Deploying a smart contract to a channel

End users interact with the blockchain ledger by invoking smart contracts. In Hyperledger Fabric, smart contracts are deployed in packages referred to as chaincode. Organizations that want to validate transactions or query the ledger need to install a chaincode on their peers. After a chaincode has been installed on the peers joined to a channel, channel members can deploy the chaincode to the channel and use the smart contracts in the chaincode to create or update assets on the channel ledger.

A chaincode is deployed to a channel using a process known as the Fabric chaincode lifecycle. The Fabric chaincode lifecycle allows multiple organizations to agree how a chaincode will be operated before it can be used to create transactions. For example, while an endorsement policy specifies which organizations need to execute a chaincode to validate a transaction, channel members need to use the Fabric chaincode lifecycle to agree on the chaincode endorsement policy. For a more in-depth overview about how to deploy and manage a chaincode on a channel, see Fabric chaincode lifecycle.

You can use this tutorial to learn how to use the peer lifecycle chaincode commands to deploy a chaincode to a channel of the Fabric test network. Once you have an understanding of the commands, you can use the steps in this tutorial to deploy your own chaincode to the test network, or to deploy chaincode to a production network. In this tutorial, you will deploy the Fabcar chaincode that is used by the Writing your first application tutorial.

Note: These instructions use the Fabric chaincode lifecycle introduced in the v2.0 release. If you would like to use the previous lifecycle to install and instantiate a chaincode, visit the v1.4 version of the Fabric documentation.

Start the network

We will start by deploying an instance of the Fabric test network. Before you begin, make sure that that you have installed the Prerequisites and Installed the Samples, Binaries and Docker Images. Use the following command to navigate to the test network directory within your local clone of the `fabric-samples` repository:

```
cd fabric-samples/test-network
```

For the sake of this tutorial, we want to operate from a known initial state. The following command will kill any active or stale docker containers and remove previously generated artifacts.

```
./network.sh down
```

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

```
./network.sh up createChannel
```

The `createChannel` command creates a channel named `mychannel` with two channel members, Org1 and Org2. The command also joins a peer that belongs to each organization to the channel. If the network and the channel are created successfully, you can see the following message printed in the logs:

```
========= Channel successfully joined ===========
```

We can now use the Peer CLI to deploy the Fabcar chaincode to the channel using the following steps:

- Step one: Package the smart contract
- Step two: Install the chaincode package
- Step three: Approve a chaincode definition
- Step four: Committing the chaincode definition to the channel

Setup Logspout (optional)

This step is not required but is extremely useful for troubleshooting chaincode. To monitor the logs of the smart contract, an administrator can view the aggregated output from a set of Docker containers using the `logspout` tool. The tool collects the output streams from different Docker containers into one place, making it easy to see what's happening from a single window. This can help administrators debug problems when they install smart contracts or developers when they invoke smart contracts. Because some containers are created purely for the purposes of starting a smart contract and only exist for a short time, it is helpful to collect all of the logs from your network.

A script to install and configure Logspout, `monitordocker.sh`, is already included in the `commercial-paper` sample in the Fabric samples. We will use the same script in this tutorial as well. The Logspout tool will continuously stream logs to your terminal, so you will need to use a new terminal window. Open a new terminal and navigate to the `test-network` directory.

```
cd fabric-samples/test-network
```

You can run the `monitordocker.sh` script from any directory. For ease of use, we will copy the `monitordocker.sh` script from the `commercial-paper` sample to your working directory

```
cp ../commercial-paper/organization/digibank/configuration/cli/monitordocker.sh .
# if you're not sure where it is
find . -name monitordocker.sh
```

You can then start Logspout by running the following command:

```
./monitordocker.sh net_test
```

You should see output similar to the following:

```
Starting monitoring on all containers on the network net_basic
Unable to find image 'gliderlabs/logspout:latest' locally
latest: Pulling from gliderlabs/logspout
4fe2ade4980c: Pull complete
decca452f519: Pull complete
ad60f6b6c009: Pull complete
Digest: sha256:374e06b17b004bddc5445525796b5f7adb8234d64c5c5d663095fccafb6e4c26
Status: Downloaded newer image for gliderlabs/logspout:latest
1f99d130f15cf01706eda3e1f040496ec885036d485cb6bcc0da4a567ad84361
```

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

Package the smart contract

We need to package the chaincode before it can be installed on our peers. The steps are different if you want to install a smart contract written in Go, Java, or JavaScript.

