**최초 네트워크 만들기**

**Note**

여기에 있는 명령어들은 최신의 안정적인 도커 이미지들과 제공된 tar 파일 에서 사전에 컴파일된 설치 유틸리티들을 대상으로 동작함을 검증하였다. 검증되지 않은 최신의 도커 이미지들이나 도구들에서 이 명령어들을 실행하는 경우, 구성 오류와 심각한 오류가 나타날 수 있다.

최초 네트워크 만들기(BYFN, Build your first network) 시나리오는 두 개의 피어 노드들을 가진 두 개의 조직들로 구성된 하이퍼레저 패브릭 네트워크 예제를 제공한다. 또한 다른 오더링 서비스 구현이 가능하지만, 기본적으로 “단일한(Solo)” 오더링 서비스를 적용한다.

**설치 사전 요구 사항**

시작하기 전에, 이전에 설치하지 않았다면, 블록체인 애플리케이션을 개발하거나 하이퍼레저 패브릭을 운영하려는 플랫폼에 따라 사전 요구 사항( [Prerequisites](https://hyperledger-fabric.readthedocs.io/en/latest/prereqs.html) )이 설치되었는지 확인하여야 한다.

또한 예제와 실행 파일, 도커 이미지를 설치할 필요가 있을 수 있다.  fabric-samples 레파지토리에 포함된 많은 예제들이 있을 알 수 있다. 여기서는 first-network 예제를 사용한다. 이제 하위 디렉토리를 열어보자.

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

**Note**

본 문서에서 제공된 명령어들은 fabric-samples 레파지토리 복제본의 first-network 하위 디렉토리에서 실행해야 한다. 다른 위치에서 명령어를 실행하는 경우, 제공된 다양한 스크립트에서 실행 파일을 찾지 못할 수 있다.

**이제 실행하기를 원하는가?**

기본적으로 서로 다른 두 조직들을 이루는 4개의 피어들과 하나의 오더러 노드로 구성된 하이퍼레저 패브릭 네트워크를 빠르게 적재할 수 있는 도커 이미지들을 만드는 완벽하게 설명이 달린 스크립트 — byfn.sh — 를 제공한다.

다음은 byfn.sh 스크립트를 위한 도움말이다.:

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"

플래그를 제공하지 않은 경우, 스크립트는 기본 값들을 사용한다.

**네트워크 구조(Network Artifacts) 생성하기**

시작할 준비가 되었는가? 그럼, 다음 명령어를 실행하자:

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

Yes/No 명령 프롬프트와 함께 어떤 일이 벌어지고 있는지 간단한 설명을 볼 수 있다. y 를 누르거나 Enter 키를 누르면 설명된 동작을 실행한다.

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

이 첫번째 단계는 다양한 네트워크 개체들을 위한 인증서들과 키들 모두와, 오더링 서비스를 시작하는데 사용할 genesis block  과, 채널을 구성하는데 필요한 구성 트랜잭션 집합을 만든다.

**네트워크 실행하기**

다음으로, 다음 명령어들 중 하나를 사용하여 네트워크를 실행할 수 있다.:

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

위 명령어는 Golang 체인코드 이미지들을 컴파일하고 관련 컨테이너들을 돌린다. Go는 기본 체인코드 언어이지만, Node.js와 자바 체인코드 또한 지원한다. 노드 체인코드로 이 교재를 따라 하려는 경우, 대신 아래 명령어를 실행하라.:

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

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

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

**Note**

Node.js 체인코드에 대한 보다 상세한 정보는 … 를 참고한다.

**Note**

자바 체인코드에 대한 보다 상세한 정보는 … 를 참고한다.

자바 체인코드를 예제 실행을 하려면, 다음과 같이 -l java 를 지정하여야 한다.:

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

**Note**

이 명령어들 둘 다 실행하지는 않아야 한다. 네트워크를 종료하고 다시 만들지 않는 한 오직 하나의 언어만 시도할 수 있다.

여러 체인코드 언어들을 지원하려는 경우 한 노드 Solo 오더러 대신 5개의 노드 Raft 오더링 서비스 또는 하나의 Kafka 오더링 서비스를 실행하는 플래그를 적용할 수 있다. 현재 지원되는 오더링 서비스 적용에 대한 추가적인 정보는 “오더링 서비스”를 확인한다.

Raft 오더링 서비스로 네트워크를 실행하려면, 다음 명령어를 실행한다.:

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

Kafka 오더링 서비스로 네트워크를 실행하려면, 다음 명령어를 실행한다.:

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

다시 한번, 계속 진행할지 중단할지를 묻는 프롬프트가 나타난다. y 를 누르거나 엔터 키를 누른다.:

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...

로그가 계속 나온다. 이제 모든 컨테이너들을 실행하고, 그러고 나서 전체 애플리케이션 시나리오의 처음부터 끝까지 돌아간다. 성공적으로 완료되면, 터미널 창에 다음과 같이 나타난다.:

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 **=====================**

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

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

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

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

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

이 로그들을 따라서 다양한 트랜잭션들을 살펴볼 수 있다. 이 결과가 나오지 않는다면, “문제 해결” 부분으로 넘어가서 무엇이 잘못되었는지 찾을 수 있도록 도와줄 수 있는지를 살펴본다.

**네트워크 종료하기**

마지막으로 모두 종료해서 한번에 한 단계씩 네트워크 설정을 살펴보자. 다음은 컨테이너를 종료하고, 암호 자료와 4개의 문서들을 제거하고, 도커 레지스트리(Docker Registry)에서 체인코드 이미지들을 삭제한다.:

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

다시 한번, 계속할지를 묻는 프롬프트가 나타나고, y 키를 누르거나 엔터 키를 눌른다.:

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

...

기반이 되는 도구들과 실행 체계에 대해 더 배우고 싶다면, 아래 내용을 계속 읽어본다. 다음 부분에서는 완벽하게 동작하는 하이퍼레저 패브릭 네트워크를 만드는 다양한 단계들과 요구 사항들을 살펴본다.

**Note**

아래 표시된 수작업 단계들은 cli 컨테이너에서 FABRIC\_LOGGING\_SPEC 이 DEBUG 로 설정되어 있음을 가정한다. 예를 들어, first-network 디렉토리에 있는 docker-compose-cli.yaml 파일을 수정해서 이 값을 설정할 수 있다.

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

**암호 생성기(Crypto Generator)**

다양한 네트워크 개체들을 위한 암호 자료(x509 인증서와 서명 키)를 생성하기 위하여 cryptogen  도구를 사용한다. 이 인증서들은 신분을 나타내고, 개체들이 서로 통신하고 거래함에 따라 발생하는 인증을 서명하거나 검증할 수 있게 한다.

**암호 생성기 동작 방법**

Cryptogen은 네트워크 배치 정보를 가지고 조직들과 그 조직들에 속한 구성 요소들 모두를 위한 인증서와 키 집합을 생성할 수 있도록 하는 crypto-config.yaml 파일을 사용한다. 각 조직(Organization)은 특정 구성 요소들(피어들과 오더러들)을 그 조직에 묶어주는 고유한 루트 인증서(ca-cert)가 제공된다. 각 조직에 고유한 CA 인증서를 지정함으로써, 참여하는 멤버가 그 자신의 인증 기관(Certificate Authority)을 사용하는 일반적인 네트워크를 흉내낸다. 하이퍼레저 패브릭 내에서의 거래와 통신은 한 개체의 개인 키(keystore)로 서명되고, 공용 키(signcerts)를 통하여 검증된다.

이 파일에는 count 변수가 있다. 이 변수는 조직 당 피어 개수를 지정하기 위해 사용한다. 여기서는 조직 당 2개 피어가 있다. 여기서 바로 x.509 인증서와 공용 키 인프라구조의 세부사항들([x.509 certificates and public key infrastructure](https://en.wikipedia.org/wiki/Public_key_infrastructure) )로 들어가지는 않는다. 관심이 있다면, 시간이 되는대로 해당 주제들을 살펴보도록 한다.

cryptogen 도구를 실행한 후, 생성된 인증서들과 키들은 crypto-config 라는 폴더에 저장된다. crypto-config.yaml  파일은 오더러 조직에 묶인 5개의 오더러들을 나열하고 있음을 살펴보자. Note that the crypto-config.yaml file lists five orderers as being tied to the orderer organization.  cryptogen  도구가 모두 5개의 오더러들을 위한 인증서들을 만든 반면, Raft나 Kafka 오더링 서비스를 사용하고 있지 않다면 이 오더러들 중 단 하나만 Solo 오더링 서비스 구현에 사용되어 시스템 채널과  mychannel 을 만드는데 사용된다.

**구성 트랜잭션 생성기**

configtxgen 도구는 다음과 같이 4 개의 구성 문서를 생성하는데 사용된다.:

* 오더러 genesis block,
* 채널 configuration transaction,
* anchor peer transactions 두 개 – 각 피어 조직 마다 하나씩

이 도구의 기능에 대한 상세한 설명은 [configtxgen](https://hyperledger-fabric.readthedocs.io/en/latest/commands/configtxgen.html) 을 참고한다.

