Hyperledger Fabric

Hyperledger Fabric is the most widely-used permissioned blockchain in the Hyperledger family. It is an open source enterprise-grade platform that leverages a highly-modular and configurable architecture. Hyperledger Fabric is optimized for a broad range of industry use cases, including the finance, banking, healthcare, insurance, and public sectors, as well as supply chains and digital asset management.

Hyperledger Fabric supports smart contact development in general-purpose programming languages, such as Java, Go, and Node.js. Hyperledger Fabric is also operating under a governance model to build trust between participants on a shared network.

Reviewing the Hyperledger Fabric architecture and components

We will review and examine various Hyperledger Fabric components and architectures throughout this recipe. Hyperledger Fabric has three core components, which are peers, ordering service, and Fabric CA:

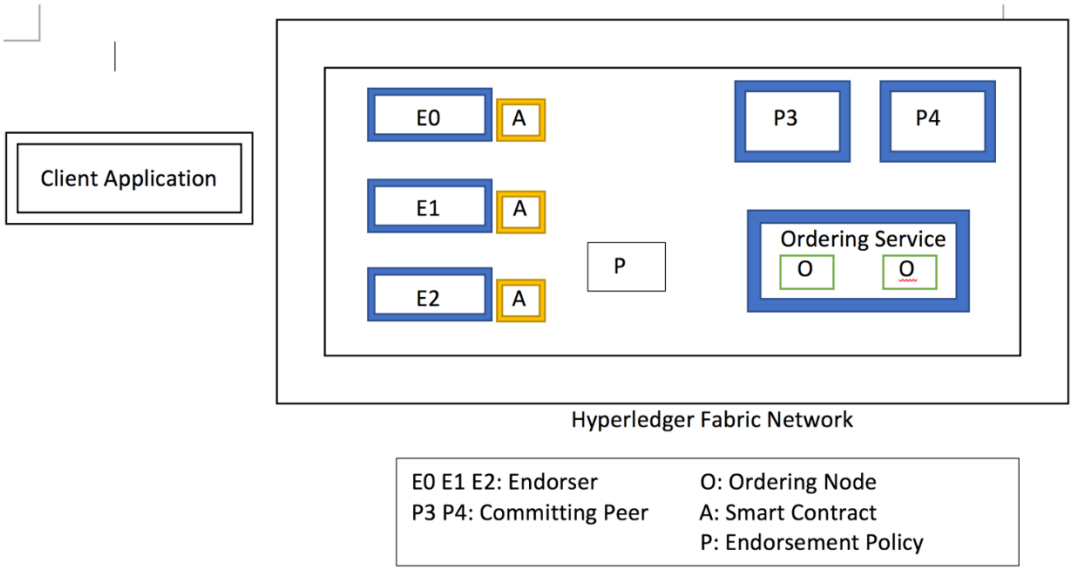
- **Peer**: A node on the network that maintains the state of the ledger and manages chaincode. Any number of peers may participate in a network. A peer can be an endorser, which executes transactions, or a committer, which verifies the endorsements and validates transactions results. An endorser is always a committer. Peers form a peer-to-peer gossip network. A peer manages the events hub and delivers events to the subscribers.
- **Ordering service**: Packages transactions into blocks to be delivered to peers, since it communicates only with peers. The ordering service is the genesis of a network. Clients of the ordering service are peers and applications. A group of orderers run a communication service, called an ordering service, to provide an atomic broadcast. The ordering service accepts transactions and delivers blocks. The ordering service processes all configuration transactions to set up network policies (including readers, writers, and admins). The orderer manages a pluggable trust engine (such as CFT or BFT) that performs the ordering of the transactions.
- **Fabric CA**: Fabric CA is the certificate authority that issues PKI-based certificates to network member organizations and users. Fabric CA supports LDAP for user authentication and HSM for security. Fabric CA issues one root certificate to member organizations and one enrollment certificate to each authorized user.

Hyperledger Fabric also have several important key features and concepts:

- **Fabric ledger**: Maintained by each peer and consists of two parts: the blockchain and the world state. Transaction read/write and channel configurations sets are written to the blockchain. A separate ledger is maintained for each channel for each peer that joins. The world state has options of either LevelDB or CouchDB, where LevelDB is a simple key-value store and CouchDB is a document store that allows complex queries. The smart contract decides what is written into the world state.
- **Channel**: Provides privacy between different ledgers and exists in the scope of a channel. Channels can be shared across an entire network of peers, and peers can participate in multiple channels. Channels can be permissioned for a specific set of participants. Chaincode is installed on peers to access the world state. Chaincode is instantiated on specific channels. Channels also support concurrent execution for performance and scalability.
- **Organization**: Define boundaries within a Fabric blockchain network. Each organization defines an MSP for the identities of administrators, users, peers, and orderers. A network can include many organizations, representing a consortium. Each organization has an individual ID.
- **Endorsement policy**: The conditions by which a transaction can be endorsed. A transaction can only be considered valid if it has been endorsed according to its policy. Each chaincode is deployed with an endorsement policy. **Endorsement system chaincode** (**ESCC**) signs the proposal response on the endorsing peer and **validation system chaincode** (**VSCC**) validates the endorsement.
- **Membership services provider** (**MSP**): Manages a set of identities within a distributed Fabric network. It provides identities for peers, orderers, client applications, and administrators. Where the identities can be Fabric CA or external CA, MSP provides authentication, validation, signing and issuance. MSP support different crypto standards with a pluggable interface. A network can include multiple MSPs (typically one per organization), which can include TLS crypto material for encrypted communications.

Getting ready

We will look into a sample transaction flow on Hyperledger Fabric. Fabric uses the execute-order-validate blockchain transaction flow architecture shown in the following diagram:



How to do it...

In this section, we will review how a transaction is created on the Hyperledger Fabric network:

1. The **Client Application** submits a transaction proposal for smart contact **A** to the network. The endorsement policy requires three endorsers—**E0**, **E1**, and **E2**—to sign together.
2. The endorsers execute proposed transactions. At this time, three endorsers—**E0**, **E1**, **E2**—will each execute the proposed transaction independently. None of these executions will update the ledger. Each execution will capture the set of **read and written** (**RW**) data, which will now flow in the fabric network. All transactions should be signed and encrypted.
3. RW sets are asynchronously returned to the client application with a transaction proposal. The RW sets are signed by each endorser and will be processed later.
4. All transactions that returned from the Fabric network are submitted for ordering. The application can submit responses as a transaction to be ordered, and ordering happens across the Fabric in parallel with transactions submitted by other applications.
5. **Ordering Service** collects transactions into proposed blocks for distribution to committing peers. This proposed blocks can then be deliver to other peers in a hierarchy. There are two ordering algorithms available: SOLO (single node for development) and Kafka (crash-fault-tolerance for production). In the production system, it is suggested to use Kafka.
6. Committing peers validate the transactions. All committing peers validate against the endorsement policy and check whether RW sets are still valid for

the current world state. World state is not update if there is invalid transctions but are retained on the ledger while validated transactions are applied to the world state.

7. Client applications can register to be notified on the status of transactions, to find out whether they succeed or fail, and when blocks are added to the ledger. Client applications will be notified by each peer to which they are independently connected.

How it works...

We reviewed how transaction flow works in Fabric. Fabric uses the execute-order-validate model with the following seven steps:

1. Client application submits a transaction proposal
2. Endorsers execute the proposed transactions
3. Client applications receive transaction proposal response
4. Transactions are submitted for ordering
5. Transactions are delivered to committing peer
6. Validated transaction are applied to world state
7. Client applications get notified with the status of the transaction

In the next recipe, we will walk through how to install Hyperledger Fabric on **Amazon Web Services** (**AWS**).

Installing Hyperledger Fabric on AWS

To install and run the recipe in this chapter, you need AWS EC2 Ubuntu Server 16.04 with 4 GB of memory. We will use the Fabric 1.3 release as it is the most stable release as of writing this recipe.

Getting ready

From the Hyperledger Fabric website (https://hyperledger-fabric.readthedocs.io/en/release-1.3/prereqs.html), the prerequisites for this recipe are as follows:

- **Operating systems**: Ubuntu Linux 14.04 / 16.04 LTS (both 64-bit), or macOS 10.12
- **cURL tool**: The latest version
- **Docker engine**: Version 17.06.2-ce or greater
- **Docker-compose**: Version 1.14 or greater
- **Go**: Version 1.10.x
- **Node**: Version 8.9 or higher (note: version 9 is not supported)
- **npm**: Version 5.x
- **Python**: 2.7.x

We chose Amazon Ubuntu Server 16.04. If you don't have experience with installing Ubuntu in EC2, please refer to the AWS document: https://aws.amazon.com/getting-started/tutorials/launch-a-virtual-machine/.

You can also chose to install Ubuntu in your local machine virtual box. A tutorial for this can be found at http://www.psychocats.net/ubuntu/virtualbox or https://askubuntu.com/questions/142549/how-to-install-ubuntu-on-virtualbox.

How to do it...

