**Building Your First Network**

**Note**

These instructions have been verified to work against the latest stable Docker images and the pre-compiled setup utilities within the supplied tar file. If you run these commands with images or tools from the current master branch, it is possible that you will see configuration and panic errors.

The build your first network (BYFN) scenario provisions a sample Hyperledger Fabric network consisting of two organizations, each maintaining two peer nodes. It also will deploy a “Solo” ordering service by default, though other ordering service implementations are available.

**Install prerequisites**

Before we begin, if you haven’t already done so, you may wish to check that you have all the [Prerequisites](https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.html) installed on the platform(s) on which you’ll be developing blockchain applications and/or operating Hyperledger Fabric.

You will also need to [Install Samples, Binaries and Docker Images](https://hyperledger-fabric.readthedocs.io/en/release-1.4/install.html). You will notice that there are a number of samples included in the fabric-samples repository. We will be using the first-network sample. Let’s open that sub-directory now.

cd fabric**-**samples**/**first**-**network

**Note**

The supplied commands in this documentation **MUST** be run from your first-network sub-directory of the fabric-samples repository clone. If you elect to run the commands from a different location, the various provided scripts will be unable to find the binaries.

**Want to run it now?**

We provide a fully annotated script — byfn.sh — that leverages these Docker images to quickly bootstrap a Hyperledger Fabric network that by default is comprised of four peers representing two different organizations, and an orderer node. It will also launch a container to run a scripted execution that will join peers to a channel, deploy a chaincode and drive execution of transactions against the deployed chaincode.

Here’s the help text for the byfn.sh script:

Usage:

byfn**.**sh **<**mode**>** [**-**c **<**channel name**>**] [**-**t **<**timeout**>**] [**-**d **<**delay**>**] [**-**f **<**docker**-**compose**-**file**>**] [**-**s **<**dbtype**>**] [**-**l **<**language**>**] [**-**o **<**consensus**-**type**>**] [**-**i **<**imagetag**>**] [**-**v]"

**<**mode**>** **-** one of 'up', 'down', 'restart', 'generate' **or** 'upgrade'"

**-** 'up' **-** bring up the network **with** docker**-**compose up"

**-** 'down' **-** clear the network **with** docker**-**compose down"

**-** 'restart' **-** restart the network"

**-** 'generate' **-** generate required certificates **and** genesis block"

**-** 'upgrade' **-** upgrade the network **from** version 1.3**.**x to 1.4**.**0"

**-**c **<**channel name**>** **-** channel name to use (defaults to \"mychannel\")"

**-**t **<**timeout**>** **-** CLI timeout duration **in** seconds (defaults to 10)"

**-**d **<**delay**>** **-** delay duration **in** seconds (defaults to 3)"

**-**f **<**docker**-**compose**-**file**>** **-** specify which docker**-**compose file use (defaults to docker**-**compose**-**cli**.**yaml)"

**-**s **<**dbtype**>** **-** the database backend to use: goleveldb (default) **or** couchdb"

**-**l **<**language**>** **-** the chaincode language: golang (default), node, **or** java"

**-**o **<**consensus**-**type**>** **-** the consensus**-**type of the ordering service: solo (default), kafka, **or** etcdraft"

**-**i **<**imagetag**>** **-** the tag to be used to launch the network (defaults to \"latest\")"

**-**v **-** verbose mode"

byfn**.**sh **-**h (print this message)"

Typically, one would first generate the required certificates **and**

genesis block, then bring up the network**.** e**.**g**.**:"

byfn**.**sh generate **-**c mychannel"

byfn**.**sh up **-**c mychannel **-**s couchdb"

byfn**.**sh up **-**c mychannel **-**s couchdb **-**i 1.4**.**0"

byfn**.**sh up **-**l node"

byfn**.**sh down **-**c mychannel"

byfn**.**sh upgrade **-**c mychannel"

Taking all defaults:"

byfn**.**sh generate"

byfn**.**sh up"

byfn**.**sh down"

If you choose not to supply a flag, the script will use default values.

**Generate Network Artifacts**

Ready to give it a go? Okay then! Execute the following command:

**./**byfn**.**sh generate

You will see a brief description as to what will occur, along with a yes/no command line prompt. Respond with a y or hit the return key to execute the described action.

Generating certs and genesis block for channel 'mychannel' with CLI timeout of '10' seconds and CLI delay of '3' seconds

Continue? [Y/n] y

proceeding ...

/Users/xxx/dev/fabric-samples/bin/cryptogen

##########################################################

##### Generate certificates using cryptogen tool #########

##########################################################

org1.example.com

2017-06-12 21:01:37.334 EDT [bccsp] GetDefault -> WARN 001 Before using BCCSP, please call InitFactories(). Falling back to bootBCCSP.

...

/Users/xxx/dev/fabric-samples/bin/configtxgen

##########################################################

######### Generating Orderer Genesis block ##############

##########################################################

2017-06-12 21:01:37.558 EDT [common/configtx/tool] main -> INFO 001 Loading configuration

2017-06-12 21:01:37.562 EDT [msp] getMspConfig -> INFO 002 intermediate certs folder not found at [/Users/xxx/dev/byfn/crypto-config/ordererOrganizations/example.com/msp/intermediatecerts]. Skipping.: [stat /Users/xxx/dev/byfn/crypto-config/ordererOrganizations/example.com/msp/intermediatecerts: no such file or directory]

...

2017-06-12 21:01:37.588 EDT [common/configtx/tool] doOutputBlock -> INFO 00b Generating genesis block

2017-06-12 21:01:37.590 EDT [common/configtx/tool] doOutputBlock -> INFO 00c Writing genesis block

#################################################################

### Generating channel configuration transaction 'channel.tx' ###

#################################################################

2017-06-12 21:01:37.634 EDT [common/configtx/tool] main -> INFO 001 Loading configuration

2017-06-12 21:01:37.644 EDT [common/configtx/tool] doOutputChannelCreateTx -> INFO 002 Generating new channel configtx

2017-06-12 21:01:37.645 EDT [common/configtx/tool] doOutputChannelCreateTx -> INFO 003 Writing new channel tx

#################################################################

####### Generating anchor peer update for Org1MSP ##########

#################################################################

2017-06-12 21:01:37.674 EDT [common/configtx/tool] main -> INFO 001 Loading configuration

2017-06-12 21:01:37.678 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 002 Generating anchor peer update

2017-06-12 21:01:37.679 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 003 Writing anchor peer update

#################################################################

####### Generating anchor peer update for Org2MSP ##########

#################################################################

2017-06-12 21:01:37.700 EDT [common/configtx/tool] main -> INFO 001 Loading configuration

2017-06-12 21:01:37.704 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 002 Generating anchor peer update

2017-06-12 21:01:37.704 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 003 Writing anchor peer update

This first step generates all of the certificates and keys for our various network entities, the genesis block used to bootstrap the ordering service, and a collection of configuration transactions required to configure a [Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/glossary.html#channel).

