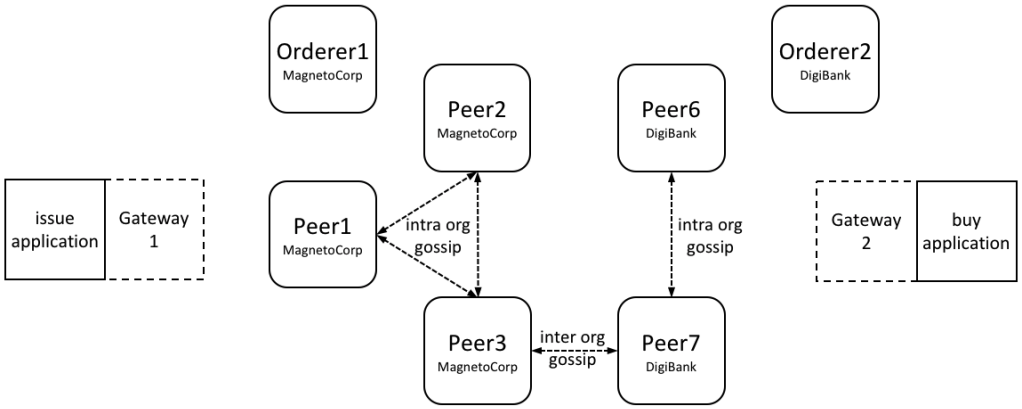
In this topic, we're going to cover:

- *Why gateways are important*
- *How applications use a gateway*
- *How to define a static gateway*
- *How to define a dynamic gateway for service discovery*
- *Using multiple gateways*

Scenario

A Hyperledger Fabric network channel can constantly change. The peer, orderer and CA components, contributed by the different organizations in the network, will come and go. Reasons for this include increased or reduced business demand, and both planned and unplanned outages. A gateway relieves an application of this burden, allowing it to focus on the business problem it is trying to solve.



A MagnetoCorp and DigiBank applications (issue and buy) delegate their respective network interactions to their gateways. Each gateway understands the network channel topology comprising the multiple peers and orderers of two organizations MagnetoCorp and DigiBank, leaving applications to focus on business logic. Peers can talk to each other both within and across organizations using the gossip protocol.

A gateway can be used by an application in two different ways:

- **Static**: The gateway configuration is *completely* defined in a connection profile. All the peers, orderers and CAs available to an application are statically defined in the connection profile used to configure the gateway. For peers, this includes their role as an endorsing peer or event notification hub, for example. You can read more about these roles in the connection profile topic.

  The SDK will use this static topology, in conjunction with gateway connection options, to manage the transaction submission and notification processes. The connection profile must contain enough of the network topology to allow a gateway to interact with the network on behalf of the application; this includes the network channels, organizations, orderers, peers and their roles.

- **Dynamic**: The gateway configuration is minimally defined in a connection profile. Typically, one or two peers from the application's organization are specified, and they use service discovery to discover the available network topology. This includes peers, orderers, channels, deployed smart contracts and their endorsement policies. (In production environments, a gateway configuration should specify at least two peers for availability.)

  The SDK will use all of the static and discovered topology information, in conjunction with gateway connection options, to manage the transaction submission and notification processes. As part of this, it will also intelligently

use the discovered topology; for example, it will *calculate* the minimum required endorsing peers using the discovered endorsement policy for the smart contract.

You might ask yourself whether a static or dynamic gateway is better? The trade-off is between predictability and responsiveness. Static networks will always behave the same way, as they perceive the network as unchanging. In this sense they are predictable – they will always use the same peers and orderers if they are available. Dynamic networks are more responsive as they understand how the network changes – they can use newly added peers and orderers, which brings extra resilience and scalability, at potentially some cost in predictability. In general it's fine to use dynamic networks, and indeed this the default mode for gateways.

Note that the *same* connection profile can be used statically or dynamically. Clearly, if a profile is going to be used statically, it needs to be comprehensive, whereas dynamic usage requires only sparse population.

Both styles of gateway are transparent to the application; the application program design does not change whether static or dynamic gateways are used. This also means that some applications may use service discovery, while others may not. In general using dynamic discovery means less definition and more intelligence by the SDK; it is the default.

Connect

When an application connects to a gateway, two options are provided. These are used in subsequent SDK processing:

```
await gateway.connect(connectionProfile, connectionOptions);
```

- **Connection profile**: `connectionProfile` is the gateway configuration that will be used for transaction processing by the SDK, whether statically or dynamically. It can be specified in YAML or JSON, though it must be converted to a JSON object when passed to the gateway:

  ```
  let connectionProfile = yaml.safeLoad(fs.readFileSync('../gateway/paperNet.yaml',
  ↪'utf8'));
  ```

  Read more about connection profiles and how to configure them.

- **Connection options**: `connectionOptions` allow an application to declare rather than implement desired transaction processing behaviour. Connection options are interpreted by the SDK to control interaction patterns with network components, for example to select which identity to connect with, or which peers to use for event notifications. These options significantly reduce application complexity without compromising functionality. This is possible because the SDK has implemented much of the low level logic that would otherwise be required by applications; connection options control this logic flow.

  Read about the list of available connection options and when to use them.

Static

Static gateways define a fixed view of a network. In the MagnetoCorp *scenario*, a gateway might identify a single peer from MagnetoCorp, a single peer from DigiBank, and a MagentoCorp orderer. Alternatively, a gateway might define *all* peers and orderers from MagnetCorp and DigiBank. In both cases, a gateway must define a view of the network sufficient to get commercial paper transactions endorsed and distributed.

Applications can use a gateway statically by explicitly specifying the connect option `discovery: { enabled:false }` on the `gateway.connect()` API. Alternatively, the environment variable setting `FABRIC_SDK_DISCOVERY=false` will always override the application choice.

Examine the connection profile used by the MagnetoCorp issue application. See how all the peers, orderers and even CAs are specified in this file, including their roles.

It's worth bearing in mind that a static gateway represents a view of a network at a *moment in time*. As networks change, it may be important to reflect this in a change to the gateway file. Applications will automatically pick up these changes when they re-load the gateway file.

Dynamic

Dynamic gateways define a small, fixed *starting point* for a network. In the MagnetoCorp *scenario*, a dynamic gateway might identify just a single peer from MagnetoCorp; everything else will be discovered! (To provide resiliency, it might be better to define two such bootstrap peers.)

If service discovery is selected by an application, the topology defined in the gateway file is augmented with that produced by this process. Service discovery starts with the gateway definition, and finds all the connected peers and orderers within the MagnetoCorp organization using the gossip protocol. If anchor peers have been defined for a channel, then service discovery will use the gossip protocol across organizations to discover components within the connected organization. This process will also discover smart contracts installed on peers and their endorsement policies defined at a channel level. As with static gateways, the discovered network must be sufficient to get commercial paper transactions endorsed and distributed.

Dynamic gateways are the default setting for Fabric applications. They can be explicitly specified using the connect option `discovery: { enabled:true }` on the `gateway.connect()` API. Alternatively, the environment variable setting `FABRIC_SDK_DISCOVERY=true` will always override the application choice.

A dynamic gateway represents an up-to-date view of a network. As networks change, service discovery will ensure that the network view is an accurate reflection of the topology visible to the application. Applications will automatically pick up these changes; they do not even need to re-load the gateway file.

Multiple gateways

Finally, it is straightforward for an application to define multiple gateways, both for the same or different networks. Moreover, applications can use the name gateway both statically and dynamically.

It can be helpful to have multiple gateways. Here are a few reasons:

- Handling requests on behalf of different users.
- Connecting to different networks simultaneously.
- Testing a network configuration, by simultaneously comparing its behaviour with an existing configuration.

This topic covers how to develop a client application and smart contract to solve a business problem using Hyperledger Fabric. In a real world **Commercial Paper** scenario, involving multiple organizations, you'll learn about all the concepts and tasks required to accomplish this goal. We assume that the blockchain network is already available.

The topic is designed for multiple audiences:

- Solution and application architect
- Client application developer
- Smart contract developer
- Business professional

You can choose to read the topic in order, or you can select individual sections as appropriate. Individual topic sections are marked according to reader relevance, so whether you're looking for business or technical information it'll be clear when a topic is for you.

The topic follows a typical software development lifecycle. It starts with business requirements, and then covers all the major technical activities required to develop an application and smart contract to meet these requirements.

If you'd prefer, you can try out the commercial paper scenario immediately, following an abbreviated explanation, by running the commercial paper tutorial. You can return to this topic when you need fuller explanations of the concepts introduced in the tutorial.

CHAPTER 7

Tutorials

Application developers can use the Fabric tutorials to get started building their own solutions. Start working with Fabric by deploying the test network on your local machine. You can then use the steps provided by the *Deploying a smart contract to a channel* tutorial to deploy and test your smart contracts. The *Writing Your First Application* tutorial provides an introduction to how to use the APIs provided by the Fabric SDKs to invoke smart contracts from your client applications. For an in depth overview of how Fabric applications and smart contracts work together, you can visit the *Developing Applications* topic.

