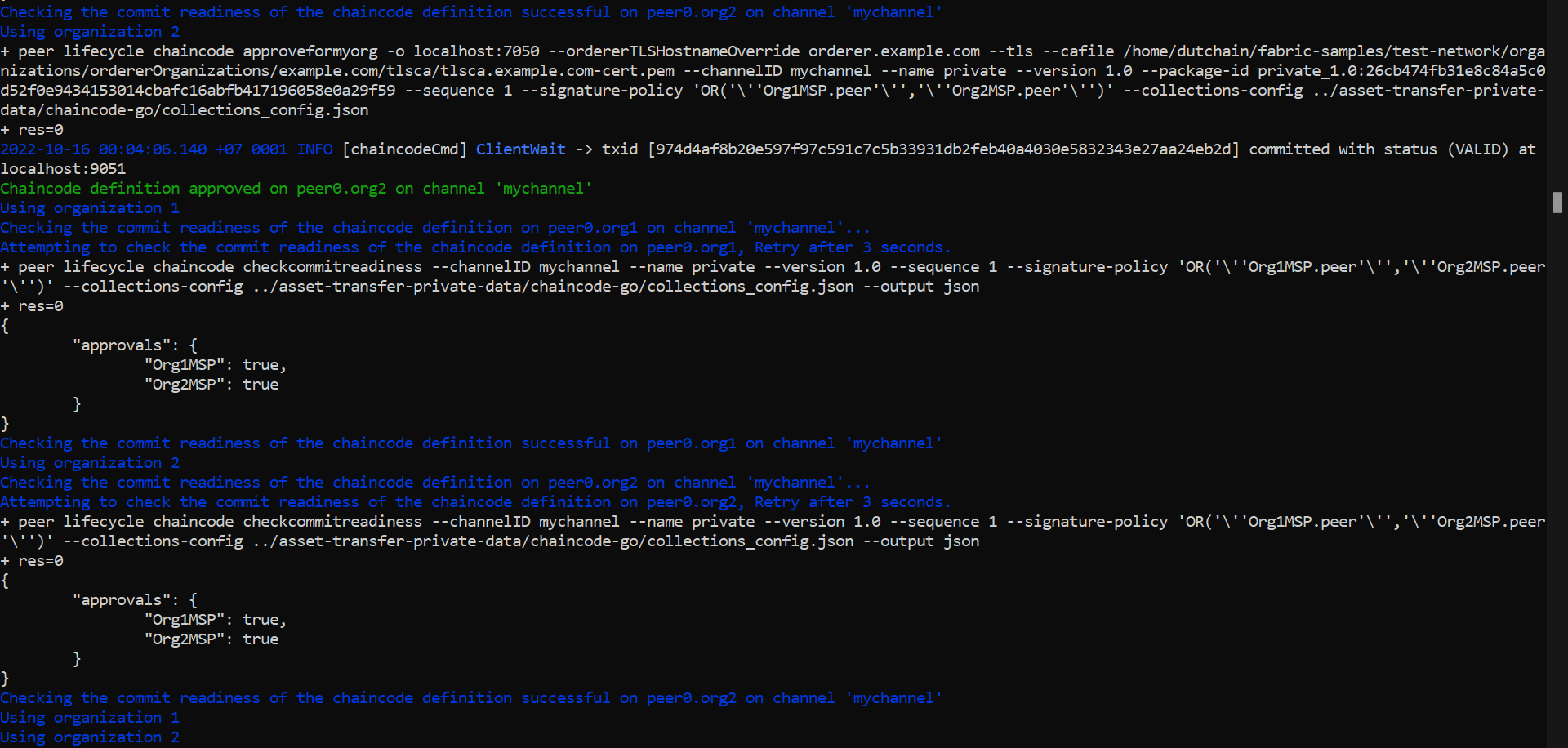
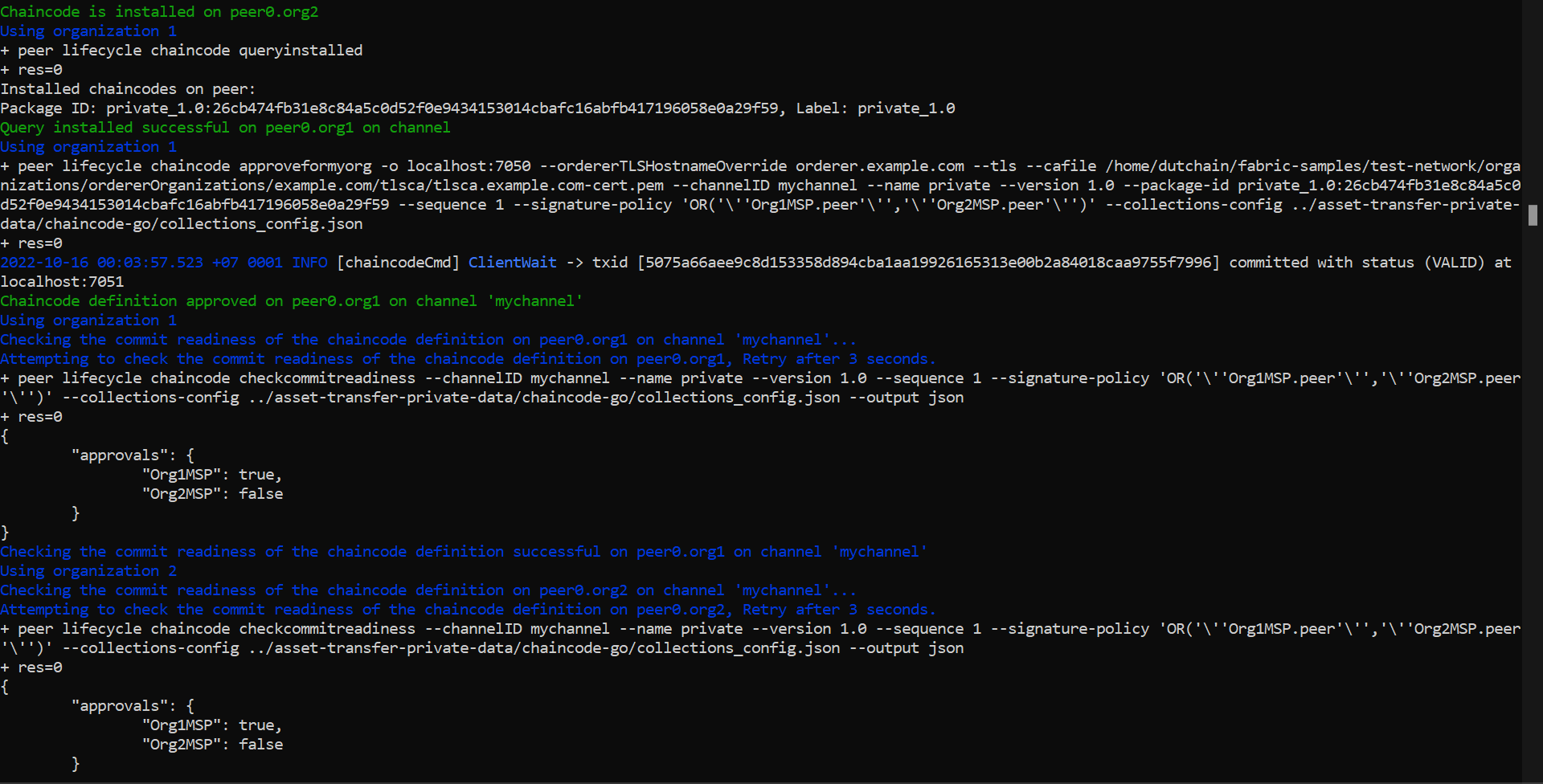
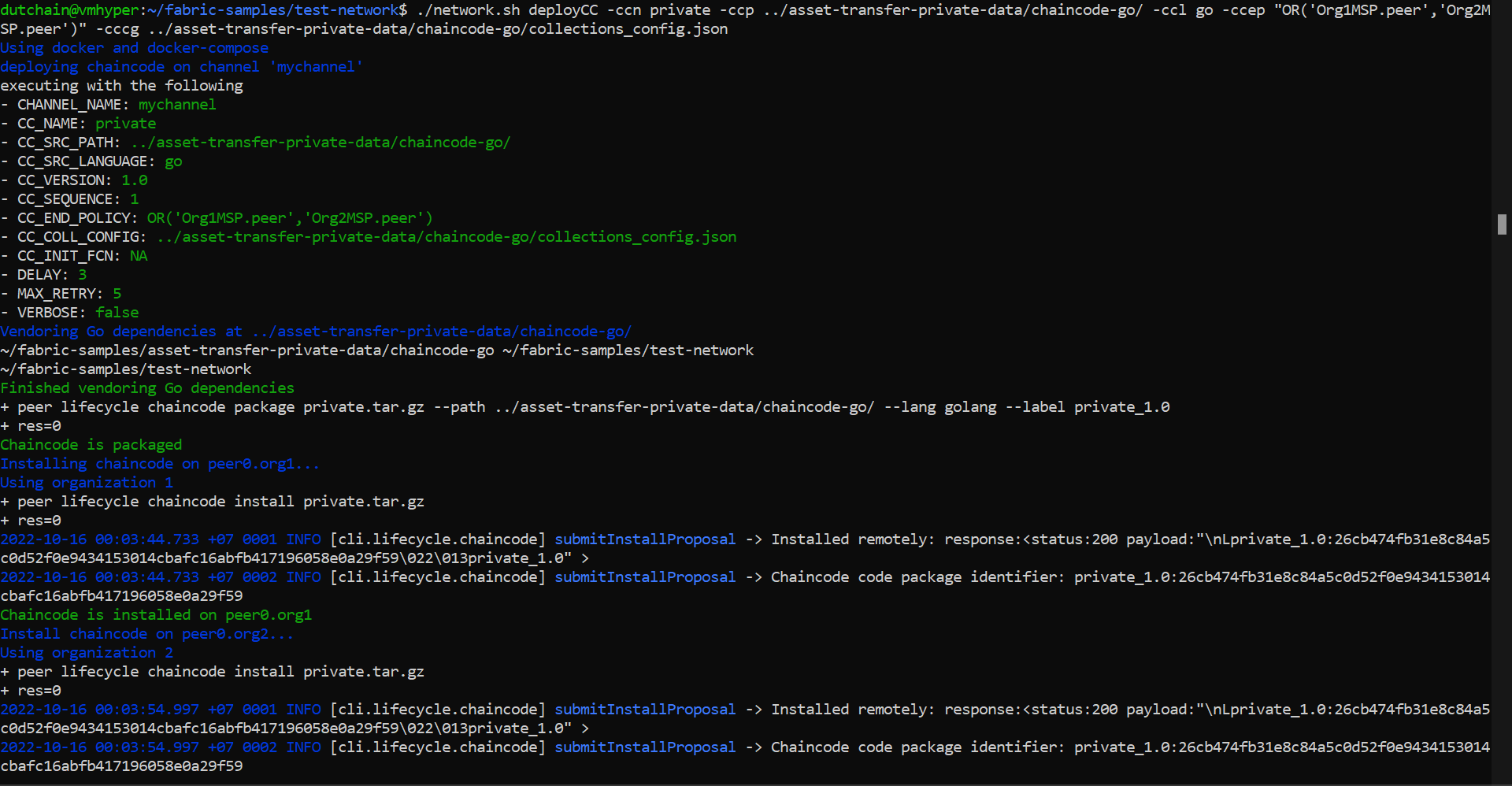
**Note**

For collections to work, it is important to have cross organizational gossip configured correctly. Refer to our documentation on [Gossip data dissemination protocol](https://hyperledger-fabric.readthedocs.io/en/latest/gossip.html), 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.