To install Hyperledger on AWS, follow these steps:

1. Execute the following commands to update the software on your system:

```
$ sudo apt-get update
```

2. Install curl and the golang software package:

```
$ sudo apt-get install curl
$ sudo apt-get install golang
$ export GOPATH=$HOME/go
$ export PATH=$PATH:$GOPATH/bin
```

3. Install Node.js, npm, and Python:

```
$ sudo apt-get install nodejs
$ sudo apt-get install npm
$ sudo apt-get install python
```

4. Install and upgrade docker and docker-compose:

```
$ sudo apt-get install docker
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
  sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64]
  https://download.docker.com/linux/ubuntu
$(lsb_release -cs) stable"
$ sudo apt-get update
$ apt-cache policy docker-ce
$ sudo apt-get install -y docker-ce
$ sudo apt-get install docker-compose
$ sudo apt-get upgrade
```

5. Let's customize and update Node.js and golang to the proper versions:

```
$ wget https://dl.google.com/go/go1.11.2.linux-amd64.tar.gz
$ tar -xzvf go1.11.2.linux-amd64.tar.gz
$ sudo mv go/ /usr/local
$ export GOPATH=/usr/local/go
$ export PATH=$PATH:$GOPATH/bin
$ curl -sL https://deb.nodesource.com/setup_8.x | sudo bash -
$ sudo apt-get install -y nodejs
```

6. Verify the installed software package versions:

```
$ curl --version
$ /usr/local/go/bin/go version
$ python -V
$ node -v
$ npm -version
$ docker --version
$ docker-compose --version
```

The result should look like this:

```
curl 7.47.0 (x86_64-pc-linux-gnu) libcurl/7.47.0 GnuTLS/3.4.10 zlib/1.2.8 libidn/1.32 librtmp/2.3
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 pop3s rtmp rtsp smb smbs smtp smtps telnet tftp
Features: AsynchDNS IDN IPv6 Largefile GSS-API Kerberos SPNEGO NTLM NTLM_WB SSL libz TLS-SRP UnixSockets
go version go1.11.2 linux/amd64
Python 2.7.12
v8.15.0
6.4.1
Docker version 18.09.0, build 4d60db4
docker-compose version 1.8.0, build unknown
```

7. Install Hyperledger Fabric 1.3:

```
$ curl -sSL http://bit.ly/2ysbOFE | sudo bash -s 1.3.0
```

It will take a few minutes to download the Docker images. When it is done, the results should look like this:

```
===> List out hyperledger docker images
hyperledger/fabric-ca          1.4.0-rc2           921e03d2731e        2 weeks ago         244MB
hyperledger/fabric-ca          latest              921e03d2731e        2 weeks ago         244MB
hyperledger/fabric-zookeeper   0.4.14              d36da0db87a4        2 months ago        1.43GB
hyperledger/fabric-zookeeper   latest              d36da0db87a4        2 months ago        1.43GB
hyperledger/fabric-kafka       0.4.14              a3b095201c66        2 months ago        1.44GB
hyperledger/fabric-kafka       latest              a3b095201c66        2 months ago        1.44GB
hyperledger/fabric-couchdb     0.4.14              f14f97292b4c        2 months ago        1.5GB
hyperledger/fabric-couchdb     latest              f14f97292b4c        2 months ago        1.5GB
hyperledger/fabric-javaenv     1.3.0               2476cefaf833        2 months ago        1.7GB
hyperledger/fabric-javaenv     latest              2476cefaf833        2 months ago        1.7GB
hyperledger/fabric-tools       1.3.0               c056cd9890e7        2 months ago        1.5GB
hyperledger/fabric-tools       latest              c056cd9890e7        2 months ago        1.5GB
hyperledger/fabric-ccenv       1.3.0               953124d80237        2 months ago        1.38GB
hyperledger/fabric-ccenv       latest              953124d80237        2 months ago        1.38GB
hyperledger/fabric-orderer     1.3.0               f430f581b46b        2 months ago        145MB
hyperledger/fabric-orderer     latest              f430f581b46b        2 months ago        145MB
hyperledger/fabric-peer        1.3.0               f3ea63abddaa        2 months ago        151MB
hyperledger/fabric-peer        latest              f3ea63abddaa        2 months ago        151MB
```

This completes the installation of the Hyperledger Fabric on the AWS EC2 machine.

We will build up the network in the next recipe.

How it works...

We installed several prerequisites, so let's explain what each software package is and how they work together to build the Hyperledger Fabric platform:

- **cURL**: A tool used to transfer data from or to a server, using one of the supported protocols (HTTP, HTTPS, FTP, FTPS, SCP, SFTP, TFTP, DICT, TELNET, LDAP, or FILE). The command is designed to work without user interaction.
- **Docker**: A tool to create, deploy, and run applications using containers. Containers allow developers to package applications with all of the parts it needs, such as libraries and other dependencies, and ship it out as one package.
- **Docker Compose**: It is a tool which is used for defining and running Multi-container application. You can create and start all the services with help of a single command from your configuration YAML file.
- **Go:** An open source programming language that makes it easy to build simple, reliable, and efficient software. Hyperledger Fabric is primarily developed using the Go language.
- **Node.js**: A platform built on Chrome's JavaScript runtime to easily build fast and scalable network applications. Node.js is considered to be more lightweight and efficient since it uses event-driven, non-blocking I/O models, which make it more feasible for data-intensive real-time applications.
- **npm package manager**: A tool that will allow you to install third-party libraries (other people's code) using the command line.

- **Python**: A general-purpose programming language for developing both desktop and web applications. Python is also used to develop complex scientific and numeric applications. It is designed with features to facilitate data analysis and visualization.

With this Hyperledger Fabric installation, it will download and install samples and binaries to your system. The sample applications installed are useful for learning the capabilities and operations of Hyperledger Fabric:

- balance-transfer: A sample Node.js app to demonstrate fabric-client and fabric-ca-client Node.js SDK APIs.
- basic-network: A basic network with certificates and key materials, predefined transactions, and one channel, mychannel.
- bin: Binary and scripts for fabric-ca, orderer, and peer.
- chaincode: Chaincode developed for fabcar, marbles, and a few other examples.
- chaincode-docker-devmode: Develops chaincode in dev mode for rapid code/build/run/debug.
- config: YAML files to define transaction, orderer, organization, and chaincode.
- fabcar: A sample Node.js app to demonstrate the capabilities with chaincode deployment, query, and updating the ledger.
- fabric-ca: Uses the Fabric CA client and server to generate all crypto material and learn how to use attribute-based access control.
- first-network: Builds the first hyperledger fabric network with byfn.sh and eyfn.sh.
- Jenkinsfile: Jenkins is a suite of plugins that supports implementing and integrating continuous-delivery pipelines. The definition of a Jenkins pipeline is typically written into a text file, Jenkinsfile, which in turn is checked into a project's source-control repository.
- scripts: There are two scripts in this directory: bootstrap.sh and Jenkins_Scripts.

Now that we have successfully installed Hyperledger Fabric on an AWS EC2 virtual machine, in the next recipe, we will set up the first Hyperledger Fabric network.

Building the Fabric network

To run this recipe, you need to complete the **Reviewing the Hyperledger Fabric architecture and components** recipe in this chapter to install Hyperledger Fabric with samples and binaries on the AWS EC2 instance.

How to do it...

There is a **Build your first network** (**BYFN**) sample installed with Hyperledger Fabric. We will use that to provision a sample Hyperledger Fabric network that consists of two organizations, each maintaining two peer nodes, and a solo ordering service. To do this, follow these steps:

1. Log in as a default user and execute the byfn.sh script to generate certificates and keys for the network:

```
$ cd ~
$ sudo chmod 777 -R fabric-samples
$ cd fabric-samples/first-network
$ sudo ./byfn.sh generate
```

2. Bring up the Fabric network by executing the byfn.sh script using the up option:

```
$ cd ~
$ cd fabric-samples/first-network
$ sudo ./byfn.sh up
```

You should see the following output, which states that the network has started successfully:

```
ubuntu@ip-172-31-78-117:~/fabric-samples/first-network$ sudo ./byfn.sh up
Starting for channel 'mychannel' with CLI timeout of '10' seconds and CLI delay of '3' seconds
Continue? [Y/n] Y
proceeding ...
LOCAL_VERSION=1.3.0
DOCKER_IMAGE_VERSION=1.3.0
Creating network "net_byfn" with the default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating volume "net_peer1.org2.example.com" with default driver
Creating volume "net_peer1.org1.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_orderer.example.com" with default driver
Creating peer0.org2.example.com
Creating peer1.org1.example.com
Creating peer0.org1.example.com
Creating peer1.org2.example.com
Creating orderer.example.com
Creating cli


 ____    _____      _      ____    _____
/ ___|  |_   _|    / \    |  _ \  |_   _|
\___ \    | |     / _ \   | |_) |   | |
 ___) |   | |    / ___ \  |  _ <    | |
|____/    |_|   /_/   \_\ |_| \_\   |_|

Build your first network (BYFN) end-to-end test
```

3. Bring down the Fabric network by executing the byfn.sh script using the down option to shut down and clean up the network. This kills the containers, removes the crypto material and artifacts, and deletes the chaincode images. The following code shows how to do this:

```
$ cd ~
$ cd fabric-samples/first-network
$ sudo ./byfn.sh down
```

Let's review the byfn.sh script, shown as follows. This script is well documented, and you should read about it in detail to understand each execution step during the network startup process:

```
# Print the usage message
function printHelp() {
  echo "Usage: "
  echo "  byfn.sh <mode> [-c <channel name>] [-t <timeout>] [-d <delay>] [-f <do
  echo "    <mode> - one of 'up', 'down', 'restart', 'generate' or 'upgrade'"
  echo "      - 'up' - bring up the network with docker-compose up"
  echo "      - 'down' - clear the network with docker-compose down"
  echo "      - 'restart' - restart the network"
  echo "      - 'generate' - generate required certificates and genesis block"
  echo "      - 'upgrade'  - upgrade the network from version 1.2.x to 1.3.x"
  echo "    -c <channel name> - channel name to use (defaults to \"mychannel\")"
  echo "    -t <timeout> - CLI timeout duration in seconds (defaults to 10)"
  echo "    -d <delay> - delay duration in seconds (defaults to 3)"
  echo "    -f <docker-compose-file> - specify which docker-compose file use (de
  echo "    -s <dbtype> - the database backend to use: goleveldb (default) or co
  echo "    -l <language> - the chaincode language: golang (default) or node"
  echo "    -i <imagetag> - the tag to be used to launch the network (defaults t
  echo "    -v - verbose mode"
  echo "  byfn.sh -h (print this message)"
  echo
  echo "Typically, one would first generate the required certificates and "
  echo "genesis block, then bring up the network. e.g.:"
  echo
  echo "        byfn.sh generate -c mychannel"
  echo "        byfn.sh up -c mychannel -s couchdb"
  echo "        byfn.sh up -c mychannel -s couchdb -i 1.2.x"
  echo "        byfn.sh up -l node"
  echo "        byfn.sh down -c mychannel"
  echo "        byfn.sh upgrade -c mychannel"
  echo
  echo "Taking all defaults:"
  echo "        byfn.sh generate"
  echo "        byfn.sh up"
  echo "        byfn.sh down"
```

We will review and exam the Hyperledger Fabric byfn.sh script using the command-line interface.