Go

Before we package the chaincode, we need to install the chaincode dependences. Navigate to the folder that contains the Go version of the Fabcar chaincode.

```
cd fabric-samples/chaincode/fabcar/go
```

The sample uses a Go module to install the chaincode dependencies. The dependencies are listed in a `go.mod` file in the `fabcar/go` directory. You should take a moment to examine this file.

```
$ cat go.mod
module github.com/hyperledger/fabric-samples/chaincode/fabcar/go

go 1.13

require github.com/hyperledger/fabric-contract-api-go v1.1.0
```

The `go.mod` file imports the Fabric contract API into the smart contract package. You can open `fabcar.go` in a text editor to see how the contract API is used to define the `SmartContract` type at the beginning of the smart contract:

```
// SmartContract provides functions for managing a car
type SmartContract struct {
    contractapi.Contract
}
```

The `SmartContract` type is then used to create the transaction context for the functions defined within the smart contract that read and write data to the blockchain ledger.

```
// CreateCar adds a new car to the world state with given details
func (s *SmartContract) CreateCar(ctx contractapi.TransactionContextInterface, carNumber string,
    car := Car{
        Make:   make,
        Model:  model,
        Colour: colour,
        Owner:  owner,
    }

    carAsBytes, _ := json.Marshal(car)

    return ctx.GetStub().PutState(carNumber, carAsBytes)
}
```

You can learn more about the Go contract API by visiting the API documentation and the smart contract processing topic.

To install the smart contract dependencies, run the following command from the `fabcar/go` directory.

```
GO111MODULE=on go mod vendor
```

If the command is successful, the go packages will be installed inside a `vendor` folder.

Now that we that have our dependences, we can create the chaincode package. Navigate back to our working directory in the `test-network` folder so that we can package the chaincode together with our other network artifacts.

```
cd ../../../test-network
```

You can use the `peer` CLI to create a chaincode package in the required format. The `peer` binaries are located in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

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

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To confirm that you are able to use the `peer` CLI, check the version of the binaries. The binaries need to be version `2.0.0` or later to run this tutorial.

```
peer version
```

You can now create the chaincode package using the peer lifecycle chaincode package command:

```
peer lifecycle chaincode package fabcar.tar.gz --path ../chaincode/fabcar/go/ --lang golang --lab
```

This command will create a package named `fabcar.tar.gz` in your current directory. The `--lang` flag is used to specify the chaincode language and the `--path` flag provides the location of your smart contract code. The path must be a fully qualified path or a path relative to your present working directory. The `--label` flag is used to specify a chaincode label that will identity your chaincode after it is installed. It is recommended that your label include the chaincode name and version.

Now that we created the chaincode package, we can install the chaincode on the peers of the test network.

JavaScript

Before we package the chaincode, we need to install the chaincode dependences. Navigate to the folder that contains the JavaScript version of the Fabcar chaincode.

```
cd fabric-samples/chaincode/fabcar/javascript
```

The dependencies are listed in the `package.json` file in the `fabcar/javascript` directory. You should take a moment to examine this file. You can find the dependences section displayed below:

```
"dependencies": {
        "fabric-contract-api": "^2.0.0",
        "fabric-shim": "^2.0.0"
```

The `package.json` file imports the Fabric contract class into the smart contract package. You can open `lib/fabcar.js` in a text editor to see the contract class imported into the smart contract and used to create the FabCar class.

```
const { Contract } = require('fabric-contract-api');

class FabCar extends Contract {
    ...
}
```

The `FabCar` class provides the transaction context for the functions defined within the smart contract that read and write data to the blockchain ledger.

```
async createCar(ctx, carNumber, make, model, color, owner) {
    console.info('============= START : Create Car ===========');

    const car = {
        color,
        docType: 'car',
        make,
        model,
        owner,
    };

    await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
```

```
    console.info('============= END : Create Car ===========');
}
```

You can learn more about the JavaScript contract API by visiting the API documentation and the smart contract processing topic.