**Bring Up the Network**

Next, you can bring the network up with one of the following commands:

**./**byfn**.**sh up

The above command will compile Golang chaincode images and spin up the corresponding containers. Go is the default chaincode language, however there is also support for [Node.js](https://fabric-shim.github.io/) and [Java](https://hyperledger.github.io/fabric-chaincode-java/) chaincode. If you’d like to run through this tutorial with node chaincode, pass the following command instead:

*# we use the -l flag to specify the chaincode language*

*# forgoing the -l flag will default to Golang*

**./**byfn**.**sh up **-**l node

**Note**

For more information on the Node.js shim, please refer to its [documentation](https://fabric-shim.github.io/ChaincodeInterface.html).

**Note**

For more information on the Java shim, please refer to its [documentation](https://hyperledger.github.io/fabric-chaincode-java/release-1.4/api/org/hyperledger/fabric/shim/Chaincode.html).

Тo make the sample run with Java chaincode, you have to specify -l java as follows:

**./**byfn**.**sh up **-**l java

**Note**

Do not run both of these commands. Only one language can be tried unless you bring down and recreate the network between.

In addition to support for multiple chaincode languages, you can also issue a flag that will bring up a five node Raft ordering service or a Kafka ordering service instead of the one node Solo orderer. For more information about the currently supported ordering service implementations, check out [The Ordering Service](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html).

To bring up the network with a Raft ordering service, issue:

**./**byfn**.**sh up **-**o etcdraft

To bring up the network with a Kafka ordering service, issue:

**./**byfn**.**sh up **-**o kafka

Once again, you will be prompted as to whether you wish to continue or abort. Respond with a y or hit the return key:

Starting for channel 'mychannel' with CLI timeout of '10' seconds and CLI delay of '3' seconds

Continue? [Y/n]

proceeding ...

Creating network "net\_byfn" with the default driver

Creating peer0.org1.example.com

Creating peer1.org1.example.com

Creating peer0.org2.example.com

Creating orderer.example.com

Creating peer1.org2.example.com

Creating cli

\_\_\_\_ \_\_\_\_\_ \_ \_\_\_\_ \_\_\_\_\_

/ \_\_\_| |\_ \_| / \ | \_ \ |\_ \_|

\\_\_\_ \ | | / \_ \ | |\_) | | |

\_\_\_) | | | / \_\_\_ \ | \_ < | |

|\_\_\_\_/ |\_| /\_/ \\_\ |\_| \\_\ |\_|

Channel name : mychannel

Creating channel...

The logs will continue from there. This will launch all of the containers, and then drive a complete end-to-end application scenario. Upon successful completion, it should report the following in your terminal window:

Query Result: 90

2017**-**05**-**16 17:08:15.158 UTC [main] main **->** INFO 008 Exiting**.....**

**=====================** Query successful on peer1**.**org2 on channel 'mychannel' **=====================**

**=====================** All GOOD, BYFN execution completed **=====================**

\_\_\_\_\_ \_ \_ \_\_\_\_

**|** \_\_\_\_**|** **|** \ **|** **|** **|** \_ \

**|** \_**|** **|** \**|** **|** **|** **|** **|** **|**

**|** **|**\_\_\_ **|** **|**\ **|** **|** **|**\_**|** **|**

**|**\_\_\_\_\_**|** **|**\_**|** \\_**|** **|**\_\_\_\_**/**

You can scroll through these logs to see the various transactions. If you don’t get this result, then jump down to the [Troubleshooting](https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html#troubleshoot) section and let’s see whether we can help you discover what went wrong.

**Bring Down the Network**

Finally, let’s bring it all down so we can explore the network setup one step at a time. The following will kill your containers, remove the crypto material and four artifacts, and delete the chaincode images from your Docker Registry:

**./**byfn**.**sh down

Once again, you will be prompted to continue, respond with a y or hit the return key:

Stopping with channel 'mychannel' and CLI timeout of '10'

Continue? [Y/n] y

proceeding ...

WARNING: The CHANNEL\_NAME variable is not set. Defaulting to a blank string.

WARNING: The TIMEOUT variable is not set. Defaulting to a blank string.

Removing network net\_byfn

468aaa6201ed

...

Untagged: dev-peer1.org2.example.com-mycc-1.0:latest

Deleted: sha256:ed3230614e64e1c83e510c0c282e982d2b06d148b1c498bbdcc429e2b2531e91

...

If you’d like to learn more about the underlying tooling and bootstrap mechanics, continue reading. In these next sections we’ll walk through the various steps and requirements to build a fully-functional Hyperledger Fabric network.

**Note**

The manual steps outlined below assume that the FABRIC\_LOGGING\_SPEC in the cli container is set to DEBUG. You can set this by modifying the docker-compose-cli.yaml file in the first-network directory. e.g.

cli:

container\_name: cli

image: hyperledger/fabric-tools:$IMAGE\_TAG

tty: true

stdin\_open: true

environment:

- GOPATH=/opt/gopath

- CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock

- FABRIC\_LOGGING\_SPEC=DEBUG

#- FABRIC\_LOGGING\_SPEC=INFO

**Crypto Generator**

We will use the cryptogen tool to generate the cryptographic material (x509 certs and signing keys) for our various network entities. These certificates are representative of identities, and they allow for sign/verify authentication to take place as our entities communicate and transact.

**How does it work?**

Cryptogen consumes a file — crypto-config.yaml — that contains the network topology and allows us to generate a set of certificates and keys for both the Organizations and the components that belong to those Organizations. Each Organization is provisioned a unique root certificate (ca-cert) that binds specific components (peers and orderers) to that Org. By assigning each Organization a unique CA certificate, we are mimicking a typical network where a participating [Member](https://hyperledger-fabric.readthedocs.io/en/release-1.4/glossary.html#member) would use its own Certificate Authority. Transactions and communications within Hyperledger Fabric are signed by an entity’s private key (keystore), and then verified by means of a public key (signcerts).