4. Use the tool for crypto and certificate generation, called cryptogen, which uses a YAML configuration file as the base to generate the certificates:

```
OrdererOrgs:
  - Name: Orderer
    Domain: example.com
    Specs:
      - Hostname: orderer
PeerOrgs:
  - Name: Org1
    Domain: org1.example.com
    EnableNodeOUs: true
    Template:
      Count: 2
    Users:
      Count: 1
  - Name: Org2
    Domain: org2.example.com
    EnableNodeOUs: true
    Template:
      Count: 2
    Users:
      Count: 1
```

The following command will generate the YAML file:

```
$ cd ~
$ cd fabric-samples/first-network
$ sudo ../bin/cryptogen generate --config=./crypto-config.yaml
```

On execution of the previous command, you will find a new directory `crypto-config` is created, and inside there are directories that correspond to `ordererOrganizations` and `peerOrganizations`. We have two organizations, (`Org1.example.com` and `Org2.example.com`) network artifacts.

5. Let's generate the configuration transaction. The tool to generate the configuration transaction is called configtxgen. The artifacts generated in this step are the orderer genesis block, the channel configuration transaction, and one anchor peer transaction for each peer organization. There will also be a `configtx.yaml` file that is broken into several sections: profiles (describe the organizational structure of the network), organizations (the details regarding individual organizations), orderer (the details regarding the orderer parameters), and application (application defaults—not needed for this recipe).

The profiles that are needed for this recipe are shown as follows:

```
Profiles:

  TwoOrgsOrdererGenesis:
    <<: *ChannelDefaults
    Orderer:
      <<: *OrdererDefaults
      Organizations:
        - *OrdererOrg
      Capabilities:
        <<: *OrdererCapabilities
    Consortiums:
      SampleConsortium:
        Organizations:
          - *Org1
          - *Org2
  TwoOrgsChannel:
    Consortium: SampleConsortium
    Application:
      <<: *ApplicationDefaults
      Organizations:
        - *Org1
        - *Org2
      Capabilities:
        <<: *ApplicationCapabilities
```

Let's go with the detailed command-line steps to understand what is happening:

```
$ export FABRIC_CFG_PATH=$PWD
$ sudo ../bin/configtxgen -profile TwoOrgsOrdererGenesis -
outputBlock ./channel-artifacts/genesis.block
$ export CHANNEL_NAME=mychannel
$ sudo ../bin/configtxgen -profile TwoOrgsChannel
 -outputCreateChannelTx ./channel-artifacts/channel.tx
 -channelID $CHANNEL_NAME
$ sudo ../bin/configtxgen -profile TwoOrgsChannel
 -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx
 -channelID $CHANNEL_NAME -asOrg Org1MSP
$ sudo ../bin/configtxgen -profile TwoOrgsChannel
 -outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchors.tx
 -channelID $CHANNEL_NAME -asOrg Org2MSP
```

Here, we write the blockchain genesis block, create the first channel transaction, and write anchor peer updates. You may not care how exactly it is done, but this is how Fabric is built from the bottom up. You can see that four new files are generated and stored in the channel-artifacts directory:

- genesis.block
- channel.tx
- Org1MSPanchors.tx
- Org2MSPanchors.tx

6. The Docker Compose tool is used to bring up Docker containers. We use docker-compose-cli.yaml to keep track of all Docker containers that we bring up:

```
$ cd ~
$ cd fabric-samples/first-network
$ sudo docker-compose -f docker-compose-cli.yaml up -d
```

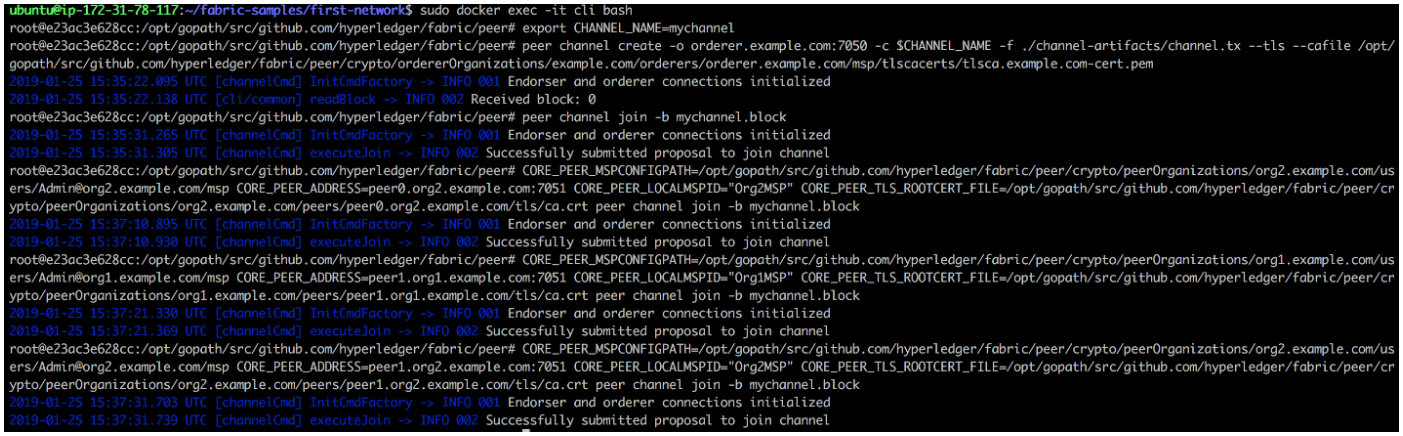
7. We have brought up six nodes: cli, orderer.example.com, peer0.org1.example.com, peer0.org2.example.com, peer1.org1.example.com, and peer1.org2.example.com:

```
ubuntu@ip-172-31-78-117:~/fabric-samples/first-network$ sudo docker-compose -f docker-compose-cli.yaml up -d
Creating network "net_byfn" with the default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating volume "net_peer1.org2.example.com" with default driver
Creating volume "net_peer1.org1.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_orderer.example.com" with default driver
Creating peer1.org1.example.com
Creating peer1.org2.example.com
Creating peer0.org2.example.com
Creating peer0.org1.example.com
Creating orderer.example.com
Creating cli
```

8. Use the peer CLI to set up the network. Using the peer command line within the Docker CLI container for this step, we will create the channel using channel.tx so that peers can join the channel. Please note that some commands are extremely long as we need to set up peer environment variables (note that the default is peer0.org1), as follows:

```
$ cd ~
$ cd fabric-samples/first-network
$ sudo docker exec -it cli bash
$ export CHANNEL_NAME=mychannel

$ 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
$ peer channel join -b mychannel.block

// for peer0.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:7051
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

// for peer1.org1
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=peer1.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/peer1.org1.example.com/tls/ca.crt
peer channel join -b mychannel.block

// for peer1.org2
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.co
m/msp CORE_PEER_ADDRESS=peer1.org2.example.com:7051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.c
om/tls/ca.crt peer channel join -b mychannel.block
```

This will create a connection between all four peers:



9. Update the anchor peer on each organization. We use the files we created in the **Installing Hyperledger Fabric on AWS** section (Org1MSPanchors.tx and Org2MSPanchors.tx) and apply them to Peer0 of both Org1 and Org2:

```
$ 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

$ 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:7051
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
```

10. Using the CLI, we need to install the chaincode to peer0Org1 and peer0Org2. The chaincode is specified in the -p option in the command and the chaincode name is mycc. This is shown in the following code:

```
$ peer chaincode install -n mycc -v 1.0 -p
github.com/chaincode/chaincode_example02/go/
```

```
$
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.co
m/msp CORE_PEER_ADDRESS=peer0.org2.example.com:7051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.c
om/tls/ca.crt peer chaincode install -n mycc -v 1.0 -p
github.com/chaincode/chaincode_example02/go//orderers/orderer.example.com/
msp/tlscacerts/tlsca.example.com-cert.pem
```

11. Instantiate the chaincode from peer0.org2. We will use -c to initialize this with a value of 100, and b with 200. We use -p to define the endorsement policy. This is shown in the following code:

```
$
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.co
m/msp CORE_PEER_ADDRESS=peer0.org2.example.com:7051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.c
om/tls/ca.crt peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganization
s/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')"
```

12. Execute a query on peer0Org1 on the a value. We should get the correct value of 100 back:

```
$ peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

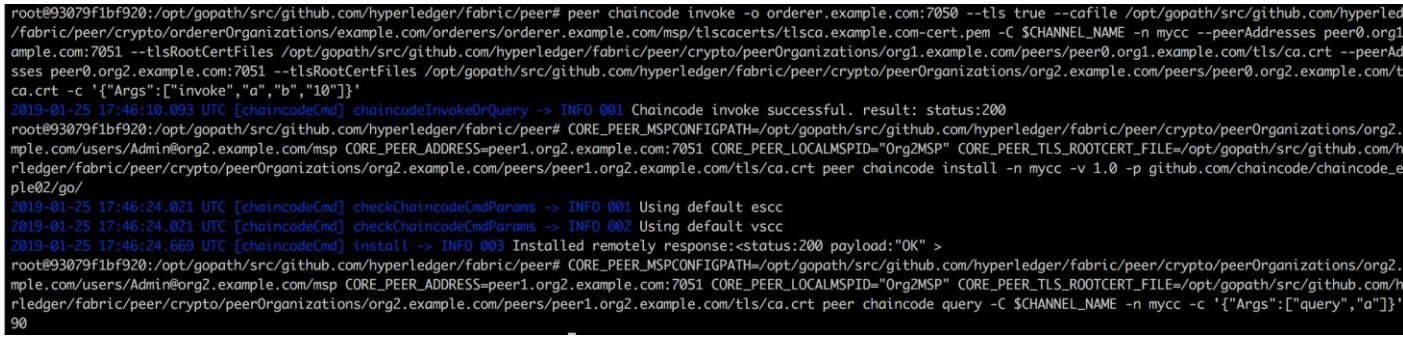
```
2019-01-25 16:52:06.751 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using default vscc
Error: could not assemble transaction, err proposal response was not successful, error code 500, msg chaincode with name 'mycc' already exists
root@c4dedc504f0a:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
100
```

13. Using the CLI, create a transaction by invoking chaincode. In this example, we will move 10 from a to b. Install chaincode on peer1org2, and then query from peer1org2 for the latest value of a:

```
$
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:7051 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 chaincode install -n mycc -v 1.0 -p github.com/chaincode/chaincode_example02/go/

$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:7051 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 chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

This takes some time, but we will eventually receive the result of 90, which is correct after 10 is removed from 100:



This concludes building our first Fabric network. We will look at how to make changes to the existing network and add an organization to a channel in the next recipe.