Network operators can use the *Deploying a smart contract to a channel* tutorial and the *Creating a channel* tutorial series to learn important aspects of administering a running network. Both network operators and application developers can use the tutorials on Private data and CouchDB to explore important Fabric features. When you are ready to deploy Hyperledger Fabric in production, see the guide for *Deploying a production network*.

There are two tutorials for updating a channel: *Updating a channel configuration* and *Updating the capability level of a channel*, while *Upgrading your components* shows how to upgrade components like peers, ordering nodes, SDKs, and more.

Finally, we provide an introduction to how to write a basic smart contract, *Writing Your First Chaincode*.

Note: If you have questions not addressed by this documentation, or run into issues with any of the tutorials, please visit the *Still Have Questions?* page for some tips on where to find additional help.

7.1 Deploying a smart contract to a channel

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

A chaincode is deployed to a channel using a process known as the Fabric chaincode lifecycle. The Fabric chaincode lifecycle allows multiple organizations to agree how a chaincode will be operated before it can be used to create

transactions. For example, while an endorsement policy specifies which organizations need to execute a chaincode to validate a transaction, channel members need to use the Fabric chaincode lifecycle to agree on the chaincode endorsement policy. For a more in-depth overview about how to deploy and manage a chaincode on a channel, see Fabric chaincode lifecycle.

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 asset-transfer (basic) 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.

7.1.1 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 asset-transfer (basic) 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*

7.1.2 Setup Logspout (optional)

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

A script to install and configure Logspout, `monitordocker.sh`, is already included in the `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.

7.1.3 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*, *JavaScript*, or *Typescript*.

Go

Before we package the chaincode, we need to install the chaincode dependencies. Navigate to the folder that contains the Go version of the asset-transfer (basic) chaincode.

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

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

```
$ cat go.mod
module github.com/hyperledger/fabric-samples/asset-transfer-basic/chaincode-go

go 1.14

require (
        github.com/golang/protobuf v1.3.2
        github.com/hyperledger/fabric-chaincode-go v0.0.0-20200424173110-d7076418f212
        github.com/hyperledger/fabric-contract-api-go v1.1.0
```

(continues on next page)

(continued from previous page)

```
        github.com/hyperledger/fabric-protos-go v0.0.0-20200424173316-dd554ba3746e
        github.com/stretchr/testify v1.5.1
)
```

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

```
// SmartContract provides functions for managing an Asset
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.

```
// CreateAsset issues a new asset to the world state with given details.
func (s *SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface, id␣
↪string, color string, size int, owner string, appraisedValue int) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if exists {
        return fmt.Errorf("the asset %s already exists", id)
    }

    asset := Asset{
        ID:             id,
        Color:          color,
        Size:           size,
        Owner:          owner,
        AppraisedValue: appraisedValue,
    }
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(id, assetJSON)
}
```

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 `asset-transfer-basic/chaincode-go` directory.

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

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

Now that we that have our dependencies, 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 basic.tar.gz --path ../asset-transfer-basic/
↪chaincode-go/ --lang golang --label basic_1.0
```

This command will create a package named `basic.tar.gz` in your current directory. 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 dependencies. Navigate to the folder that contains the JavaScript version of the asset-transfer (basic) chaincode.

```
cd fabric-samples/asset-transfer-basic/chaincode-javascript
```

The dependencies are listed in the `package.json` file in the `asset-transfer-basic/chaincode-javascript` directory. You should take a moment to examine this file. You can find the dependencies 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/assetTransfer.js` in a text editor to see the contract class imported into the smart contract and used to create the asset-transfer (basic) class.

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

class AssetTransfer extends Contract {
    ...
}
```

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

```
async CreateAsset(ctx, id, color, size, owner, appraisedValue) {
        const asset = {
            ID: id,
            Color: color,
            Size: size,
            Owner: owner,
            AppraisedValue: appraisedValue,
        };

        await ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
    }
```

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 `asset-transfer-basic/chaincode-javascript` directory.

```
npm install
```

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

Now that we that have our dependencies, 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 basic.tar.gz --path ../asset-transfer-basic/
↪chaincode-javascript/ --lang node --label basic_1.0
```

This command will create a package named `basic.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.

Typescript

Before we package the chaincode, we need to install the chaincode dependencies. Navigate to the folder that contains the TypeScript version of the asset-transfer (basic) chaincode.

```
cd fabric-samples/asset-transfer-basic/chaincode-typescript
```

The dependencies are listed in the `package.json` file in the `asset-transfer-basic/chaincode-typescript` directory. You should take a moment to examine this file. You can find the dependencies 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 `src/assetTransfer.ts` in a text editor to see the contract class imported into the smart contract and used to create the asset-transfer (basic) class. Also notice that the Asset class is imported from the type definition file `asset.ts`.

```
import { Context, Contract } from 'fabric-contract-api';
import { Asset } from './asset';

export class AssetTransfer extends Contract {
    ...
}
```

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

```
 // CreateAsset issues a new asset to the world state with given details.
    public async CreateAsset(ctx: Context, id: string, color: string, size: number,␣
↪owner: string, appraisedValue: number) {
        const asset = {
            ID: id,
            Color: color,
            Size: size,
            Owner: owner,
            AppraisedValue: appraisedValue,
        };

        await ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
    }
```

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 `asset-transfer-basic/chaincode-typescript` directory.

```
npm install
```

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

Now that we that have our dependencies, 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 basic.tar.gz --path ../asset-transfer-basic/
↪chaincode-typescript/ --lang node --label basic_1.0
```

This command will create a package named `basic.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.

7.1.4 Install the chaincode package

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

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

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

```
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

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

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

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

```
2020-07-16 10:09:57.534 CDT [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪001 Installed remotely: response:<status:200 payload:"\nJbasic_1.
↪0:e2db7f693d4aa6156e652741d5606e9c5f0de9ebb88c5721cb8248c3aead8123\022\tbasic_1.0" >
2020-07-16 10:09:57.534 CDT [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪002 Chaincode code package identifier: basic_1.
↪0:e2db7f693d4aa6156e652741d5606e9c5f0de9ebb88c5721cb8248c3aead8123
```

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

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

Issue the following command to install the chaincode:

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

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

7.1.5 Approve a chaincode definition

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

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

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

```
peer lifecycle chaincode queryinstalled
```

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

```
Installed chaincodes on peer:
Package ID: basic_1.
↪0:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3, Label: basic_1.0
```

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

```
export CC_PACKAGE_ID=basic_1.
↪0:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3
```

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

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic --
↪version 1.0 --package-id $CC_PACKAGE_ID --sequence 1 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

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

We could have provided a `--signature-policy` or `--channel-config-policy` argument to the `approveformyorg` command to specify a chaincode endorsement policy. The endorsement policy specifies how many peers belonging to different channel members need to validate a transaction against a given chaincode. Because we did not set a policy, the definition of asset-transfer (basic) will use the default endorsement policy, which requires that a transaction be endorsed by a majority of channel members present when the transaction is submitted. This implies that if new organizations are added or removed from the channel, the endorsement policy is updated automatically to require more or fewer endorsements. In this tutorial, the default policy will require a majority of 2 out of 2 and transactions will need to be endorsed by a peer from Org1 and Org2. If you want to specify a custom endorsement policy, you can use the Endorsement Policies operations guide to learn about the policy syntax.

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

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

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_ADDRESS=localhost:7051
```

You can now approve the chaincode definition as Org1.

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic --
↪version 1.0 --package-id $CC_PACKAGE_ID --sequence 1 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

We now have the majority we need to deploy the asset-transfer (basic) the chaincode to the channel. While only a

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

7.1.6 Committing the chaincode definition to the channel

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

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

```
peer lifecycle chaincode checkcommitreadiness --channelID mychannel --name basic --
↪version 1.0 --sequence 1 --tls --cafile ${PWD}/organizations/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -
↪-output json
```

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

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

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

```
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride␣
↪orderer.example.com --channelID mychannel --name basic --version 1.0 --sequence 1 --
↪tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.
↪example.com/msp/tlscacerts/tlsca.example.com-cert.pem --peerAddresses␣
↪localhost:7051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 -
↪-tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org2.example.com/peers/
↪peer0.org2.example.com/tls/ca.crt
```

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

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

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

```
peer lifecycle chaincode querycommitted --channelID mychannel --name basic --cafile $
↪{PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/
↪msp/tlscacerts/tlsca.example.com-cert.pem
```

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

```
Committed chaincode definition for chaincode 'basic' on channel 'mychannel':
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc,␣
↪Approvals: [Org1MSP: true, Org2MSP: true]
```

7.1.7 Invoking the chaincode

After the chaincode definition has been committed to a channel, the chaincode will start on the peers joined to the channel where the chaincode was installed. The asset-transfer (basic) chaincode is now ready to be invoked by client applications. Use the following command create an initial set of assets 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 --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic␣
↪--peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"function":"InitLedger","Args":[]}'
```