You will notice a count variable within this file. We use this to specify the number of peers per Organization; in our case there are two peers per Org. We won’t delve into the minutiae of [x.509 certificates and public key infrastructure](https://en.wikipedia.org/wiki/Public_key_infrastructure) right now. If you’re interested, you can peruse these topics on your own time.

After we run the cryptogen tool, the generated certificates and keys will be saved to a folder titled crypto-config. Note that the crypto-config.yaml file lists five orderers as being tied to the orderer organization. While the cryptogen tool will create certificates for all five of these orderers, unless the Raft or Kafka ordering services are being used, only one of these orderers will be used in a Solo ordering service implementation and be used to create the system channel and mychannel.

**Configuration Transaction Generator**

The configtxgen tool is used to create four configuration artifacts:

* orderer genesis block,
* channel configuration transaction,
* and two anchor peer transactions - one for each Peer Org.

Please see [configtxgen](https://hyperledger-fabric.readthedocs.io/en/release-1.4/commands/configtxgen.html) for a complete description of this tool’s functionality.

The orderer block is the [Genesis Block](https://hyperledger-fabric.readthedocs.io/en/release-1.4/glossary.html#genesis-block) for the ordering service, and the channel configuration transaction file is broadcast to the orderer at [Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/glossary.html#channel) creation time. The anchor peer transactions, as the name might suggest, specify each Org’s [Anchor Peer](https://hyperledger-fabric.readthedocs.io/en/release-1.4/glossary.html#anchor-peer) on this channel.

**How does it work?**

Configtxgen consumes a file - configtx.yaml - that contains the definitions for the sample network. There are three members - one Orderer Org (OrdererOrg) and two Peer Orgs (Org1 & Org2) each managing and maintaining two peer nodes. This file also specifies a consortium - SampleConsortium - consisting of our two Peer Orgs. Pay specific attention to the “Profiles” section at the bottom of this file. You will notice that we have several unique profiles. A few are worth noting:

* TwoOrgsOrdererGenesis: generates the genesis block for a Solo ordering service.
* SampleMultiNodeEtcdRaft: generates the genesis block for a Raft ordering service. Only used if you issue the -o flag and specify etcdraft.
* SampleDevModeKafka: generates the genesis block for a Kafka ordering service. Only used if you issue the -o flag and specify kafka.
* TwoOrgsChannel: generates the genesis block for our channel, mychannel.

These headers are important, as we will pass them in as arguments when we create our artifacts.

**Note**

Notice that our SampleConsortium is defined in the system-level profile and then referenced by our channel-level profile. Channels exist within the purview of a consortium, and all consortia must be defined in the scope of the network at large.

This file also contains two additional specifications that are worth noting. Firstly, we specify the anchor peers for each Peer Org (peer0.org1.example.com & peer0.org2.example.com). Secondly, we point to the location of the MSP directory for each member, in turn allowing us to store the root certificates for each Org in the orderer genesis block. This is a critical concept. Now any network entity communicating with the ordering service can have its digital signature verified.

**Run the tools**

You can manually generate the certificates/keys and the various configuration artifacts using the configtxgen and cryptogen commands. Alternately, you could try to adapt the byfn.sh script to accomplish your objectives.

**Manually generate the artifacts**

You can refer to the generateCerts function in the byfn.sh script for the commands necessary to generate the certificates that will be used for your network configuration as defined in the crypto-config.yaml file. However, for the sake of convenience, we will also provide a reference here.

First let’s run the cryptogen tool. Our binary is in the bin directory, so we need to provide the relative path to where the tool resides.

**../**bin**/**cryptogen generate **--**config**=./**crypto**-**config**.**yaml

You should see the following in your terminal:

org1**.**example**.**com

org2**.**example**.**com

The certs and keys (i.e. the MSP material) will be output into a directory - crypto-config - at the root of the first-network directory.

Next, we need to tell the configtxgen tool where to look for the configtx.yaml file that it needs to ingest. We will tell it look in our present working directory:

export FABRIC\_CFG\_PATH=$PWD

Then, we’ll invoke the configtxgen tool to create the orderer genesis block:

**../**bin**/**configtxgen **-**profile TwoOrgsOrdererGenesis **-**channelID byfn**-**sys**-**channel **-**outputBlock **./**channel**-**artifacts**/**genesis**.**block

To output a genesis block for a Raft ordering service, this command should be:

**../**bin**/**configtxgen **-**profile SampleMultiNodeEtcdRaft **-**channelID byfn**-**sys**-**channel **-**outputBlock **./**channel**-**artifacts**/**genesis**.**block

Note the SampleMultiNodeEtcdRaft profile being used here.

To output a genesis block for a Kafka ordering service, issue:

**../**bin**/**configtxgen **-**profile SampleDevModeKafka **-**channelID byfn**-**sys**-**channel **-**outputBlock **./**channel**-**artifacts**/**genesis**.**block

If you are not using Raft or Kafka, you should see an output similar to the following:

2017**-**10**-**26 19:21:56.301 EDT [common**/**tools**/**configtxgen] main **->** INFO 001 Loading configuration

2017**-**10**-**26 19:21:56.309 EDT [common**/**tools**/**configtxgen] doOutputBlock **->** INFO 002 Generating genesis block

2017**-**10**-**26 19:21:56.309 EDT [common**/**tools**/**configtxgen] doOutputBlock **->** INFO 003 Writing genesis block

**Note**

The orderer genesis block and the subsequent artifacts we are about to create will be output into the channel-artifacts directory at the root of this project. The *channelID* in the above command is the name of the system channel.

**Create a Channel Configuration Transaction**

Next, we need to create the channel transaction artifact. Be sure to replace $CHANNEL\_NAME or set CHANNEL\_NAME as an environment variable that can be used throughout these instructions:

# The channel.tx artifact contains the definitions for our sample channel

export CHANNEL\_NAME=mychannel && ../bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/channel.tx -channelID $CHANNEL\_NAME

Note that you don’t have to issue a special command for the channel if you are using a Raft or Kafka ordering service. The TwoOrgsChannel profile will use the ordering service configuration you specified when creating the genesis block for the network.