오더러 블록은 오더링 서비스를 제네시스 블록([Genesis Block](https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html#genesis-block))이고, 채널 구성 트랜잭션 파일은 채널을 만들 때 오더러에게 전파된다. 앵커 피어 트랜잭션은 그 이름에서 의미하듯이 이 채널([Channel](https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html#channel))에 있는 각 조직의 앵커 피어([Anchor Peer](https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html#anchor-peer))를 지정한다.

**구성 트랜잭션 동작 방법**

Configtxgen 은 예제 네트워크를 위한 정의들을 가진 configtx.yaml 파일을 사용한다. 여기에는 하나의 오더러 조직(Orderer Org, OrdererOrg)과 두 피어 노드를 각각 관리하고 유지하는 두 개의 피어 조직(Peer Orgs, Org1 & Org2), 이렇게 3개의 구성 요소가 있다.이 파일은 또한 두 개의 피어 조직으로 구성된 SampleConsortium 라는 콘소시움을 지정한다. 이 파일의 끝 부분에 있는 “Profiles” 부분에 특별히 주의해야 한다. 여러 개의 고유한 프로파일들이 있음을 알 수 있다. 다음 프로파일들은 살펴볼 필요가 있다.:

* TwoOrgsOrdererGenesis: Solo 오더링 서비스를 위한 제네시스 블록을 생성한다.
* SampleMultiNodeEtcdRaft: Raft 오더링 서비스를 위한 제네시스 블록을 생성한다. -o  플래그를 붙이고  etcdraft 를 지정한 경우에만 사용된다.
* SampleDevModeKafka: Kafka 오더링 서비스를 위한 제네시스 블록을 생성한다. -o 를 붙이고 kafka 를 지정한 경우에만 사용된다.
* TwoOrgsChannel:  여기서 사용할 채널, mychannel 을 위한 제네시스 블록을 생성한다.

이 헤더들은 문서를 만들 때 매개변수들로 전달되기 때문에 중요하다.

**Note**

여기서 사용하는 SampleConsortium 는 시스템 수준의 프로파일로 정의되고 채널 수준의 프로파일에서 참조됨을 알아야 한다. 채널들은 하나의 콘소시움 범위 내에 존재하고, 모든 콘소시움들은 대체적으로 네트워크 범위 내에서 정의되어야 한다.

이 파일은 또한 살펴볼 필요가 있는 두 가지 추가적인 내역을 가지고 있다. 첫번째로 각 피어 조직을 위한 앵커 피어들을 지정한다. (peer0.org1.example.com & peer0.org2.example.com). 두번째로 각 구성원을 위한 MSP 디렉토리의 위치를 지정하여 오더러 제네시스 블록에 각 조직을 위한 루트 인증서들을 저장할 수 있도록 한다. 이것은 매우 중요한 개념이다. 이제 오더링 서비스와 통신하는 네트워크 개체가 자체적인 디지털 서명을 검증한다.

**도구들 실행하기**

configtxgen 와 cryptogen 명령어들을 사용하여 인증서들/키들과 다양한 구성 문서를 수작업으로 생성할 수 있다. 그렇지 않으면 자신의 목표를 달성할 수 있도록 byfn.sh 스크립트를 적용을 시도해볼 수 있다.

**수작업으로 문서 생성하기**

 crypto-config.yaml 파일에 정의된 것처럼 네트워크 구성을 위해 사용될 인증서들을 생성하기 위해 필요한 명령어들은 byfn.sh 스크립트에 있는  generateCerts  기능을 참고할 수 있다. 그러나, 편의상, 또한 여기에 참조를 제공한다.

먼저, cryptogen 도구를 실행한다. 실행 파일은 bin 디렉토리에 있으므로, 도구가 있는 곳을 지정하는 상대 경로를 제공할 필요가 있다.

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

터미널 창에서 다음 목록을 볼 수 있다.:

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

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

인증서들과 키들(즉, MSP 자료)는 first-network  디렉토리의 루트에 있는 crypto-config  디렉토리로 출력된다.

다음으로 구성값을 가져오는데 필요한 configtx.yaml 파일을 찾을 위치를 configtxgen 도구에 알려줄 필요가 있다. 현재 작업 중인 디렉토리를 살펴보도록 지정한다.:

export FABRIC\_CFG\_PATH=$PWD

그리고나서,  configtxgen 도구를 실행하여 오더러 제네시스 블록을 만든다.:

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

Raft 오더링 서비스를 위한 제네시스 블록을 만들려면, 다음 명령어와 같아야 한다.:

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

여기에 사용된 SampleMultiNodeEtcdRaft 프로파일을 살펴보자.

Kafka 오더링 서비스를 위한 제네시스 블록을 만들려면, 다음 명령어를 실행한다.:

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

Raft나 Kafka를 사용하고 있지 않은 경우, 결과는 다음과 같이 나타나야 한다.:

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**

만들려고 하는 오더러 제네시스 블록과 결과적인 문서들은 본 프로젝트의 루트에 있는 channel-artifacts 디렉토리에 출력된다. 위 명령어에 있는 channelID는 시스템 채널의 이름이다.

**채널 구성 트랜잭션 만들기**

다음으로, 채널 트랜잭션 문서를 만들 필요가 있다. $CHANNEL\_NAME 값을 바꾸거나 CHANNEL\_NAME 를 설정하여 이 명령을 통해 사용될 수 있는 환경 변수로 지정해야 한다.:

# 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

Raft나 Kafka 오더링 서비스를 사용하고 있는 경우 채널을 위한 특수한 명령어를 실행하지 않아야 한다. TwoOrgsChannel 프로파일은 네트워크를 위한 제네시스 블록을 만들 때 지정한 오더링 서비스 구성을 사용한다.

Raft나 Kafka 오더링 서비스를 사용하고 있지 않은 경우, 결과는 다음과 같이 나타나야 한다.:

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

다음으로, 만들고 있는 채널 상에 Org1을 위한 앵커 피어를 정의한다. 또한 다음 명령어들로  $CHANNEL\_NAME 값을 교체하거나 환경 변수를 설정하여야 한다. 터미널 출력은 채널 트랜잭션 문서의 결과와 비슷하다.:

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

이제 동일한 채널 상에 Org2를 위한 앵커 피어를 정의한다.:

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

**네트워크 시작하기**

**Note**

앞서 byfn.sh 예제를 실행하였다면, 계속 진행하기 전에 테스트 네트워크를 종료하여야 한다( [Bring Down the Network](https://hyperledger-fabric.readthedocs.io/en/latest/build_network.html#bring-down-the-network) 참조).

네트워크를 시작하기 위해 스크립트를 활용한다. Docker-compose 파일은 앞서 내려받은 이미지들을 참조하고, 앞서 생성된  genesis.block 을 가지고 오더러를 시작한다.

각 호출의 구문과 기능을 보여주기 위해 수작업으로 명령어들을 진행하고자 한다.

먼저, 네트워크를 시작한다.:

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

네트워크에 대한 실시간 로그를 보고자 하는 경우, -d 플래그를 포함하지 않아야 한다. 로그를 기록하도록 한 경우, CLI 호출을 실행하기 위해 다른 터미널 창을 열 필요가 있다.

**채널 만들어 참여하기**

앞서 [Create a Channel Configuration Transaction](https://hyperledger-fabric.readthedocs.io/en/latest/build_network.html#createchanneltx) 에서  configtxgen 도구를 사용하여 채널 구성 트랜잭션을 생성하였음을 되살려보자. configtxgen 도구에 전달한 configtx.yaml 파일에 있는 동일하거나 다른 프로파일을 사용하여 추가적인 채널 구성 트랜잭션을 만들기 위해 그 과정을 반복할 수 있다. 그리고나서 네트워크에 다른 채널을 설정하기 위해 여기서 정의한 과정을 반복할 수 있다.

docker exec 명령어를 사용하여 CLI 컨테이너에 들어간다.:

docker exec **-**it cli bash

성공하였다면, 다음과 같은 화면이 나타난다.:

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

작업할 peer0.org1.example.com  에 여기에 나올 CLI 명령어들을 실행하기 위하여 아래 주어진 4개의 환경 변수들을 명령어들 보다 먼저 다룬다.  peer0.org1.example.com  을 위한 이 변수들은 CLI 컨테이너에 묶여져 있어서, 변수들을 전달하지 않고서 작업할 수 있다. 그러나, 다른 피어들이나 오더러에 호출을 전달하려는 경우,  CLI 컨테이너 기본값이 peer0.org1.example.com 은 대상으로 하도록 유지하여야 하고, 그렇지만 CLI 호출을 하려고 할 때 아래에 있는 예제에서 처럼 환경 변수들을 덮어써야 한다.:

*# 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

다음으로, [Create a Channel Configuration Transaction](https://hyperledger-fabric.readthedocs.io/en/latest/build_network.html#createchanneltx)  에서 만든 생성된 채널 구성 트랜잭션 문서(channel.tx 라고 부른다)에서 채널 생성 요청의 일부로써 오더러에게 전달하려고 한다.