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

```
2020-02-12 18:22:20.576 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001␣
↪Chaincode invoke successful. result: status:200
```

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

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

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

```
[{"Key":"asset1","Record":{"ID":"asset1","color":"blue","size":5,"owner":"Tomoko",
↪"appraisedValue":300}},
{"Key":"asset2","Record":{"ID":"asset2","color":"red","size":5,"owner":"Brad",
↪"appraisedValue":400}},
{"Key":"asset3","Record":{"ID":"asset3","color":"green","size":10,"owner":"Jin Soo",
↪"appraisedValue":500}},
{"Key":"asset4","Record":{"ID":"asset4","color":"yellow","size":10,"owner":"Max",
↪"appraisedValue":600}},
{"Key":"asset5","Record":{"ID":"asset5","color":"black","size":15,"owner":"Adriana",
↪"appraisedValue":700}},
{"Key":"asset6","Record":{"ID":"asset6","color":"white","size":15,"owner":"Michel",
↪"appraisedValue":800}}]
```

7.1.8 Upgrading a smart contract

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

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

To provide a scenario for upgrading the asset-transfer (basic) chaincode that we just deployed, let's assume that Org1 and Org2 would like to install a version of the chaincode that is written in another language. They will use the Fabric chaincode lifecycle to update the chaincode version and ensure that both organizations have installed the new chaincode before it becomes active on the channel.

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

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

```
cd ../asset-transfer-basic/chaincode-javascript
npm install
cd ../../test-network
```

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

```
export PATH=${PWD}/../bin:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
peer lifecycle chaincode package basic_2.tar.gz --path ../asset-transfer-basic/
↪chaincode-javascript/ --lang node --label basic_2.0
```

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

```
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

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

```
peer lifecycle chaincode install basic_2.tar.gz
```

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

```
peer lifecycle chaincode queryinstalled
```

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

```
Installed chaincodes on peer:
Package ID: basic_1.
↪0:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3, Label: basic_1.0
Package ID: basic_2.
↪0:1d559f9fb3dd879601ee17047658c7e0c84eab732dca7c841102f20e42a9e7d4, Label: basic_2.0
```

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

```
export NEW_CC_PACKAGE_ID=basic_2.
↪0:1d559f9fb3dd879601ee17047658c7e0c84eab732dca7c841102f20e42a9e7d4
```

Org1 can now approve a new chaincode definition:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic --
↪version 2.0 --package-id $NEW_CC_PACKAGE_ID --sequence 2 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

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

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

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

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

```
peer lifecycle chaincode install basic_2.tar.gz
```

You can now approve the new chaincode definition for Org2.

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic --
↪version 2.0 --package-id $NEW_CC_PACKAGE_ID --sequence 2 --tls --cafile ${PWD}/
↪organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

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

```
peer lifecycle chaincode checkcommitreadiness --channelID mychannel --name basic --
↪version 2.0 --sequence 2 --tls --cafile ${PWD}/organizations/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -
↪-output json
```

(continues on next page)

(continued from previous page)

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

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

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

```
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride␣
↪orderer.example.com --channelID mychannel --name basic --version 2.0 --sequence 2 --
↪tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.
↪example.com/msp/tlscacerts/tlsca.example.com-cert.pem --peerAddresses␣
↪localhost:7051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 -
↪-tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org2.example.com/peers/
↪peer0.org2.example.com/tls/ca.crt
```

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

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

```
$ docker ps
CONTAINER ID        IMAGE
↪                                                                                ␣
↪                    COMMAND                  CREATED             STATUS
↪PORTS                               NAMES
7bf2f1bf792b        dev-peer0.org1.example.com-basic_2.0-
↪572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c-
↪69c9e3e44ed18cafd1e58de37a70e2ec54cd49c7da0cd461fbd5e333de32879b   "docker-
↪entrypoint.s..."   2 minutes ago       Up 2 minutes
↪          dev-peer0.org1.example.com-basic_2.0-
↪572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c
985e0967c27a        dev-peer0.org2.example.com-basic_2.0-
↪572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c-
↪158e9c6a4cb51dea043461fc4d3580e7df4c74a52b41e69a25705ce85405d760   "docker-
↪entrypoint.s..."   2 minutes ago       Up 2 minutes
↪          dev-peer0.org2.example.com-basic_2.0-
↪572cafd6a972a9b6aa3fa4f6a944efb6648d363c0ba4602f56bc8b3f9e66f46c
31fdd19c3be7        hyperledger/fabric-peer:latest
↪                                                                                ␣
↪                    "peer node start"        About an hour ago   Up About an hour
↪0.0.0.0:7051->7051/tcp             peer0.org1.example.com
1b17ff866fe0        hyperledger/fabric-peer:latest
↪                                                                                ␣
↪                    "peer node start"        About an hour ago   Up About an hour
↪7051/tcp, 0.0.0.0:9051->9051/tcp   peer0.org2.example.com
4cf170c7ae9b        hyperledger/fabric-orderer:latest
```

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

new car:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic␣
↪--peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"function":"CreateAsset","Args":["asset8","blue","16","Kelley","750"]}'
```

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

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

You should see the following result from the JavaScript chaincode:

```
[{"Key":"asset1","Record":{"ID":"asset1","color":"blue","size":5,"owner":"Tomoko",
↪"appraisedValue":300}},
{"Key":"asset2","Record":{"ID":"asset2","color":"red","size":5,"owner":"Brad",
↪"appraisedValue":400}},
{"Key":"asset3","Record":{"ID":"asset3","color":"green","size":10,"owner":"Jin Soo",
↪"appraisedValue":500}},
{"Key":"asset4","Record":{"ID":"asset4","color":"yellow","size":10,"owner":"Max",
↪"appraisedValue":600}},
{"Key":"asset5","Record":{"ID":"asset5","color":"black","size":15,"owner":"Adriana",
↪"appraisedValue":700}},
{"Key":"asset6","Record":{"ID":"asset6","color":"white","size":15,"owner":"Michel",
↪"appraisedValue":800}},
"Key":"asset8","Record":{"ID":"asset8","color":"blue","size":16,"owner":"Kelley",
↪"appraisedValue":750}}]
```

7.1.9 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
```

7.1.10 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.

7.1.11 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, msg failed to invoke backing implementation of
↪'CommitChaincodeDefinition': chaincode definition not agreed to by this org␣
↪(Org1MSP)
```

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 asset-transfer (basic) has been successfully defined on channel mychannel␣
↪and try again: chaincode definition for 'asset-transfer (basic)' exists, but␣
↪chaincode is not installed"
```

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␣
↪          STATUS              PORTS                                       NAMES
7fe1ae0a69fa        hyperledger/fabric-orderer:latest      "orderer"           5 minutes␣
↪ago       Up 4 minutes        0.0.0.0:7050->7050/tcp                      orderer.example.com
2b9c684bd07e        hyperledger/fabric-peer:latest         "peer node start"   5 minutes␣
↪ago       Up 4 minutes        0.0.0.0:7051->7051/tcp                      peer0.org1.example.
↪com
39a3e41b2573        hyperledger/fabric-peer:latest         "peer node start"   5 minutes␣
↪ago       Up 4 minutes        7051/tcp, 0.0.0.0:9051->9051/tcp            peer0.org2.example.
↪com
```

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.

7.1.12 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␣
↪[5f569e50ae58efa6261c4ad93180d49ac85ec29a07b58f576405b826a8213aeb] committed with␣
↪status (ENDORSEMENT_POLICY_FAILURE) at localhost:7051
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
↪'ApproveChaincodeDefinitionForMyOrg': could not set defaults for chaincode␣
↪definition in channel mychannel: policy '/Channel/Application/Endorsement' must be␣
↪defined for channel 'mychannel' before chaincode operations can be attempted
```

7.2 Writing Your First Application

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

It is also worth noting that this tutorial serves as an introduction to Fabric applications and uses simple smart contracts and applications. For a more in-depth look at Fabric applications and smart contracts, check out our *Developing Applications* section or the *Commercial paper tutorial*.

This tutorial provides an introduction to how Fabric applications interact with deployed blockchain networks. The tutorial uses sample programs built using the Fabric SDKs – described in detail in the *Application* topic – to invoke a smart contract which queries and updates the ledger with the smart contract API – described in detail in *Smart Contract Processing*. We will also use our sample programs and a deployed Certificate Authority to generate the X.509 certificates that an application needs to interact with a permissioned blockchain.

About Asset Transfer

This Asset Transfer (basic) sample demonstrates how to initialize a ledger with assets, query those assets, create a new asset, query a single asset based on an asset ID, update an existing asset, and transfer an asset to a new owner. It involves the following two components:

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