To install the smart contract dependencies, run the following command from the `fabcar/javascript` directory.

```
npm install
```

If the command is successful, the JavaScript packages will be installed inside a `npm_modules` folder.

Now that we that have our dependences, we can create the chaincode package. Navigate back to our working directory in the `test-network` folder so that we can package the chaincode together with our other network artifacts.

```
cd ../../../test-network
```

You can use the `peer` CLI to create a chaincode package in the required format. The `peer` binaries are located in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

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

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To confirm that you are able to use the `peer` CLI, check the version of the binaries. The binaries need to be version `2.0.0` or later to run this tutorial.

```
peer version
```

You can now create the chaincode package using the peer lifecycle chaincode package command:

```
peer lifecycle chaincode package fabcar.tar.gz --path ../chaincode/fabcar/javascript/ --lang node
```

This command will create a package named `fabcar.tar.gz` in your current directory. The `--lang` flag is used to specify the chaincode language and the `--path` flag provides the location of your smart

contract code. The `--label` flag is used to specify a chaincode label that will identity your chaincode after it is installed. It is recommended that your label include the chaincode name and version.

Now that we created the chaincode package, we can install the chaincode on the peers of the test network.

Java

Before we package the chaincode, we need to install the chaincode dependences. Navigate to the folder that contains the Java version of the Fabcar chaincode.

```
cd fabric-samples/chaincode/fabcar/java
```

The sample uses Gradle to install the chaincode dependencies. The dependencies are listed in the `build.gradle` file in the `fabcar/java` directory. You should take a moment to examine this file. You can find the dependences section displayed below:

```
dependencies {
    compileOnly 'org.hyperledger.fabric-chaincode-java:fabric-chaincode-shim:2.0.+'
    implementation 'com.owlike:genson:1.5'
    testImplementation 'org.hyperledger.fabric-chaincode-java:fabric-chaincode-shim:2.0.+'
    testImplementation 'org.junit.jupiter:junit-jupiter:5.4.2'
    testImplementation 'org.assertj:assertj-core:3.11.1'
    testImplementation 'org.mockito:mockito-core:2.+'
}
```

The `build.gradle` file imports the Java chaincode shim into the smart contract package, which includes the contract class. You can find Fabcar smart contract in the `src` directory. You can navigate to the `FabCar.java` file and open it in a text editor to see how the contract class is used to create the transaction context for the functions defined that read and write data to the blockchain ledger.

You can learn more about the Java chaincode shim and the contract class by visiting the Java chaincode documentation and the smart contract processing topic.

To install the smart contract dependencies, run the following command from the `fabcar/java` directory.

```
./gradlew installDist
```

If the command is successful, you will be able to find the built smart contract in the `build` folder.

Now that we have installed the dependences and built the smart contract, we can create the chaincode package. Navigate back to our working directory in the `test-network` folder so that we can package the chaincode together with our other network artifacts.

```
cd ../../../test-network
```

You can use the `peer` CLI to create a chaincode package in the required format. The `peer` binaries are located in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

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

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To confirm that you are able to use the `peer` CLI, check the version of the binaries. The binaries need to be version `2.0.0` or later to run this tutorial.

```
peer version
```

You can now create the chaincode package using the peer lifecycle chaincode package command:

```
peer lifecycle chaincode package fabcar.tar.gz --path ../chaincode/fabcar/java/build/install/fabc
```

This command will create a package named `fabcar.tar.gz` in your current directory. The `--lang` flag is used to specify the chaincode language and the `--path` flag provides the location of your smart contract code. The `--label` flag is used to specify a chaincode label that will identity your chaincode after it is installed. It is recommended that your label include the chaincode name and version.

Now that we created the chaincode package, we can install the chaincode on the peers of the test network.