 -c 플래그로 채널 이름을 지정하고, -f  플래그로 채널 구성 트랜잭션을 지정한다. 이 경우, 채널 구성 트랜잭션은  channel.tx 이다. 그러나 다른 이름으로 자신의 구성 트랜잭션을 올릴 수 있다. 다시, CLI 컨테이너 내에  CHANNEL\_NAME 환경 변수를 설정하여 이 매개변수를 명시적으로 전달하지 않아야 한다. 채널 이름들은 모두 소문자로, 최대 250 문자 보다 적어야 하고, 일반적인 [a-z][a-z0-9.-]\* 표현에 맞아야 한다.

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**

이 명령어의 일부로써 전달하는 --cafile을 살펴보자. --cafile 는 오더러의 루트 인증에 대한 경로로, TLS 교환을 검증할 수 있게 한다.

이 명령어는 채널에 참여하는데 사용할  <CHANNEL\_NAME.block> 제네시스 블록을 돌려준다. 여기에는  channel.tx 에서 지정한 구성 정보를 가지고 있다. 기본 채널 이름에서 수정하지 않은 경우, 이 명령어는 mychannel.block 라는 원래 이름을 돌려준다.

**Note**

나머지 수작업 명령어들을 CLI 컨테이너에 남겨둘 수 있다. 또한 모든 명령어들 전에 peer0.org1.example.com 가 아닌 다른 피어를 대상으로 할 때는 모든 명령어들 이전에 관련 환경 변수를 작업해야 함을 또한 기억하여야 한다.

이제 peer0.org1.example.com 을 채널에 참여시키자.

*# 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

앞서 “채널생성하고 참여하기”에서 사용한 4개의 환경 변수들에 적절한 변경을 하여 필요에 따라 채널에 다른 피어들을 참여시킬 수 있다.

모든 피어들을 참여시키지는 않고,  peer0.org2.example.com 를 단순하게 참여시켜 채널에 있는 앵커 피어 정의들을 적절히 업데이트할 수 있다. CLI 컨테이너 내로 묶여진 기본 환경 변수들을 덮어쓰고 있기 때문에, 이 전체 명령어는 다음과 같다.:

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

아니면, 전체 문자열로 전달하기 보다는 개별적으로 이 환경 변수들을 지정하도록 선택할 수 있다. 일단 설정되면, 단순히 다시 peer channel join  명령어를 실행할 필요가 있고, CLI 컨테이너는 peer0.org2.example.com 을 대신하여 동작한다.

**앵커 피어 갱신하기**

다음 명령어들은 채널 갱신으로, 채널의 정의로 전파된다. 기본적으로 채널의 제네시스 블록 위에 추가적인 구성 정보를 추가한다. 제네시스 블록을 수정하고 있는 것이 아니라 단순히 앵커 피어를 정의할 체인 내에 변경을 추가한 것임에 주목하여야 한다.

peer0.org1.example.com 처럼 Org1을 위한 앵커 피어를 정의하기 위해 채널 정의를 갱신한다.:

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

이제 peer0.org2.example.com 처럼 Org2를 위한 앵커 피어를 정의하기 위해 채널 정의를 갱신한다. Org2 피어를 위한  peer channel join 명령어와 동일하게, 적절한 환경 변수들을 이 명령어 전에 설정할 필요가 있다.

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

**체인코드 설치하고 정의하기**

**Note**

간단한 기존 체인코드를 사용한다. 자신의 체인코드를 작성하는 방법을 배우려면, [Chaincode for Developers](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4ade.html)  내용을 살펴본다.

**Note**

여기의 지침들은 v2.0 알파 릴리즈에서 소개된 패브릭 체인코드 수명주기를 사용한다. 체인코드를 설치하고 인스턴스화하기 위해 기존 수명주기를 사용하려는 경우, [Building your first network tutorial](https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html) 의 v1.4 버전을 참조한다.

애플리케이션들은  chaincode 을 통해 블록체인 원장과 상호작용한다. 따라서 트랜잭션을 실행하고 기입할 모든 피어에 체인코드를 설치할 필요가 있다. 그러나 체인코드와 상호작용하기 전에 채널의 구성원들은 체인코드 거버넌스를 설정한 체인코드 정의에 동의할 필요가 있다.

피어들에 설치될 수 있기 전에 체인코드를 패키지로 묶을 필요가 있다. 만든 각 패키지에 대하여, 체인코드의 설명으로 체인코드 패키지 이름을 제공할 필요가 있다. Go 또는 Node.js, 자바 예제 체인코드를 패키지로 묶기 위하여 다음 명령어들을 사용한다.

**Golang**

*# this packages a Golang chaincode.*

*# make note of the --lang flag to indicate "golang" chaincode*

*# for go chaincode --path takes the relative path from $GOPATH/src*

*# The --label flag is used to create the package label*

peer lifecycle chaincode package mycc**.**tar**.**gz **--**path github**.**com**/**hyperledger**/**fabric**-**samples**/**chaincode**/**abstore**/**go**/** **--**lang golang **--**label mycc\_1

**Node.js**

*# this packages a Node.js chaincode*

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

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

*# The --label flag is used to create the package label*

peer lifecycle chaincode package mycc**.**tar**.**gz **--**path **/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**-**samples**/**chaincode**/**abstore**/**node**/** **--**lang node **--**label mycc\_1

**Java**

*# this packages a java chaincode*

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

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

*# The --label flag is used to create the package label*

peer lifecycle chaincode package mycc**.**tar**.**gz **--**path **/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**-**samples**/**chaincode**/**abstore**/**java**/** **--**lang java **--**label mycc\_1

위에 있는 명령어들 각각은 [``](https://hyperledger-fabric.readthedocs.io/en/latest/build_network.html#id3)mycc.tar.gz` 라는 이름의 체인코드 패키지를 만들고, 피어들 상에 체인코드를 설치하기 위해 사용할 수 있다. Org1의 peer0 상에 패키지를 설치하기 위하여 다음 명령어들을 실행시킨다.

*# this command installs a chaincode package on your peer*

peer lifecycle chaincode install mycc**.**tar**.**gz

install 명령어가 성공하면 체인코드 패키지 인식번호가 반환된다. 다음과 비슷한 결과를 볼 수 있어야 한다.:

2019**-**03**-**13 13:48:53.691 UTC [cli**.**lifecycle**.**chaincode] submitInstallProposal **->** INFO 001 Installed remotely: response:**<**status:200 payload:"\nEmycc\_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173" **>**

2019**-**03**-**13 13:48:53.691 UTC [cli**.**lifecycle**.**chaincode] submitInstallProposal **->** INFO 002 Chaincode code package identifier: mycc\_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173

또한 설치한 패키지에 대한 정보를 피어에 질의를 보내 체인코드 패키지 인식번호를 찾을 수 있다.

*# this returns the details of the packages installed on your peers*

peer lifecycle chaincode queryinstalled

위 명령어는 install 명령어처럼 동일한 패키지 인식번호를 반환한다. 다음과 비슷한 결과를 볼 수 있어야 한다.:

Get installed chaincodes on peer:

Package ID: mycc\_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173, Label: mycc\_1

이후의 명령어를 위해 패키지 ID가 필요하므로, 더 나아가 패키지 ID를 환경 변수로 저장하자. 패키지 ID가 나중에 사용할 명령어에 필요하므로, 더 나아가 패키지 ID를 환경 변수처럼 저장한다. peer lifecycle chaincode queryinstalled 명령어에 의해 나온 패키지 ID를 아래와 같이 명령창에 붙여넣기 한다. 패키지 ID가 모든 사용자들에게 동일하지는 않을테니, 자신의 콘솔에서 나온 패키지 ID를 사용하여 이 단계를 완료할 필요가 있다.

*# Save the package ID as an environment variable.*

CC\_PACKAGE\_ID**=**mycc\_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173

Org1와 Org2 모두에 있는 한 피어로부터 보증을 필요로 할 수 있도록 mycc 의 보증(Endorsement) 정책을 설정해야 한다. 따라서 Org2에 있는 한 피어에 체인코드를 또한 설치할 필요가 있다.

다음 4개 환경 변수들을 수정하여 Org2에 install 명령을 실행하도록 한다.:

*# 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

이제 Org2의 peer0 상에 체인코드 패키지를 설치한다. 다음 명령어는 Org1에서 실행한 install 명령어처럼 체인코드를 설치하고 동일한 인식번호를 반환한다.

*# this installs a chaincode package on your peer*

peer lifecycle chaincode install mycc**.**tar**.**gz

패키지를 설치한 후, 자신의 조직을 위한 체인코드 정의를 승인할 필요가 있다. 체인코드 정의(chaincode definition)는 체인코드 이름과 버전을 포함한 체인코드 거버넌스의 중요한 매개변수들을 포함한다. 체인코드 정의는 또한 피어들에 설치된 체인코드 패키지를 자신의 조직에 의해 승인된 체인코드 정의에 연결시키기 위해 사용되는 패키지 인식번호를 포함한다.