## **3.5** **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](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode_lifecycle.html). 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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html#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

## **3.6** **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

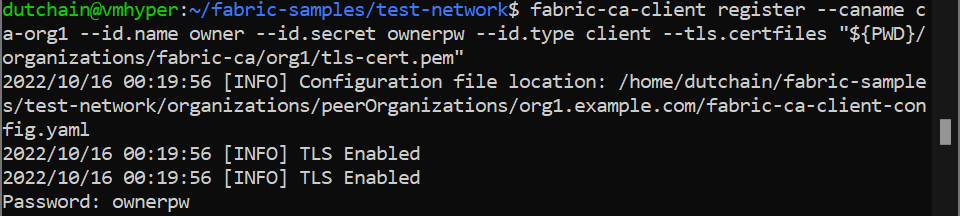
export FABRIC\_CFG\_PATH=$PWD/../config/

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/

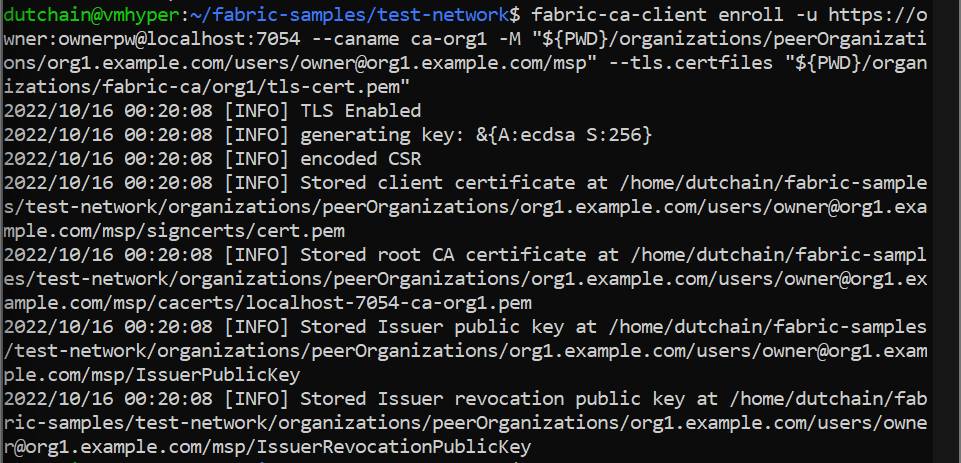
You can register a new owner client identity using the fabric-ca-client tool:

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.

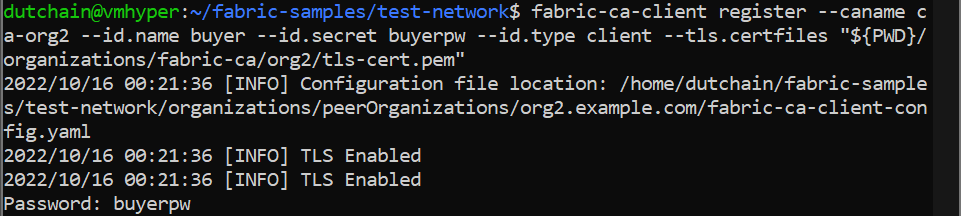
cp "$*{PWD}*/organizations/peerOrganizations/org1.example.com/msp/config.yaml" "$*{PWD}*/organizations/peerOrganizations/org1.example.com/users/owner@org1.example.com/msp/config.yaml"

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/

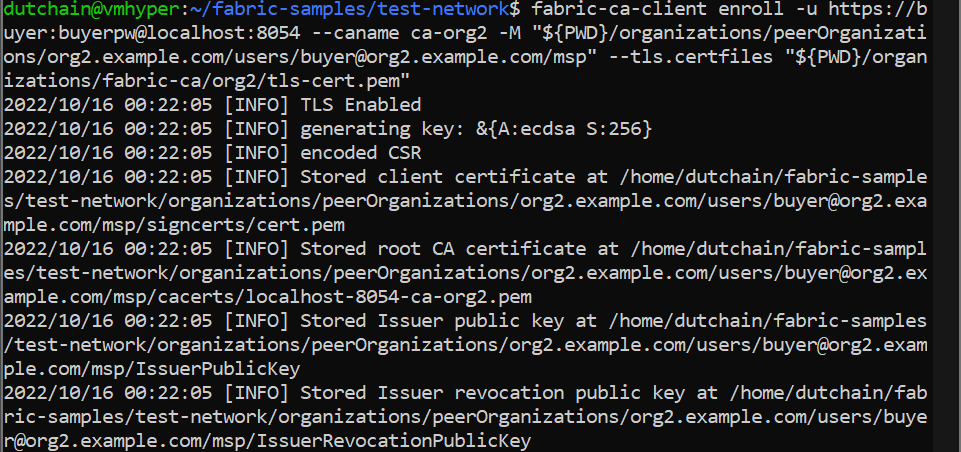
You can register a new owner client identity using the fabric-ca-client tool:

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"

## **3.7** **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.

## **3.8** **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

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)

}

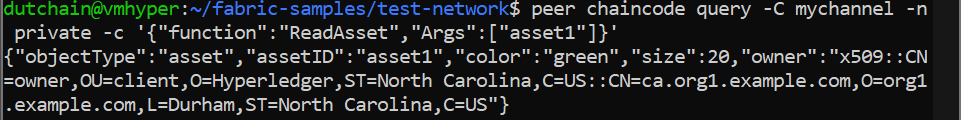
**return** assetDetails, **nil**

}

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"]}'



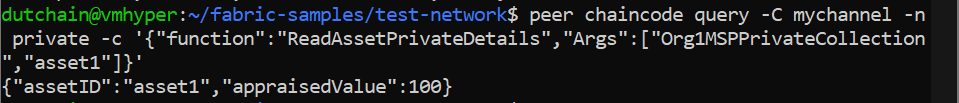
When successful, the command will return the following result:

{"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}

## **3.9** **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.

### ***3.9.1*** ***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

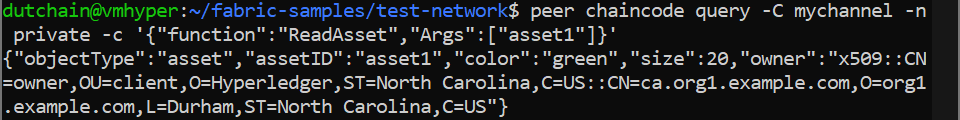
export CORE\_PEER\_ADDRESS=localhost:9051

### ***3.9.2*** ***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"]}'

result:

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" }

### ***3.9.3*** ***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:

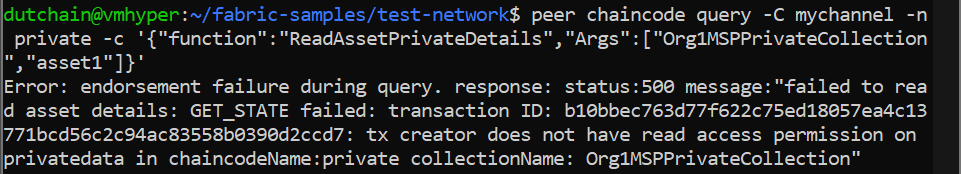
**Try it yourself**

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"]}'

result: 

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.

## **3.10** **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, lets continue.

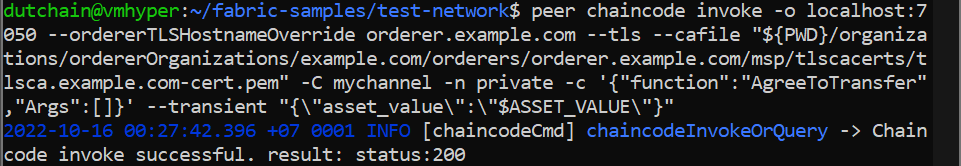
**Try it yourself**

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:

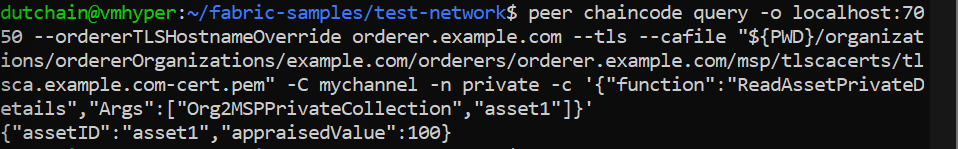
export ASSET\_VALUE=$(echo -n "{\"assetID\":\"asset1\",\"appraisedValue\":100}" | base64 | tr -d \\n)

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\"}"

result: 

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"]}'

result: 