If you are not using a Raft or Kafka ordering service, you should see an output similar to the following in your terminal:

2017**-**10**-**26 19:24:05.324 EDT [common**/**tools**/**configtxgen] main **->** INFO 001 Loading configuration

2017**-**10**-**26 19:24:05.329 EDT [common**/**tools**/**configtxgen] doOutputChannelCreateTx **->** INFO 002 Generating new channel configtx

2017**-**10**-**26 19:24:05.329 EDT [common**/**tools**/**configtxgen] doOutputChannelCreateTx **->** INFO 003 Writing new channel tx

Next, we will define the anchor peer for Org1 on the channel that we are constructing. Again, be sure to replace $CHANNEL\_NAME or set the environment variable for the following commands. The terminal output will mimic that of the channel transaction artifact:

../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx -channelID $CHANNEL\_NAME -asOrg Org1MSP

Now, we will define the anchor peer for Org2 on the same channel:

../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchors.tx -channelID $CHANNEL\_NAME -asOrg Org2MSP

**Start the network**

**Note**

If you ran the byfn.sh example above previously, be sure that you have brought down the test network before you proceed (see [Bring Down the Network](https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html#bring-down-the-network)).

We will leverage a script to spin up our network. The docker-compose file references the images that we have previously downloaded, and bootstraps the orderer with our previously generated genesis.block.

We want to go through the commands manually in order to expose the syntax and functionality of each call.

First let’s start our network:

docker**-**compose **-**f docker**-**compose**-**cli**.**yaml up **-**d

If you want to see the realtime logs for your network, then do not supply the -d flag. If you let the logs stream, then you will need to open a second terminal to execute the CLI calls.

**Create & Join Channel**

Recall that we created the channel configuration transaction using the configtxgen tool in the [Create a Channel Configuration Transaction](https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html#createchanneltx) section, above. You can repeat that process to create additional channel configuration transactions, using the same or different profiles in the configtx.yaml that you pass to the configtxgen tool. Then you can repeat the process defined in this section to establish those other channels in your network.

We will enter the CLI container using the docker exec command:

docker exec **-**it cli bash

If successful you should see the following:

root@0d78bb69300d:**/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer*#*

For the following CLI commands to work, we need to preface our commands with the four environment variables given below. These variables for peer0.org1.example.com are baked into the CLI container, therefore we can operate without passing them. **HOWEVER**, if you want to send calls to other peers or the orderer, override the environment variables as seen in the example below when you make any CLI calls:

*# Environment variables for PEER0*

CORE\_PEER\_MSPCONFIGPATH**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org1**.**example**.**com**/**users**/**Admin@org1**.**example**.**com**/**msp

CORE\_PEER\_ADDRESS**=**peer0**.**org1**.**example**.**com:7051

CORE\_PEER\_LOCALMSPID**=**"Org1MSP"

CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org1**.**example**.**com**/**peers**/**peer0**.**org1**.**example**.**com**/**tls**/**ca**.**crt

Next, we are going to pass in the generated channel configuration transaction artifact that we created in the [Create a Channel Configuration Transaction](https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html#createchanneltx) section (we called it channel.tx) to the orderer as part of the create channel request.

We specify our channel name with the -c flag and our channel configuration transaction with the -f flag. In this case it is channel.tx, however you can mount your own configuration transaction with a different name. Once again we will set the CHANNEL\_NAME environment variable within our CLI container so that we don’t have to explicitly pass this argument. Channel names must be all lower case, less than 250 characters long and match the regular expression [a-z][a-z0-9.-]\*.

export CHANNEL\_NAME=mychannel

# the channel.tx file is mounted in the channel-artifacts directory within your CLI container

# as a result, we pass the full path for the file

# we also pass the path for the orderer ca-cert in order to verify the TLS handshake

# be sure to export or replace the $CHANNEL\_NAME variable appropriately

peer channel create -o orderer.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/channel.tx --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem

**Note**

Notice the --cafile that we pass as part of this command. It is the local path to the orderer’s root cert, allowing us to verify the TLS handshake.

This command returns a genesis block - <CHANNEL\_NAME.block> - which we will use to join the channel. It contains the configuration information specified in channel.tx If you have not made any modifications to the default channel name, then the command will return you a proto titled mychannel.block.

**Note**

You will remain in the CLI container for the remainder of these manual commands. You must also remember to preface all commands with the corresponding environment variables when targeting a peer other than peer0.org1.example.com.

Now let’s join peer0.org1.example.com to the channel.

*# By default, this joins ``peer0.org1.example.com`` only*

*# the <CHANNEL\_NAME.block> was returned by the previous command*

*# if you have not modified the channel name, you will join with mychannel.block*

*# if you have created a different channel name, then pass in the appropriately named block*

peer channel join **-**b mychannel**.**block

You can make other peers join the channel as necessary by making appropriate changes in the four environment variables we used in the [Create & Join Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html#peerenvvars) section, above.

Rather than join every peer, we will simply join peer0.org2.example.com so that we can properly update the anchor peer definitions in our channel. Since we are overriding the default environment variables baked into the CLI container, this full command will be the following:

CORE\_PEER\_MSPCONFIGPATH**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**users**/**Admin@org2**.**example**.**com**/**msp CORE\_PEER\_ADDRESS**=**peer0**.**org2**.**example**.**com:9051 CORE\_PEER\_LOCALMSPID**=**"Org2MSP" CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**peers**/**peer0**.**org2**.**example**.**com**/**tls**/**ca**.**crt peer channel join **-**b mychannel**.**block

**Note**

Prior to v1.4.1 all peers within the docker network used port 7051. If using a version of fabric-samples prior to v1.4.1, modify all occurrences of CORE\_PEER\_ADDRESS in this tutorial to use port 7051.

Alternatively, you could choose to set these environment variables individually rather than passing in the entire string. Once they’ve been set, you simply need to issue the peer channel join command again and the CLI container will act on behalf of peer0.org2.example.com.

**Update the anchor peers**

The following commands are channel updates and they will propagate to the definition of the channel. In essence, we adding additional configuration information on top of the channel’s genesis block. Note that we are not modifying the genesis block, but simply adding deltas into the chain that will define the anchor peers.

Update the channel definition to define the anchor peer for Org1 as peer0.org1.example.com:

peer channel update -o orderer.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/Org1MSPanchors.tx --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem

Now update the channel definition to define the anchor peer for Org2 as peer0.org2.example.com. Identically to the peer channel join command for the Org2 peer, we will need to preface this call with the appropriate environment variables.

CORE\_PEER\_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp CORE\_PEER\_ADDRESS=peer0.org2.example.com:9051 CORE\_PEER\_LOCALMSPID="Org2MSP" CORE\_PEER\_TLS\_ROOTCERT\_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt peer channel update -o orderer.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/Org2MSPanchors.tx --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem

**Install & Instantiate Chaincode**

**Note**

We will utilize a simple existing chaincode. To learn how to write your own chaincode, see the [Chaincode for Developers](https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4ade.html) tutorial.