Org2에서도 마찬가지로 동작하도록 환경 변수들을 설정하였기 때문에, Org2를 위한 mycc  체인코드의 정의를 승인하기 위해 아래 명령어를 사용할 수 있다. 승인은 gossip을 사용하여 각 조직 내에서 배포되고, 따라서 명령어가 조직 내의 모든 피어를 대상으로 할 필요는 없다.

# this approves a chaincode definition for your org

# make note of the --package-id flag that provides the package ID

# use the --init-required flag to request the ``Init`` function be invoked to initialize the chaincode

peer lifecycle chaincode approveformyorg --channelID $CHANNEL\_NAME --name mycc --version 1.0 --init-required --package-id $CC\_PACKAGE\_ID --sequence 1 --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 --waitForEvent

체인코드 보증 정책을 설정하기 위해 위 명령어에 --signature-policy 나 --channel-config-policy  인자를 제공할 수 있었다. 보증 정책은 서로 다른 채널 구성원들에 얼마나 많은 피어들이 주어진 체인코드에 대해 거래를 검증할 필요가 있는지 지정한다. 정책을 설정하지 않았기 때문에,  mycc 정의는 기본 보증 정책을 사용하고, 그 값은 거래가 제출되었을 때 존재하는 다수의 채널 구성원들이 거래를 보증할 것을 필요로 한다. 이는 새 조직이 채널에 추가되거나 제거된 경우, 보증 정책이 더 많거나 더 적은 보증을 필요로 하도록 자동적으로 갱신됨을 의미한다. 이 교재에서 기본 정책은 Org1 과 Org2 (즉, 두 개의 보증)에 속한 한 피어로부터의 보증을 필요하다. 정책 적용에 대한 보다 상세한 내용은 [Endorsement policies](https://hyperledger-fabric.readthedocs.io/en/latest/endorsement-policies.html) 을 참고한다.

모든 조직들은 체인코드를 사용할 수 있기 전에 체인코드 정의에 동의할 필요가 있다. Org1에서처럼 동작하도록 다음 4개의 환경 변수들을 수정한다.:

*# 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

이제 Org1에서처럼 mycc 체인코드를 위한 정의를 승인할 수 있다. 여러 피어들을 가진 경우라도 한번은 명령어를 실행할 수 있다.

# this defines a chaincode for your org

# make note of the --package-id flag that provides the package ID

# use the --init-required flag to request the Init function be invoked to initialize the chaincode

peer lifecycle chaincode approveformyorg --channelID $CHANNEL\_NAME --name mycc --version 1.0 --init-required --package-id $CC\_PACKAGE\_ID --sequence 1 --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 --waitForEvent

충분한 수의 채널 구성원들이 체인코드 정의를 승인하였다면, 한 구성원이 채널에 정의를 확정할 수 있다. 기본적으로 대부분 채널 구성원들이 확정될 수 있기 전에 체인코드 정의를 승인할 필요가 있다. 다음 질의를 실행하여 모든 조직들에서 체인코드 정의의 승인 상태를 확인할 수 있다.:

# the flags used for this command are identical to those used for approveformyorg

# except for --package-id which is not required since it is not stored as part of

# the definition

peer lifecycle chaincode queryapprovalstatus --channelID $CHANNEL\_NAME --name mycc --version 1.0 --init-required --sequence 1 --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

이 명령어는 채널에 있는 조직들이 queryapprovalstatus 명령어에서 제공된 체공인코드 정의를 승인하였는지를 보여주는 JSON 형식으로 결과를 제공한다. 여기서는 두 조지글 모두 승인하였다면, 다음과 같은 결과가 나온다.:

{

"Approved": {

"Org1MSP": true,

"Org2MSP": true

}

}

두 채널 구성원들 모두 체인코드 정의를 승인한 후, 다음 명령어를 사용하여 채널에 확정할 수 있다. Org1 이나 Org2 어느 하나에서 이 명령어를 실행할 수 있다. 거래는 Org1 과 Org2 에 있는 피어들을 대상으로 보증을 수집하고 있음을 알 수 있다.

# this commits the chaincode definition to the channel

peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID $CHANNEL\_NAME --name mycc --version 1.0 --sequence 1 --init-required --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 --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 --waitForEvent

**체인코드 호출하기**

체인코드 정의가 한 채널에 확정하면, 체인코드를 호출하여 원장과 상호동작하는 것을 시작할 준비가 된다.  --init-required 플래그를 사용하여 체인코드 정의에서 Init 기능의 실행을 요청하였다. 결과로써,  --isInit 플래그를 최초 호출에 전달하고 인자들을 Init 기능에 제공할 필요가 있다. 다음 명령어를 실행하여 체인코드를 초기화하고 원장에 초기 데이터를 넣는다.

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

# use the --isInit flag if you are invoking an Init function

peer chaincode invoke -o orderer.example.com:7050 --isInit --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":["Init","a","100","b","100"]}' --waitForEvent

처음 호출은 체인코드 컨테이너를 시작한다. 컨테이너 시작을 기다릴 필요가 있다. Node.js 이미지는 더 오래 걸릴 수 있다.

**질의**

컨테이너가 적절히 시작되었고 상태 DB가 올라와있는지를 확인하기 위해 체인코드에 질의를 보내보자. 질의를 위한 구문은 다음과 같다.:

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

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

**호출**

a 로부터 b 로 10 로 옮겨보자. 이 거래는 새로운 블록을 짤라서 상태 DB를 갱신한다. 호출을 위한 구문은 다음과 같다.:

# 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"]}' --waitForEvent

**질의**

이전의 호출이 잘 실행되었는지 확인해보자. 키 a 를 100  값으로 초기화하였고 이제 앞서 호출로 10 을 뺐다. 따라서 a 을 조회하면 90 가 나와야 한다. 질의를 위한 구문은 다음과 같다.

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

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

다음과 같은 결과를 볼 수 있어야 한다.:

Query Result: 90

**추가 피어에 체인코드 설치하기**

피어를 추가해서 원장과 연동하려는 경우, 피어를 채널에 참여시키고 피어에 동일한 체인코드 패키지를 설치하여야 한다. 단지 조직으로부터 체인코드 정의를 한번 승인할 필요가 있다. 체인코드 컨테이너가 각 피어에서 실행되자마자 그 특정 체인코드와 상호작용한다. Node.js 이미지는 구축하고 최초 호출을 시작하는 속도가 더 늦을 수 있다는 사실을 다시 알려준다.

Org2 에 있는 3번째 피어, peer1에 체인코드를 설치한다. Org2에 있는 peer1에 대해 아래 4개 환경 변수들을 수정하여 install 명령어를 실행한다.:

*# 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

이제 Org2의 peer1 에 mycc  패키지를 설치한다.:

*# this command installs a chaincode package on your peer*

peer lifecycle chaincode install mycc**.**tar**.**gz

**질의**

Org2에 있는 peer1 에 질의를 실행하여 확인해보자. 키 a 를 100  값으로 초기화하였고 이제 앞서 호출로 10 을 뺐다. 따라서 a 을 조회하면 90 가 나와야 한다. 질의를 위한 구문은 다음과 같다.

Org2 에 있는 peer1은 질의에 답변하기 전에 채널에 먼저 참여하여야 한다. 채널은 다음 명령어를 실행하여 참여할 수 있다.:

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

join 명령어 결과가 나타나면, 질의를 실행할 수 있다. 질의를 위한 구문은 다음과 같다.

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

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

다음과 같은 결과를 볼 수 있어야 한다.:

Query Result: 90

오류가 있는 경우에는 피어가 채널에 참여하고 현재의 블록체인 수준을 따라잡는데 몇 초가 걸릴 수 있기 때문이다. 필요에 따라 다시 질의를 실행할 수 있다. 또한 추가적인 호출을 수행할 수 있다.

**화면 뒤에서는 어떤 일이 벌어지고 있나?**

**Note**

이 단계들은 script.sh 를 ‘./byfn.sh up’로 실행하는 시나리오를 설명한다. ./byfn.sh down 으로 네트워크를 정리하고, 이 명령이 실행중인지를 확인한다. 그리고나서 네트워크를 다시 실행하기 위해 동일한 docker-compose 창을 사용한다.