```
asset-transfer-basic/application-javascript
```

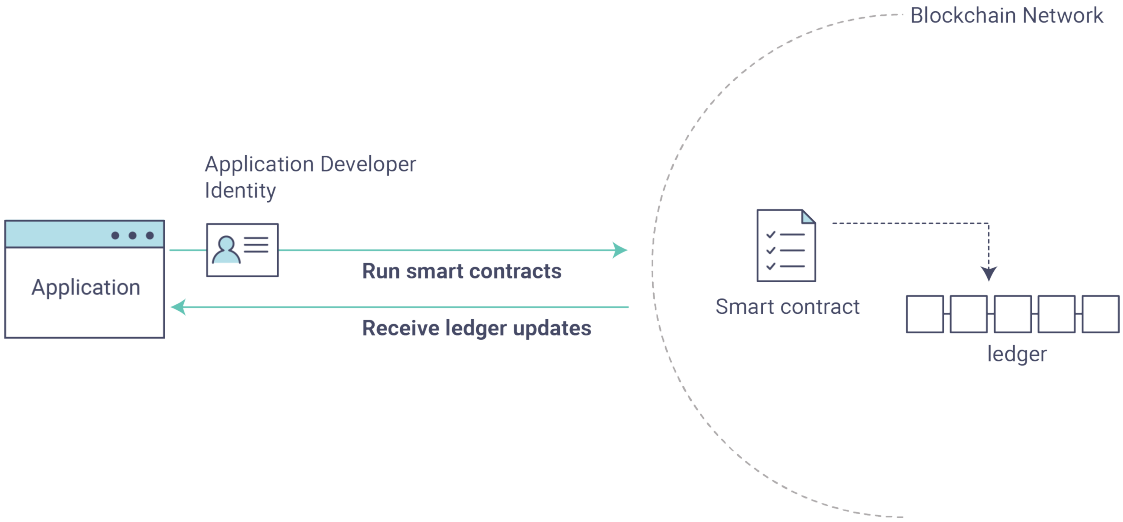
2. Smart contract itself, implementing the transactions that involve interactions with the ledger. The smart contract (chaincode) is located in the following `fabric-samples` directory:

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

Please note that for the purposes of this tutorial, the terms chaincode and smart contract are used interchangeably. For this example, we will be using the javascript chaincode.

We'll go through three principle steps:

1. Setting up a development environment. Our application needs a network to interact with, so we'll deploy a basic network for our smart contracts and application.



2. Explore a sample smart contract. We'll inspect the sample assetTransfer (javascript) smart contract to learn about the transactions within it, and how they are used by an application to query and update the ledger.

3. Interact with the smart contract with a sample application. Our application will use the assetTransfer smart contract to create, query, and update assets on the ledger. We'll get into the code of the app and the transactions they create, including initializing the ledger with assets, querying an asset, querying a range of assets, creating a new asset, and transferring an asset to a new owner.

After completing this tutorial you should have a basic understanding of how Fabric applications and smart contracts work together to manage data on the distributed ledger of a blockchain network.

7.2.1 Before you begin

In addition to the standard *Prerequisites* for Fabric, this tutorial leverages the Hyperledger Fabric SDK for Node.js. See the Node.js SDK README for a up to date list of prerequisites.

- If you are using macOS, complete the following steps:
  1. Install Homebrew.
  2. Check the Node SDK prerequisites to find out what level of Node to install.
  3. Run `brew install node` to download the latest version of node or choose a specific version, for example: `brew install node@10` according to what is supported in the prerequisites.
  4. Run `npm install`.
- If you are on Windows, you can install the windows-build-tools with npm which installs all required compilers and tooling by running the following command:

```
npm install --global windows-build-tools
```

- If you are on Linux, you need to install Python v2.7, make, and a C/C++ compiler toolchain such as GCC. You can run the following command to install the other tools:

```
sudo apt install build-essential
```

7.2.2 Set up the blockchain network

If you've already run through *Using the Fabric test network* tutorial and have a network up and running, this tutorial will bring down your running network before bringing up a new one.

Launch the network

Note: This tutorial demonstrates the JavaScript versions of the Asset Transfer smart contract and application, but the `fabric-samples` repository also contains Go, Java and TypeScript versions of this sample smart contract. To try the Go, Java or TypeScript versions, change the `javascript` argument for `./network.sh deployCC -ccl javascript` below to either `go`, `java` or `typescript` and follow the instructions written to the terminal. You may use any chaincode language sample with the javascript application sample (e.g javascript application calling go chaincode functions or javascript application calling typescript chaincode functions, etc.)

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

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

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

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

Launch the Fabric test network using the `network.sh` shell script.

```
./network.sh up createChannel -c mychannel -ca
```

This command will deploy the Fabric test network with two peers, an ordering service, and three certificate authorities (Orderer, Org1, Org2). Instead of using the cryptogen tool, we bring up the test network using Certificate Authorities, hence the `-ca` flag. Additionally, the org admin user registration is bootstrapped when the Certificate Authority is started. In a later step, we will show how the sample application completes the admin enrollment.

Next, let's deploy the chaincode by calling the `./network.sh` script with the chaincode name and language options.

```
./network.sh deployCC -ccn basic -ccl javascript
```

Note: Behind the scenes, this script uses the chaincode lifecycle to package, install, query installed chaincode, approve chaincode for both Org1 and Org2, and finally commit the chaincode.

If the chaincode is successfully deployed, the end of the output in your terminal should look similar to below:

```
Committed chaincode definition for chaincode 'basic' on channel 'mychannel':
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc,␣
↪Approvals: [Org1MSP: true, Org2MSP: true]
====================== Query chaincode definition successful on peer0.org2 on channel
↪'mychannel' ======================

===================== Chaincode initialization is not required =====================
```

Sample application

Next, let's prepare the sample Asset Transfer Javascript application that will be used to interact with the deployed chaincode.

- JavaScript application

Note that the sample application is also available in Go and Java at the links below:

- Go application
- Java application

Open a new terminal, and navigate to the `application-javascript` folder.

```
cd asset-transfer-basic/application-javascript
```

This directory contains sample programs that were developed using the Fabric SDK for Node.js. Run the following command to install the application dependencies. It may take up to a minute to complete:

```
npm install
```

This process is installing the key application dependencies defined in the application's `package.json`. The most important of which is the `fabric-network` Node.js module; it enables an application to use identities, wallets,

and gateways to connect to channels, submit transactions, and wait for notifications. This tutorial also uses the `fabric-ca-client` module to enroll users with their respective certificate authorities, generating a valid identity which is then used by the `fabric-network` module to interact with the blockchain network.

Once `npm install` completes, everything is in place to run the application. Let's take a look at the sample JavaScript application files we will be using in this tutorial. Run the following command to list the files in this directory:

```
ls
```

You should see the following:

```
app.js                  node_modules            package.json        package-lock.json
```

Note: The first part of the following section involves communication with the Certificate Authority. You may find it useful to stream the CA logs when running the upcoming programs by opening a new terminal shell and running `docker logs -f ca_org1`.

When we started the Fabric test network back in the first step, an admin user — literally called `admin` — was created as the **registrar** for the Certificate Authority (CA). Our first step is to generate the private key, public key, and X.509 certificate for `admin` by having the application call the `enrollAdmin` . This process uses a **Certificate Signing Request** (CSR) — the private and public key are first generated locally and the public key is then sent to the CA which returns an encoded certificate for use by the application. These credentials are then stored in the wallet, allowing us to act as an administrator for the CA.

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

```
node app.js
```

7.2.3 First, the application enrolls the admin user

Note: It is important to note that enrolling the admin and registering the app user are interactions that take place between the application and the Certificate Authority, not between the application and the chaincode. If you examine the chaincode in `asset-transfer-basic/chaincode-javascript/lib` you will find that the chaincode does not contain any functionality that supports enrolling the admin or registering the user.

In the sample application code below, you will see that after getting reference to the common connection profile path, making sure the connection profile exists, and specifying where to create the wallet, `enrollAdmin()` is executed and the admin credentials are generated from the Certificate Authority.

```
async function main() {
  try {
    // build an in memory object with the network configuration (also known as a␣
↪connection profile)
    const ccp = buildCCP();

    // build an instance of the fabric ca services client based on
    // the information in the network configuration
    const caClient = buildCAClient(FabricCAServices, ccp);

    // setup the wallet to hold the credentials of the application user
```

(continues on next page)

(continued from previous page)

```
    const wallet = await buildWallet(Wallets, walletPath);

    // in a real application this would be done on an administrative flow, and only␣
↪once
    await enrollAdmin(caClient, wallet);
```

This command stores the CA administrator's credentials in the `wallet` directory. You can find administrator's certificate and private key in the `wallet/admin.id` file.

Note: If you decide to start over by taking down the network and bringing it back up again, you will have to delete the `wallet` folder and its identities prior to re-running the javascript application or you will get an error. This happens because the Certificate Authority and its database are taken down when the test-network is taken down but the original wallet still remains in the application-javascript directory so it must be deleted. When you re-run the sample javascript application, a new wallet and credentials will be generated.

If you scroll back up to the beginning of the output in your terminal, it should be similar to below:

```
Wallet path: /Users/<your_username>/fabric-samples/asset-transfer-basic/application-
↪javascript/wallet
Successfully enrolled admin user and imported it into the wallet
```

Because the admin registration step is bootstrapped when the Certificate Authority is started, we only need to enroll the admin.

Note: Since the Fabric CA interactions are common across the samples, enrollAdmin() and the other CA related functions are included in the `fabric-samples/test-application/javascript/CAUtil.js` common utility.

As for the app user, we need the application to register and enroll the user in the next step.

7.2.4 Second, the application registers and enrolls an application user

Now that we have the administrator's credentials in a wallet, the application uses the `admin` user to register and enroll an app user which will be used to interact with the blockchain network. The section of the application code is shown below.