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

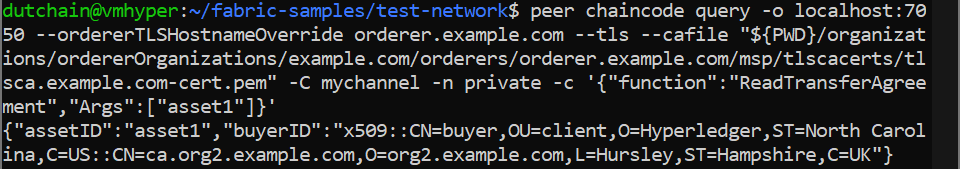
export CORE\_PEER\_TLS\_ROOTCERT\_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt

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":"eDUwOTo6Q049YnV5ZXIsT1U9Y2xpZW50LE89SHlwZXJsZWRnZXIsU1Q9Tm9ydGggQ2Fyb2xpbmEsQz1VUzo6Q049Y2Eub3JnMi5leGFtcGxlLmNvbSxPPW9yZzIuZXhhbXBsZS5jb20sTD1IdXJzbGV5LFNUPUhhbXBzaGlyZSxDPVVL"}

result: 

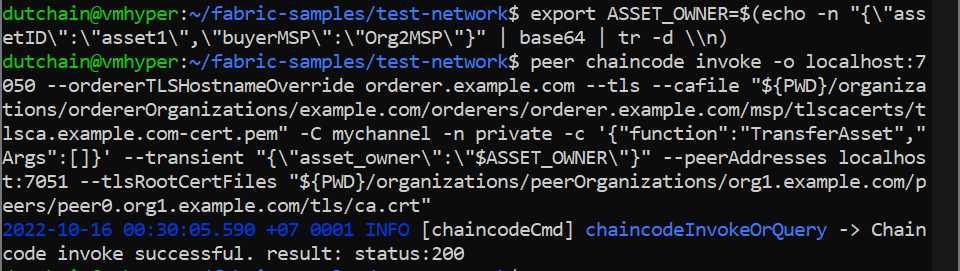
We now have all we need to transfer the asset. The smart contract uses the GetPrivateDataHash() function to check that the hash of the asset appraisal value in Org1MSPPrivateCollection matches the hash of the appraisal value in the Org2MSPPrivateCollection. If the hashes are the same, it confirms that 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)

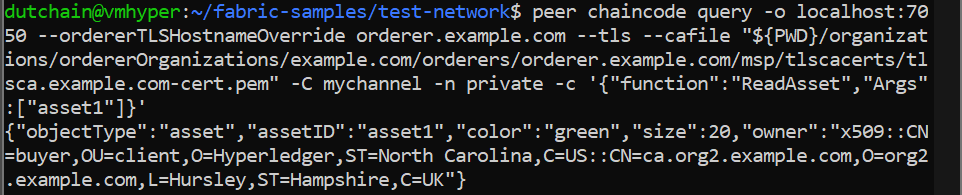
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n private -c '{"function":"TransferAsset","Args":[]}' --transient "{\"asset\_owner\":\"$ASSET\_OWNER\"}" --peerAddresses localhost:7051 --tlsRootCertFiles "${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt"

result:



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"]}'

result: 

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.

## **3.11** **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"

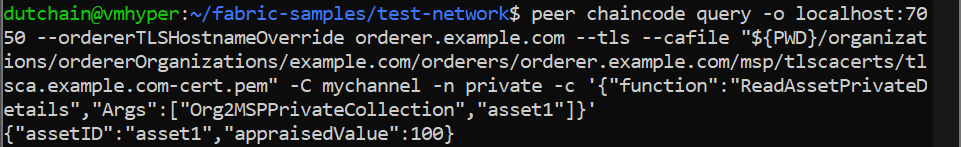
export CORE\_PEER\_TLS\_ROOTCERT\_FILE=${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt

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"]}'

result: 

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.

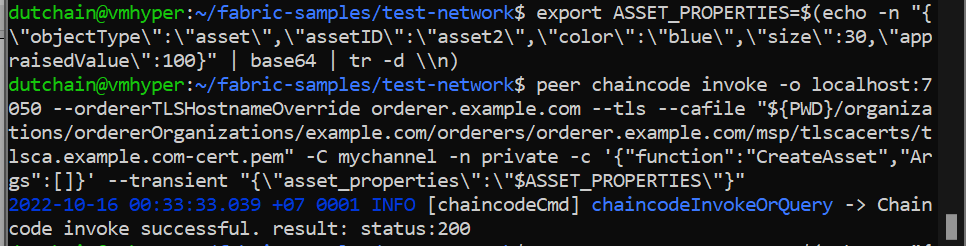
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'

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\"}"



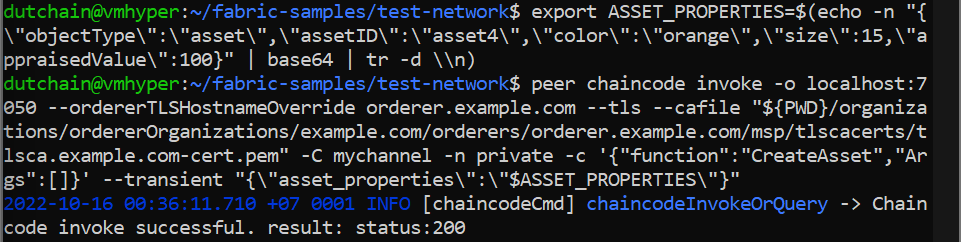
export ASSET\_PROPERTIES=$(echo -n "{\"objectType\":\"asset\",\"assetID\":\"asset3\",\"color\":\"red\",\"size\":25,\"appraisedValue\":100}" | base64 | tr -d \\n)

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"]}'

## 

## **3.12** **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](https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-private-data/chaincode-go/META-INF/statedb/couchdb/collections/assetCollection/indexes/indexOwner.json) .

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](https://jira.hyperledger.org/browse/FAB-17916) for more information.

## **3.13** **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.

# **4.** **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 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](https://github.com/hyperledger/fabric-samples/tree/main/asset-transfer-secured-agreement/chaincode-go) 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](https://hyperledger-fabric.readthedocs.io/en/latest/install.html#install-samples-binaries-and-docker-images).

## **4.1** **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 is a hash of the asset’s immutable properties, and along with the current owner is stored as public channel data for all channel members to see.

· The asset immutable 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 against the hashed asset id before purchasing. This confirms that the buyer has the correct asset description.

· An interested buyer will want to verify an asset’s provenance, specifically the asset’s origin and chain of custody. This confirms that the asset has not changed since issuance.

· To transfer an asset, a buyer and seller must first agree on the asset’s properties and sales price.

· Only the current owner may transfer their asset to another organization.

· The actual private asset transfer must verify that the asset’s properties and price have been agreed to. Both buyer and seller must endorse the transfer.

## **4.2** **How privacy is maintained**

The smart contract uses the following techniques to ensure that the asset properties remain private:

· The asset 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.

## **4.3** **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:

### ***4.3.1*** ***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 initiating the transfer of the asset using a smart contract that has been maliciously altered on their own peers. To further secure asset transfer, consider including other parties in the asset’s state based endorsement policy, such as a trusted third party.

### ***4.3.2*** ***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 and the private asset properties 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 and asset properties secret from other members of the channel. The endorsement policy of the implicit 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 and of the asset properties is automatically stored on the ledger when using private data collections. These hashes will match only if the two organizations have agreed to the same price and the two asset descriptions correspond. This allows the organizations to verify that they have come to agreement on the transfer details when they execute and endorse the transfer transaction. 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.

### ***4.3.3*** ***Transferring the asset***

After the two organizations have agreed to the same price and asset properties, the asset owner can invoke 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 on-chain asset properties present in the seller’s private collection correspond to the ones present in the buyer’s private collection by comparing their hash, to ensure that the asset owner is *selling* the same asset declared.

· The transfer function also 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 deletes the asset from the collection of the seller and updates the owner in the public ownership record.

· 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 to record the transaction price and timestamp.

· Because the transfer transaction updates data in the seller and buyer implicit data collections, the transfer must 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 initialize a subsequent transfer of their new asset.

## **4.4** **Running the secured asset transfer smart contract**

You can use the Fabric test network to run the secured 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 asset properties and price, we will transfer the asset from Org1 to Org2.

## **4.5** **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](https://github.com/hyperledger/fabric-samples). We will operate from the test-network directory for the remainder of the tutorial.

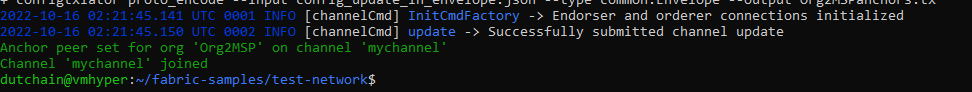
cd fabric-samples/test-network

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.

## **4.6** **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.

### ***4.6.1*** ***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

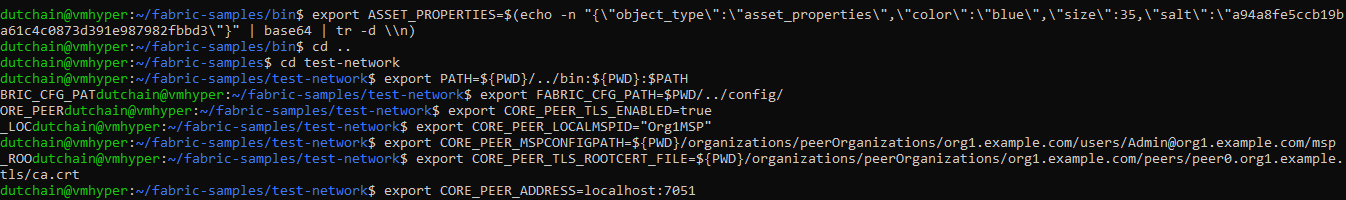
export CORE\_PEER\_LOCALMSPID="Org1MSP"

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.



### ***4.6.2*** ***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

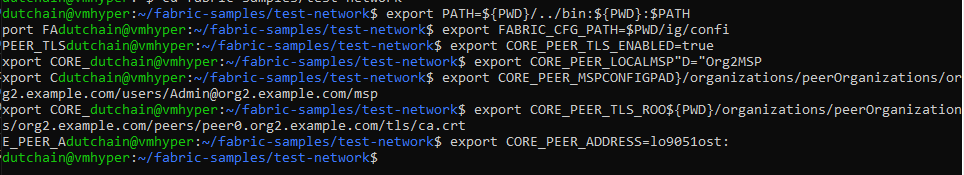
export CORE\_PEER\_LOCALMSPID="Org2MSP"

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.