Applications interact with the blockchain ledger through chaincode. As such we need to install the chaincode on every peer that will execute and endorse our transactions, and then instantiate the chaincode on the channel.

First, install the sample Go, Node.js or Java chaincode onto the peer0 node in Org1. These commands place the specified source code flavor onto our peer’s filesystem.

**Note**

You can only install one version of the source code per chaincode name and version. The source code exists on the peer’s file system in the context of chaincode name and version; it is language agnostic. Similarly the instantiated chaincode container will be reflective of whichever language has been installed on the peer.

**Golang**

*# this installs the Go chaincode. For go chaincode -p takes the relative path from $GOPATH/src*

peer chaincode install **-**n mycc **-**v 1.0 **-**p github**.**com**/**chaincode**/**chaincode\_example02**/**go**/**

**Node.js**

*# this installs the Node.js chaincode*

*# make note of the -l flag to indicate "node" chaincode*

*# for node chaincode -p takes the absolute path to the node.js chaincode*

peer chaincode install **-**n mycc **-**v 1.0 **-**l node **-**p **/**opt**/**gopath**/**src**/**github**.**com**/**chaincode**/**chaincode\_example02**/**node**/**

**Java**

*# make note of the -l flag to indicate "java" chaincode*

*# for java chaincode -p takes the absolute path to the java chaincode*

peer chaincode install **-**n mycc **-**v 1.0 **-**l java **-**p **/**opt**/**gopath**/**src**/**github**.**com**/**chaincode**/**chaincode\_example02**/**java**/**

When we instantiate the chaincode on the channel, the endorsement policy will be set to require endorsements from a peer in both Org1 and Org2. Therefore, we also need to install the chaincode on a peer in Org2.

Modify the following four environment variables to issue the install command against peer0 in Org2:

*# Environment variables for PEER0 in Org2*

CORE\_PEER\_MSPCONFIGPATH**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**users**/**Admin@org2**.**example**.**com**/**msp

CORE\_PEER\_ADDRESS**=**peer0**.**org2**.**example**.**com:9051

CORE\_PEER\_LOCALMSPID**=**"Org2MSP"

CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**peers**/**peer0**.**org2**.**example**.**com**/**tls**/**ca**.**crt

Now install the sample Go, Node.js or Java chaincode onto a peer0 in Org2. These commands place the specified source code flavor onto our peer’s filesystem.

**Golang**

*# this installs the Go chaincode. For go chaincode -p takes the relative path from $GOPATH/src*

peer chaincode install **-**n mycc **-**v 1.0 **-**p github**.**com**/**chaincode**/**chaincode\_example02**/**go**/**

**Node.js**

*# this installs the Node.js chaincode*

*# make note of the -l flag to indicate "node" chaincode*

*# for node chaincode -p takes the absolute path to the node.js chaincode*

peer chaincode install **-**n mycc **-**v 1.0 **-**l node **-**p **/**opt**/**gopath**/**src**/**github**.**com**/**chaincode**/**chaincode\_example02**/**node**/**

**Java**

*# make note of the -l flag to indicate "java" chaincode*

*# for java chaincode -p takes the absolute path to the java chaincode*

peer chaincode install **-**n mycc **-**v 1.0 **-**l java **-**p **/**opt**/**gopath**/**src**/**github**.**com**/**chaincode**/**chaincode\_example02**/**java**/**

Next, instantiate the chaincode on the channel. This will initialize the chaincode on the channel, set the endorsement policy for the chaincode, and launch a chaincode container for the targeted peer. Take note of the -P argument. This is our policy where we specify the required level of endorsement for a transaction against this chaincode to be validated.

In the command below you’ll notice that we specify our policy as -P "AND ('Org1MSP.peer','Org2MSP.peer')". This means that we need “endorsement” from a peer belonging to Org1 **AND** Org2 (i.e. two endorsement). If we changed the syntax to OR then we would need only one endorsement.

**Golang**

# be sure to replace the $CHANNEL\_NAME environment variable if you have not exported it

# if you did not install your chaincode with a name of mycc, then modify that argument as well

peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n mycc -v 1.0 -c '{"Args":["init","a", "100", "b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.peer')"

**Node.js**

**Note**

The instantiation of the Node.js chaincode will take roughly a minute. The command is not hanging; rather it is installing the fabric-shim layer as the image is being compiled.

# be sure to replace the $CHANNEL\_NAME environment variable if you have not exported it

# if you did not install your chaincode with a name of mycc, then modify that argument as well

# notice that we must pass the -l flag after the chaincode name to identify the language

peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n mycc -l node -v 1.0 -c '{"Args":["init","a", "100", "b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.peer')"

**Java**

**Note**

Please note, Java chaincode instantiation might take time as it compiles chaincode and downloads docker container with java environment.

peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n mycc -l java -v 1.0 -c '{"Args":["init","a", "100", "b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.peer')"

See the [endorsement policies](http://hyperledger-fabric.readthedocs.io/en/latest/endorsement-policies.html) documentation for more details on policy implementation.

If you want additional peers to interact with ledger, then you will need to join them to the channel, and install the same name, version and language of the chaincode source onto the appropriate peer’s filesystem. A chaincode container will be launched for each peer as soon as they try to interact with that specific chaincode. Again, be cognizant of the fact that the Node.js images will be slower to compile.

Once the chaincode has been instantiated on the channel, we can forgo the l flag. We need only pass in the channel identifier and name of the chaincode.

**Query**

Let’s query for the value of a to make sure the chaincode was properly instantiated and the state DB was populated. The syntax for query is as follows:

# be sure to set the -C and -n flags appropriately

peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}'

**Invoke**

Now let’s move 10 from a to b. This transaction will cut a new block and update the state DB. The syntax for invoke is as follows:

# be sure to set the -C and -n flags appropriately

peer chaincode invoke -o orderer.example.com:7050 --tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n mycc --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"Args":["invoke","a","b","10"]}'

**Query**

Let’s confirm that our previous invocation executed properly. We initialized the key a with a value of 100 and just removed 10 with our previous invocation. Therefore, a query against a should return 90. The syntax for query is as follows.

# be sure to set the -C and -n flags appropriately

peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}'

We should see the following:

Query Result: 90

Feel free to start over and manipulate the key value pairs and subsequent invocations.