```
// in a real application this would be done only when a new user was required to be␣
↪added
// and would be part of an administrative flow
await registerUser(caClient, wallet, userId, 'org1.department1');
```

Similar to the admin enrollment, this function uses a CSR to register and enroll `appUser` and store its credentials alongside those of `admin` in the wallet. We now have identities for two separate users — `admin` and `appUser` — that can be used by our application.

Scrolling further down in your terminal output, you should see confirmation of the app user registration similar to this:

```
Successfully registered and enrolled user appUser and imported it into the wallet
```

7.2.5 Third, the sample application prepares a connection to the channel and smart contract

In the prior steps, the application generated the admin and app user credentials and placed them in the wallet. If the credentials exist and have the correct permissions attributes associated with them, the sample application user will be able to call chaincode functions after getting reference to the channel name and contract name.

Note: Our connection configuration specifies only the peer from your own Org. We tell node client sdk to use the service discovery (running on the peer), which fetches other peers that are currently online, metadata like relevant endorsement policies and any static information it would have otherwise needed to communicate with the rest of the nodes. The `asLocalhost` set to `true` tells it to connect as localhost, since our client is running on same network as the other fabric nodes. In deployments where you are not running the client on the same network as the other fabric nodes, the `asLocalhost` option would be set to `false`.

You will notice that in the following lines of application code, the application is getting reference to the Contract using the contract name and channel name via Gateway:

```
// Create a new gateway instance for interacting with the fabric network.
// In a real application this would be done as the backend server session is setup for
// a user that has been verified.
const gateway = new Gateway();

try {
  // setup the gateway instance
  // The user will now be able to create connections to the fabric network and be␣
↪able to
  // submit transactions and query. All transactions submitted by this gateway will be
  // signed by this user using the credentials stored in the wallet.
  await gateway.connect(ccp, {
    wallet,
    identity: userId,
    discovery: {enabled: true, asLocalhost: true} // using asLocalhost as this␣
↪gateway is using a fabric network deployed locally
  });

  // Build a network instance based on the channel where the smart contract is␣
↪deployed
  const network = await gateway.getNetwork(channelName);


  // Get the contract from the network.
  const contract = network.getContract(chaincodeName);
```

When a chaincode package includes multiple smart contracts, on the getContract() API you can specify both the name of the chaincode package and a specific smart contract to target. For example:

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

7.2.6 Fourth, the application initializes the ledger with some sample data

Now that we are at the point where we are actually having the sample application submit transactions, let's go through them in sequence. The application code snippets and invoked chaincode snippets are provided for each called function, as well as the terminal output.

The submitTransaction() function is used to invoke the chaincode `InitLedger` function to populate the ledger with some sample data. Under the covers, the submitTransaction() function will use service discovery to find a set of required endorsing peers for the chaincode, invoke the chaincode on the required number of peers, gather the chaincode endorsed results from those peers, and finally submit the transaction to the ordering service.

Sample application `'InitLedger'` call

```
// Initialize a set of asset data on the channel using the chaincode 'InitLedger'␣
↪function.
// This type of transaction would only be run once by an application the first time␣
↪it was started after it
// deployed the first time. Any updates to the chaincode deployed later would likely␣
↪not need to run
// an "init" type function.
console.log('\n--> Submit Transaction: InitLedger, function creates the initial set␣
↪of assets on the ledger');
await contract.submitTransaction('InitLedger');
console.log('*** Result: committed');
```

Chaincode `'InitLedger'` function

```
async InitLedger(ctx) {
     const assets = [
         {
             ID: 'asset1',
             Color: 'blue',
             Size: 5,
             Owner: 'Tomoko',
             AppraisedValue: 300,
         },
         {
             ID: 'asset2',
             Color: 'red',
             Size: 5,
             Owner: 'Brad',
             AppraisedValue: 400,
         },
         {
             ID: 'asset3',
             Color: 'green',
             Size: 10,
             Owner: 'Jin Soo',
             AppraisedValue: 500,
         },
         {
             ID: 'asset4',
             Color: 'yellow',
             Size: 10,
             Owner: 'Max',
             AppraisedValue: 600,
         },
         {
             ID: 'asset5',
             Color: 'black',
             Size: 15,
             Owner: 'Adriana',
             AppraisedValue: 700,
         },
```

(continues on next page)

(continued from previous page)

```
        {
            ID: 'asset6',
            Color: 'white',
            Size: 15,
            Owner: 'Michel',
            AppraisedValue: 800,
        },
    ];

    for (const asset of assets) {
        asset.docType = 'asset';
        await ctx.stub.putState(asset.ID, Buffer.from(JSON.stringify(asset)));
        console.info(`Asset ${asset.ID} initialized`);
    }
}
```

The terminal output entry should look similar to below:

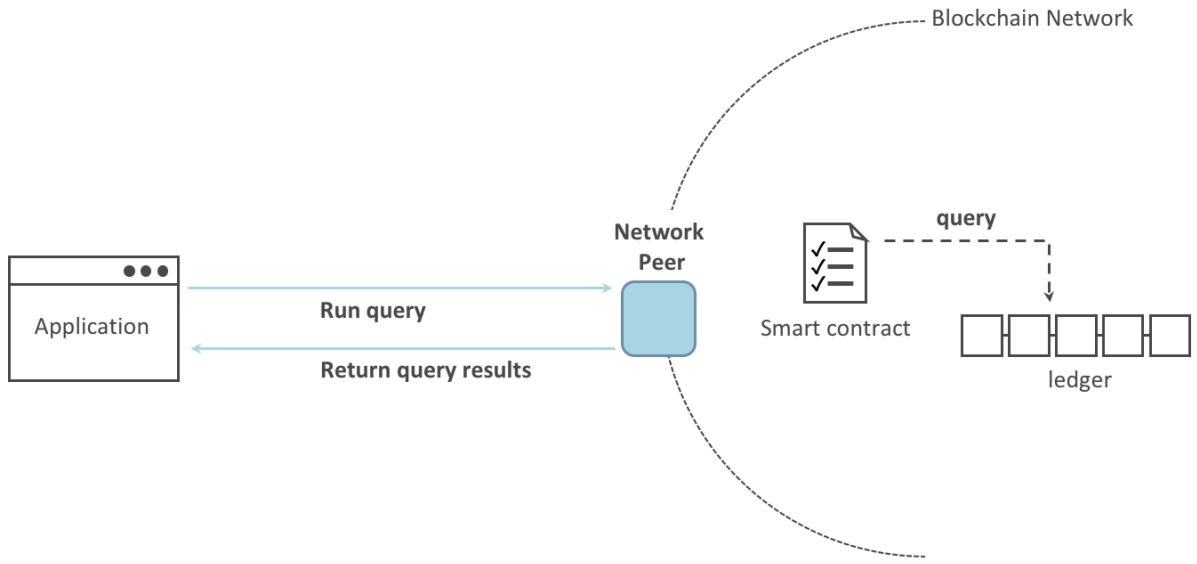
```
Submit Transaction: InitLedger, function creates the initial set of assets on the␣
↪ledger
```

7.2.7 Fifth, the application invokes each of the chaincode functions

First, a word about querying the ledger.

Each peer in a blockchain network hosts a copy of the ledger. An application program can view the most recent data from the ledger using read-only invocations of a smart contract running on your peers called a query.

Here is a simplified representation of how a query works:



The most common queries involve the current values of data in the ledger – its world state. The world state is represented as a set of key-value pairs, and applications can query data for a single key or multiple keys. Moreover, you can use complex queries to read the data on the ledger when you use CouchDB as your state database and model

your data in JSON. This can be very helpful when looking for all assets that match certain keywords with particular values; all assets with a particular owner, for example.

Below, the sample application is just getting all the assets that we populated in the prior step when we initialized the ledger with data. The evaluateTransaction() function is used when you'd like to query a single peer, without submitting a transaction to the ordering service.

Sample application `'GetAllAssets'` call