How it works...

We covered the following steps to build our Fabric network:

- Generating the crypto/certificate using cryptogen

- Generating the configuration transaction using configtxgen
- Bring up the nodes based on what is defined in the docker-compose file
- Using the CLI to set up the first network
- Using the CLI to install and instantiate the chaincode
- Using the CLI to invoke and query the chaincode

This recipe helps you to understand the Hyperledge Fabric components and shows how we can quickly set up a Hyperledger Fabric network using sample chaincode (mycc). You should be able to modify the scripts and run other samples, such as fabcar and marble02, which are provided under the fabric-sample/chaincode directory.

Fabric provides the following commands used in the byfn.sh script. In the following chapters and recipes, these commands will be used to operate and manage the Fabric network environment:

- **peer**: Operates and configures a peer
- **peer chaincode**: Manages chaincode on the peer
- **peer channel**: Manages channels on the peer
- **peer node**: Manages the peer
- **peer version**: Returns the peer version
- **cryptogen**: Utility for generating crypto material
- **configtxgen**: Creates configuration data, such as the genesis block
- **configtxlator**: Utility for generating channel configurations
- **fabric-ca-client**: Manage identities
- **fabric-ca-server**: Manages the fabric-ca server

Now that we have set up our first network, let's add an organization to the channel.

Adding an organization to a channel

This recipe serves as an extension to the BYFN recipe. We will demonstrate how to add a new organization – Org3 – to the application's channel (mychannel).

Getting ready...

To run this recipe, you need complete the **Reviewing the Hyperledger Fabric architecture and components** recipe in this chapter to install Hyperledger Fabric with samples and binaries on the AWS EC2 instance.

How to do it...

Since we need add the new organization, Org3, to BYFN, we will first bring up the BYFN network. Follow these steps:

1. Bring up the first network using the following command:

```
$ cd ~
$ cd fabric-samples/first-network
$ sudo ./byfn.sh generate
$ sudo ./byfn.sh up
```

2. Execute the script to add Org3 into the mychannel channel:

```
$ cd ~
  $ cd fabric-samples/first-network
  $ sudo ./eyfn.sh up
```

The following screenshot confirms org3 is added to mychannel successfully:

```
========= Finished adding Org3 to your first network! =========


 _____    _____       _      ____    _____
/ ___|  |_   _|     / \    |  _ \  |_   _|
\___ \    | |      / _ \   | |_) |   | |
 ___) |   | |     / ___ \  |  _ <    | |
|____/    |_|    /_/   \_\ |_| \_\   |_|

Extend your first network (EYFN) test

Channel name : mychannel
Querying chaincode on peer0.org3...
===================== Querying on peer0.org3 on channel 'mychannel'... =====================
Attempting to Query peer0.org3 ...3 secs
+ peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
+ res=0
+ set +x

90
===================== Query successful on peer0.org3 on channel 'mychannel' =====================
```

We can test this by running a query against Org3peer0.

3. To shut down and clean up the network, execute the following:

```
$ cd fabric-samples/first-network
$ sudo ./eyfn.sh down
$ sudo ./byfn.sh down
```

How it works...

Like what we did in the **Building the Fabric network** recipe, the eyfn.sh script is a good resource to understand how things work.

We will also look into the command-line steps to see the internal building blocks to add an organization to a channel:

```
# Print the usage message
function printHelp () {
  echo "Usage: "
  echo "  eyfn.sh up|down|restart|generate [-c <channel name>] [-t <timeout>] [-d <delay>] [-f <docker-compose-file>] [-s <db
  echo "  eyfn.sh -h|--help (print this message)"
  echo "    <mode> - one of 'up', 'down', 'restart' or 'generate'"
  echo "      - 'up' - bring up the network with docker-compose up"
  echo "      - 'down' - clear the network with docker-compose down"
  echo "      - 'restart' - restart the network"
  echo "      - 'generate' - generate required certificates and genesis block"
  echo "    -c <channel name> - channel name to use (defaults to \"mychannel\")"
  echo "    -t <timeout> - CLI timeout duration in seconds (defaults to 10)"
  echo "    -d <delay> - delay duration in seconds (defaults to 3)"
  echo "    -f <docker-compose-file> - specify which docker-compose file use (defaults to docker-compose-cli.yaml)"
  echo "    -s <dbtype> - the database backend to use: goleveldb (default) or couchdb"
  echo "    -l <language> - the chaincode language: golang (default) or node"
  echo "    -i <imagetag> - the tag to be used to launch the network (defaults to \"latest\")"
  echo "    -v - verbose mode"
  echo
  echo "Typically, one would first generate the required certificates and "
  echo "genesis block, then bring up the network. e.g.:"
  echo
  echo "        eyfn.sh generate -c mychannel"
  echo "        eyfn.sh up -c mychannel -s couchdb"
  echo "        eyfn.sh up -l node"
  echo "        eyfn.sh down -c mychannel"
  echo
  echo "Taking all defaults:"
  echo "        eyfn.sh generate"
  echo "        eyfn.sh up"
  echo "        eyfn.sh down"
}
```

4. Generate the org3 certificates:

```
$ cryptogen generate --config=./org3-crypto.yaml
```

5. Generate the org3 configuration materials:

```
$ configtxgen -printOrg Org3MSP
```

6. Generate and submit the transaction configuration for organization 3:

```
$ peer channel fetch config config_block.pb -o
 orderer.example.com:7050 -c mychannel --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
 $ configtxlator proto_encode --input config.json
```

```
 --type common.Config
$ configtxlator proto_encode --input modified_config.json
 --type common.Config
$ configtxlator compute_update --channel_id mychannel
--original original_config.pb --updated modified_config.pb
$ configtxlator proto_decode --input config_update.pb
--type common.ConfigUpdate
```

7. Configure the transaction to add org3, which has been created:

```
$ peer channel signconfigtx -f org3_update_in_envelope.pb
```

8. Submit the transaction from a different peer (peer0.org2), who also signs it:

```
$ peer channel update -f org3_update_in_envelope.pb -c mychannel -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
```

9. Get the org3 peer to join the network:

```
$ peer channel fetch 0 mychannel.block -o orderer.example.com:7050
-c mychannel --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
$ peer channel join
 -b mychannel.blockcd fabric-samples/first-network
```

10. Install and update the chaincode:

```
 $ peer chaincode install -n mycc -v 2.0 -l golang -p
 github.com/chaincode/chaincode_example02/go/
$ peer chaincode upgrade -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 mychannel -n mycc -v 2.0 -c
'{"Args":["init","a","90","b","210"]}'
-P 'AND ('\''Org1MSP.peer'\','\''Org2MSP.peer'\',
 '\''Org3MSP.peer'\'')'
```

11. Query peer0org3:

```
$ peer chaincode query -C mychannel -n mycc
-c '{"Args":["query","a"]}'
```

12. Invoke the transaction to move 10 from a to b again on a different peer:

```
$ 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 mychannel -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:7051
--tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/
fabric/peer/crypto/peerOrganizations/org2.example.com/peers/
 peer0.org2.example.com/tls/ca.crt
--peerAddresses peer0.org3.example.com:7051
--tlsRootCertFiles/opt/gopath/src/github.com/hyperledger/
fabric/peer/crypto/peerOrganizations/org3.example.com/peers/
peer0.org3.example.com/tls/ca.crt
-c '{"Args":["invoke","a","b","10"]}'
```

This concludes how to add an organization to an existing network in a channel. We will look at how to use CouchDB to review transactions in the next recipe.

Following all the previous steps will create our first network, which consists of two organizations, two peers per organization, and single Solo ordering service. In this recipe, we showed you how to add a third organization to an application channel with its own peers to an already running first network, and then join it to the new channel.

When you view the log file, you will be able to see details in the following order:

- Generating Org3 config material
- Generating and submitting config tx to add Org3
- Creating config transaction to add Org3 to the network
- Installing jq
- Config transaction to add Org3 to the network
- Signing the config transaction
- Submitting the transaction from a different peer (peer0.org2), which also signs it
- Configure transaction to add Org3 to network submitted
- Having Org3 peers join the network
- Getting Org3 on to your first network
- Fetching the channel config block from orderer
- peer0.org3 joined the mychannel channel
- peer1.org3 joined the mychannel channel
- Installing chaincode 2.0 on peer0.org3
- Upgrading chaincode to have Org3 peers on the network
- Finishing adding Org3 to your first network
- Chaincode is installed on peer0.org1
- Chaincode is installed on peer0.org2
- Chaincode is upgraded on peer0.org1 on the mychannel channel
- Finished adding Org3 to your first network!

Updating modification policies or altering batch sizes or any other channel configuration can be updated using the same approach but for now we will focus solely on the integration of a new organization.

There's more...

The following block shows the `org3-crypto.yaml` section for `Org3`:

```
# ---------------------------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
# ---------------------------------------------------------------------------
PeerOrgs:
# ---------------------------------------------------------------------------
# Org3
```

```
# ---------------------------------------------------------------------------
- Name: Org3
Domain: org3.example.com
EnableNodeOUs: true
Template:
Count: 2
Users:
Count: 1
```

The following block shows the configtx.yaml section for Org3:

```
################################################################################
#################
# Section: Organizations
#
# - This section defines the different organizational identities which will
# be referenced later in the configuration.
#
################################################################################
################
Organizations:
- &Org3
# DefaultOrg defines the organization which is used in the sampleconfig
# of the fabric.git development environment
Name: Org3MSP
# ID to load the MSP definition as
ID: Org3MSP
MSPDir: crypto-config/peerOrganizations/org3.example.com/msp
AnchorPeers:
# AnchorPeers defines the location of peers which can be used for cross org gossip
#communication. Note, this value is only
# encoded in the genesis block in the Application section context
- Host: peer0.org3.example.com
```

Writing your first application

In this recipe, we will explore how to create a smart contract and then deploy it into the blockchain.

To run this recipe, you need to have completed the **Installing Hyperledger Fabric on AWS** recipe in this chapter to install Hyperledger Fabric with samples and binaries on the AWS EC2 instance.

How to do it...