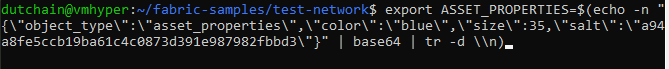
## **4.7** **Create an asset**

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.

### ***4.7.1*** ***Operate from the Org1 terminal***

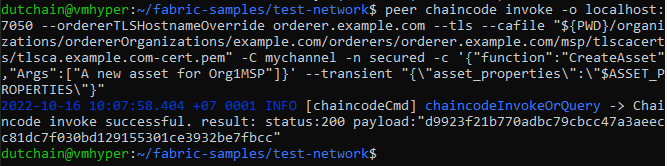
Before we create the asset, we need to specify the details of what our asset will be. Issue the following command to create a JSON that will describe the asset. The "salt" parameter is a random string that would prevent another member of the channel from guessing the asset using the hash on the ledger. 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\",\"color\":\"blue\",\"size\":35,\"salt\":\"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3\"}" | base64 | tr -d \\n)



We can now use the following command to create an asset that belongs to Org1.

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c '{"function":"CreateAsset","Args":["A new asset for Org1MSP"]}' --transient "{**\"**asset\_properties**\"**:**\"**$ASSET\_PROPERTIES**\"**}"



The hash of the asset properties will become the asset ID and is returned by the CreateAsset call. Look for the hash in the response “payload” that was reported to the CLI, and set it as an environment variable for later reference.

export ASSET\_ID=d9923f21b770adbc79cbcc47a3aeecc81dc7f030bd129155301ce3932be7fbcc

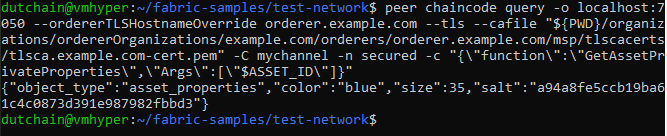


We can query the Org1 implicit data collection to see the asset that was created, write the following command by substituting assetId with the id returned from the createAsset function.

peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**GetAssetPrivateProperties**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**]}"

When successful, the command will return the following result:

{"object\_type":"asset\_properties","color":"blue","size":35,"salt":"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3"}

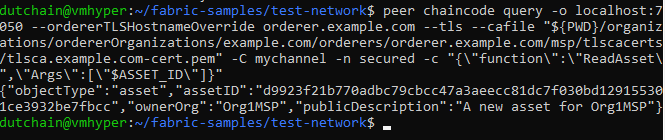


We can also query the ledger to see the public ownership record:

peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**ReadAsset**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**]}"

The command will return the record that the asset is owned by Org1:

{"object\_type":"asset","asset\_id":"d9923f21b770adbc79cbcc47a3aeecc81dc7f030bd129155301ce3932be7fbcc","owner\_org":"Org1MSP","public\_description":"A new asset for Org1MSP"}



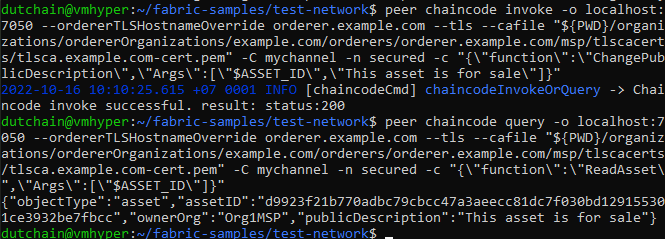
Because the market for assets is hot, Org1 wants to flip this asset and put it up for sale. As the asset owner, Org1 can update the public description to advertise that the asset is for sale. Run the following command to change the asset description:

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**ChangePublicDescription**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**,**\"**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 "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**ReadAsset**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**]}"

We can now see that the asset is for sale:

{"object\_type":"asset","asset\_id":"d9923f21b770adbc79cbcc47a3aeecc81dc7f030bd129155301ce3932be7fbcc","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.*

### ***4.7.2*** ***Operate from the Org2 terminal***

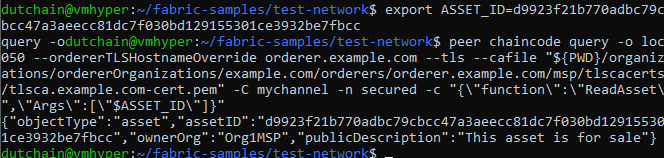
If we operate from the Org2 terminal, we can use the smart contract query the public asset data:

export ASSET\_ID=d9923f21b770adbc79cbcc47a3aeecc81dc7f030bd129155301ce3932be7fbcc

peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**ReadAsset**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**]}"

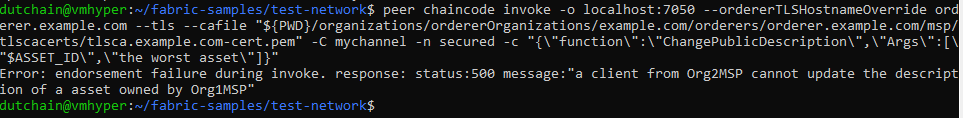
From this query, Org2 learns that the asset is for sale:

{"object\_type":"asset","asset\_id":"d9923f21b770adbc79cbcc47a3aeecc81dc7f030bd129155301ce3932be7fbcc","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 "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**ChangePublicDescription**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**,**\"**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 an asset owned by Org1MSP"

## **4.8** **Agree to sell the asset**

To sell an asset, both the buyer and the seller must agree on an asset price and make sure that they both have the same asset properties in their respective private data collections. 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 and asset properties before the asset can be transferred.

## **4.9** **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 together with the asset properties, 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=$(echo -n "{\"asset\_id\":\"$ASSET\_ID\",\"trade\_id\":\"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\",\"price\":110}" | base64 | tr -d \\n)

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{\"function\":\"AgreeToSell\",\"Args\":[\"$ASSET\_ID\"]}" --transient "{\"asset\_price\":\"$ASSET\_PRICE\"}"

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 "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**GetAssetSalesPrice**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**]}"

## **4.10** **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\",\"color\":\"blue\",\"size\":35,\"salt\":\"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3\"}" | base64 | tr -d \\n)

peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{\"function\":\"VerifyAssetProperties\",\"Args\":[\"$ASSET\_ID\"]}" --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 Org1. 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.

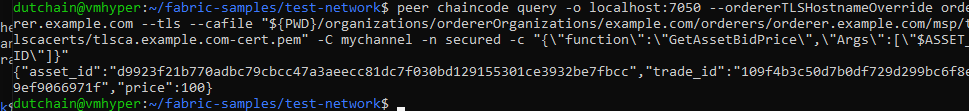
export ASSET\_PRICE=$(echo -n "{\"asset\_id\":\"$ASSET\_ID\",\"trade\_id\":\"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\",\"price\":100}" | base64 | tr -d \\n)

export ASSET\_PROPERTIES=$(echo -n "{\"object\_type\":\"asset\_properties\",\"color\":\"blue\",\"size\":35,\"salt\":\"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3\"}" | base64 | tr -d \\n)

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{\"function\":\"AgreeToBuy\",\"Args\":[\"$ASSET\_ID\"]}" --transient "{\"asset\_price\":\"$ASSET\_PRICE\", \"asset\_properties\":\"$ASSET\_PROPERTIES\"}"

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 "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**GetAssetBidPrice**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**]}"



*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.*

## **4.11** **Transfer the asset from Org1 to Org2**

After both organizations have agreed to their price and asset properties, 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 check the asset properties hash of both seller and buyer private collection to check that the asset that is transferred is the same asset that Org1 owns.

### ***4.11.1*** ***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 price is passed in the transfer request as transient properties. This is 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 "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**TransferAsset**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**,**\"**Org2MSP**\"**]}" --transient "{**\"**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"

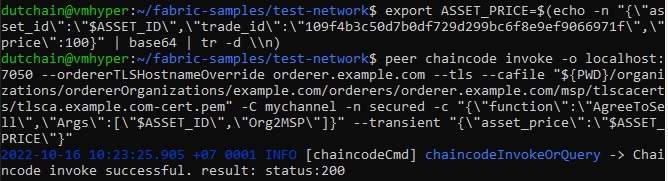
Because the two organizations have not agreed to the same price, the transfer cannot be completed:

Error: endorsement failure during invoke. response: status:500 message:"failed transfer verification: hash cf74b8ce092b637bd28f98f7cdd490534c102a0665e7c985d4f2ab9810e30b1c for passed price JSON {**\"**asset\_id**\"**:**\"**d9923f21b770adbc79cbcc47a3aeecc81dc7f030bd129155301ce3932be7fbcc**\"**,**\"**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=$(echo -n "{\"asset\_id\":\"$ASSET\_ID\",\"trade\_id\":\"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\",\"price\":100}" | base64 | tr -d \\n)

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{\"function\":\"AgreeToSell\",\"Args\":[\"$ASSET\_ID\",\"Org2MSP\"]}" --transient "{\"asset\_price\":\"$ASSET\_PRICE\"}"

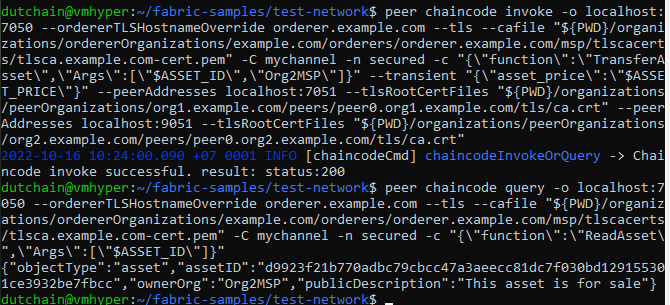


Now that the buyer and seller have agreed to the same price, Org1 can transfer the asset to Org2.