```
// Let's try a query type operation (function).
// This will be sent to just one peer and the results will be shown.
console.log('\n--> Evaluate Transaction: GetAllAssets, function returns all the␣
↪current assets on the ledger');
let result = await contract.evaluateTransaction('GetAllAssets');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);
```

Chaincode `'GetAllAssets'` function

```
// GetAllAssets returns all assets found in the world state.
 async GetAllAssets(ctx) {
     const allResults = [];
     // range query with empty string for startKey and endKey does an open-ended␣
↪query of all assets in the chaincode namespace.
     const iterator = await ctx.stub.getStateByRange('', '');
     let result = await iterator.next();
     while (!result.done) {
         const strValue = Buffer.from(result.value.value.toString()).toString('utf8');
         let record;
         try {
             record = JSON.parse(strValue);
         } catch (err) {
             console.log(err);
             record = strValue;
         }
         allResults.push({ Key: result.value.key, Record: record });
         result = await iterator.next();
     }
     return JSON.stringify(allResults);
 }
```

The terminal output should look like this:

```
  Evaluate Transaction: GetAllAssets, function returns all the current assets on the␣
↪ledger
  Result: [
  {
    "Key": "asset1",
    "Record": {
      "ID": "asset1",
      "Color": "blue",
      "Size": 5,
      "Owner": "Tomoko",
      "AppraisedValue": 300,
      "docType": "asset"
    }
  },
  {
    "Key": "asset2",
```

(continues on next page)

(continued from previous page)


```
    "Record": {
      "ID": "asset2",
      "Color": "red",
      "Size": 5,
      "Owner": "Brad",
      "AppraisedValue": 400,
      "docType": "asset"
    }
  },
  {
    "Key": "asset3",
    "Record": {
      "ID": "asset3",
      "Color": "green",
      "Size": 10,
      "Owner": "Jin Soo",
      "AppraisedValue": 500,
      "docType": "asset"
    }
  },
  {
    "Key": "asset4",
    "Record": {
      "ID": "asset4",
      "Color": "yellow",
      "Size": 10,
      "Owner": "Max",
      "AppraisedValue": 600,
      "docType": "asset"
    }
  },
  {
    "Key": "asset5",
    "Record": {
      "ID": "asset5",
      "Color": "black",
      "Size": 15,
      "Owner": "Adriana",
      "AppraisedValue": 700,
      "docType": "asset"
    }
  },
  {
    "Key": "asset6",
    "Record": {
      "ID": "asset6",
      "Color": "white",
      "Size": 15,
      "Owner": "Michel",
      "AppraisedValue": 800,
      "docType": "asset"
    }
  }
]
```


Next, the sample application submits a transaction to create 'asset13'.

Sample application `'CreateAsset'` call

```
// Now let's try to submit a transaction.
// This will be sent to both peers and if both peers endorse the transaction, the␣
↪endorsed proposal will be sent
// to the orderer to be committed by each of the peer's to the channel ledger.
console.log('\n--> Submit Transaction: CreateAsset, creates new asset with ID, color,␣
↪owner, size, and appraisedValue arguments');
await contract.submitTransaction('CreateAsset', 'asset13', 'yellow', '5', 'Tom', '1300
↪');
console.log('*** Result: committed');
```

Chaincode `'CreateAsset'` function

```
// CreateAsset issues a new asset to the world state with given details.
async CreateAsset(ctx, id, color, size, owner, appraisedValue) {
  const asset = {
      ID: id,
      Color: color,
      Size: size,
      Owner: owner,
      AppraisedValue: appraisedValue,
  };
  return ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
}
```

Terminal output:

```
Submit Transaction: CreateAsset, creates new asset with ID, color, owner, size, and␣
↪appraisedValue arguments
```

Note: In the application and chaincode snippets above, it is important to note that the sample application submits the `'CreateAsset'` transaction with the same type and number of arguments the chaincode is expecting, and in the correct sequence. In this case, the transaction name and correctly sequenced arguments are: `'CreateAsset'`, `'asset13'`, `'yellow'`, `'5'`, `'Tom'`, `'1300'` because the corresponding chaincode CreateAsset is expecting the correct sequence and type of arguments that define the asset object: sequence: ID, Color, Size, Owner, and AppraisedValue

type: ID (string), Color (string), Size (int), Owner (string), AppraisedValue (int).

The sample application then evaluates a query for 'asset13'.

Sample application `'ReadAsset'` call

```
console.log('\n--> Evaluate Transaction: ReadAsset, function returns an␣
↪asset with a given assetID');
result = await contract.evaluateTransaction('ReadAsset', 'asset13');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);
```

Chaincode `'ReadAsset'` function

```
// ReadAsset returns the asset stored in the world state with given id.
async ReadAsset(ctx, id) {
  const assetJSON = await ctx.stub.getState(id); // get the asset from␣
↪chaincode state
  if (!assetJSON || assetJSON.length === 0) {
      throw new Error(`The asset ${id} does not exist`);
  }
```

(continues on next page)

(continued from previous page)

```
  return assetJSON.toString();
}
```

Terminal output:

```
Evaluate Transaction: ReadAsset, function returns an asset with a given␣
↪assetID
Result: {
  "ID": "asset13",
  "Color": "yellow",
  "Size": "5",
  "Owner": "Tom",
  "AppraisedValue": "1300"
}
```

In the next part of the sequence, the sample application evaluates to see if `asset1` exists, which will return a boolean value of true, because we populated the ledger with `asset1` when we initialized the ledger with assets. You may recall that the original appraised value of `asset1` was `300`. The application then submits a transaction to update `asset1` with a new appraised value, and then immediately evaluates to read `asset1` from the ledger to show the new appraised value of `350`.

Sample application `'AssetExists'`, `'UpdateAsset'`, and `'ReadAsset'` calls

```
console.log('\n--> Evaluate Transaction: AssetExists, function returns "true" if an␣
↪asset with given assetID exist');
result = await contract.evaluateTransaction('AssetExists', 'asset1');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);

console.log('\n--> Submit Transaction: UpdateAsset asset1, change the appraisedValue␣
↪to 350');
await contract.submitTransaction('UpdateAsset', 'asset1', 'blue', '5', 'Tomoko', '350
↪');
console.log('*** Result: committed');

console.log('\n--> Evaluate Transaction: ReadAsset, function returns "asset1"␣
↪attributes');
result = await contract.evaluateTransaction('ReadAsset', 'asset1');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);
```

Chaincode `'AssetExists'`, `'UpdateAsset'`, and `'ReadAsset'` functions

```
// AssetExists returns true when asset with given ID exists in world state.
   async AssetExists(ctx, id) {
       const assetJSON = await ctx.stub.getState(id);
       return assetJSON && assetJSON.length > 0;
   }
// UpdateAsset updates an existing asset in the world state with provided parameters.
   async UpdateAsset(ctx, id, color, size, owner, appraisedValue) {
       const exists = await this.AssetExists(ctx, id);
       if (!exists) {
           throw new Error(`The asset ${id} does not exist`);
       }

       // overwriting original asset with new asset
       const updatedAsset = {
           ID: id,
           Color: color,
```

(continues on next page)

(continued from previous page)

```
            Size: size,
            Owner: owner,
            AppraisedValue: appraisedValue,
        };
        return ctx.stub.putState(id, Buffer.from(JSON.stringify(updatedAsset)));
    }
 // ReadAsset returns the asset stored in the world state with given id.
 async ReadAsset(ctx, id) {
     const assetJSON = await ctx.stub.getState(id); // get the asset from chaincode␣
↪state
     if (!assetJSON || assetJSON.length === 0) {
         throw new Error(`The asset ${id} does not exist`);
     }
     return assetJSON.toString();
 }
```

Terminal Output:

```
Evaluate Transaction: AssetExists, function returns "true" if an asset with given␣
↪assetID exist
Result: true

Submit Transaction: UpdateAsset asset1, change the appraisedValue to 350

Evaluate Transaction: ReadAsset, function returns "asset1" attributes
Result: {
  "ID": "asset1",
  "Color": "blue",
  "Size": "5",
  "Owner": "Tomoko",
  "AppraisedValue": "350"
}
```

In this part of the sequence, the sample application attempts to submit an `'UpdateAsset'` transaction for an asset that we know does not exist (`asset70`). We expect that we will get an error because you cannot update an asset that does not exist, which is why it is a good idea to check if an asset exists prior to attempting an asset update or deletion.

Sample application `'UpdateAsset'` call

```
try {
  // How about we try a transactions where the executing chaincode throws an error
  // Notice how the submitTransaction will throw an error containing the error thrown␣
↪by the chaincode
  console.log('\n--> Submit Transaction: UpdateAsset asset70, asset70 does not exist␣
↪and should return an error');
  await contract.submitTransaction('UpdateAsset', 'asset70', 'blue', '5', 'Tomoko',
↪'300');
  console.log('******** FAILED to return an error');
} catch (error) {
  console.log(`*** Successfully caught the error: \n    ${error}`);
}
```

Chaincode `'UpdateAsset'` function