To write your first application, follow these steps:

1. Set up the development environment:

```
$ cd ~
cd fabric-samples/first-network
sudo docker ps
sudo ./byfn.sh down
sudo docker rm -f $(sudo docker ps -aq)
sudo docker network prune
cd ../fabcar && ls
```

You will notice that there are a few**Node.js** file present in `fabcar` folder such as `enrollAdmin.js`, `invoke.js`, `query.js`, `registerUser.js`, and `package.json` and all others packaged into one `startFabric.sh` file.

2. Install the Fabric client:

```
$ sudo npm install -g npm@5.3.0
$ sudo npm update
```

You will notice from the following screenshot that the Fabric client 1.3.0 and Fabric CA client 1.30 packages are installed:

```
  CXX(target) Release/obj.target/pkcs11/src/pkcs11/param_ecdh.o
  CXX(target) Release/obj.target/pkcs11/src/pkcs11/pkcs11.o
  CXX(target) Release/obj.target/pkcs11/src/async.o
  CXX(target) Release/obj.target/pkcs11/src/node.o
  SOLINK_MODULE(target) Release/obj.target/pkcs11.node
  COPY Release/pkcs11.node
make: Leaving directory '/home/ubuntu/fabric-samples/fabcar/node_modules/pkcs11js/build'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN ajv-keywords@2.1.1 requires a peer of ajv@^5.0.0 but none was installed.
npm WARN fabcar@1.0.0 No repository field.

+ grpc@1.18.0
+ fabric-ca-client@1.3.0
+ fabric-client@1.3.0
added 743 packages in 44.441s
```

3. Execute the following command to launch the network:

```
$ sudo ./startFabric.sh node
```

4. Open a new Terminal to stream the Docker logs:

```
$ sudo docker logs -f ca.example.com
```

This will open the Docker file, which will look similar to the following screenshot:

```
# don't rewrite paths for Windows Git Bash users
export MSYS_NO_PATHCONV=1

docker-compose -f docker-compose.yml down
Removing network net_basic
WARNING: Network net_basic not found.

docker-compose -f docker-compose.yml up -d ca.example.com orderer.example.com peer0.org1.example.com couchdb
Creating network "net_basic" with the default driver
Creating orderer.example.com
Creating couchdb
Creating ca.example.com
Creating peer0.org1.example.com

# wait for Hyperledger Fabric to start
# incase of errors when running later commands, issue export FABRIC_START_TIMEOUT=<larger number>
export FABRIC_START_TIMEOUT=10
#echo ${FABRIC_START_TIMEOUT}
sleep ${FABRIC_START_TIMEOUT}

# Create the channel
docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/users/Admin@org1.example.com/msp" peer0.org1.
example.com peer channel create -o orderer.example.com:7050 -c mychannel -f /etc/hyperledger/configtx/channel.tx
2019-01-05 19:27:29.498 UTC [channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
2019-01-05 19:27:29.544 UTC [cli/common] readBlock -> INFO 002 Received block: 0
# Join peer0.org1.example.com to the channel.
docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/users/Admin@org1.example.com/msp" peer0.org1.
example.com peer channel join -b mychannel.block
2019-01-05 19:27:29.963 UTC [channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
2019-01-05 19:27:30.073 UTC [channelCmd] executeJoin -> INFO 002 Successfully submitted proposal to join channel
Creating cli
2019-01-05 19:27:31.654 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using default escc
2019-01-05 19:27:31.654 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using default vscc
2019-01-05 19:27:31.680 UTC [chaincodeCmd] install -> INFO 003 Installed remotely response:<status:200 payload:"OK" >
2019-01-05 19:27:32.002 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using default escc
2019-01-05 19:27:32.002 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using default vscc
2019-01-05 19:28:12.288 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Chaincode invoke successful. result: status:200

Total setup execution time : 58 secs ...


Start by installing required packages run 'npm install'
Then run 'node enrollAdmin.js', then 'node registerUser'

The 'node invoke.js' will fail until it has been updated with valid arguments
The 'node query.js' may be run at anytime once the user has been registered
```

Next, we will use the Node.js script to run, query, and update the records on Fabric network.

Accessing the API with SDK

In this recipe, the application uses an SDK to access the APIs that permit queries and updates to the ledger. Now we will perform the following steps:

1. Enroll an admin user with the enrollAdmin.js script:

```
$ sudo node enrollAdmin.js
```

When we launch the network, an admin user needs to be registered with certificate authority. We send an enrollment call to the CA server and retrieve the **enrollment certificate** (**eCert**) for this user. We then use this admin user to subsequently register and enroll other users:

```
ubuntu@ip-172-31-45-218:~/fabric-samples/fabcar$ sudo node enrollAdmin.js
 Store path:/home/ubuntu/fabric-samples/fabcar/hfc-key-store
(node:4356) DeprecationWarning: grpc.load: Use the @grpc/proto-loader module with grpc.loadPackageDefinition instead
Successfully enrolled admin user "admin"
Assigned the admin user to the fabric client ::{"name":"admin","mspid":"Org1MSP","roles":null,"affiliation":"","enrollmentSecret":"","enrollment":{"signingIdentity":"8312cd0dbdeeddf02397eaab8e36ff1f4b44507777313e55667d68e9dcce5853","identity":{"certificate":"-----BEGIN CERTIFICATE-----\nMIICATCCAaigAwIBAgIUE2CeOFH7I6tsjOxaTXWmd1uFeMkwCgYIKoZIzj0EAwIw\nczELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcTDVNh\nbiBGcmFuY2lzY28xGTAXBgNVBAoTEG9yZzEuZXhhbXBsZS5jb20xHDAaBgNVBAMT\nE2NhLm9yZzEuZXhhbXBsZS5jb20wHhcNMTkwMTA1MTkzNjAwWhcNMjAwMTA1MTk0\nMTAwWjAhMQ8wDQYDVQQLEwZjbGllbnQxDjAMBgNVBAMTBWFkbWluMFkwEwYHKoZI\nzj0CAQYIKoZIzj0DAQcDQgAEjEewCJzMAxWIAvnELX1tTRxAZfY3aOUZG1wUlA7w\nxR9q1XOfUG15+doDN7YCYSLW4lEGa+QaW8b4CqNNt/irIaNsMGowDgYDVR0PAQH/\nBAQDAgeAMAwGA1UdEwEB/wQCMAAwHQYDVR0OBBYEFCxsDPYQBdFkAA+lcnGmqOpz\nnq7tcMCsGA1UdIwQkMCKAIEI5qg3NdtruuLoM2nAYUdFFBNMarRst3dusalc2Xkl8\nnMAoGCCqGSM49BAMCA0cAMEQCID9+iCfAq4xkrsUNphUCGJMqb0dbSENezFXIZgFy\nnKdRUAiBOuIC8SWrNMHA3gPTg6tJA1xS6+tut0mK6rpzZB5hm4Q==\n-----END CERTIFICATE-----\n"}}}
```

2. Register and enroll a user called user1 using the registerUser.js script:

```
$ sudo node registerUser.js
```

3. With the newly-generated eCert for the admin user, let's communicate with the CA server once more to register and enroll user1. We can use the ID of user1 to query and update the ledger:

```
 Store path:/home/ubuntu/fabric-samples/fabcar/hfc-key-store
(node:4370) DeprecationWarning: grpc.load: Use the @grpc/proto-loader module with grpc.loadPackageDefinition instead
Successfully loaded admin from persistence
Successfully registered user1 - secret:ZLBOPPJGbJOY
Successfully enrolled member user "user1"
User1 was successfully registered and enrolled and is ready to interact with the fabric network
```

4. Let's run a query against the ledger:

```
$ sudo node query.js
```

5. It returns the following screenshot. You will find that there are 10 cars on the network, from CAR0 to CAR9. Each has a color, doctype, make, model, and owner:

```
Store path:/home/ubuntu/fabric-samples/fabcar/hfc-key-store
(node:4384) DeprecationWarning: grpc.load: Use the @grpc/proto-loader module with grpc.loadPackageDefinition instead
Successfully loaded user1 from persistence
Query has completed, checking results
Response is  [{"Key":"CAR0","Record":{"color":"blue","docType":"car","make":"Toyota","model":"Prius","owner":"Tomoko"}},{"Key":"CAR1","Record":{"color":"red","docType":"car","make":"Ford","model":"Mustang","owner":"Brad"}},{"Key":"CAR2","Record":{"color":"green","docType":"car","make":"Hyundai","model":"Tucson","owner":"Jin Soo"}},{"Key":"CAR3","Record":{"color":"yellow","docType":"car","make":"Volkswagen","model":"Passat","owner":"Max"}},{"Key":"CAR4","Record":{"color":"black","docType":"car","make":"Tesla","model":"S","owner":"Adriana"}},{"Key":"CAR5","Record":{"color":"purple","docType":"car","make":"Peugeot","model":"205","owner":"Michel"}},{"Key":"CAR6","Record":{"color":"white","docType":"car","make":"Chery","model":"S22L","owner":"Aarav"}},{"Key":"CAR7","Record":{"color":"violet","docType":"car","make":"Fiat","model":"Punto","owner":"Pari"}},{"Key":"CAR8","Record":{"color":"indigo","docType":"car","make":"Tata","model":"Nano","owner":"Valeria"}},{"Key":"CAR9","Record":{"color":"brown","docType":"car","make":"Holden","model":"Barina","owner":"Shotaro"}}]
```

6. The following chaincode constructs the query using the queryAllCars function to query all cars:

```
 // queryCar chaincode function - requires 1 argument,
   ex: args: ['CAR4'],
// queryAllCars chaincode function - requires no arguments,
   ex: args: [''],
 const request = {
   //targets : --- letting this default to the
    peers assigned to the channel
   chaincodeId: 'fabcar',
   fcn: 'queryAllCars',
   args: ['']
}
```

7. Update the ledger. To do this, we will update the invoke.js script. This time, the fabcar chaincode uses the createCar function to insert a new car, CAR10, into the ledger:

```
var request = {
  //targets: let default to the peer assigned to the client
  chaincodeId: 'fabcar',
   fcn: 'createCar',
  args: ['CAR10', 'Chevy', 'Volt', 'Red', 'Nick'],
  chainId: 'mychannel',
  txId: tx_id
};

sudo node invoke.js
```

Here we will complete the transaction when CAR10 is created.