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**TransferAsset**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**,**\"**Org2MSP**\"**]}" --transient "{**\"**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 "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**ReadAsset**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**]}"



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 Org2.*

### ***4.11.2*** ***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:

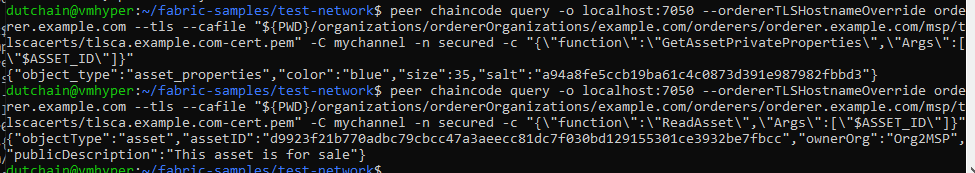
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**GetAssetPrivateProperties**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**]}"

Org2 can now update the asset public description:

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**ChangePublicDescription**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**,**\"**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 "$*{PWD}*/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n secured -c "{**\"**function**\"**:**\"**ReadAsset**\"**,**\"**Args**\"**:[**\"**$ASSET\_ID**\"**]}"



## **4.12** **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

# **5.** **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](https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_tutorial.html).

The tutorial will take you through the following steps:

1. [Enable CouchDB in Hyperledger Fabric](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-enable-couch)

2. [Create an index](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-create-index)

3. [Add the index to your chaincode folder](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-add-index)

4. [Deploy the smart contract](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-install-deploy)

5. [Query the CouchDB State Database](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-query)

6. [Use best practices for queries and indexes](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-best)

7. [Query the CouchDB State Database With Pagination](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-pagination)

8. [Update an Index](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-update-index)

9. [Delete an Index](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-delete-index)

For a deeper dive into CouchDB refer to [CouchDB as the State Database](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_as_state_database.html) and for more information on the Fabric ledger refer to the [Ledger](https://hyperledger-fabric.readthedocs.io/en/latest/ledger/ledger.html) 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](https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-ledger-queries/chaincode-go) 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 Fabric and Fabric Samples](https://hyperledger-fabric.readthedocs.io/en/latest/install.html).

## **5.1** **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.

## **5.2** **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](https://hub.docker.com/_/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](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_as_state_database.html#couchdb-configuration).

## **5.3** **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](https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-ledger-queries/chaincode-go/asset_transfer_ledger_chaincode.go). 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](http://guide.couchdb.org/draft/design.html) 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"

}

{

"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](http://docs.couchdb.org/en/3.2.2/api/database/find.html#db-index).

## **5.4** **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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html) 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](https://github.com/hyperledger/fabric-samples/tree/main/asset-transfer-ledger-queries/chaincode-go) below illustrates how the index is packaged with the chaincode.

This sample includes one index named indexOwnerDoc, to support queries by asset owner:

{"index":{"fields":["docType","owner"]},"ddoc":"indexOwnerDoc", "name":"indexOwner","type":"json"}

### ***5.4.1*** ***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.

result: 

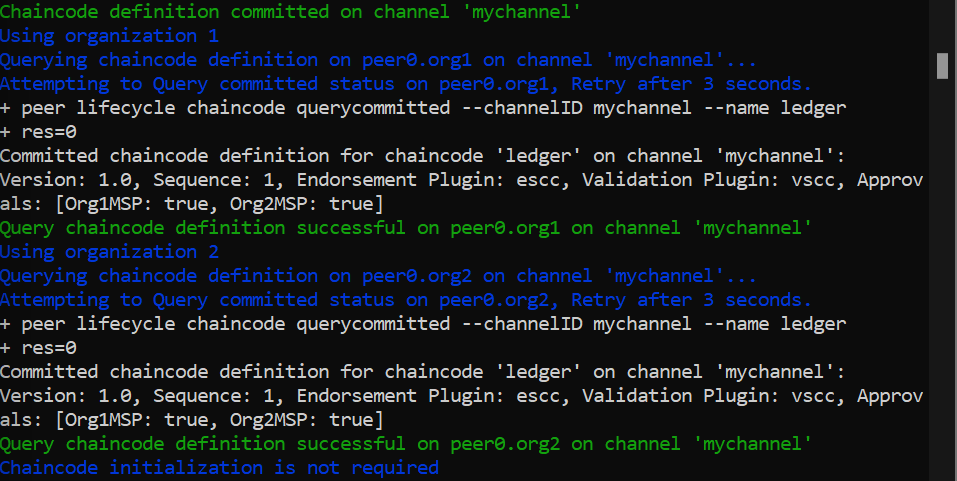
## 

## **5.5** **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.

result: 

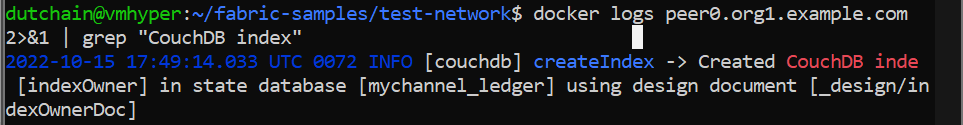
### ***5.5.1*** ***Verify index was deployed***

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"

result: 

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]

## **5.6** **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.

### ***5.6.1*** ***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](https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-ledger-queries/chaincode-go/asset_transfer_ledger_chaincode.go) 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](http://docs.couchdb.org/en/3.2.2/api/database/find.html#find-selectors).

· **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.

### ***5.6.2*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerchaincode.html?%20chaincode%20query#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 an asset owned by “tom”:

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

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"]}'

result: 

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](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html#cdb-create-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}]

## **5.7** **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 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\"]}"]}'

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](https://hyperledger-fabric.readthedocs.io/en/latest/peer_event_services.html) 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](https://github.com/hyperledger/fabric-samples/tree/main/off_chain_data) in the Fabric Samples.

## **5.8** **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](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_as_state_database.html#couchdb-pagination).

We will use the [Asset transfer ledger queries sample](https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-ledger-queries/chaincode-go/asset_transfer_ledger_chaincode.go) 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

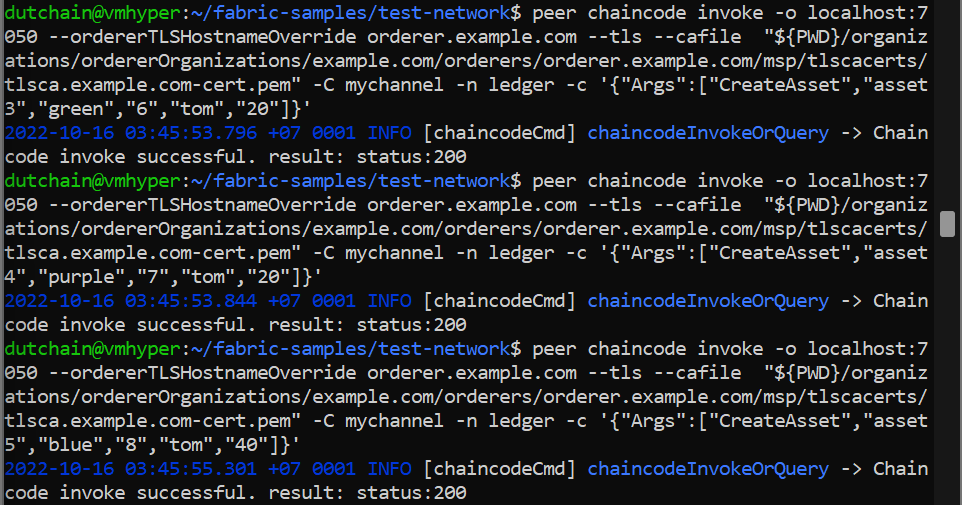
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile "${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n ledger -c '{"Args":["CreateAsset","asset2","yellow","5","tom","35"]}'

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"]}'

result:



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**

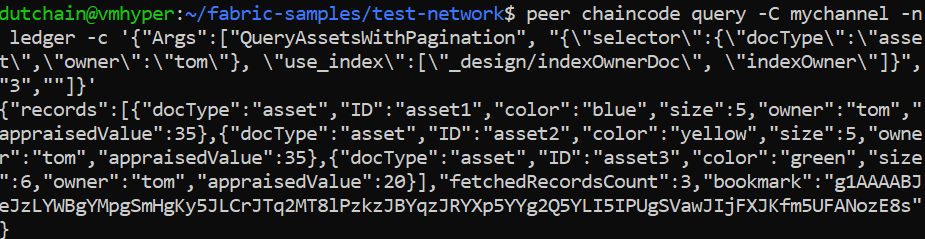
When no bookmark is specified, the query starts with the “first” page of records.

**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",""]}'

result:



The following response is received (carriage returns added for clarity), three of the five assets are returned because the pagsize 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}],

"fetchedRecordsCount":3,

"bookmark":"g1AAAABJeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqzJRYXp5YYg2Q5YLI5IPUgSVawJIjFXJKfm5UFANozE8s"

}

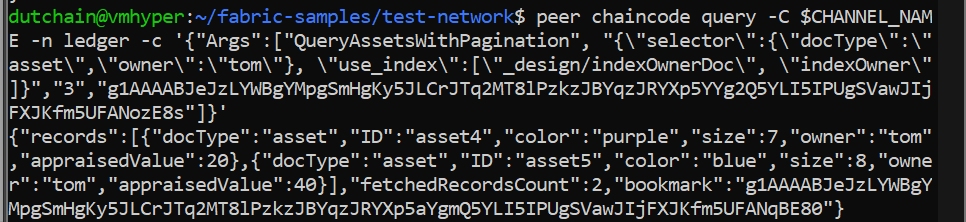
**Note**

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"]}'

result: 

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}],

"fetchedRecordsCount":2,

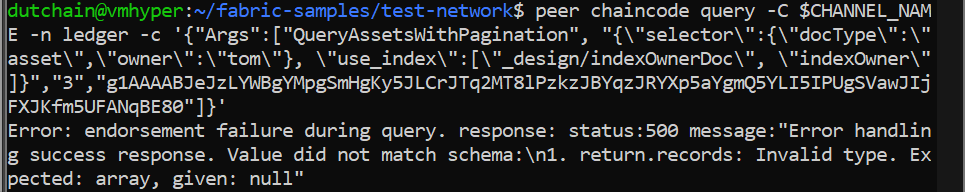
"bookmark":"g1AAAABJeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqzJRYXp5aYgmQ5YLI5IPUgSVawJIjFXJKfm5UFANqBE80"

}

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"]}'

result: 

For an example of how a client application can iterate over JSON query result sets using pagination, search for the getQueryResultForQueryStringWithPagination function in the [Asset transfer ledger queries sample](https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-ledger-queries/chaincode-go/asset_transfer_ledger_chaincode.go).