**Install**

Now we will install the chaincode on a third peer, peer1 in Org2. Modify the following four environment variables to issue the install command against peer1 in Org2:

*# Environment variables for PEER1 in Org2*

CORE\_PEER\_MSPCONFIGPATH**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**users**/**Admin@org2**.**example**.**com**/**msp

CORE\_PEER\_ADDRESS**=**peer1**.**org2**.**example**.**com:10051

CORE\_PEER\_LOCALMSPID**=**"Org2MSP"

CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**peers**/**peer1**.**org2**.**example**.**com**/**tls**/**ca**.**crt

Now install the sample Go, Node.js or Java chaincode onto peer1 in Org2. These commands place the specified source code flavor onto our peer’s filesystem.

**Golang**

*# this installs the Go chaincode. For go chaincode -p takes the relative path from $GOPATH/src*

peer chaincode install **-**n mycc **-**v 1.0 **-**p github**.**com**/**chaincode**/**chaincode\_example02**/**go**/**

**Node.js**

*# this installs the Node.js chaincode*

*# make note of the -l flag to indicate "node" chaincode*

*# for node chaincode -p takes the absolute path to the node.js chaincode*

peer chaincode install **-**n mycc **-**v 1.0 **-**l node **-**p **/**opt**/**gopath**/**src**/**github**.**com**/**chaincode**/**chaincode\_example02**/**node**/**

**Java**

*# make note of the -l flag to indicate "java" chaincode*

*# for java chaincode -p takes the absolute path to the java chaincode*

peer chaincode install **-**n mycc **-**v 1.0 **-**l java **-**p **/**opt**/**gopath**/**src**/**github**.**com**/**chaincode**/**chaincode\_example02**/**java**/**

**Query**

Let’s confirm that we can issue the query to Peer1 in Org2. We initialized the key a with a value of 100 and just removed 10 with our previous invocation. Therefore, a query against a should still return 90.

peer1 in Org2 must first join the channel before it can respond to queries. The channel can be joined by issuing the following command:

CORE\_PEER\_MSPCONFIGPATH**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**users**/**Admin@org2**.**example**.**com**/**msp CORE\_PEER\_ADDRESS**=**peer1**.**org2**.**example**.**com:10051 CORE\_PEER\_LOCALMSPID**=**"Org2MSP" CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**peers**/**peer1**.**org2**.**example**.**com**/**tls**/**ca**.**crt peer channel join **-**b mychannel**.**block

After the join command returns, the query can be issued. The syntax for query is as follows.

# be sure to set the -C and -n flags appropriately

peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}'

We should see the following:

Query Result: 90

Feel free to start over and manipulate the key value pairs and subsequent invocations.

**What’s happening behind the scenes?**

**Note**

These steps describe the scenario in which script.sh is run by ‘./byfn.sh up’. Clean your network with ./byfn.sh down and ensure this command is active. Then use the same docker-compose prompt to launch your network again

* A script - script.sh - is baked inside the CLI container. The script drives the createChannel command against the supplied channel name and uses the channel.tx file for channel configuration.
* The output of createChannel is a genesis block - <your\_channel\_name>.block - which gets stored on the peers’ file systems and contains the channel configuration specified from channel.tx.
* The joinChannel command is exercised for all four peers, which takes as input the previously generated genesis block. This command instructs the peers to join <your\_channel\_name> and create a chain starting with <your\_channel\_name>.block.
* Now we have a channel consisting of four peers, and two organizations. This is our TwoOrgsChannel profile.
* peer0.org1.example.com and peer1.org1.example.com belong to Org1; peer0.org2.example.com and peer1.org2.example.com belong to Org2
* These relationships are defined through the crypto-config.yaml and the MSP path is specified in our docker compose.
* The anchor peers for Org1MSP (peer0.org1.example.com) and Org2MSP (peer0.org2.example.com) are then updated. We do this by passing the Org1MSPanchors.tx and Org2MSPanchors.tx artifacts to the ordering service along with the name of our channel.
* A chaincode - **chaincode\_example02** - is installed on peer0.org1.example.com and peer0.org2.example.com
* The chaincode is then “instantiated” on mychannel. Instantiation adds the chaincode to the channel, starts the container for the target peer, and initializes the key value pairs associated with the chaincode. The initial values for this example are [“a”,”100” “b”,”200”]. This “instantiation” results in a container by the name of dev-peer0.org2.example.com-mycc-1.0 starting.
* The instantiation also passes in an argument for the endorsement policy. The policy is defined as -P "AND ('Org1MSP.peer','Org2MSP.peer')", meaning that any transaction must be endorsed by a peer tied to Org1 and Org2.
* A query against the value of “a” is issued to peer0.org2.example.com. A container for Org2 peer0 by the name of dev-peer0.org2.example.com-mycc-1.0 was started when the chaincode was instantiated. The result of the query is returned. No write operations have occurred, so a query against “a” will still return a value of “100”.
* An invoke is sent to peer0.org1.example.com and peer0.org2.example.com to move “10” from “a” to “b”
* A query is sent to peer0.org2.example.com for the value of “a”. A value of 90 is returned, correctly reflecting the previous transaction during which the value for key “a” was modified by 10.
* The chaincode - **chaincode\_example02** - is installed on peer1.org2.example.com
* A query is sent to peer1.org2.example.com for the value of “a”. This starts a third chaincode container by the name of dev-peer1.org2.example.com-mycc-1.0. A value of 90 is returned, correctly reflecting the previous transaction during which the value for key “a” was modified by 10.

**What does this demonstrate?**

Chaincode **MUST** be installed on a peer in order for it to successfully perform read/write operations against the ledger. Furthermore, a chaincode container is not started for a peer until an init or traditional transaction - read/write - is performed against that chaincode (e.g. query for the value of “a”). The transaction causes the container to start. Also, all peers in a channel maintain an exact copy of the ledger which comprises the blockchain to store the immutable, sequenced record in blocks, as well as a state database to maintain a snapshot of the current state. This includes those peers that do not have chaincode installed on them (like peer1.org1.example.com in the above example) . Finally, the chaincode is accessible after it is installed (like peer1.org2.example.com in the above example) because it has already been instantiated.