Install the chaincode package

After we package the Fabcar smart contract, we can install the chaincode on our peers. The chaincode needs to be installed on every peer that will endorse a transaction. Because we are going to set the endorsement policy to require endorsements from both Org1 and Org2, we need to install the chaincode on the peers operated by both organizations:

- peer0.org1.example.com
- peer0.org2.example.com

Let's install the chaincode on the Org1 peer first. Set the following environment variables to operate the `peer` CLI as the Org1 admin user. The `CORE_PEER_ADDRESS` will be set to point to the Org1 peer, `peer0.org1.example.com`.

```
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/peers/
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.com/users/Admi
export CORE_PEER_ADDRESS=localhost:7051
```

Issue the peer lifecycle chaincode install command to install the chaincode on the peer:

```
peer lifecycle chaincode install fabcar.tar.gz
```

If the command is successful, the peer will generate and return the package identifier. This package ID will be used to approve the chaincode in the next step. You should see output similar to the following:

```
2020-02-12 11:40:02.923 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO 001 Installed
2020-02-12 11:40:02.925 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO 002 Chaincode
```

We can now install the chaincode on the Org2 peer. Set the following environment variables to operate as the Org2 admin and target target the Org2 peer, `peer0.org2.example.com`.

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.example.com/peers/
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.example.com/peers/
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.com/users/Admi
export CORE_PEER_ADDRESS=localhost:9051
```

Issue the following command to install the chaincode:

```
peer lifecycle chaincode install fabcar.tar.gz
```

The chaincode is built by the peer when the chaincode is installed. The install command will return any build errors from the chaincode if there is a problem with the smart contract code.

Approve a chaincode definition

After you install the chaincode package, you need to approve a chaincode definition for your organization. The definition includes the important parameters of chaincode governance such as the name, version, and the chaincode endorsement policy.

The set of channel members who need to approve a chaincode before it can be deployed is governed by the `Application/Channel/lifeycleEndorsement` policy. By default, this policy requires that a majority of channel members need to approve a chaincode before it can used on a channel. Because we have only two organizations on the channel, and a majority of 2 is 2, we need approve a chaincode definition of Fabcar as Org1 and Org2.

If an organization has installed the chaincode on their peer, they need to include the packageID in the chaincode definition approved by their organization. The package ID is used to associate the chaincode installed on a peer with an approved chaincode definition, and allows an organization to use the chaincode to endorse transactions. You can find the package ID of a chaincode by using the peer lifecycle chaincode queryinstalled command to query your peer.

```
peer lifecycle chaincode queryinstalled
```

The package ID is the combination of the chaincode label and a hash of the chaincode binaries. Every peer will generate the same package ID. You should see output similar to the following:

```
Installed chaincodes on peer:
Package ID: fabcar_1:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3, Label: fab
```

We are going to use the package ID when we approve the chaincode, so let's go ahead and save it as an environment variable. Paste the package ID returned by `peer lifecycle chaincode queryinstalled` into the command below. **Note:** The package ID will not be the same for all users, so you need to complete this step using the package ID returned from your command window in the previous step.

```
export CC_PACKAGE_ID=fabcar_1:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3
```

Because the environment variables have been set to operate the `peer` CLI as the Org2 admin, we can approve the chaincode definition of Fabcar as Org2. Chaincode is approved at the organization level, so the command only needs to target one peer. The approval is distributed to the other peers within the organization using gossip. Approve the chaincode definition using the peer lifecycle chaincode approveformyorg command:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --ordererTLSHostnameOverride orderer.e
```

The command above uses the `--package-id` flag to include the package identifier in the chaincode definition. The `--sequence` parameter is an integer that keeps track of the number of times a chaincode has been defined or updated. Because the chaincode is being deployed to the channel for the first time, the sequence number is 1. When the Fabcar chaincode is upgraded, the sequence number will be incremented to 2. If you are using the low level APIs provided by the Fabric Chaincode Shim API, you could pass the `--init-required` flag to the command above to request the execution of the Init function to initialize the chaincode. The first invoke of the chaincode would need to target the Init function and include the `--isInit` flag before you could use the other functions in the chaincode to interact with the ledger.

We could have provided a `--signature-policy` or `--channel-config-policy` argument to the `approveformyorg` command to specify a chaincode endorsement policy. The endorsement policy specifies how many peers belonging to different channel members need to validate a transaction against a given chaincode. Because we did not set a policy, the definition of Fabcar will use the

default endorsement policy, which requires that a transaction be endorsed by a majority of channel members present when the transaction is submitted. This implies that if new organizations are added or removed from the channel, the endorsement policy is updated automatically to require more or fewer endorsements. In this tutorial, the default policy will require a majority of 2 out of 2 and transactions will need to be endorsed by a peer from Org1 and Org2. If you want to specify a custom endorsement policy, you can use the Endorsement Policies operations guide to learn about the policy syntax.