8. Execute a query to verify the changes made. Change query.js using the queryCar function to query CAR10:

```
var request = {
 //targets: let default to the peer assigned to the client
 chaincodeId: 'fabcar',
 fcn: 'queryCar',
 args: ['CAR10'],
 chainId: 'mychannel',
```

```
  txId: tx_id
};
```

9. Run query.js again. We can now extract CAR10 from the ledger with the response as {"color":"Red","docType":"car","make":"Chevy","model":"Volt","owner":"Nick"}:

```
sudo node query.js
```

This will result in the following query:

```
Store path:/home/ubuntu/fabric-samples/fabcar/hfc-key-store
(node:9047) DeprecationWarning: grpc.load: Use the @grpc/proto-loader module with grpc.loadPackageDefinition instead
Successfully loaded user1 from persistence
Query has completed, checking results
Response is  {"color":"Red","docType":"car","make":"Chevy","model":"Volt","owner":"Nick"}
```

10. Shut down the Fabric network:

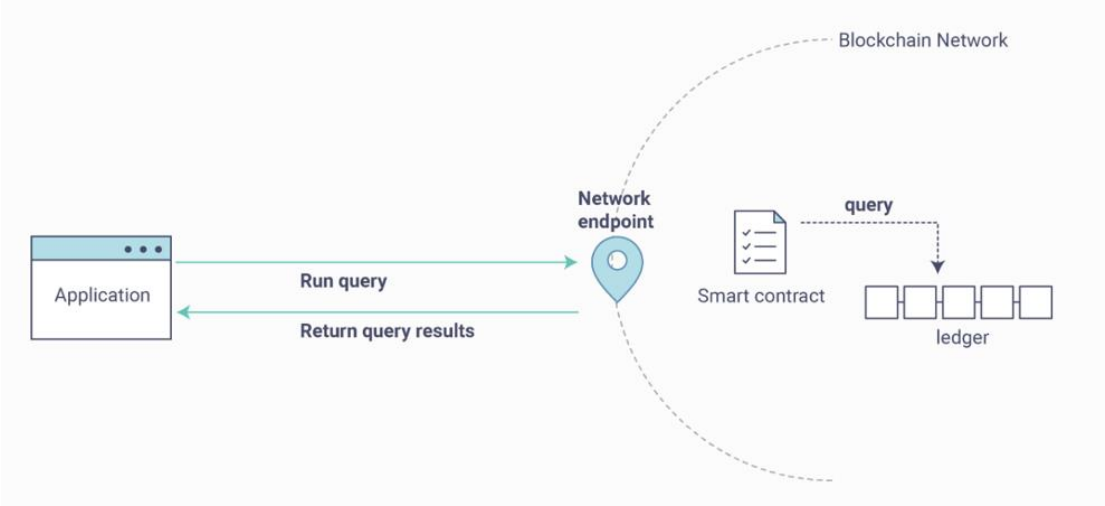
```
sudo docker stop $(sudo docker ps -a -q)
sudo docker rm $(sudo docker ps -a -q)
sudo docker ps
```

We have gone through the steps to query and update the transaction using smart contract chaincode. Now, let's see how it works under the hood.

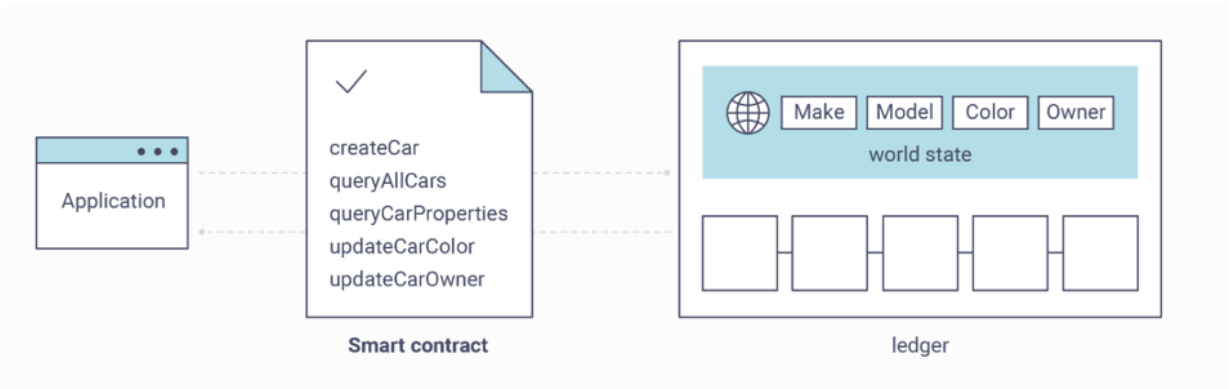
How it works...

This concludes the recipe to create and deploy your first smart contract chaincode.

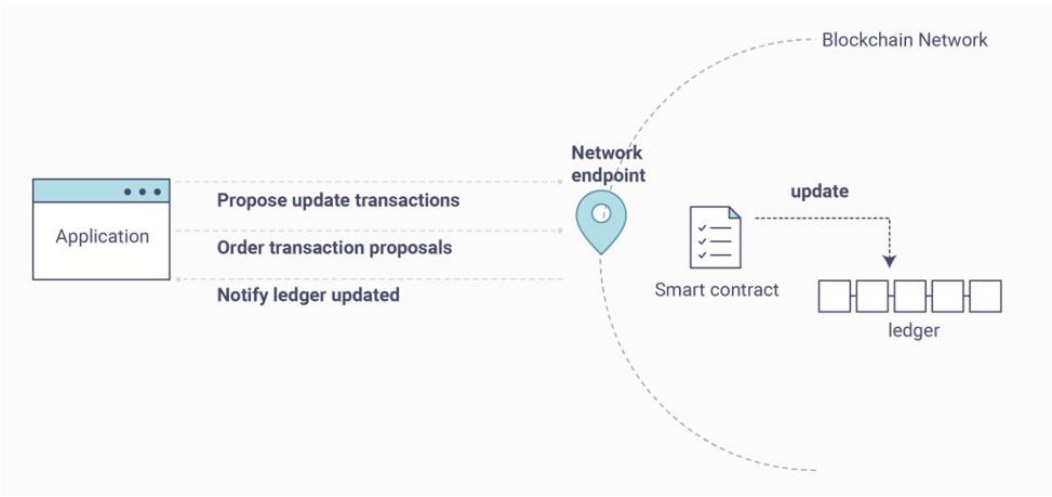
In the previous steps, we used query.js to query the key-value pair store. We can also query for the values of one or more keys, or perform complex searches on JSON data-storage formats. The following diagram shows how the query works:



The following is a representation of different functions in chaincode, which explains that we should first define the code functions to all the available APIs in the chaincode interface:



The following diagram shows the process of updating the ledger. Once an update to the ledger is proposed and endorsed, it will be returned to the application, and will in turn send the updated ledger to be ordered and written to every peer's ledger:



We learned how to write a small smart contract chaincode on the Fabric network to perform a transaction data query and update. In the next chapter, you will learn how to write an end-to-end Hyperledger Fabric application using all that we have learned in this chapter.

Implementing Hyperledger Fabric

In the previous chapter, we learned about how to set up and configure Hyperledger Fabric. We explored its key components, including channels, **Membership Service Providers** (**MSPs**), the ordering service, and Fabric **Certificate Authority** (**CA**).

In this chapter, we are going to build a simple device asset management DApp. We will exploit this example by writing chaincode implemented by various programming languages and we'll also build, test, and deploy our DApp.

First, we will look at inventory asset management, and then the rest of the chapter will be divided into the following recipes:

- Writing chaincode as a smart contract
- Compiling and deploying Fabric chaincode
- Running and testing the smart contract
- Developing an application with Hyperledger Fabric through the SDK

Inventory asset management

Blockchain technology is considered to be a game-changer for building an immutable, decentralized, trustless, and peer-to-peer ledger for business logic. Records in the blockchain are linked using cryptography. Each block contains a block timestamp, transaction data, and the previous block's cryptographic hash information.

IT asset management is an important part of an organization's strategy. It usually involves incorporating detailed IT assets and inventory information for business practices, such as hardware purchases and redistribution. Typical business practices include the request and approval process, procurement management, life cycle management, and so on.

Today, there are many participants in an asset's life cycle—from the manufacturer, the transporter, the IT service department, all the way to the end user—with each having their own management system. As a result, it's quite difficult to integrate all of these different bits of data to maintain a single version of the truth for the asset's entire life cycle.

By design, blockchain is a shared ledger technology. It is really good at registering, controlling, and transferring assets. Applying blockchain in an asset-tracking management system allows us to track digital transactions more securely and transparently. It provides new opportunities for organizations to correct problems within the asset management industry as it revolves around a **single source of truth**.

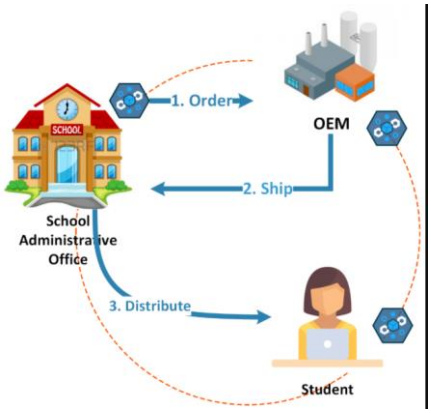
In this chapter, we will look at the processes involved in an IT asset management system. One of the main processes is tracking the complete life cycle of the assets. This includes ordering the asset, shipping the asset, receiving the asset, requesting a new asset, approving the asset, and then recycling and retiring the asset. Other record-tracking activities involve geographically locating the asset across the organization's various locations. This allows organizations to maintain the

inventory better, to identify where the asset is currently located, and which asset is available at any time.

For the sake of our demonstration, we are going to simplify the entire process, as it can be very complex in a real-world scenario. In the following school IT-asset management system, we have defined three participants: the **School Administrative Office** (**SAO**), the **original equipment manufacturer** (**OEM**), and the end user, who is a student. In this scenario, the following occurs:

1. The SAO places an **Order** to the **OEM**
2. The **OEM** receives the **Order**, makes the products, and ships the orders
3. The school receives the **Order** and distributes the products to the students

The overall process is shown in the following diagram:



In the following recipe, we will implement this flow using Fabric chaincode.