### ***5.8.1*** ***Range Query Pagination***

Bookmarks are also returned by the GetStateByRangeWithPagination shim API so that applications can page through range query results when using either LevelDB or CouchDB state database. The returned bookmark represents the next startKey that can be used to retrieve the next page of range query results. Once the results are exhausted the returned bookmark will be an empty string. If an endKey was specified in the range query and the results are exhausted, the returned bookmark will be the passed endKey when using CouchDB, and will be an empty string when using LevelDB.

For an example of how a client application can iterate over range query result sets using pagination, search for the GetAssetsByRangeWithPagination function in the [Asset transfer ledger queries sample](https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-ledger-queries/chaincode-go/asset_transfer_ledger_chaincode.go).

## **5.9** **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.

### ***5.9.1*** ***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](http://docs.couchdb.org/en/3.2.2/fauxton/index.html) 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**\"**, **\**

**\"**ddoc**\"**:**\"**indexOwnerDoc**\"**, **\**

**\"**type**\"**:**\"**json**\"**}" http://username:password**@hostname**:port/mychannel\_ledger/\_index

**Note**

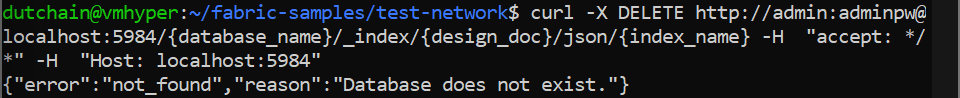
If you are using the test network configured with CouchDB, replace hostname:port with localhost:5984 and username:password with admin:adminpw.

## **5.10** **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://admin:adminpw**@localhost**:5984/{database\_name}/\_index/{design\_doc}/json/{index\_name} -H "accept: \*/\*" -H "Host: localhost:5984"

result:

To delete the index used in this tutorial, the curl command would be:

curl -X DELETE http://admin:adminpw**@localhost**:5984/mychannel\_ledger/\_index/indexOwnerDoc/json/indexOwner -H "accept: \*/\*" -H "Host: localhost:5984"

## result:

## **5.11** **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.

# **6.** **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**:

o Easy to list the channels that the ordering node is a consenter on.

o Simple process to remove a channel and the blocks associated with that channel.

o 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.

o 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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#remove-the-system-channel).

While creating the channel, this tutorial will take you through the following steps and concepts:

· [Prerequisites](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#prerequisites)

· [Step one: Generate the genesis block of the channel](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#step-one-generate-the-genesis-block-of-the-channel)

· [Step two: Use the osnadmin CLI to add the first orderer to the channel](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#step-two-use-the-osnadmin-cli-to-add-the-first-orderer-to-the-channel)

· [Step three: Join additional ordering nodes](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#step-three-join-additional-ordering-nodes)

· [Next steps](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_test_net.html) tutorial.

## **6.1** **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.

├── organizations

│ ├── ordererOrganizations

│ │ └── ordererOrg1.example.com

│ │ ├── msp

│ │ │ ├── cacerts

│ │ │ | └── ca-cert.pem

│ │ │ ├── config.yaml

│ │ │ ├── tlscacerts

│ │ │ | └── tls-ca-cert.pem

│ │ └── ordering-service-nodes

│ │ ├── osn1.ordererOrg1.example.com

│ │ │ ├── msp

│ │ │ │ ├── IssuerPublicKey

│ │ │ │ ├── IssuerRevocationPublicKey

│ │ │ │ ├── cacerts

│ │ │ │ │ └── ca-cert.pem

│ │ │ │ ├── config.yaml

│ │ │ │ ├── keystore

│ │ │ │ │ └── key.pem

│ │ │ │ ├── signcerts

│ │ │ │ │ └── cert.pem

│ │ │ │ └── user

│ │ │ └── tls

│ │ │ ├── IssuerPublicKey

│ │ │ ├── IssuerRevocationPublicKey

│ │ │ ├── cacerts

│ │ │ │ └── tls-ca-cert.pem

│ │ │ ├── keystore

│ │ │ │ └── tls-key.pem

│ │ │ ├── signcerts

│ │ │ │ └── cert.pem

│ │ │ └── user

├── admin-client

│ ├── client-tls-cert.pem

│ ├── client-tls-key.pem

│ └── client-tls-ca-cert.pem

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](https://hyperledger-fabric-ca.readthedocs.io/en/release-1.4/deployguide/use_CA.html#nodeous) 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.**

## **6.2** **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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#use-existing-ordering-service)

· [Deploy a new set of orderers](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#deploy-a-new-set-of-orderers)

### ***6.2.1*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#deploy-a-new-ordering-service).

#### *6.2.1.1* *Remove the system channel*

· Before attempting these steps ensure that you have [upgraded](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade.html) 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](https://github.com/hyperledger/fabric/blob/main/sampleconfig/orderer.yaml) for more information about these required parameters.

o 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.

o 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.

o 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.

o Admin.TLS.PrivateKey: - The path to and file name of the orderer private key issued by the TLS CA.

o Admin.TLS.Certificate: - The path to and file name of the orderer signed certificate issued by the TLS CA.

o 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.

o 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.

o 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](https://hyperledger-fabric.readthedocs.io/en/latest/kafka_raft_migration.html#entry-to-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:

o ORDERER\_ADMIN\_LISTENADDRESS corresponds to the Orderer.Admin.ListenAddress defined in the orderer.yaml for this orderer.

o 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.

o ADMIN\_TLS\_SIGN\_CERT with the path and file name of the admin client signed certificate from the TLS CA.

o 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.

### ***6.2.2*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#create-the-ordering-organization-msp-and-generate-ordering-node-certificates)

· [Configure the orderer.yaml file for each orderer](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#configure-the-orderer-yaml-file-for-each-orderer)

#### *6.2.2.1* *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](https://hyperledger-fabric-ca.readthedocs.io/en/release-1.4/deployguide/use_CA.html). 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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#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](https://hyperledger-fabric.readthedocs.io/en/latest/orderer/ordering_service.html#raft). 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.

#### *6.2.2.2* *Configure the orderer.yaml file for each orderer*

Follow the instructions in the [ordering service deployment guide](https://hyperledger-fabric.readthedocs.io/en/latest/deployorderer/ordererdeploy.html) 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](https://hyperledger-fabric.readthedocs.io/en/latest/deployorderer/ordererchecklist.html#channelparticipation) and [General.BoostrapMethod](https://hyperledger-fabric.readthedocs.io/en/latest/deployorderer/ordererchecklist.html#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:

2. export PATH=<path to download location>/bin:$PATH

3. 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:

4. export FABRIC\_CFG\_PATH=../config

5. You can now start the orderer by running the following command on each ordering node:

6. 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.

### ***6.2.3*** ***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](https://hyperledger-fabric-ca.readthedocs.io/en/release-1.4/deployguide/use_CA.html#create-the-org-msp-needed-to-add-an-org-to-a-channel) documentation to build your peer organization MSP definition. If the peers have been deployed, you should also include their address in the AnchorPeers: section.

## **6.3** **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.

### ***6.3.1*** ***Set up the configtxgen tool***

While it is possible to build the channel creation transaction file manually, it is easier to use the [configtxgen](https://hyperledger-fabric.readthedocs.io/en/latest/commands/configtxgen.html) 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

### ***6.3.2*** ***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](https://developers.google.com/protocol-buffers) 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:**](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#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](https://hyperledger-fabric.readthedocs.io/en/latest/membership/membership.html). Exactly which members are part of a channel configuration is defined in the **Profiles** section below.

· [**Orderer:**](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#orderer) Contains the list of ordering nodes that will form the consenter set of the channel.

· [**Profiles:**](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#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 consenters. 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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_config.html) 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.

#### *6.3.2.1* *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"

#### *6.3.2.2* *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.

#### *6.3.2.3* *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](https://github.com/hyperledger/fabric/blob/main/sampleconfig/configtx.yaml) 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.

### ***6.3.3*** ***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 SampleAppGenesisEtcdRaft: 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

## **6.4** **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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#remove-the-system-channel) 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 consenters (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.

## **6.5** **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 consenters set. What is more important is the height that the new orderer reached, compared to the height of the consenters. 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 consenters 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](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html), or by using the [fabric-config](https://github.com/hyperledger/fabric-config) library.

**Note:** you should never attempt any configuration changes to the Raft consenters, such as adding a consenter, unless all consenters 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.

## **6.6** **Next steps**

### ***6.6.1*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html) 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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerchannel.html#peer-channel-fetch) command. The organization can then use the genesis block to join the peer to the channel using the [peer channel join](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerchannel.html#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.

### ***6.6.2*** ***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.** **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 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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/configtxgen.html) tool to create a channel genesis block and then use the [osnadmin channel](https://hyperledger-fabric.readthedocs.io/en/latest/commands/osnadminchannel.html) 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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#deploy-a-new-set-of-orderers). 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](https://hyperledger-fabric.readthedocs.io/en/release-2.2/create_channel/create_channel.html) 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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_test_net.html#prerequisites)

· [Step one: Generate the genesis block of the channel](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_test_net.html#step-one-generate-the-genesis-block-of-the-channel)

· [Step two: Create the application channel](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_test_net.html#step-two-create-the-application-channel)

· [Next steps](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_test_net.html#next-steps)

## **7.1** **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](https://hyperledger-fabric.readthedocs.io/en/latest/prereqs.html) and [Installed the Samples, Binaries, and Docker Images](https://hyperledger-fabric.readthedocs.io/en/latest/install.html).

**Note:** After you create a channel and join peers to it, you will need to you 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](https://stedolan.github.io/jq/) is installed on your local machine.