**How do I see these transactions?**

Check the logs for the CLI Docker container.

docker logs **-**f cli

You should see the following output:

2017**-**05**-**16 17:08:01.366 UTC [msp] GetLocalMSP **->** DEBU 004 Returning existing local MSP

2017**-**05**-**16 17:08:01.366 UTC [msp] GetDefaultSigningIdentity **->** DEBU 005 Obtaining default signing identity

2017**-**05**-**16 17:08:01.366 UTC [msp**/**identity] Sign **->** DEBU 006 Sign: plaintext: 0AB1070A6708031A0C08F1E3ECC80510**...**6D7963631A0A0A0571756572790A0161

2017**-**05**-**16 17:08:01.367 UTC [msp**/**identity] Sign **->** DEBU 007 Sign: digest: E61DB37F4E8B0D32C9FE10E3936BA9B8CD278FAA1F3320B08712164248285C54

Query Result: 90

2017**-**05**-**16 17:08:15.158 UTC [main] main **->** INFO 008 Exiting**.....**

**=====================** Query successful on peer1**.**org2 on channel 'mychannel' **=====================**

**=====================** All GOOD, BYFN execution completed **=====================**

\_\_\_\_\_ \_ \_ \_\_\_\_

**|** \_\_\_\_**|** **|** \ **|** **|** **|** \_ \

**|** \_**|** **|** \**|** **|** **|** **|** **|** **|**

**|** **|**\_\_\_ **|** **|**\ **|** **|** **|**\_**|** **|**

**|**\_\_\_\_\_**|** **|**\_**|** \\_**|** **|**\_\_\_\_**/**

You can scroll through these logs to see the various transactions.

**How can I see the chaincode logs?**

Inspect the individual chaincode containers to see the separate transactions executed against each container. Here is the combined output from each container:

$ docker logs dev-peer0.org2.example.com-mycc-1.0

04:30:45.947 [BCCSP\_FACTORY] DEBU : Initialize BCCSP [SW]

ex02 Init

Aval = 100, Bval = 200

$ docker logs dev-peer0.org1.example.com-mycc-1.0

04:31:10.569 [BCCSP\_FACTORY] DEBU : Initialize BCCSP [SW]

ex02 Invoke

Query Response:{"Name":"a","Amount":"100"}

ex02 Invoke

Aval = 90, Bval = 210

$ docker logs dev-peer1.org2.example.com-mycc-1.0

04:31:30.420 [BCCSP\_FACTORY] DEBU : Initialize BCCSP [SW]

ex02 Invoke

Query Response:{"Name":"a","Amount":"90"}

**Understanding the Docker Compose topology**

The BYFN sample offers us two flavors of Docker Compose files, both of which are extended from the docker-compose-base.yaml (located in the base folder). Our first flavor, docker-compose-cli.yaml, provides us with a CLI container, along with an orderer, four peers. We use this file for the entirety of the instructions on this page.

**Note**

the remainder of this section covers a docker-compose file designed for the SDK. Refer to the [Node SDK](https://github.com/hyperledger/fabric-sdk-node) repo for details on running these tests.

The second flavor, docker-compose-e2e.yaml, is constructed to run end-to-end tests using the Node.js SDK. Aside from functioning with the SDK, its primary differentiation is that there are containers for the fabric-ca servers. As a result, we are able to send REST calls to the organizational CAs for user registration and enrollment.

If you want to use the docker-compose-e2e.yaml without first running the byfn.sh script, then we will need to make four slight modifications. We need to point to the private keys for our Organization’s CA’s. You can locate these values in your crypto-config folder. For example, to locate the private key for Org1 we would follow this path - crypto-config/peerOrganizations/org1.example.com/ca/. The private key is a long hash value followed by \_sk. The path for Org2 would be - crypto-config/peerOrganizations/org2.example.com/ca/.

In the docker-compose-e2e.yaml update the FABRIC\_CA\_SERVER\_TLS\_KEYFILE variable for ca0 and ca1. You also need to edit the path that is provided in the command to start the ca server. You are providing the same private key twice for each CA container.

**Using CouchDB**

The state database can be switched from the default (goleveldb) to CouchDB. The same chaincode functions are available with CouchDB, however, there is the added ability to perform rich and complex queries against the state database data content contingent upon the chaincode data being modeled as JSON.

To use CouchDB instead of the default database (goleveldb), follow the same procedures outlined earlier for generating the artifacts, except when starting the network pass docker-compose-couch.yaml as well:

docker**-**compose **-**f docker**-**compose**-**cli**.**yaml **-**f docker**-**compose**-**couch**.**yaml up **-**d

**chaincode\_example02** should now work using CouchDB underneath.

**Note**

If you choose to implement mapping of the fabric-couchdb container port to a host port, please make sure you are aware of the security implications. Mapping of the port in a development environment makes the CouchDB REST API available, and allows the visualization of the database via the CouchDB web interface (Fauxton). Production environments would likely refrain from implementing port mapping in order to restrict outside access to the CouchDB containers.

You can use **chaincode\_example02** chaincode against the CouchDB state database using the steps outlined above, however in order to exercise the CouchDB query capabilities you will need to use a chaincode that has data modeled as JSON, (e.g. **marbles02**). You can locate the **marbles02** chaincode in the fabric/examples/chaincode/go directory.

We will follow the same process to create and join the channel as outlined in the createandjoin section above. Once you have joined your peer(s) to the channel, use the following steps to interact with the **marbles02** chaincode:

* Install and instantiate the chaincode on peer0.org1.example.com:

# be sure to modify the $CHANNEL\_NAME variable accordingly for the instantiate command

peer chaincode install -n marbles -v 1.0 -p github.com/chaincode/marbles02/go

peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n marbles -v 1.0 -c '{"Args":["init"]}' -P "OR ('Org1MSP.peer','Org2MSP.peer')"

* Create some marbles and move them around:

# be sure to modify the $CHANNEL\_NAME variable accordingly

peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n marbles -c '{"Args":["initMarble","marble1","blue","35","tom"]}'

peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n marbles -c '{"Args":["initMarble","marble2","red","50","tom"]}'

peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n marbles -c '{"Args":["initMarble","marble3","blue","70","tom"]}'

peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n marbles -c '{"Args":["transferMarble","marble2","jerry"]}'

peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n marbles -c '{"Args":["transferMarblesBasedOnColor","blue","jerry"]}'

peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n marbles -c '{"Args":["delete","marble1"]}'

* If you chose to map the CouchDB ports in docker-compose, you can now view the state database through the CouchDB web interface (Fauxton) by opening a browser and navigating to the following URL:

http://localhost:5984/\_utils

You should see a database named mychannel (or your unique channel name) and the documents inside it.

**Note**

For the below commands, be sure to update the $CHANNEL\_NAME variable appropriately.