Writing chaincode as a smart contract

Chaincode in Hyperledger Fabric is similar to smart contracts. It is a program that implements the business logic and is run on top of blockchain. The application can interact with the blockchain by invoking chaincode to manage the ledger state and keep the transaction record in the ledger. This chaincode needs to be installed on each endorsing peer node that runs in a secured Docker container. The Hyperledger Fabric chaincode can be programmed in Go, Node.js, and Java.

Every chaincode program must implement the Chaincode interface. In this section, we will explore chaincode implementation using Go.

Getting ready

Working with Hyperledger Fabric, we set up our Hyperledger Fabric and the runtime environments. If you haven't done this, please revisit the previous chapter. After that, you can start following this recipe.

Writing chaincode using Go

Every chaincode needs to implement a Chaincode interface. There are two methods defined in the interface:

```
type Chaincode interface {
 Init (stub ChaincodeStubInterface) pb.Response
 Invoke (stub ChaincodeStubInterface) pb.Response
}
```

Here, the Init method is called to allow the chaincode to create an initial state and the data initialization after the chaincode container has been established for the first time. The Invoke method is called to interact with the ledger (to query or update the asset) in the proposed transaction.

ChaincodeStubInterface provides the API for apps to access and modify their ledgers. Here are some important APIs:

```
type ChaincodeStubInterface interface {
   InvokeChaincode(chaincodeName string, args [][]byte, channel string) pb.Response
   GetState(key string) ([]byte, error)
   PutState(key string, value []byte) error
   DelState(key string) error
   GetQueryResult(query string) (StateQueryIteratorInterface, error)
   GetTxTimestamp() (*timestamp.Timestamp, error)
   GetTxID() string
   GetChannelID() string
}
```

Examples of important APIs include the following:

- InvokeChaincode: Calls the chaincode function
- GetState: Returns the value of the specified key from the ledger
- PutState: Adds the key and the value to the ledger

Now that we understand some basic chaincode APIs, let's start to write our chaincode for IT asset management.

How to do it...

We will implement our school IT-asset management system using chaincode, and define the Asset object, and the Init, Invoke, and query functions. To do this, follow these steps:

1. Since we will use Go to write chaincode, install it in Unix (Ubuntu). Make sure Go version 1.10.x is installed. If you haven't yet installed Go, run the following command:

```
wget https://dl.google.com/go/go1.11.4.linux-amd64.tar.gz
sudo tar -zxvf go1.11.4.linux-amd64.tar.gz -C /usr/local/
```

2. Create a local folder called itasset and navigate to that folder:

```
mkdir ~/itasset && cd ~/itasset
```

3. To set up the PATH variable for Go, enter the following command:

```
ubuntu@ip-172-31-0-111:~$ export GOPATH=/home/ubuntu/itasset/
ubuntu@ip-172-31-0-111:~$ export
PATH=/usr/local/go/bin:$GOPATH/bin/:$PATH
ubuntu@ip-172-31-0-111:~$ cd /home/ubuntu/itasset/
ubuntu@ip-172-31-0-111:~/itasset$ mkdir -p $GOPATH/src/assetmgr
ubuntu@ip-172-31-0-111:~/itasset$ cd $GOPATH/src/assetmgr
```

4. Create the chaincode source file, assetmgr.go, for writing IT asset management:

```
touch assetmgr.go
```

5. Our assetmgr chaincode needs to implement the Chaincode interface and the business functions for IT asset management. As we discussed in the previous section, we will implement three chaincode functions in blockchain, shown as follows:

```
Order: function called by school administer to order a device from OEM
Ship: function called by OEM to transport the device to school
Distribute: function called by School to distribute the device to students.
```

Once the student receives the device, the asset management process is completed. We will keep track of the device's asset information, so we also need to define the device with related tracking information in the chaincode.

6. Based on our chaincode implementation analysis, let's define the skeleton of the AssetMgr chaincode. Define the import section:

```
package main
import (
        "encoding/json"
        "fmt"
        "github.com/hyperledger/fabric/core/chaincode/shim"
        pb "github.com/hyperledger/fabric/protos/peer"
)
type AssetMgr struct {
}
```

7. Define the asset:

```
//define organization asset information, the record can be trace in bloackchain
type OrgAsset struct {
}
```

8. Define the Init and Invoke methods:

```
func (c *AssetMgr) Init(stub shim.ChaincodeStubInterface) pb.Response {
        return shim.Success(nil)
}
func (c *AssetMgr) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
        return shim.Error("Invalid function name")
}
func (c *AssetMgr) Order(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
}
func (c *AssetMgr) Ship(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
}
func (c *AssetMgr) Distribute(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
}
```

9. Define the chaincode's main function:

```
func main() {
        err := shim.Start(new(AssetMgr))
```

```
        if err != nil {
                fmt.Printf("Error creating new AssetMgr Contract: %s", err)
        }
}
```

We have now defined our AssetMgr skeleton. Next, we need to implement all of these unimplemented functions in our chaincode. We will start by defining the OrgAsset entity.

The OrgAsset entity

All assets should have an Id field to identify them. Each device also has a physical device ID (DeviceId) that indicates the type of device, such as iPhone, iPad, or macOS. During the IT asset management flow process, the device is transferred from one entity to another, and the Location of the device keeps changing. Each processor may want to enter Comment to provide additional information at each step. Based on this, we can define the OrgAsset entity as follows:

```
type OrgAsset struct {
        Id      string `json:"id"`      //the assetId
        AssetType string `json:"assetType"` //type of device
        Status   string `json:"status"`   //status of asset
        Location  string `json:"location"`  //device location
        DeviceId  string `json:"deviceId"`  //DeviceId
        Comment   string `json:"comment"`   //comment
        From     string `json:"from"`     //from
        To      string `json:"to"`      //to
}
```

After we have defined the OrgAsset entity, we will take a look at the implementation of the Init function.

The Init function

The implementation of our Init function is as follows:

```
func (c *AssetMgr) Init(stub shim.ChaincodeStubInterface) pb.Response {
 args := stub.GetStringArgs()
 if len(args) != 3 {
```

```
  return shim.Error("Incorrect arguments. Expecting a key and a value")
 }
 assetId := args[0]
 assetType := args[1]
 deviceId := args[2]

 //create asset
 assetData := OrgAsset{
  Id: assetId,
  AssetType: assetType,
  Status: "START",
  Location: "N/A",
  DeviceId: deviceId,
  Comment: "Initialized asset",
  From: "N/A",
  To: "N/A"}
 assetBytes, _ := json.Marshal(assetData)
 assetErr := stub.PutState(assetId, assetBytes)
 if assetErr != nil {
  return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
 }
 return shim.Success(nil)
}

func (c *AssetMgr) Init(stub shim.ChaincodeStubInterface) pb.Response {
    args := stub.GetStringArgs()
        assetId := args[0]  assetType := args[1]        deviceId := args[2]
        //create asset
        assetData := OrgAsset{Id:     assetId,AssetType:
assetType,              Status:   "START",Location:  "N/A",DeviceId:
deviceId,Comment:  "Initialized asset",From:    "N/A",         To:     "N/A"}
        assetBytes, _ := json.Marshal(assetData)
        assetErr := stub.PutState(assetId, assetBytes)
...
        return shim.Success(nil)
}
```

The Invoke function

The implementation of the Invoke function is as follows:

```
func (c *AssetMgr) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
        function, args := stub.GetFunctionAndParameters()
        if function == "Order" {
                return c.Order(stub, args)
```

```
        } else if function == "Ship" {
                return c.Ship(stub, args)
        } else if function == "Distribute" {
                return c.Distribute(stub, args)
        } else if function == "query" {
                return c.query(stub, args)
        } else if function == "getHistory" {
                return c.getHistory(stub, args)
        }
        return shim.Error("Invalid function name")
}
```

The Order, Ship, and Distribute functions will be quite similar. These will update the ledger state. We will use order() as an example to show how we implement the chaincode function:

```
func (c *AssetMgr) Order(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
        return c.UpdateAsset(stub, args, "ORDER", "SCHOOL", "OEM")
}
```

Here is the UpdateAsset function:

```
func (c *AssetMgr) UpdateAsset(stub shim.ChaincodeStubInterface, args []string,
currentStatus string, from string, to string) pb.Response {
        assetId := args[0]  comment := args[1]          location := args[2]
        assetBytes, err := stub.GetState(assetId)
        orgAsset := OrgAsset{}
        ...
        if currentStatus == "ORDER" && orgAsset.Status != "START" {
        return shim.Error(err.Error())
        } else if currentStatus == "SHIP" && orgAsset.Status != "ORDER" {.}
else if currentStatus == "DISTRIBUTE" && orgAsset.Status != "SHIP" {.}
        orgAsset.Comment = comment
        orgAsset.Status = currentStatus
....
        orgAsset0, _ := json.Marshal(orgAsset)
        err = stub.PutState(assetId, orgAsset0)
        ...
        return shim.Success(orgAsset0)
}
```

The query and getHistory functions

ChaincodeStubInterface provides GetState, query functions. We can call this functions by passing assetId. This will trigger chaincode to get the corresponding result.

The getHistory function is used to view the records returned from the transaction history; all records are associated with assetId. Each record contains a related transaction ID and timestamp. With the timestamp, we know when the asset status was updated in the past.

Once the data is saved to blockchain, the application needs to query the chaincode data to check the OrgAsset information, shown as follows:

```
func (c *AssetMgr) getHistory(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
        type AuditHistory struct {
                TxId  string   `json:"txId"`
                Value OrgAsset `json:"value"`
        }
        var history []AuditHistory
        var orgAsset OrgAsset
        assetId := args[0]
        // Get History
        resultsIterator, err := stub.GetHistoryForKey(assetId)
        defer resultsIterator.Close()
        for resultsIterator.HasNext() {
                historyData, err := resultsIterator.Next()
                var tx AuditHistory
                tx.TxId = historyData.TxId
                json.Unmarshal(historyData.Value, &orgAsset)
                tx.Value = orgAsset        // orgAsset over
                history = append(history, tx) //add this tx to the list
        }
        ..
}
```

How it works...

Let's now take a closer look at what happens in each function in detail.