## **7.2** **Prerequisites**

### ***7.2.1*** ***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

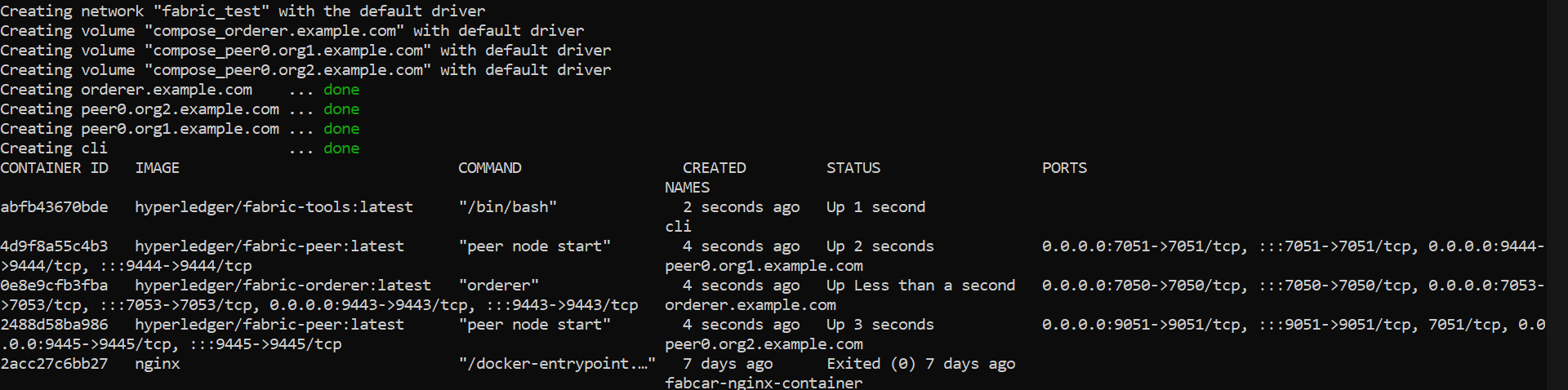
Run the following command to bring down the network:

./network.sh down

You can then use the following command to start the test network:

./network.sh up

result:



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 "fabric\_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

1667543b5634 hyperledger/fabric-tools:latest "/bin/bash" 1 second ago Up Less than a second cli

b6b117c81c7f hyperledger/fabric-peer:latest "peer node start" 2 seconds ago Up 1 second 0.0.0.0:7051->7051/tcp peer0.org1.example.com

703ead770e05 hyperledger/fabric-orderer:latest "orderer" 2 seconds ago Up Less than a second 0.0.0.0:7050->7050/tcp, 0.0.0.0:7053->7053/tcp orderer.example.com

718d43f5f312 hyperledger/fabric-peer:latest "peer node start" 2 seconds ago Up 1 second 7051/tcp, 0.0.0.0:9051->9051/tcp 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.

### ***7.2.2*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/configtxgen.html) 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](https://hyperledger-fabric.readthedocs.io/en/latest/install.html), 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

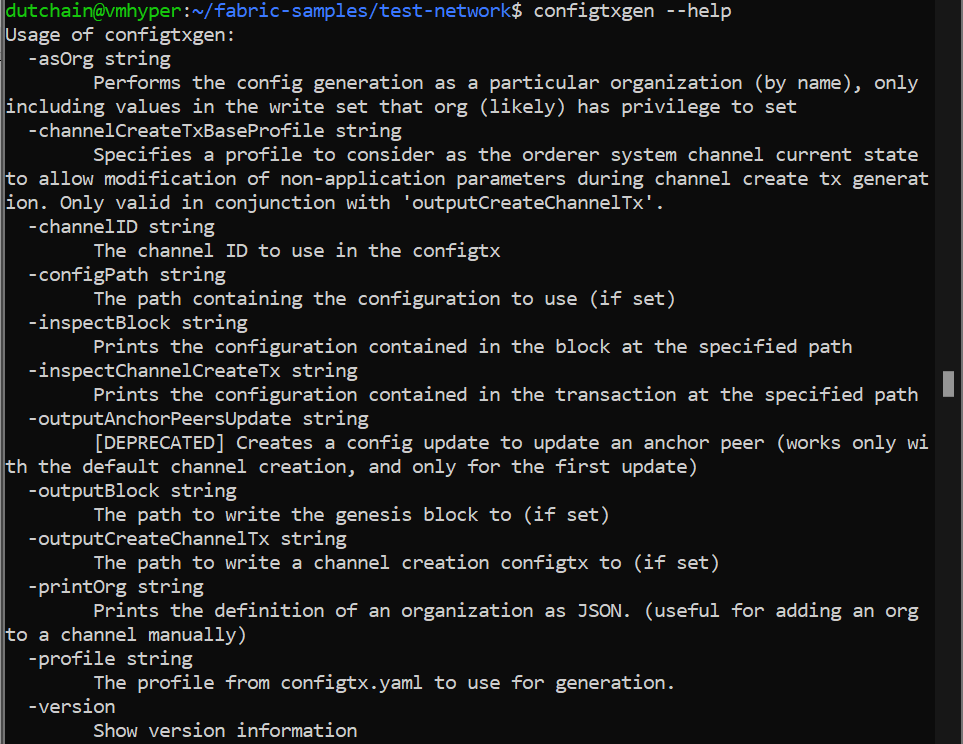
Next, before you can use configtxgen, you need to the set the FABRIC\_CFG\_PATH environment variable to the location of the test network folder that contains the configtx.yaml file. Because we are using the test network, we reference the configtx folder:

export FABRIC\_CFG\_PATH=${PWD}/configtx

Now verify that you can use the tool by printing the configtxgen help text:

configtxgen --help

### result:



### ***7.2.3*** ***The configtx.yaml file***

For the test network, the configtxgen tool uses the channel profiles that are defined in the configtxt\configtx.yaml file to create the channel configuration and write it to the [protobuf format](https://developers.google.com/protocol-buffers) 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](https://hyperledger-fabric.readthedocs.io/en/latest/membership/membership.html).

· **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

· ClientTLSCert: ../organizations/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt

· ServerTLSCert: ../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 application 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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_config.html) 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.

## **7.3** **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:

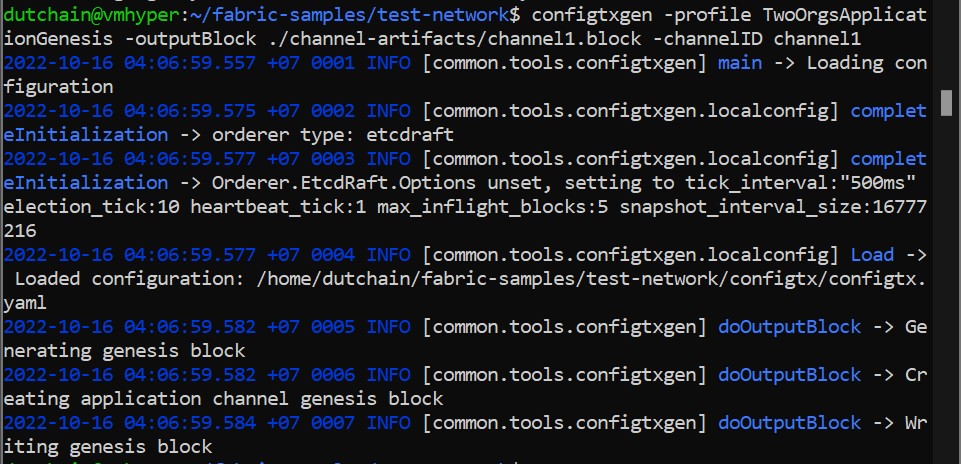
configtxgen -profile TwoOrgsApplicationGenesis -outputBlock ./channel-artifacts/channel1.block -channelID channel1

· **-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:

result: 

[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

[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

## **7.4** **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",

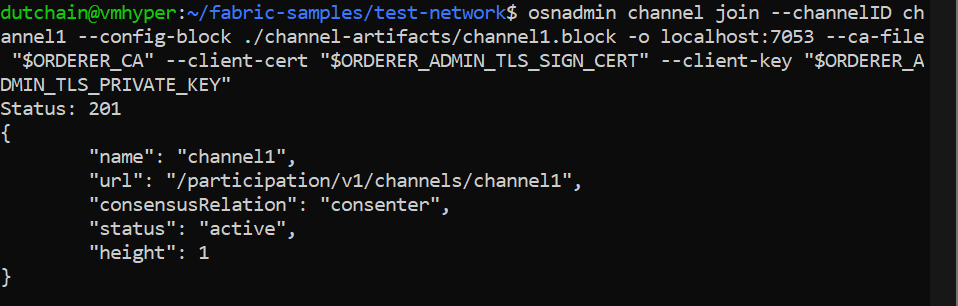
"url": "/participation/v1/channels/channel1",

"consensusRelation": "consenter",

"status": "active",

"height": 1

}

result:

The channel is active and ready for peers to join.

### ***7.4.1*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html#step-three-join-additional-ordering-nodes).

### ***7.4.2*** ***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.

## **7.5** **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.

### ***7.5.1*** ***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"

}

]

}

### result:

### ***7.5.2*** ***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:

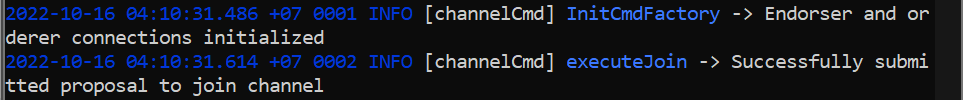
peer channel join -b ./channel-artifacts/channel1.block

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

result:



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

## result:

## **7.6** **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](https://hyperledger-fabric.readthedocs.io/en/latest/gossip.html#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](https://hyperledger-fabric.readthedocs.io/en/latest/gossip.html).

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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/configtxlator.html) tool to update the channel configuration and select an anchor peer for Org1 and Org2.

**Note:** If [jq](https://stedolan.github.io/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"

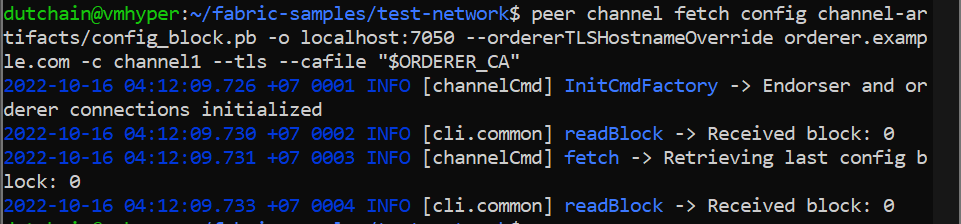
Because the most recent channel configuration block is the channel genesis block, the command returns block 0 from the channel.