* 스크립트 script.sh 는 CLI 컨테이너 내에 묶여 있다. 이 스크립트는 제공된 채널 이름에 대하여  createChannel 명령어를 실행하고, 채널 구성을 위해 channel.tx 파일을 사용한다.
* createChannel  의 결과는 제네시스 블록- <your\_channel\_name>.block -으로, 피어들의 파일 시스템들에 저장되고 channel.tx로부터 지정된 채널 구성을 갖는다.
* joinChannel  을 4개 피어들 모두에 대하여 실행하여, 기존에 생성된 제네시스 블록을 입력으로 사용한다. 이 명령어는 피어들이 <your\_channel\_name> 에 참여하도록 하고 <your\_channel\_name>.block 로 시작하는 체인을 만든다.
* 이제 4개의 피어들과 2개의 조직들로 구성된 채널이 생겼다. 이것이  TwoOrgsChannel 프로파일이다.
* peer0.org1.example.com 과 peer1.org1.example.com 는 Org1에 속하고, peer0.org2.example.com 와 peer1.org2.example.com 는 Org2에 속한다.
* 이 관계들은 crypto-config.yaml  을 통하여 정의하고, MSP 경로는 도커 컴포즈에서 지정된다.
* Org1MSP (peer0.org1.example.com) 과 Org2MSP (peer0.org2.example.com) 를 위한 앵커 피어들이 그러고나면 갱신된다. 이 작업은 Org1MSPanchors.tx 문서와 Org2MSPanchors.tx 문서를 채널 이름과 함께 오더링 서비스에 전달함으로써 수행한다.
* 체인코드 - **abstore** – 가 패키지로 만들어져서 peer0.org1.example.com 와 peer0.org2.example.com 에 설치된다.
* 그러면 체인코드가 Org1과 Org2에 의해 각각 승인되고, 채널 상에서 확정된다. 보증 정책이 지정되지 않았으므로, 대부분 조직들의 채널 기본 보증 정책이 활용되는데, 이는 어떤 거래도 Org1과 Org2 모두에 묶여진 하나의 피어에 의해 보증되어야 함을 의미한다.
* 이제 채널코드 초기화(init)가 호출되어 대상 피어를 위한 컨테이너를 시작하고, 체인코드와 연결된 키 값 쌍들을 초기화한다. 본 예제를 위한 초기 값들은 [“a”,”100” “b”,”200”]이다. 여기서는 첫번째 호출 결과 dev-peer0.org2.example.com-mycc-1.0 의 이름으로 된 컨테이너가 시작된다.
* “a” 값에 대한 질의가 peer0.org2.example.com 를 대상으로 실행된다. dev-peer0.org2.example.com-mycc-1.0 란 이름인Org2를 위한 컨테이너 peer0 가 체인코드가 초기화될 때 시작된다. 질의 결과가 나온다. 쓰기 동작이 발생하지 않으므로, “a”에 대한 질의는 여전히 “100” 값을 보여준다.
* “10”을 “a”에서 “b”로 옮기라는 호출을 peer0.org1.example.com 과 peer0.org2.example.com 로 보낸다.
* “a” 값에 대한 질의를 peer0.org2.example.com 로 보낸다. 90 값이 반환되는데, 키 “a” 값이 10만큼 수정한 앞서 거래가 제대로 반영되었다.
* 체인코드 - **abstore** – 가 peer1.org2.example.com 에 설치된다.
* “a” 값에 대한 질의를 peer1.org2.example.com 로 보낸다. dev-peer1.org2.example.com-mycc-1.0 란 이름의 3번째 체인코드 컨테이너를 시작한다. 90 값이 반환되는데, 키 “a” 값이 10만큼 수정한 앞서 거래가 제대로 반영되었다.

**여기서 설명하려는게 무엇인가?**

체인코드는 원장에 읽기/쓰기를 성공적으로 수행하기 위해 한 피어에 반드시 설치되어야 한다. 더욱이 체인코드 컨테이너는  init 또는 일반적인 거래 – 읽기/쓰기 –(예를 들어, “a” 값에 대한 질의)를 체인코드에 대하여 실행시킬 때까지 피어에서 시작되지 않는다. 거래는 컨테이너를 시작하게 한다. 또한 채널에 있는 모든 피어들은 블록에 변하지 않고 순차적인 레코드를 저장하기 위해 블록체인을 구성하는 원장의 정확한 복사본을 유지하고, 그 뿐만 아니라 현재 상태의 단면을 유지하기 위한 상태 데이터베이스도 가진다. 여기에는 (앞서의 예제에서 peer1.org1.example.com  와 같이) 피어들에 설치된 체인코드를 갖지 않은 피어들도 포함한다. 마지막으로 체인코드는 체인코드 정의가 이미 채널에서 확정되었기 때문에 (앞서의 예제에서 peer1.org2.example.com 와 같이) 설치된 이후부터 접근가능하다.

**거래를 어떻게 볼 수 있나?**

CLI 도커 컨테이너를 위한 로그를 살펴본다.

docker logs **-**f cli

다음과 같은 결과가 나온다.:

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 **=====================**

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

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

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

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

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

이 로그들을 통하여 다양한 거래들을 볼 수 있다.

**체인코드 로그를 어떻게 볼 수 있나?**

개별 체인코드 컨테이너들을 살펴보면 각 컨테이너를 대상으로 실행된 개별 거래를 볼 수 있다. 다음은 각 컨테이너에 대한 합쳐진 결과이다.:

$ 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"}

또한 피어 로그를 통해 체인코드의 호출 메시지들과 블록 확정 메시지들을 볼 수 있다.:

$ docker logs peer0.org1.example.com

**도커 컴포즈 배치 이해하기**

BYFN 예제는 두 개의 변종 도커 컴포즈 파일들을 제공하는데, 둘다 ( base 폴더에 위치하는) docker-compose-base.yaml 에서 확장되어 있다. 첫번째 변종인 docker-compose-cli.yaml는 하나의 오더러와 4개의 피어들로 구성된 CLI 컨테이너를 제공한다. 이 파일은 여기에 있는 전체 명령들을 위해 사용한다.

**Note**

여기서 나머지 부분은 SDK를 위해 설계된 docker-compose 파일을 다룬다. 이 테스트를 실행하는데 대한 상세한 내용은 [Node SDK](https://github.com/hyperledger/fabric-sdk-node) 를 참고한다.

두 번째 변종인 docker-compose-e2e.yaml는 Node.js SDK를 사용하는 전체 테스트를 실행하기 위해 만들어졌다. SDK와 함께 작동하는 것과는 별개로 주요한 차이점은 fabric-ca 서버를 위한 컨테이너가 있다는 것이다. 결과적으로 사용자 신고와 등록을 위해 조직 CA에 REST 호출을 보낼 수 있다.

byfn.sh 스크립트를 먼저 실행하지 않고서 docker-compose-e2e.yaml 를 사용하기를 원한다면, 4가지 부분에서 조금의 수정을 필요로 한다. 조직의 CA를 위한 개인 키들로 지정할 필요가 있다. 이 값들은 crypto-config 폴더에서 찾을 수 있다. 예를 들어, Org1을 위한 개인 키를 찾으려면 crypto-config/peerOrganizations/org1.example.com/ca/ 경로를 따라간다. 개인 키는 \_sk 가 붙은 긴 해시 값이다. Org2를 위한 경로는 crypto-config/peerOrganizations/org2.example.com/ca/ 이다.

docker-compose-e2e.yaml 에서 ca0와 ca1에 대한 FABRIC\_CA\_SERVER\_TLS\_KEYFILE 변수를 갱신한다. 또한 ca 서버를 시작하는 명령어에서 제공되는 경로를 편집할 필요가 있다. 각 CA 컨테이너에 대하여 두 번 동일한 개인 키를 제공하고 있다.

**CouchDB 사용하기**

상태 데이터베이스는 기본 데이터베이스(goleveldb)에서 CouchDB로 전환될 수 있다. 동일한 체인코드 기능들이 CouchDB에서 이용할 수 있지만, JSON 모델로 된 체인코드 데이터에 따라서 상태 데이터베이스 데이터 내용에 대하여 다양하고 복잡한 질의들을 수행할 수 있는 추가적인 기능이 있다.

기본 데이타베이스(goleveldb) 대신 CouchDB를 사용하려면, 네트워크를 시작할 때 docker-compose-couch.yaml 를 전달하는 것을 제외하고는 앞서 문서 생성에서 설명한 동일한 절차를 따른다.:

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

여기서 **abstore** 는 CouchDB를 기반으로 사용하여 동작하고 있다.

**Note**

Fabric-couchdb 컨테이너 포트를 호스트 포트에 연결을 하려고 선택한 경우, 보안 위험을 알고 있어야 한다. 개발 환경에서의 포트 연결은 CouchDB REST API를 사용할 수 있게 해야 하고, CouchDB 웹 인터페이스 (Fauxton)를 통해 데이터베이스를 시각화해준다. 운영 환경은 CouchDB 컨테이너들에 대한 외부로부터의 접근을 제한하기 위해 포트 연결 적용을 제한하는게 좋다.

앞서 설명한 단계들을 사용하여 CouchDB 상태 데이터베이스에 **abstore** 체인코드를 사용할 수 있지만, CouchDB 질의 기능을 연습하기 위하여 JSON 모델의 데이터를 가진 체인코드를 사용할 필요가 있다. 예제 체인코드인 **marbles02** 는 CouchDB 데이터베이스를 사용하는 경우 체인코드로부터 실행할 수 있는 질의들을 보여주기 위해 작성되었다. **marbles02** 체인코드는 chaincode in the fabric/examples/chaincode/go 디렉토리에서 찾을 수 있다.