You can run regular queries from the CLI (e.g. reading marble2):

peer chaincode query -C $CHANNEL\_NAME -n marbles -c '{"Args":["readMarble","marble2"]}'

The output should display the details of marble2:

Query Result: {"color":"red","docType":"marble","name":"marble2","owner":"jerry","size":50}

You can retrieve the history of a specific marble - e.g. marble1:

peer chaincode query -C $CHANNEL\_NAME -n marbles -c '{"Args":["getHistoryForMarble","marble1"]}'

The output should display the transactions on marble1:

Query Result: [{"TxId":"1c3d3caf124c89f91a4c0f353723ac736c58155325f02890adebaa15e16e6464", "Value":{"docType":"marble","name":"marble1","color":"blue","size":35,"owner":"tom"}},{"TxId":"755d55c281889eaeebf405586f9e25d71d36eb3d35420af833a20a2f53a3eefd", "Value":{"docType":"marble","name":"marble1","color":"blue","size":35,"owner":"jerry"}},{"TxId":"819451032d813dde6247f85e56a89262555e04f14788ee33e28b232eef36d98f", "Value":}]

You can also perform rich queries on the data content, such as querying marble fields by owner jerry:

peer chaincode query -C $CHANNEL\_NAME -n marbles -c '{"Args":["queryMarblesByOwner","jerry"]}'

The output should display the two marbles owned by jerry:

Query Result: [{"Key":"marble2", "Record":{"color":"red","docType":"marble","name":"marble2","owner":"jerry","size":50}},{"Key":"marble3", "Record":{"color":"blue","docType":"marble","name":"marble3","owner":"jerry","size":70}}]

**Why CouchDB**

CouchDB is a kind of NoSQL solution. It is a document-oriented database where document fields are stored as key-value maps. Fields can be either a simple key-value pair, list, or map. In addition to keyed/composite-key/key-range queries which are supported by LevelDB, CouchDB also supports full data rich queries capability, such as non-key queries against the whole blockchain data, since its data content is stored in JSON format and fully queryable. Therefore, CouchDB can meet chaincode, auditing, reporting requirements for many use cases that not supported by LevelDB.

CouchDB can also enhance the security for compliance and data protection in the blockchain. As it is able to implement field-level security through the filtering and masking of individual attributes within a transaction, and only authorizing the read-only permission if needed.

In addition, CouchDB falls into the AP-type (Availability and Partition Tolerance) of the CAP theorem. It uses a master-master replication model with Eventual Consistency. More information can be found on the [Eventual Consistency page of the CouchDB documentation](http://docs.couchdb.org/en/latest/intro/consistency.html). However, under each fabric peer, there is no database replicas, writes to database are guaranteed consistent and durable (not Eventual Consistency).

CouchDB is the first external pluggable state database for Fabric, and there could and should be other external database options. For example, IBM enables the relational database for its blockchain. And the CP-type (Consistency and Partition Tolerance) databases may also in need, so as to enable data consistency without application level guarantee.

**A Note on Data Persistence**

If data persistence is desired on the peer container or the CouchDB container, one option is to mount a directory in the docker-host into a relevant directory in the container. For example, you may add the following two lines in the peer container specification in the docker-compose-base.yaml file:

volumes:

**-** **/**var**/**hyperledger**/**peer0:**/**var**/**hyperledger**/**production

For the CouchDB container, you may add the following two lines in the CouchDB container specification:

volumes:

**-** **/**var**/**hyperledger**/**couchdb0:**/**opt**/**couchdb**/**data

**Troubleshooting**

* Always start your network fresh. Use the following command to remove artifacts, crypto, containers and chaincode images:
* **./**byfn**.**sh down

**Note**

You **will** see errors if you do not remove old containers and images.

* If you see Docker errors, first check your docker version ([Prerequisites](https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.html)), and then try restarting your Docker process. Problems with Docker are oftentimes not immediately recognizable. For example, you may see errors resulting from an inability to access crypto material mounted within a container.

If they persist remove your images and start from scratch:

docker rm -f $(docker ps -aq)

docker rmi -f $(docker images -q)

* If you see errors on your create, instantiate, invoke or query commands, make sure you have properly updated the channel name and chaincode name. There are placeholder values in the supplied sample commands.
* If you see the below error:
* Error: Error endorsing chaincode: rpc error: code **=** 2 desc **=** Error installing chaincode code mycc:1.0(chaincode **/**var**/**hyperledger**/**production**/**chaincodes**/**mycc**.**1.0 exits)

You likely have chaincode images (e.g. dev-peer1.org2.example.com-mycc-1.0 or dev-peer0.org1.example.com-mycc-1.0) from prior runs. Remove them and try again.

docker rmi -f $(docker images | grep peer[0-9]-peer[0-9] | awk '{print $3}')

* If you see something similar to the following:
* Error connecting: rpc error: code **=** 14 desc **=** grpc: RPC failed fast due to transport failure
* Error: rpc error: code **=** 14 desc **=** grpc: RPC failed fast due to transport failure

Make sure you are running your network against the “1.0.0” images that have been retagged as “latest”.

* If you see the below error:
* [configtx**/**tool**/**localconfig] Load **->** CRIT 002 Error reading configuration: Unsupported Config Type ""
* panic: Error reading configuration: Unsupported Config Type ""

Then you did not set the FABRIC\_CFG\_PATH environment variable properly. The configtxgen tool needs this variable in order to locate the configtx.yaml. Go back and execute an export FABRIC\_CFG\_PATH=$PWD, then recreate your channel artifacts.

* To cleanup the network, use the down option:
* **./**byfn**.**sh down
* If you see an error stating that you still have “active endpoints”, then prune your Docker networks. This will wipe your previous networks and start you with a fresh environment:
* docker network prune

You will see the following message:

WARNING! This will remove all networks not used by at least one container.

Are you sure you want to continue? [y/N]

Select y.

* If you see an error similar to the following:
* **/**bin**/**bash: **./**scripts**/**script**.**sh: **/**bin**/**bash**^**M: bad interpreter: No such file **or** directory

Ensure that the file in question (**script.sh** in this example) is encoded in the Unix format. This was most likely caused by not setting core.autocrlf to false in your Git configuration (see [Windows extras](https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.html#windows-extras)). There are several ways of fixing this. If you have access to the vim editor for instance, open the file:

vim **./**fabric**-**samples**/**first**-**network**/**scripts**/**script**.**sh

Then change its format by executing the following vim command:

:set ff**=**unix

**Note**

If you continue to see errors, share your logs on the **fabric-questions** channel on [Hyperledger Rocket Chat](https://chat.hyperledger.org/home) or on [StackOverflow](https://stackoverflow.com/questions/tagged/hyperledger-fabric).