You need to approve a chaincode definition with an identity that has an admin role. As a result, the `CORE_PEER_MSPCONFIGPATH` variable needs to point to the MSP folder that contains an admin identity. You cannot approve a chaincode definition with a client user. The approval needs to be submitted to the ordering service, which will validate the admin signature and then distribute the approval to your peers.

We still need to approve the chaincode definition as Org1. Set the following environment variables to operate as the Org1 admin:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.com/users/Admi
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/peers/
export CORE_PEER_ADDRESS=localhost:7051
```

You can now approve the chaincode definition as Org1.

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --ordererTLSHostnameOverride orderer.e
```

We now have the majority we need to deploy the Fabcar the chaincode to the channel. While only a majority of organizations need to approve a chaincode definition (with the default policies), all organizations need to approve a chaincode definition to start the chaincode on their peers. If you commit the definition before a channel member has approved the chaincode, the organization will not be able to endorse transactions. As a result, it is recommended that all channel members approve a chaincode before committing the chaincode definition.

Committing the chaincode definition to the channel

After a sufficient number of organizations have approved a chaincode definition, one organization can commit the chaincode definition to the channel. If a majority of channel members have approved the definition, the commit transaction will be successful and the parameters agreed to in the chaincode definition will be implemented on the channel.

You can use the peer lifecycle chaincode checkcommitreadiness command to check whether channel members have approved the same chaincode definition. The flags used for the `checkcommitreadiness` command are identical to the flags used to approve a chaincode for your organization. However, you do not need to include the `--package-id` flag.

```
peer lifecycle chaincode checkcommitreadiness --channelID mychannel --name fabcar --version 1.0 -
```

The command will produce a JSON map that displays if a channel member has approved the parameters that were specified in the `checkcommitreadiness` command:

```
{
        "Approvals": {
                "Org1MSP": true,
                "Org2MSP": true
        }
}
```

Since both organizations that are members of the channel have approved the same parameters, the chaincode definition is ready to be committed to the channel. You can use the peer lifecycle chaincode commit command to commit the chaincode definition to the channel. The commit command also needs to be submitted by an organization admin.

```
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride orderer.example.co
```

The transaction above uses the `--peerAddresses` flag to target `peer0.org1.example.com` from Org1 and `peer0.org2.example.com` from Org2. The `commit` transaction is submitted to the peers joined to the channel to query the chaincode definition that was approved by the organization that operates the peer. The command needs to target the peers from a sufficient number of organizations to satisfy the policy for deploying a chaincode. Because the approval is distributed within each organization, you can target any peer that belongs to a channel member.

The chaincode definition endorsements by channel members are submitted to the ordering service to be added to a block and distributed to the channel. The peers on the channel then validate whether a sufficient number of organizations have approved the chaincode definition. The `peer lifecycle chaincode commit` command will wait for the validations from the peer before returning a response.

You can use the peer lifecycle chaincode querycommitted command to confirm that the chaincode definition has been committed to the channel.

```
peer lifecycle chaincode querycommitted --channelID mychannel --name fabcar --cafile ${PWD}/organ
```

If the chaincode was successful committed to the channel, the `querycommitted` command will return the sequence and version of the chaincode definition:

```
Committed chaincode definition for chaincode 'fabcar' on channel 'mychannel':
Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc, Approvals: [Org1MSP:
```

Invoking the chaincode

After the chaincode definition has been committed to a channel, the chaincode will start on the peers joined to the channel where the chaincode was installed. The Fabcar chaincode is now ready to be invoked by client applications. Use the following command create an initial set of cars on the ledger. Note that the invoke command needs target a sufficient number of peers to meet chaincode endorsement policy.