채널 만들어 참여하기 부분에서 설명한 것과 같이 채널을 만들어 참여하기 위한 동일한 과정을 따른다. 피어를 채널에 참여한 후 다음 단계들을 사용하여 **marbles02** 체인코드와 상호작용한다.:

* 체인코드를 패키지화해서 peer0.org1.example.com 에 설치한다.

peer lifecycle chaincode package marbles**.**tar**.**gz **--**path github**.**com**/**hyperledger**/**fabric**-**samples**/**chaincode**/**marbles02**/**go**/** **--**lang golang **--**label marbles\_1

peer lifecycle chaincode install marbles**.**tar**.**gz

The install command will **return** a chaincode packageID that you will use to

approve a chaincode definition**.**

2019**-**04**-**08 20:10:32.568 UTC [cli**.**lifecycle**.**chaincode] submitInstallProposal **->** INFO 001 Installed remotely: response:**<**status:200 payload:"\nJmarbles\_1:cfb623954827aef3f35868764991cc7571b445a45cfd3325f7002f14156d61ae\022\tmarbles\_1" **>**

2019**-**04**-**08 20:10:32.568 UTC [cli**.**lifecycle**.**chaincode] submitInstallProposal **->** INFO 002 Chaincode code package identifier: marbles\_1:cfb623954827aef3f35868764991cc7571b445a45cfd3325f7002f14156d61ae

* 패키지 ID를 환경 변수로 저장하여 이후 명령어 사용에서 전달할 수 있다.:
* CC\_PACKAGE\_ID**=**marbles\_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173
* Org1 으로 체인코드 정의를 승인한다.:

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

peer lifecycle chaincode approveformyorg --channelID $CHANNEL\_NAME --name marbles --version 1.0 --package-id $CC\_PACKAGE\_ID --sequence 1 --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 --waitForEvent

* peer0.org2.example.com 에 체인코드를 설치한다.

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 lifecycle chaincode install marbles**.**tar**.**gz

* Org2 로 체인코드 정의를 승인하고, 채널에 체인코드 정의를 확정한다.:

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

peer lifecycle chaincode approveformyorg --channelID $CHANNEL\_NAME --name marbles --version 1.0 --package-id $CC\_PACKAGE\_ID --sequence 1 --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 --waitForEvent

peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID $CHANNEL\_NAME --name marbles --version 1.0 --sequence 1 --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 --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 --waitForEvent

* 이제 marble을 만들 수 있다. 체인코드에 대한 첫번째 호출은 체인코드 컨테이너를 시작시킨다. 컨테이너가 시작될 때까지 기다릴 필요가 있다.

# 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 --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":["initMarble","marble1","blue","35","tom"]}'

컨테이너가 시작되면, 추가적인 명령어들을 실행하여 marble 들을 만들고 이동시키도록 할 수 있다.:

# 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 --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":["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 --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":["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 --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":["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 --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":["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 --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":["delete","marble1"]}'

* Docker-compose 에 CouchDB 포트들을 연결하도록 선택한 경우, 이제 브라우저를 열어 다음 URL로 들어가서 CouchDB 웹 인터페이스 (Fauxton)를 통하여 상태 데이터베이스를 볼 수 있다.:

http://localhost:5984/\_utils

 mychannel 란 이름(또는 자신의 고유한 채널 이름)을 가진 데이터베이스와 그 내의 문서들을 볼 수 있어야 한다.

**Note**

아래 명령어들을 위해 $CHANNEL\_NAME 변수를 적절히 갱신하여야 한다.

CLI 로부터 정상적인 질의(예를 들어,  marble2 읽기)를 실행할 수 있다.:

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

결과로 marble2 의 세부 내용을 표시한다.:

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

 marble1 과 같이 특정 marble의 이력을 살펴볼 수 있다.:

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

결과로 marble1 에 대한 명령 실행 결과가 표시된다.:

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":}]

소유자가  jerry 인 marble 필드들을 질의하는 것과 같이 데이터 내용에 대하여 다양한 질의들을 실행할 수 있다.:

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

결과로  jerry 가 소유한 두 개의 marble이 표시된다.:

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}}]

**왜 CouchDB**

CouchDB는 NoSQL 솔루션 중의 하나이다. CouchDB는 문서-중심 데이터베이스로 문서 필드들이 키-값 매핑으로 저장되어 있다. 필드들은 단일한 키-값의 쌍이거나 목록이거나 매핑일 수 있다. LevelDB에서 지원되는 키 질의/복합 키 질의/키-범위 질의 뿐만 아니라, CouchDB는 데이터 내용이 JSON 형식으로 저장되고 전체에 대하여 질의가 가능하기 때문에 전체 블록체인 데이터에 대한 키를 사용하지 않은 질의들과 같이 전체 데이터에 대한 다양한 질의들을 또한 지원한다. 그러므로 CouchDB는 LevelDB가 지원하지 않는 많은 사용 경우들을 위한 체인코드의 감사와 보고 요구사항들을 만들 시킬 수 있다.

CouchDB는 또한 블록체인에서 규정 준수와 데이터 보호를 위한 보안을 강화할 수 있다. 거래 내의 개별 속성들을 필터링하고 마스킹 처리하고 필요한 경우 읽기 전용 권한만을 부여하여 필드 수준의 보안을 구현할 수 있다.

또한, CouchDB는 CAP 정리에서 AP-유형(가용성과 파티션 허용)에 해당한다. CouchDB는 Eventual Consistency 방식의 마스터-마스터 복제 모델을 사용한다. 보다 상세한 정보는 [Eventual Consistency page of the CouchDB documentation](http://docs.couchdb.org/en/latest/intro/consistency.html) 에서 찾을 수 있다. 그러나 각 패브릭 피어 내에는 데이터베이스 복제본들이 없기 때문에 데이터베이스에 쓰기 작업은 일관성과 내구성을 보증하게 된다(Eventual Consistency는 아니다).

CouchDB는 패브릭용으로 최초의 외부적인 접속 가능한 상태 데이타베이스로, 그 외 다른 외부적인 데이터베이스 선택이 가능하고 가능하여야 한다. 예를 들어, IBM은 블록체인을 위하여 관계형 데이터베이스를 사용할 수 있다. 그리고 또한 애플리케이션 수준에서의 보장 없이 데이터 일관성을 유지할 수 있도록 CP-유형(일관성과 파티션 허용) 데이터베이스가 필요할 수 있다.

**데이터 지속성에 대한 설명**

데이터 지속성이 피어 컨테이너나 CouchDB 컨테이너 상에서 필요한 경우, 한가지 선택은 docker-host에 있는 한 디렉토리를 컨테이너에 관련 디렉토리로 마운드하는 것이다. 예를 들어, docker-compose-base.yaml  파일에 있는 피어 컨테이너 사양에 다음 두 줄을 추가한다.:

volumes:

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

CouchDB 컨테이너에 대해서는 CouchDB 컨테이너 사양에 다음 두 줄을 추가한다.:

volumes:

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

**문제 해결하기**

* 항상 네트워크를 깨끗한 상태로 시작하라. 기존 문서, 암호, 컨테이너들, 체인코드 이미지들을 제거하기 위해 다음 명령어를 사용한다.:
* **./**byfn**.**sh down

**Note**

기존 컨테이너들과 이미지들을 제거하지 않으면 오류가 생길 수 있다.

* 도커 오류가 있는 경우, 우선 도커 버전([Prerequisites](https://hyperledger-fabric.readthedocs.io/en/latest/prereqs.html))을 확인하고, 도커 프로세스 재시작을 시도해본다. 도커 문제는 간혹 바로 인식되지 않을 수 있다. 예를 들어, 컨테이너 내에 올라온 암호 자료를 접근할 수 없는 오류가 생길 수 있다.

이미지를 제거하고 처음부터 시작을 하려면,:

docker rm -f $(docker ps -aq)

docker rmi -f $(docker images -q)

* create, approve, commit, invoke, query 명령어들에서 오류가 생기는 경우, 채널 이름과 체인코드 이름을 제대로 사용했는지 확인한다. 제공된 예제 명령어들에 임시 값들이 있다.
* 아래 오류가 생기는 경우,:

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)

앞서 실행에서 체인코드 이미지들(예를 들어, dev-peer1.org2.example.com-mycc-1.0 또는 dev-peer0.org1.example.com-mycc-1.0) 이 있을 수 있다. 이들을 제거하고 다시 시도해본다.

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

* 아래와 유사한 오류가 발생하는 경우:

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

“latest”로 명시된 “1.0.0” 이미지들에 대하여 네트워크를 운영하고 있는지 확인한다.

* 아래 오류가 발생한 경우,:

[configtx**/**tool**/**localconfig] Load **->** CRIT 002 Error reading configuration: Unsupported Config Type ""

panic: Error reading configuration: Unsupported Config Type ""

FABRIC\_CFG\_PATH  환경 변수를 적절히 설정하지 않았다. configtxgen 도구는 configtx.yaml 을 찾기 위해 이 변수가 필요하다. 되돌아가서  export FABRIC\_CFG\_PATH=$PWD 를 실행하고, 채널 문서를 다시 만든다.