The Init function

The Init function is called when the chaincode is instantiated by the blockchain network and the function initializes the asset management data. In this function, we need to set up the OrgAsset initialization information. The three parameters we pass to call the Init function are assetId, assetType, and deviceId. This will set our device asset information, then we call the PutState(key, value) method to store the key and the value on the ledger.

The Invoke function

The Invoke function is called when the client invokes a specific function to process the transaction proposal. The ChaincodeStubInterface interface has the GetFunctionAndParameters method. This method extracts the function name and arguments and dispatches code to different functions based on the first argument. In our assetmgr, we need to call the Order, Ship, and Distribute functions, and then update the status for each step and the orgAsset information in the ledger. We can also define the query and query history functions, to get orgAsset information from the ledger.

You will notice that the Order method passes parameters from the command-line input. We use stub.GetState(assetId) to query the asset data from the blockchain, then we verify to make sure the current asset status is correct. We update the orgAsset info and then convert the asset data to byte data by calling json.Marshal(orgAsset). Finally, we save the data into the blockchain via stub.PutState. If there are no errors, the function will return a successful response to the client.

The query function

ChaincodeStubInterface defines the GetState method. The query function simply calls this function by passing assetId. This will trigger the chaincode to get the corresponding result.

All records returned from the transaction history are associated with `assetId`. Each record contains a related transaction ID and a timestamp. The timestamp tells us when the asset status was updated.

Compiling and deploying Fabric chaincode

We have now successfully written our asset management chaincode using the Go language. It is now time to build and deploy our `assetmgr` chaincode to Hyperledger Fabric.

Getting ready

Let's first get the fabric library in our environment. Navigate to the assetmgr directory, run the get chaincode library command, and then start build:

```
cd $GOPATH/src/assetmgr
go get -u github.com/hyperledger/fabric/core/chaincode/shim
go build
```

This will load the chaincode library and compile the Go code. Next, we will deploy the chaincode using the dev mode. Normally, we need to define our own channel, peer, and configuration Docker container to run our chaincode. Hyperledger, however, provides a sample dev network with a pre-generated orderer and channel artifact. This allows the user to start using chaincode for quick development and testing. You should have already set up the Fabric runtime environment with the fabric-samples project. If you haven't already done so, check out the previous chapter, or refer to fabric-samples in the GitHub link and follow the instructions: https://github.com/hyperledger/fabric-samples.

In our example project, we use the build in the fabric-samples project and set this same project as the default user home directory, as follows:

Let's now open three Terminals and navigate to the chaincode-docker-devmode directory of fabric-samples:

```
$ cd chaincode-docker-devmode
```

How to do it...

We will start a sample Fabric network to provide the Fabric runtime environment, and then package and build our Composer. Finally, we will deploy it to the network.

Starting the sample Fabric network

Open Terminal one. This Terminal will start the sample Fabric network. Issue the following command:

```
docker-compose -f docker-compose-simple.yaml up
```

This will bring up a network with the SingleSampleMSPSolo orderer profile. It also launches peer nodes, cli, and chaincode containers.

Building and deploying the chaincode

1. Open Terminal two. This Terminal will build and deploy the chaincode. Since we write and build the chaincode from our local Unix system, the chaincode is not yet in the Docker containers. Run the following command in the chaincode-docker-devmode folder:

```
docker exec -it chaincode bash
```

2. The output will be a list of folders:

```
abac chaincode_example02 fabcar marbles02 marbles02_private sacc
```

3. Let's create an assetmgr folder:

```
mkdir assetmgr
```

This will create an assetmgr folder in the Fabric container. First, type exit. This will exit the container and return to the chaincode-docker-devmode folder. Check the Fabric chaincode container ID by typing docker ps. You will get a similar result to the following:

In our example, the chaincode container ID is dbf9a0a1da76. The peer port is 7051.

4. With the container ID, we can the local chaincode to the chaincode container. Run the following command:

```
docker cp ~/itasset/src/assetmgr/assetmgr.go
dbf9a0a1da76:/opt/gopath/src/chaincode/assetmgr
```

5. Launch the chaincode container again:

```
docker exec -it chaincode bash
```

6. Navigate to the assetmgr folder and execute the go build command. This will compile our assermgr.go in the chaincode container, as shown in the following screenshot:

The assetmgr chaincode can be found at the following path: /opt/gopath/src/chaincode/assetmgr.

7. Run the chaincode by providing the peer address and chaincode ID name. The command is as follows:

```
CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=mycc:0
./assetmgr
```

8. This command will deploy the chaincode to the peer node at 7052. If you don't see any errors, the chaincode will start with peer. The log indicates that assetmgr.go successfully registered with peer:

How it works...

Here, we deployed the chaincode to the peer node. Let's now take a look at how chaincode-docker-devmode defines the blockchain configuration. chaincode-docker-devmode has some predefined configuration files and scripts. Here are the files in chaincode-docker-devmode:

Let's take a closer look at the docker-compose-simple.yaml file:

- This file defines services, peer, cli, and chaincode container configuration.
- Services define the orderer service with the container name as port 7050. It points to the Docker image at hyperledger/fabric-orderer.
- Peer defines a peer node. The peer container port is 7051. It points to a Docker image at hyperledger/fabric-peer.
- The cli section defines the cli container name as cli. The cli container can issue a command to interact with the chaincode deployed in the peer node. It points to a Docker image at hyperledger/fabric-tools.
- The chaincode container defines the container name as chaincode. It points to a Docker image at hyperledger/fabric-ccenv.

Here is the screenshot we see after we bring up the Docker containers. We can see the previously mentioned four containers running:

The script file in chaincode-docker-devmode only contains the following two commands:

```
peer channel create -c myc -f myc.tx -o orderer:7050
peer channel join -b myc.block
```

The first command creates the myc channel using the specified configuration file in the myc.tx file. The myc.tx file is generated by the configtxgen tool. This tool also generates orderer.block.

The second command joins the created channel with `myc.block` to the `cli` container. With these four containers, we can deploy our chaincode to the Fabric in the development environment.

Let's now carry out some tests from the `cli` container.

Running and testing the smart contract

We have opened two Terminals so far and deployed the chaincode to the `peer` node. It is time to install and test our chaincode function.

How to do it...

Open the third container to issue the `cli` command and test our smart contract. In this Terminal, we will start the `cli` container and issue the `cli` command to interact with the `chaincode` container. Launch an example `cli` container as follows:

```
docker exec -it cli bash
```

Installing the assermgr chaincode

Install the assermgr chaincode through the cli container by running the following command:

```
peer chaincode install -p chaincodedev/chaincode/assetmgr -n mycc -v 0
```

Here is the result after the chaincode is installed:

Instantiating the assermgr chaincode

Next, we will instantiate the assermgr chaincode. As we discussed earlier, in order to create an asset record, we need to pass assetId, assetType, and deviceId. Let's assume that the school needs to trace an ipad with the 0e83ff device ID and the 100 asset ID. We can instantiate our ipad asset by running the following command:

```
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["100","ipad", "0e83ff"]}' -C myc
```

The result is as follows:

We have now successfully installed and instantiated our assetmgr chaincode.

Invoking the assermgr chaincode

Next, we can start to invoke the remaining chaincode methods: Order, Ship, and Distribute.

1. To order the device from OEM, we need to pass three parameters to the chaincode—assetId, Comment, and Location. Here, assetId is 100, and we will assume that Location is New York.
2. Now, issue invoke to call the Order method in the assetmgr chaincode. The command is as follows:

```
peer chaincode invoke -n mycc -c '{"Args":["Order", "100", "initial order from school", "New York"]}' -C myc
```

3. If all goes well, you should see the following result. The log shows that the chaincode has been invoked successfully. We can see that the result is successfully saved to blockchain:

4. In our assetmgr, we have defined a query method. We can invoke this method to verify whether the records have been saved in the Fabric blockchain. Issue the following query command with assetId as 100:

```
peer chaincode query -C myc -n mycc -c '{"Args":["query","100"]}'
```

We can find the asset with an assetId of 100 from the Fabric ledger:

5. Once the OEM receives the order, it starts to work and produce the iPad device. Then, the OEM ships the device to the school. To do this, issue the following Ship command with assetId, Comment, and Location:

```
peer chaincode invoke -n mycc -c '{"Args":["Ship", "100", "OEM deliver ipad to school", "New Jersey"]}' -C myc
```

The following screenshot will be the output of the previous code:

6. Once the device is received, the school will distribute the device to the student. Issue the following Distribute command with assetId, Comment, and Location:

```
peer chaincode invoke -n mycc -c '{"Args":["Distribute", "100", "Distribute device to student", "New York"]}' -C myc
```

We should see the following result:

7. We have now completed the entire process for our demo use case. As we discussed earlier, blockchain is a ledger system; it will keep track of all transactions. Once records are saved to the blockchain, they cannot be altered. We should be able to see this historical transaction data. In our asset manager example, we issued the Order, Ship, and Distribute commands and the related chaincode was invoked. All related asset transaction records should be kept in the blockchain. Let's verify this by issuing the getHistory command:

```
peer chaincode query -C myc -n mycc -c '{"Args":["getHistory","100"]}'
```

This command will provide the following results:

As we can see, the getHistory command returns all of the transaction records we invoked from Fabric blockchain.

How it works...

The Fabric command-line interface is built using Fabric SDK Go. The CLI provides various commands to run a peer node, interact with the channel and the chaincode, and to query blockchain data. Here are some functionalities provided by the CLI:

Component	Functionality	Example command
Channel	Creates a channel	`peer channel join -b myc.block`
	Joins a peer to a channel	`peer channel join -b myc.block`
Chaincode	Installs chaincode	`peer chaincode install -p chaincodedev/chaincode/assetmgr -n mycc -v 0`
	Instantiates chaincode	`peer chaincode instantiate -n mycc -v 0 -c '{"Args":["100","ipad", "0e83ff"]}' -C myc`
	Invokes the chaincode function	`peer chaincode invoke -n mycc -c '{"Args":["Order", "100", "initial order from school", "New York"]}' -C myc`
	Queries chaincode data	`peer chaincode query -C myc -n mycc -c '{"Args":["getHistory","100"]}'`

With these supported CLI commands, we can test our chaincode in the development environment.Next, we will write client-side code and interact with the `assetmgr` chaincode in the Fabric.