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --
```

If the command is successful, you should be able to a response similar to the following:

```
2020-02-12 18:22:20.576 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Chaincode invoke su
```

We can use a query function to read the set of cars that were created by the chaincode:

```
peer chaincode query -C mychannel -n fabcar -c '{"Args":["queryAllCars"]}'
```

The response to the query should be the following list of cars:

```
[{"Key":"CAR0","Record":{"make":"Toyota","model":"Prius","colour":"blue","owner":"Tomoko"}},
{"Key":"CAR1","Record":{"make":"Ford","model":"Mustang","colour":"red","owner":"Brad"}},
{"Key":"CAR2","Record":{"make":"Hyundai","model":"Tucson","colour":"green","owner":"Jin Soo"}},
{"Key":"CAR3","Record":{"make":"Volkswagen","model":"Passat","colour":"yellow","owner":"Max"}},
{"Key":"CAR4","Record":{"make":"Tesla","model":"S","colour":"black","owner":"Adriana"}},
{"Key":"CAR5","Record":{"make":"Peugeot","model":"205","colour":"purple","owner":"Michel"}},
{"Key":"CAR6","Record":{"make":"Chery","model":"S22L","colour":"white","owner":"Aarav"}},
{"Key":"CAR7","Record":{"make":"Fiat","model":"Punto","colour":"violet","owner":"Pari"}},
{"Key":"CAR8","Record":{"make":"Tata","model":"Nano","colour":"indigo","owner":"Valeria"}},
{"Key":"CAR9","Record":{"make":"Holden","model":"Barina","colour":"brown","owner":"Shotaro"}}]
```

Upgrading a smart contract

You can use the same Fabric chaincode lifecycle process to upgrade a chaincode that has already been deployed to a channel. Channel members can upgrade a chaincode by installing a new chaincode package and then approving a chaincode definition with the new package ID, a new chaincode version, and with the sequence number incremented by one. The new chaincode can be used after the chaincode definition is committed to the channel. This process allows channel members to coordinate on when a chaincode is upgraded, and ensure that a sufficient number of channel members are ready to use the new chaincode before it is deployed to the channel.

Channel members can also use the upgrade process to change the chaincode endorsement policy. By approving a chaincode definition with a new endorsement policy and committing the chaincode definition to the channel, channel members can change the endorsement policy governing a chaincode without installing a new chaincode package.

To provide a scenario for upgrading the Fabcar chaincode that we just deployed, let's assume that Org1 and Org2 would like to install a version of the chaincode that is written in another language.

They will use the Fabric chaincode lifecycle to update the chaincode version and ensure that both organizations have installed the new chaincode before it becomes active on the channel.

We are going to assume that Org1 and Org2 initially installed the GO version of the Fabcar chaincode, but would be more comfortable working with a chaincode written in JavaScript. The first step is to package the JavaScript version of the Fabcar chaincode. If you used the JavaScript instructions to package your chaincode when you went through the tutorial, you can install new chaincode binaries by following the steps for packaging a chaincode written in Go or Java.

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

```
cd ../chaincode/fabcar/javascript
npm install
cd ../../../test-network
```

You can then issue the following commands to package the JavaScript chaincode from the `test-network` directory. We will set the environment variables needed to use the `peer` CLI again in case you closed your terminal.

```
export PATH=${PWD}/../bin:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.com/users/Admi
peer lifecycle chaincode package fabcar_2.tar.gz --path ../chaincode/fabcar/javascript/ --lang no
```

Run the following commands to operate the `peer` CLI as the Org1 admin:

```
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/peers/
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.com/users/Admi
export CORE_PEER_ADDRESS=localhost:7051
```

We can now use the following command to install the new chaincode package on the Org1 peer.

```
peer lifecycle chaincode install fabcar_2.tar.gz
```

The new chaincode package will create a new package ID. We can find the new package ID by querying our peer.

```
peer lifecycle chaincode queryinstalled
```

The `queryinstalled` command will return a list of the chaincode that have been installed on your peer.

```
Installed chaincodes on peer:
Package ID: fabcar_1:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3, Label: fab
```

```
Package ID: fabcar_2:1d559f9fb3dd879601ee17047658c7e0c84eab732dca7c841102f20e42a9e7d4, Label: fab
```

You can use the package label to find the package ID of the new chaincode and save it as a new environment variable.

```
export NEW_CC_PACKAGE_ID=fabcar_2:1d559f9fb3dd879601ee17047658c7e0c84eab732dca7c841102f20e42a9e7d
```

Org1 can now approve a new chaincode definition:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --ordererTLSHostnameOverride orderer.e
```

The new chaincode definition uses the package ID of the JavaScript chaincode package and updates the chaincode version. Because the sequence parameter is used by the Fabric chaincode lifecycle to keep track of chaincode upgrades, Org1 also needs to increment the sequence number from 1 to 2. You can use the peer lifecycle chaincode querycommitted command to find the sequence of the chaincode that was last committed to the channel.