* 네트워크를 정리하려면, down 옵션을 사용한다:

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

* “active endpoints”가 여전히 있다는 오류를 본 경우, 도커 네트워크를 잘라내야 한다. 그러면 이전 네트워크들을 정리하고, 깨끗한 환경으로 시작하게 된다.:

docker network prune

다음 메시지가 나타난다.:

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

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

y 를 선택한다.

* 아래와 유사한 오류가 발생하는 경우:

**/**bin**/**bash: **./**scripts**/**script**.**sh: **/**bin**/**bash**^**M: bad interpreter: No such file **or** directory

문제가 있어 보이는 파일(이 예제에서는 script.sh)이 Unix 형식으로 되어 있는지 확인한다. 이 오류는 대부분 Git 구성에서  core.autocrlf 를 false  로 설정하지 않아서 발생한다([Windows extras](https://hyperledger-fabric.readthedocs.io/en/latest/prereqs.html#windows-extras) 참고). 이 오류를 해결하는 여러 방법들이 있다. 예를 들어, vim 편집기를 사용할 수 있는 경우, 파일을 연다.:

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

그리고 다음 vim 명령어를 실행하여 형식을 변경한다.:

:set ff**=**unix

**Note**

오류들이 계속 발생하면, [Hyperledger Rocket Chat](https://chat.hyperledger.org/home) 이나 [StackOverflow](https://stackoverflow.com/questions/tagged/hyperledger-fabric) 상에 있는 **fabric-questions** 채널에 로그를 공유하도록 한다.

# 시작하기

시작하기 전에, 이전에 해보지 않았다면, 블록체인 애플리케이션을 개발하거나 하이퍼레저 패브릭을 운영하려는 플랫폼 상에 “사전 요구 사항”이 모두 설치되었는지 확인할 필요가 있다.

사전 요구 사항들을 다 설치하였다면, 하이퍼레저 패브릭을 다운로드하고 설치할 준비가 되었다. 패브릭 실행을 위해 실제 설치자를 개발하는 작업을 하면서, 시스템에 예제들과 실행파일들, 도커 이미지들을 설치하는 스크립트를 제공한다. 그 스크립트는 도커 이미지들을 다운로드하여 개인의 레지스터리에 둔다. [Install Samples, Binaries and Docker Images](https://hyperledger-fabric.readthedocs.io/en/latest/install.html)

## 하이퍼레저 패브릭 SDKs

하이퍼레저 패브릭은 다양한 프로그래밍 언어를 지원하기 위하여 여러 가지 SDK들을 제공한다. Node.js 와 자바용으로 공식적으로 출시된 두 가지 SDK가 있다.:

* [Hyperledger Fabric Node SDK](https://github.com/hyperledger/fabric-sdk-node) 와 [Node SDK documentation](https://fabric-sdk-node.github.io/).
* [Hyperledger Fabric Java SDK](https://github.com/hyperledger/fabric-sdk-java).

또한 (파이썬과 Go, REST용으로) 공식적으로 출시되지 않은 3 이상의 SDK가 있지만, 여전히 다운로드해서 테스트하는데 이용할 수 있다.:

* [Hyperledger Fabric Python SDK](https://github.com/hyperledger/fabric-sdk-py).
* [Hyperledger Fabric Go SDK](https://github.com/hyperledger/fabric-sdk-go).
* [Hyperledger Fabric REST SDK](https://github.com/hyperledger/fabric-sdk-rest).

## 하이퍼레저 패브릭 CA

하이퍼레저 패브릭은 블록체인 네트워크에서 개체를 구성하고 관리할 수 있도록 인증서와 키 자료를 생성하는데 사용할 수 있는 인증서 부여 서비스([certificate authority service](http://hyperledger-fabric-ca.readthedocs.io/en/latest)) 를 선택 사항으로써 제공한다. 그러나 ECDSA 인증서를 생성할 수 있는 어떠한 CA도 사용할 수 있다.

# 사전 요구 사항

시작하기 전에, 이미 설치를 하지 않았다면, 블랙체인 애플리케이션을 개발하거나 하이퍼레저 패브릭을 운영하려는 플랫폼 상에 아래 사전 요구 사항들이 이미 설치되었는지 확인할 필요가 있다.

## cURL 설치하기

아직 cURL이 설치되어 있지 않거나 이 문서에 있는 curl 명령어를 실행할 때 오류가 있는 경우, <https://curl.haxx.se/download.html> 에서 최신 버전을 내려 받는다.

**Note**

윈도우를 사용하는 경우, 아래에 있는 ‘윈도우 추가 사항’을 참고한다.

## 도커와 도커 컴포즈

하이퍼레저 패브릭을 운영하거나 그 위에서 개발하려는 플랫폼에 다음 요소들을 설치하여야 한다.:

* MacOSX, \*nix, 또는 Windows 10: [Docker](https://www.docker.com/get-docker) 도커 버전 17.06.2-ce 이상 필요.
* 이전 윈도우 버전: [Docker Toolbox](https://docs.docker.com/toolbox/toolbox_install_windows/) 와, 도커 버전 Docker 17.06.2-ce 이상 필요.

터미널 프롬프트에서 다음 명령어를 실행하여 설치한 도커의 버전을 확인할 수 있다.:

docker **--**version

**Note**

맥이나 윈도우용 도커, 또는 그외 도커 툴박스를 설치하면 도커 컴포즈(Docker Compose)도 또한 설치한다. 이미 도커가 설치된 경우, 도커 컴포즈 버전 1.14.0 이상이 설치되어 있는지 확인하여야 한다. 설치되어 있지 않다면 보다 최신 버전의 도커 설치를 권장한다.

터미널 프롬프트에서 다음 명령어를 실행하여 설치한 도커 컴포즈의 버전을 확인할 수 있다.:

docker**-**compose **--**version

## Go 프로그래밍 언어

하이퍼레저 패브릭은 자체의 많은 구성 요소들에서 Go 프로그래밍 언어를 사용한다.

* Go 버전 1.12.x 가 필요하다.(<https://golang.org/dl/>)

Go로 체인코드 프로그램을 작성하고 있다면, 두 개의 환경 변수들을 적절히 설정할 필요가 있다. 이 설정은 적절한 시작 파일에 둠으로써 영구적인 설정을 할 수 있다. 리눅스 bash 쉘을 사용하고 있다면, 개인의 ~/.bashrc 파일이다.

먼저 아래와 같은 다운로드한 패브릭 코드를 둔 Go작업 영역을 지정하도록 GOPATH 환경 변수를 설정하여야 한다.:

export GOPATH=$HOME/go

**Note**

GOPATH 환경 변수를 꼭 설정하여야 한다.

리눅스에서 GOPATH 환경 변수가 콜론으로 분리된 디렉토리의 목록으로 될 수 있고, 설정되지 않은 경우 $HOME/go의 기본값이 사용되더라도, 현재의 패브릭 구축 프레임워크는 여전히 그 환경 변수를 설정하고 export 처리하기를 원하고, Go 작업영역을 위한 하나의 단일한 디렉토리 이름만을 지정하여야 한다. (이 제한은 향후 출시되는 버전에서는 없어질 것이다.)

다음으로, (다시 적절한 시작 파일 내에서) 명령어 검색 경로를 확장하여, 리눅스에서의 bash 에서는 아래와 같이 해서, Go의 bin 디렉토리를 포함시켜야 한다.:

export PATH=$PATH:$GOPATH/bin

새롭게 Go 작업영역을 설치한 경우 이 디렉토리가 없을 수 있지만, 구축 체계의 다른 일부로써 사용될 Go 실행 파일들과 함께 패브릭 구축 체계에서 활용된다. 그래서 현재 그와 같은 디렉토리가 없는 경우라도, 위와 같이 해서 쉘 검색 경로에 추가해 둔다.

## Node.js 실행 환경과 NPM

Node.js 용 하이퍼레저 패브릭 SDK를 활용하여 하이퍼레저 패브릭용 애플리케이션을 개발하고 있는 경우, Node.js 버전 8.9.x 가 설치될 필요가 있다.

**Note**

8.x 이전의 버전은 현재 지원되지 않는다.

* [Node.js](https://nodejs.org/en/download/) - 버전 8.x

**Note**

Node.js 설치하면 NPM 또한 설치한다. 그러나 설치된 NPM 버전을 확인할 것을 권장한다. 다음 명령어를 통해 npm 도구를 업그레이드할 수 있다.:

npm install npm@5**.**6.0 **-**g

### 파이썬(Python)

**Note**

아래 내용은 우분투 16.04 사용자들에게만 적용된다.