[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

result:

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:

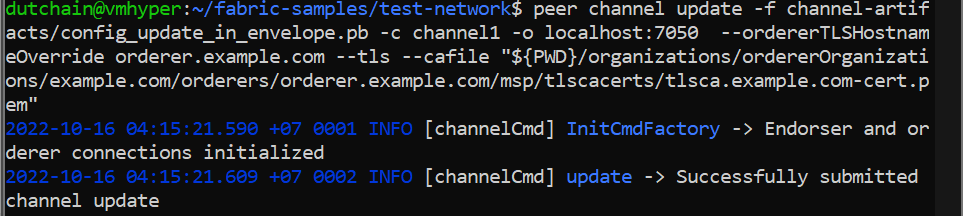
cd ..

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

result:

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](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html).

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="}

## result:

## **7.7** **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

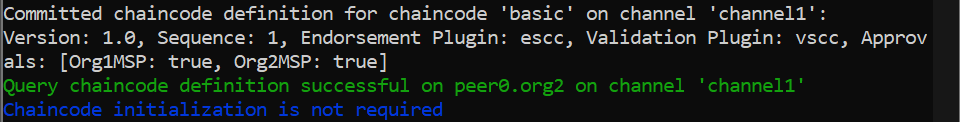
After you run the command, you should see the chaincode being deployed to the channel in your logs.

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

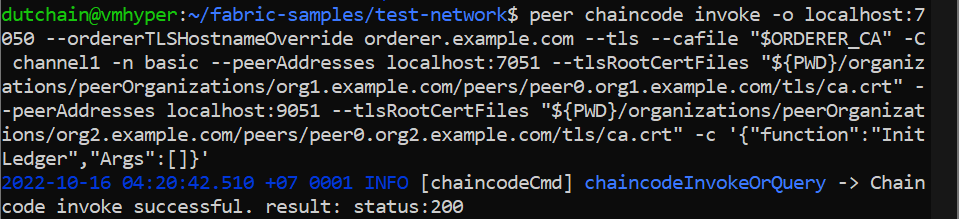
result:

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

result:

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}]

### result:

### ***7.7.1*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html) tutorial to learn more about using the osnadmin channel commands.

# **8.** **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](https://hyperledger-fabric.readthedocs.io/en/latest/commands/configtxgen.html) 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](https://developers.google.com/protocol-buffers) that can be read by Fabric.

## **8.1** **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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_config.html#organizations)

· [Capabilities](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_config.html#capabilities)

· [Application](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_config.html#application)

· [Orderer](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_config.html#orderer)

· [Channel](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_config.html#channel)

· [Profiles](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_config.html#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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/channel_policies.html).

Building off of the [Creating a channel tutorial](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html), 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](https://github.com/hyperledger/fabric/blob/main/sampleconfig/configtx.yaml).

## **8.2** **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](https://hyperledger-fabric.readthedocs.io/en/latest/membership/membership.html). The channel MSP is stored in the channel configuration and contains the certificates that are used to the 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*

*# /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')"

*# OrdererEndpoints is a list of all orderers this org runs which clients*

*# and peers may to connect to to push transactions and receive blocks respectively.*

OrdererEndpoints:

- "orderer.example.com:7050"

· 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:

o 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.

o 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.

o 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.

o 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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/channel_policies.html).

· The OrdererEndpoints indicates the orderer node endpoints that this organization makes available to clients and peers. Service discovery uses this information so that clients can pass the appropriate TLS certificates when connecting to an orderer endpoint.

## **8.3** **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](https://hyperledger-fabric.readthedocs.io/en/latest/capabilities_concept.html) concept topic.

## **8.4** **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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/channel_policies.html) tutorial.

## **8.5** **Orderer**

Each channel configuration includes the orderer nodes in the channel [consenter set](https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html#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 consenters that can participate in the consensus process. Because the test network only uses a single ordering node, the consenters list contains only one endpoint:

EtcdRaft:

Consenters:

- 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

Each ordering node in the list of consenters 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 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 BatchSize property includes MaxMessageCount, AbsoluteMaxBytes, and PreferredMaxBytes settings. A block will be cut when any of the BatchTimeout or BatchSize criteria has been met. For AbsoluteMaxBytes it is recommended not to exceed 49 MB, given the default gRPC maximum message size of 100 MB configured on orderer and peer nodes (and allowing for message expansion during communication).

· 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](https://hyperledger-fabric.readthedocs.io/en/latest/raft_configuration.html).

## **8.6** **Channel**

The channel section defines that 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.

## **8.7** **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](https://en.wikipedia.org/wiki/YAML) 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

### ***8.7.1*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html#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.

### ***8.7.2*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/channel_update_tutorial.html).

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.

### ***8.7.3*** ***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](https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_participation.html) 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

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.

# **9.** **Adding an Org to a Channel**

**Note**

Ensure that you have downloaded the appropriate images and binaries as outlined in [Install Fabric and Fabric Samples](https://hyperledger-fabric.readthedocs.io/en/latest/install.html) and [Prerequisites](https://hyperledger-fabric.readthedocs.io/en/latest/prereqs.html) 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](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html)). 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).

## **9.1** **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

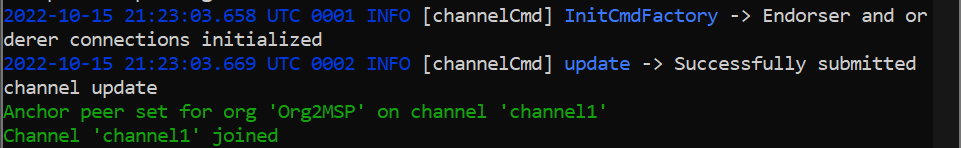
You can now use the script to bring up the test network with one channel named channel1:

./network.sh up createChannel -c channel1

If the command was successful, you can see the following message printed in your logs:

Channel 'channel1' joined

result:



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.

## **9.2** **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

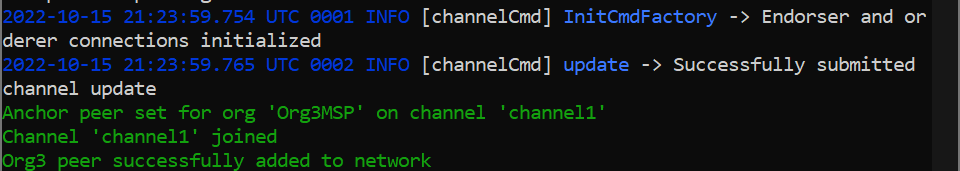
./addOrg3.sh up -c channel1

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

result:



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.

## **9.3** **Bring Org3 into the Channel Manually**

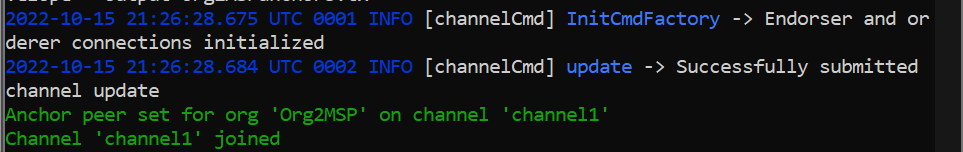
If you just used the addOrg3.sh script, you’ll need to bring your network down. The following command will bring down all running components and remove the crypto material for all organizations:

cd ..

./network.sh down

After the network is brought down, bring it back up again:

./network.sh up createChannel -c channel1

result:

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.

## **9.4** **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:

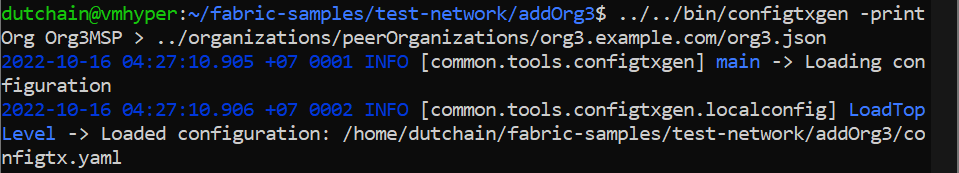
../../bin/cryptogen generate --config=org3-crypto.yaml --output="../organizations"

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.

export FABRIC\_CFG\_PATH=$PWD

../../bin/configtxgen -printOrg Org3MSP > ../organizations/peerOrganizations/org3.example.com/org3.json

result:

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

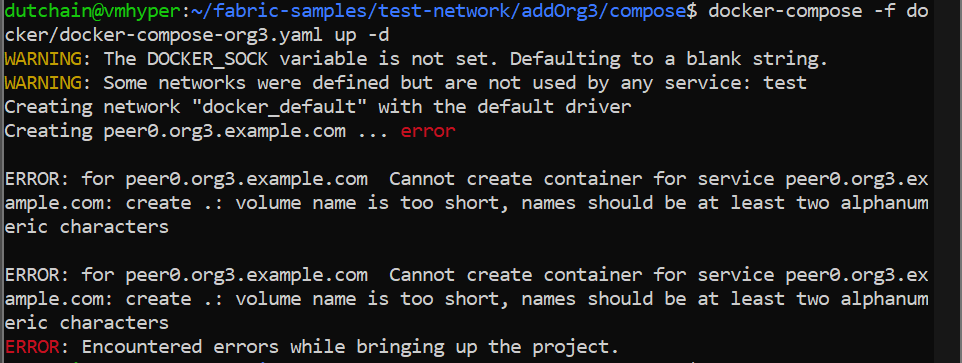
· a TLS root cert, used by the gossip protocol to identify Org3 for block dissemination and service discovery

We will add Org3 to the channel by appending this organization definition to the channel configuration.

## **9.5** **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

error: 

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.

## **9.6** **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.

cd ..

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

## **9.7** **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:

cd channel-artifacts

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](https://stedolan.github.io/jq/) 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](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html).

## **9.8** **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

## **9.9** **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:

cd ..

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

## **9.10** **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](https://hyperledger-fabric.readthedocs.io/en/latest/peer_ledger_snapshot.html#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>

## **9.11** **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.

## **9.12** **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](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html).

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 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 to 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 lets 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.

## **9.13** **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.

## **9.14** **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:

cd channel-artifacts

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

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:

cd ..

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.

# **10.** **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](https://hyperledger-fabric.readthedocs.io/en/release-2.2/config_update.html).

## **10.1** **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](https://hyperledger-fabric.readthedocs.io/en/latest/policies/policies.html), 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](http://github.com/hyperledger/fabric/blob/main/sampleconfig/configtx.yaml).

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.

## **10.2** **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.

### ***10.2.1*** ***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.