We now need to install the chaincode package and approve the chaincode definition as Org2 in order to upgrade the chaincode. Run the following commands to operate the `peer` CLI as the Org2 admin:

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.example.com/peers/
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.example.com/peers/
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.com/users/Admi
export CORE_PEER_ADDRESS=localhost:9051
```

We can now use the following command to install the new chaincode package on the Org2 peer.

```
peer lifecycle chaincode install fabcar_2.tar.gz
```

You can now approve the new chaincode definition for Org2.

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --ordererTLSHostnameOverride orderer.e
```

Use the peer lifecycle chaincode checkcommitreadiness command to check if the chaincode definition with sequence 2 is ready to be committed to the channel:

```
peer lifecycle chaincode checkcommitreadiness --channelID mychannel --name fabcar --version 2.0 -
```

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

```
{
        "Approvals": {
                "Org1MSP": true,
                "Org2MSP": true
```

```
        }
    }
```

The chaincode will be upgraded on the channel after the new chaincode definition is committed. Until then, the previous chaincode will continue to run on the peers of both organizations. Org2 can use the following command to upgrade the chaincode:

```
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride orderer.example.co
```

A successful commit transaction will start the new chaincode right away. If the chaincode definition changed the endorsement policy, the new policy would be put in effect.

You can use the `docker ps` command to verify that the new chaincode has started on your peers:

```
$docker ps
CONTAINER ID        IMAGE
197a4b70a392        dev-peer0.org1.example.com-fabcar_2-1d559f9fb3dd879601ee17047658c7e0c84eab732
b7e4dbfd4ea0        dev-peer0.org2.example.com-fabcar_2-1d559f9fb3dd879601ee17047658c7e0c84eab732
8b6e9abaef8d        hyperledger/fabric-peer:latest
429dae4757ba        hyperledger/fabric-peer:latest
7de5d19400e6        hyperledger/fabric-orderer:latest
```

If you used the `--init-required` flag, you need to invoke the Init function before you can use the upgraded chaincode. Because we did not request the execution of Init, we can test our new JavaScript chaincode by creating a new car:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --
```

You can query all the cars on the ledger again to see the new car:

```
peer chaincode query -C mychannel -n fabcar -c '{"Args":["queryAllCars"]}'
```

You should see the following result from the JavaScript chaincode:

```
[{"Key":"CAR0","Record":{"make":"Toyota","model":"Prius","colour":"blue","owner":"Tomoko"}},
{"Key":"CAR1","Record":{"make":"Ford","model":"Mustang","colour":"red","owner":"Brad"}},
{"Key":"CAR11","Record":{"color":"Black","docType":"car","make":"Honda","model":"Accord","owner":
{"Key":"CAR2","Record":{"make":"Hyundai","model":"Tucson","colour":"green","owner":"Jin Soo"}},
{"Key":"CAR3","Record":{"make":"Volkswagen","model":"Passat","colour":"yellow","owner":"Max"}},
{"Key":"CAR4","Record":{"make":"Tesla","model":"S","colour":"black","owner":"Adriana"}},
{"Key":"CAR5","Record":{"make":"Peugeot","model":"205","colour":"purple","owner":"Michel"}},
{"Key":"CAR6","Record":{"make":"Chery","model":"S22L","colour":"white","owner":"Aarav"}},
{"Key":"CAR7","Record":{"make":"Fiat","model":"Punto","colour":"violet","owner":"Pari"}},
{"Key":"CAR8","Record":{"make":"Tata","model":"Nano","colour":"indigo","owner":"Valeria"}},
{"Key":"CAR9","Record":{"make":"Holden","model":"Barina","colour":"brown","owner":"Shotaro"}}]
```

Clean up

When you are finished using the chaincode, you can also use the following commands to remove the Logspout tool.