기본적으로 우분투 16.04는  python3 실행 파일로 해서 파이썬 3.5.1이 설치되어 있다. 패브릭 Node.js SDK는 npm install 이 제대로 동작하기 위하여 파이썬 2.7을 필요로 한다. 다음 명령어를 사용해서 2.7 버전을 설치한다.:

sudo apt**-**get install python

설치된 버전을 확인한다.:

python **--**version

## 윈도우 추가 사항

윈도우 7 상에서 개발하고 있는 경우, Git Bash(<https://git-scm.com/downloads>)를 사용하고 윈도우 내장 쉘 보다 더 나은 도커 Quickstart 터미널을 내에서 작업하기를 원할 수 있다.

그렇지만, 경험 상으로 이것은 제한된 기능으로 인하여 좋지 않은 개발 환경이었다. “시작하기” 와 같은 도커 기반 시나리오를 실행하는데는 적합할 수 있지만, make 와 docker 명령어들을 호출하는 작업에는 어려움이 있을 수 있다.

윈도우 10에서는 전형적인 도커 분산을 사용하여야 하고, Windows PowerShell을 사용할 수 있다. 그러나, binaries 명령어를 제대로 하려면 여전히 uname 명령어를 이용할 수 있을 필요가 있다. 이를 Git의 일부로써 얻을 수 있지만, 64비트 버전에서만 지원됨을 알아야 한다.

git clone 명령어를 실행하기 전에, 다음 명령어들을 실행한다.

git config **--global** core**.**autocrlf false

git config **--global** core**.**longpaths true

다음 명령어들을 사용하여 이 매개변수들의 설정 결과를 확인할 수 있다.:

git config **--**get core**.**autocrlf

git config **--**get core**.**longpaths

그러면 false 와 true 를 각각 보여준다.

Git과 도커 툴박스와 함께 제공된 curl 명령어는 오래 되어, “시작하기” 에서 사용된 redirect를 제대로 처리하지 못한다. [cURL downloads page](https://curl.haxx.se/download.html) 에서 보다 최신 버전을 설치하고 사용하는지 확인하여야 한다.

Node.js를 사용하기 위해서는 무료로 사용할 수 있는 Visual Studio C++ Build Tools를 필요로 하고, 다음 명령어를 사용하여 설치할 수 있다.:

npm install **--global** windows**-**build**-**tools

(Python 2.7도 함께 설치된다)

보다 상세한 정보를 [NPM windows-build-tools page](https://www.npmjs.com/package/windows-build-tools) 를 살펴 본다..

완료하였으면, 또한 다음 명령어로 NPM GRPC 모듈을 설치하여야 한다.:

npm install **--global** grpc

이제 “시작하기” 의 예제와 자료를 진행할 환경 준비가 되었다.

**Note**

이 문서에서 다루지 않은 질문이 있거나, 지침서와 관련한 문제가 발생하는 경우, 추가적인 도움을 찾을 수 있는 팁들은  [Still Have Questions?](https://hyperledger-fabric.readthedocs.io/en/latest/questions.html) 페이지에서 살펴본다.

# 예제와 실행 파일, 도커 이미지 설치하기

패브릭 실행을 위해 실제 설치자를 개발하는 작업을 하면서, 시스템에 예제들과 실행파일들, 도커 이미지들을 설치하는 스크립트를 제공한다. 그 스크립트는 도커 이미지들을 다운로드하여 개인의 레지스터리에 둔다. 설치된 예제 애플리케이션들이 하이퍼레저 패브릭의 기능들과 운영에 대해 배우는데 유용함을 알 수 있을 것으로 생각한다.

**Note**

**윈도우에서 작업하는 경우, 이후에 사용할 터미널 명령어들을 위하여 도커** Quickstart 터미널의 사용을 원할 수 있다. 앞서 설치하지 않았다면, “사전 요구 사항”을 살펴본다.

윈도우 7이나 맥OS에서 도커 툴박스를 사용하고 있는 경우, 예제들을 설치하고 실행할 때 C:\Users (윈도우 7) 나 /Users (맥OS) 아래 한 위치를 사용할 필요가 있다.

맥용 도커를 사용하고 있는 경우, /Users나, /Volumes, /private 또는 /tmp 의 아래 한 위치를 사용할 필요가 있다. 다른 위치를 사용하려는 경우, [file sharing](https://docs.docker.com/docker-for-mac/#file-sharing)의 도커 문서를 살펴본다.

윈도우용 도커를 사용하고 있는 경우, [shared drives](https://docs.docker.com/docker-for-windows/#shared-drives) 에 대한 도커 문서를 살펴보고, 공유 드라이브들 중 하나의 아래에 있는 위치를 사용한다.

Fabric-samples 저장 공간을 둘 컴퓨터 상의 한 장소를 결정하고, 터미널 창에서 그 디렉토리로 들어간다. 이어지는 명령어들을 다음 단계에 따라 실행한다.:

1. 필요한 경우, [hyperledger/fabric-samples](https://github.com/hyperledger/fabric-samples) 레파지토리를 복사한다.
2. 적절한 버전 태그를 확인한다.
3. 플랫폼에 맞는 지정된 버전의 하이퍼레저 패브릭 실행 파일들과 구성 파일들을 fabric-samples의 /bin과 /config 디렉토리에 설치한다.
4. 지정된 버전의 하이퍼레저 패브릭 도커 이미지를 다운로드한다.

준비가 되었고, 패브릭 예제와 실행 파일을 설치하려는 디렉토리로 가서, 실행 파일들과 이미지들을 내려 받는 명령어를 실행한다.

**Note**

가장 최신 릴리즈를 원한다면, 모든 버전 지시자를 삭제한다.

curl **-**sSL http:**//**bit**.**ly**/**2ysbOFE **|** bash **-**s

**Note**

특정 릴리즈를 원한다면 패브릭과 패브릭-CA, 제3자 도커 이미지에 해당하는 버전 지시자를 전달한다. 아래 명령어는 **패브릭 v2.0.0 Alpha 릴리즈 v2.0.0-alpha**를 다운로드하는 방법을 보여준다.

curl **-**sSL http:**//**bit**.**ly**/**2ysbOFE **|** bash **-**s **--** **<**fabric\_version**>** **<**fabric**-**ca\_version**>** **<**thirdparty\_version**>**

curl **-**sSL http:**//**bit**.**ly**/**2ysbOFE **|** bash **-**s **--** 2.0**.**0**-**alpha 2.0**.**0**-**alpha 0.4**.**15

**Note**

위의 curl 명령어를 실행하는데 오류가 발생하면, redirect나 지원되지 않는 환경을 처리하지 못하는 너무 이전 버전의 curl을 사용하는 것일 수 있다.

최신 버전의 curl을 찾고 올바른 환경을 갖는데 대한 추가적인 정보는 “사전 요구 사항”을 살펴본다. 아니면, 짤지 않은 URL 주소:<https://raw.githubusercontent.com/hyperledger/fabric/master/scripts/bootstrap.sh>를 대신해서 사용할 수 있다.

위 명령어는 bash 스크립트를 다운로드하여 실행하여 네트워크를 설정하고 앞서 만든 복사해서 넣을 디렉토리에 넣는데 필요한 플랫폼 별 실행 파일들을 다운로드하고 압축 해제를 한다. 아래와 같은 플랫폼 별 실행 파일들을 가져온다.:

* configtxgen,
* configtxlator,
* cryptogen,
* discover,
* idemixgen
* orderer,
* peer,
* token,
* fabric-ca-client

그리고 현재 작업 디렉토리의 하위 디렉토리인 bin 에 위치한다.

아래와 같이 PATH 환경 변수에 경로를 추가하여 각 실행 파일에 전체 경로를 입력하지 않고 실행할 수 있게 할 수 있다.:

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

마지막으로, 스크립트는 하이퍼레저 도커 이미지들을 [Docker Hub](https://hub.docker.com/u/hyperledger/)  에서 사용자의 로컬 도커 레지스터리에 다운로드하여 ‘latest’라고 명시한다..

스크립트는 최종적으로 설치된 도커 이미지들을 보여준다.

각 이미지들의 이름들을 살펴본다. 이 이름들이 궁극적으로 하이퍼레저 패브릭 네트워크를 구성할 구성 요소들이다. 또한 하나는 “amd64-1.x.x(2.0 Alpha의 경우, 2.0.0-alpha)”로 명시되어 있고, 하나는 “latest”라고 명시되어 있는 동일한 이미지 ID의 두 개의 인스턴스가 있음을 볼 수 있다. 1.2.0 이전에 다운로드될 이미지는 uname -m  에 의해 결정되고, “x86\_64-1.x.x” 처럼 표시된다.

**Note**

다른 플랫폼에서는 x86\_64/amd64 가 해당 플랫폼을 나타내는 문자로 바뀌어져 있다.

**Note**

이 문서에서 다루지 않은 질문이 있거나, 지침서와 관련한 문제가 발생하는 경우, 추가적인 도움을 찾을 수 있는 팁들은  [Still Have Questions?](https://hyperledger-fabric.readthedocs.io/en/latest/questions.html) 페이지에서 살펴본다.