```
docker stop logspout
docker rm logspout
```

You can then bring down the test network by issuing the following command from the `test-network` directory:

```
./network.sh down
```

Next steps

After you write your smart contract and deploy it to a channel, you can use the APIs provided by the Fabric SDKs to invoke the smart contracts from a client application. This allows end users to interact with the assets on the blockchain ledger. To get started with the Fabric SDKs, see the Writing Your first application tutorial.

troubleshooting

Chaincode not agreed to by this org

Problem: When I try to commit a new chaincode definition to the channel, the `peer lifecycle chaincode commit` command fails with the following error:

```
Error: failed to create signed transaction: proposal response was not successful, error code 500,
```

Solution: You can try to resolve this error by using the `peer lifecycle chaincode checkcommitreadiness` command to check which channel members have approved the chaincode definition that you are trying to commit. If any organization used a different value for any parameter of the chaincode definition, the commit transaction will fail. The `peer lifecycle chaincode checkcommitreadiness` will reveal which organizations did not approve the chaincode definition you are trying to commit:

```
{
    "approvals": {
        "Org1MSP": false,
        "Org2MSP": true
    }
}
```

Invoke failure

Problem: The `peer lifecycle chaincode commit` transaction is successful, but when I try to invoke the chaincode for the first time, it fails with the following error:

```
Error: endorsement failure during invoke. response: status:500 message:"make sure the chaincode f
```

Solution: You may not have set the correct `--package-id` when you approved your chaincode definition. As a result, the chaincode definition that was committed to the channel was not associated with the chaincode package you installed and the chaincode was not started on your peers. If you are running a docker based network, you can use the `docker ps` command to check if your chaincode is running:

```
docker ps
CONTAINER ID        IMAGE                               COMMAND             CREATED             S
7fe1ae0a69fa        hyperledger/fabric-orderer:latest   "orderer"           5 minutes ago       U
2b9c684bd07e        hyperledger/fabric-peer:latest      "peer node start"   5 minutes ago       U
39a3e41b2573        hyperledger/fabric-peer:latest      "peer node start"   5 minutes ago       U
```

If you do not see any chaincode containers listed, use the `peer lifecycle chaincode approveformyorg` command approve a chaincode definition with the correct package ID.

Endorsement policy failure

Problem: When I try to commit the chaincode definition to the channel, the transaction fails with the following error:

```
2020-04-07 20:08:23.306 EDT [chaincodeCmd] ClientWait -> INFO 001 txid [5f569e50ae58efa6261c4ad93
Error: transaction invalidated with status (ENDORSEMENT_POLICY_FAILURE)
```

Solution: This error is a result of the commit transaction not gathering enough endorsements to meet the Lifecycle endorsement policy. This problem could be a result of your transaction not targeting a sufficient number of peers to meet the policy. This could also be the result of some of the peer organizations not including the `Endorsement:` signature policy referenced by the default `/Channel/Application/Endorsement` policy in their `configtx.yaml` file:

```
Readers:
        Type: Signature
        Rule: "OR('Org2MSP.admin', 'Org2MSP.peer', 'Org2MSP.client')"
Writers:
        Type: Signature
        Rule: "OR('Org2MSP.admin', 'Org2MSP.client')"
Admins:
        Type: Signature
        Rule: "OR('Org2MSP.admin')"
Endorsement:
        Type: Signature
        Rule: "OR('Org2MSP.peer')"
```

When you enable the Fabric chaincode lifecycle, you also need to use the new Fabric 2.0 channel policies in addition to upgrading your channel to the `V2_0` capability. Your channel needs to include the new `/Channel/Application/LifecycleEndorsement` and `/Channel/Application/Endorsement` policies:

```
Policies:
        Readers:
                Type: ImplicitMeta
                Rule: "ANY Readers"
        Writers:
                Type: ImplicitMeta
                Rule: "ANY Writers"
```

```
        Admins:
                Type: ImplicitMeta
                Rule: "MAJORITY Admins"
        LifecycleEndorsement:
                Type: ImplicitMeta
                Rule: "MAJORITY Endorsement"
        Endorsement:
                Type: ImplicitMeta
                Rule: "MAJORITY Endorsement"
```

If you do not include the new channel policies in the channel configuration, you will get the following error when you approve a chaincode definition for your organization:

```
Error: proposal failed with status: 500 - failed to invoke backing implementation of 'ApproveChai
```