```
// UpdateAsset updates an existing asset in the world state with provided parameters.
async UpdateAsset(ctx, id, color, size, owner, appraisedValue) {
    const exists = await this.AssetExists(ctx, id);
```

(continues on next page)

(continued from previous page)

```
    if (!exists) {
        throw new Error(`The asset ${id} does not exist`);
    }

      // overwriting original asset with new asset
      const updatedAsset = {
          ID: id,
          Color: color,
          Size: size,
          Owner: owner,
          AppraisedValue: appraisedValue,
      };
      return ctx.stub.putState(id, Buffer.from(JSON.stringify(updatedAsset)));
  }
```

Terminal output:

```
Submit Transaction: UpdateAsset asset70
2020-08-02T11:12:12.322Z - error: [Transaction]: Error: No valid responses from any␣
↪peers. Errors:
  peer=peer0.org1.example.com:7051, status=500, message=error in simulation:␣
↪transaction returned with failure: Error: The asset asset70 does not exist
  peer=peer0.org2.example.com:9051, status=500, message=error in simulation:␣
↪transaction returned with failure: Error: The asset asset70 does not exist
Expected an error on UpdateAsset of non-existing Asset: Error: No valid responses␣
↪from any peers. Errors:
  peer=peer0.org1.example.com:7051, status=500, message=error in simulation:␣
↪transaction returned with failure: Error: The asset asset70 does not exist
  peer=peer0.org2.example.com:9051, status=500, message=error in simulation:␣
↪transaction returned with failure: Error: The asset asset70 does not exist
```

In this final part of the sample application transaction sequence, the application submits a transaction to transfer an existing asset to a new owner and then reads the asset back from the ledger to display the new owner `Tom`.

Sample application `'TransferAsset'`, and `'ReadAsset'` calls

```
console.log('\n--> Submit Transaction: TransferAsset asset1, transfer to new owner of␣
↪Tom');
await contract.submitTransaction('TransferAsset', 'asset1', 'Tom');
console.log('*** Result: committed');

console.log('\n--> Evaluate Transaction: ReadAsset, function returns "asset1"␣
↪attributes');
result = await contract.evaluateTransaction('ReadAsset', 'asset1');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);
```

Chaincode `'TransferAsset'`, and `'ReadAsset'` functions

```
// TransferAsset updates the owner field of asset with given id in the world state.
async TransferAsset(ctx, id, newOwner) {
    const assetString = await this.ReadAsset(ctx, id);
    const asset = JSON.parse(assetString);
    asset.Owner = newOwner;
    return ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
}
 // ReadAsset returns the asset stored in the world state with given id.
async ReadAsset(ctx, id) {
```

(continues on next page)

(continued from previous page)

```
    const assetJSON = await ctx.stub.getState(id); // get the asset from chaincode␣
↪state
    if (!assetJSON || assetJSON.length === 0) {
        throw new Error(`The asset ${id} does not exist`);
    }
    return assetJSON.toString();
}
```

Terminal output:

```
Submit Transaction: TransferAsset asset1, transfer to new owner of Tom
Evaluate Transaction: ReadAsset, function returns "asset1" attributes
Result: {
  "ID": "asset1",
  "Color": "blue",
  "Size": "5",
  "Owner": "Tom",
  "AppraisedValue": "350"
}
```

7.2.8 A closer look

Let's take a closer look at how the sample javascript application uses the APIs provided by the Fabric Node SDK to interact with our Fabric network. Use an editor (e.g. atom or visual studio) to open `app.js` located in the `asset-transfer-basic/application-javascript` directory.

The application starts by bringing in scope two key classes from the `fabric-network` module; `Wallets` and `Gateway`. These classes will be used to locate the `appUser` identity in the wallet, and use it to connect to the network:

```
const { Gateway, Wallets } = require('fabric-network');
```

First, the program sets up the gateway connection with the userId stored in the wallet and specifies discovery options.

```
// setup the gateway instance
// The user will now be able to create connections to the fabric network and be able␣
↪to
// submit transactions and query. All transactions submitted by this gateway will be
// signed by this user using the credentials stored in the wallet.
await gateway.connect(ccp, {
  wallet,
  identity: userId,
  discovery: {enabled: true, asLocalhost: true} // using asLocalhost as this gateway␣
↪is using a fabric network deployed locally
});
```

Note at the top of the sample application code we require external utility files to build the CAClient, registerUser, enrollAdmin, buildCCP (common connection profile), and buildWallet. These utility programs are located in `AppUtil.js` in the `test-application/javascript` directory.

In `AppUtil.js`, `ccpPath` describes the path to the connection profile that our application will use to connect to our network. The connection profile was loaded from inside the `fabric-samples/test-network` directory and parsed as a JSON file:

```
const ccpPath = path.resolve(__dirname, '..', '..', 'test-network','organizations',
↪'peerOrganizations','org1.example.com', 'connection-org1.json');
```

If you'd like to understand more about the structure of a connection profile, and how it defines the network, check out the connection profile topic.

A network can be divided into multiple channels, and the next important line of code connects the application to a particular channel within the network, `mychannel`, where our smart contract was deployed. Note that we assigned constants near the top of the sample application to account for the channel name and the contract name:

```
const channelName = 'mychannel';
const chaincodeName = 'basic';
```

```
const network = await gateway.getNetwork(channelName);
```

Within this channel, we can access the asset-transfer ('basic') smart contract to interact with the ledger:

```
const contract = network.getContract(chaincodeName);
```

Within asset-transfer ('basic') there are many different **transactions**, and our application initially uses the `InitLedger` transaction to populate the ledger world state with data:

```
await contract.submitTransaction('InitLedger');
```

The `evaluateTransaction` method represents one of the simplest interactions with a smart contract in blockchain network. It simply picks a peer defined in the connection profile and sends the request to it, where it is evaluated. The smart contract queries the assets on the peer's copy of the ledger and returns the result to the application. This interaction does not result in an update of the ledger.

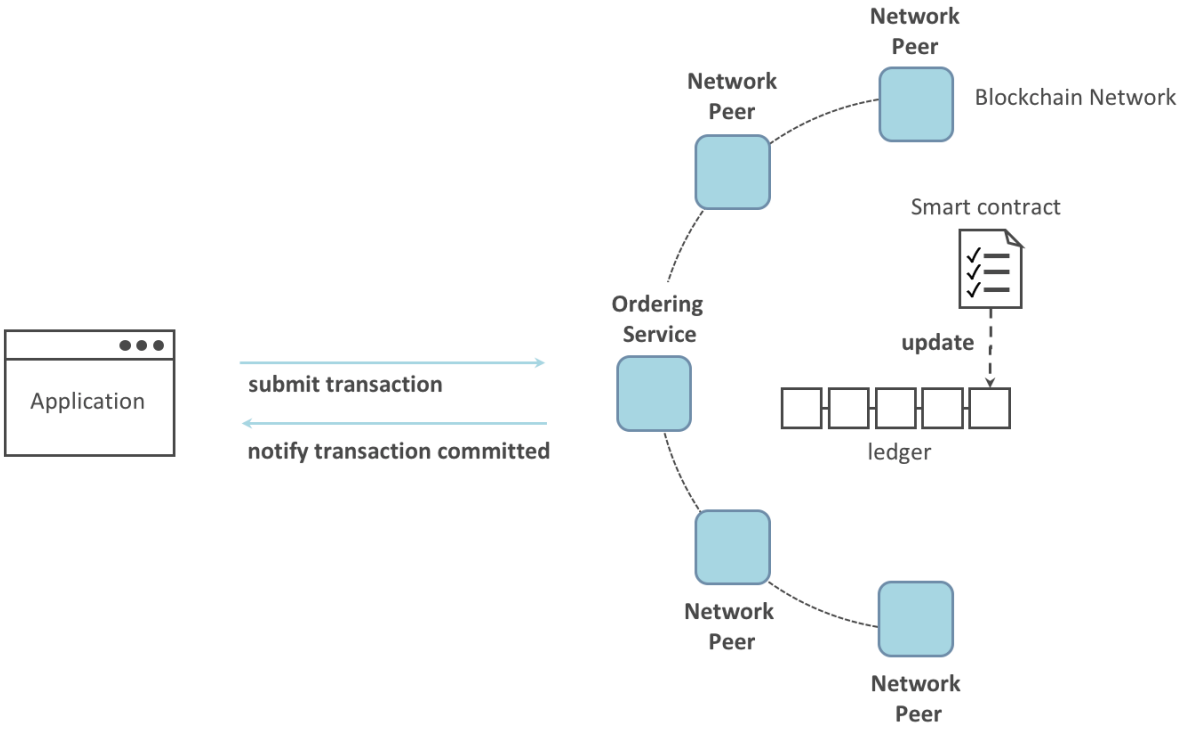
`submitTransaction` is much more sophisticated than `evaluateTransaction`. Rather than interacting with a single peer, the SDK will send the `submitTransaction` proposal to every required organization's peer in the blockchain network based on the chaincode's endorsement policy. Each of these peers will execute the requested smart contract using this proposal, to generate a transaction response which it endorses (signs) and returns to the SDK. The SDK collects all the endorsed transaction responses into a single transaction, which it then submits to the orderer. The orderer collects and sequences transactions from various application clients into a block of transactions. These blocks are distributed to every peer in the network, where every transaction is validated and committed. Finally, the SDK is notified via an event, allowing it to return control to the application.

Note: `submitTransaction` includes an event listener that checks to make sure the transaction has been validated and committed to the ledger. Applications should either utilize a commit listener, or leverage an API like `submitTransaction` that does this for you. Without doing this, your transaction may not have been successfully ordered, validated, and committed to the ledger.

`submitTransaction` does all this for the application! The process by which the application, smart contract, peers and ordering service work together to keep the ledger consistent across the network is called consensus, and it is explained in detail in this section.

7.2.9 Updating the ledger

From an application perspective, updating the ledger is simple. An application submits a transaction to the blockchain network, and when it has been validated and committed, the application receives a notification that the transaction has been successful. Behind the scenes, this involves the process of consensus whereby the different components of the blockchain network work together to ensure that every proposed update to the ledger is valid and performed in an agreed and consistent order.



7.2.10 The asset-transfer ('basic') smart contract

The smart contract sample is available in the following languages:

- Golang
- Java
- JavaScript
- Typescript

7.2.11 Clean up

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

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

This command will bring down the CAs, peers, and ordering node of the network that we created. Note that all of the data on the ledger will be lost. If you want to go through the tutorial again, you will start from a clean initial state.

7.2.12 Summary

Now that we've seen how the sample application and chaincode are written and how they interact with each other, you should have a pretty good sense of how applications interact with a blockchain network using a smart contract to query or update the ledger. You've seen the basics of the roles smart contracts, APIs, and the SDK play in queries and updates and you should have a feel for how different kinds of applications could be used to perform other business tasks and operations.

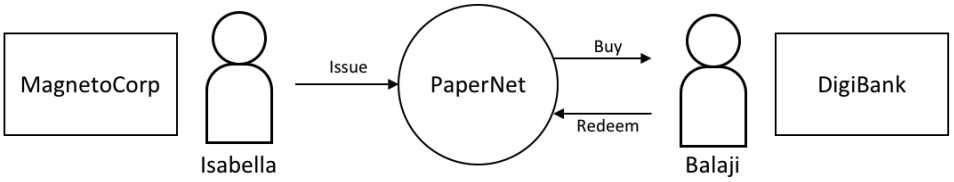
7.2.13 Additional resources

As we said in the introduction, we have a whole section on *Developing Applications* that includes in-depth information on smart contracts, process and data design, a tutorial using a more in-depth Commercial Paper tutorial and a large amount of other material relating to the development of applications.

7.3 Commercial paper tutorial

Audience: Architects, application and smart contract developers, administrators

This tutorial will show you how to install and use a commercial paper sample application and smart contract. It is a task-oriented topic, so it emphasizes procedures above concepts. When you'd like to understand the concepts in more detail, you can read the Developing Applications topic.



In this tutorial two organizations, MagnetoCorp and DigiBank, trade commercial paper with each other using PaperNet, a Hyperledger Fabric blockchain network.

Once you've set up the test network, you'll act as Isabella, an employee of MagnetoCorp, who will issue a commercial paper on its behalf. You'll then switch roles to take the role of Balaji, an employee of DigiBank, who will buy this commercial paper, hold it for a period of time, and then redeem it with MagnetoCorp for a small profit.

You'll act as a developer, end user, and administrator, each in different organizations, performing the following steps designed to help you understand what it's like to collaborate as two different organizations working independently, but according to mutually agreed rules in a Hyperledger Fabric network.

- *Set up machine* and *download samples*
- *Create the network*
- *Examine the commercial paper smart contract*
- *Deploy the smart contract to the channel* by approving the chaincode definition as MagnetoCorp and Digibank.
- Understand the structure of a MagnetoCorp *application*, including its *dependencies*
- Configure and use a *wallet and identities*
- Run a MagnetoCorp application to *issue a commercial paper*
- Understand how DigiBank uses the smart contract in their *applications*
- As Digibank, run applications that *buy* and *redeem* commercial paper

This tutorial has been tested on MacOS and Ubuntu, and should work on other Linux distributions. A Windows version is under development.

7.3.1 Prerequisites

Before you start, you must install some prerequisite technology required by the tutorial. We've kept these to a minimum so that you can get going quickly.

You **must** have the following technologies installed:

- **Node** The Node.js SDK README contains the up to date list of prerequisites.

You **will** find it helpful to install the following technologies:

- A source code editor, such as **Visual Studio Code** version 1.28, or higher. VS Code will help you develop and test your application and smart contract. Install VS Code here.

  Many excellent code editors are available including Atom, Sublime Text and Brackets.

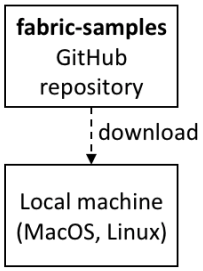
You **may** find it helpful to install the following technologies as you become more experienced with application and smart contract development. There's no requirement to install these when you first run the tutorial:

- **Node Version Manager**. NVM helps you easily switch between different versions of node – it can be really helpful if you're working on multiple projects at the same time. Install NVM here.

7.3.2 Download samples

The commercial paper tutorial is one of the samples in the `fabric-samples` repository. Before you begin this tutorial, ensure that you have followed the instructions to install the Fabric Prerequisites and Download the Samples, Binaries and Docker Images. When you are finished, you will have cloned the `fabric-samples` repository that contains the tutorial scripts, smart contract, and application files.

https://github.com/hyperledger/fabric-samples



Download the `fabric-samples` GitHub repository to your local machine.

After downloading, feel free to examine the directory structure of `fabric-samples`:

```
$ cd fabric-samples
$ ls

CODEOWNERS                SECURITY.md                 first-network
CODE_OF_CONDUCT.md        chaincode                   high-throughput
CONTRIBUTING.md           chaincode-docker-devmode    interest_rate_swaps
LICENSE                   ci                          off_chain_data
MAINTAINERS.md            commercial-paper            test-network
README.md                 fabcar
```

Notice the `commercial-paper` directory – that's where our sample is located!

You've now completed the first stage of the tutorial! As you proceed, you'll open multiple command windows for different users and components. For example:

- To show peer, orderer and CA log output from your network.

- To approve the chaincode as an administrator from MagnetoCorp and as an administrator from DigiBank.
- To run applications on behalf of Isabella and Balaji, who will use the smart contract to trade commercial paper with each other.

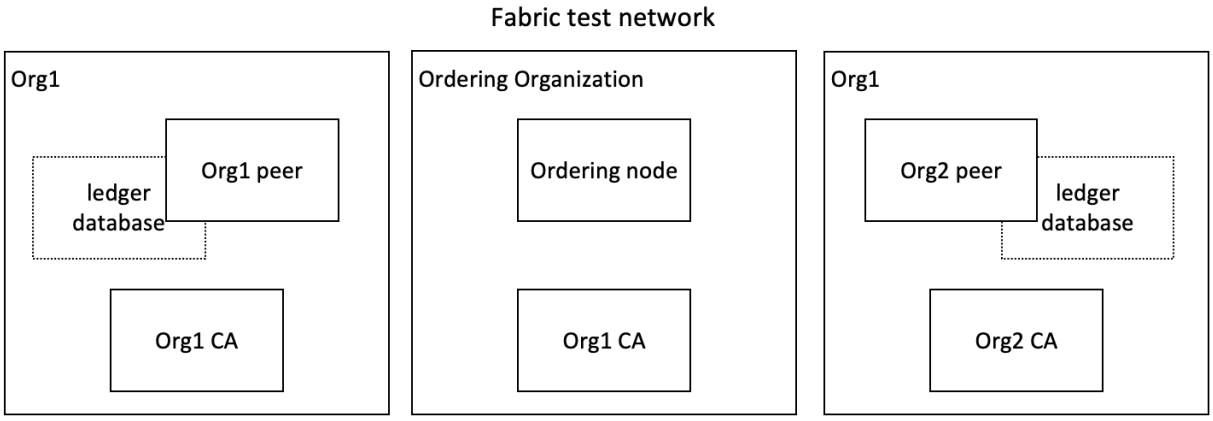
We'll make it clear when you should run a command from particular command window; for example:

```
(isabella)$ ls
```

indicates that you should run the `ls` command from Isabella's window.

7.3.3 Create the network

This tutorial will deploy a smart contract using the Fabric test network. The test network consists of two peer organizations and one ordering organization. The two peer organizations operate one peer each, while the ordering organization operates a single node Raft ordering service. We will also use the test network to create a single channel named `mychannel` that both peer organizations will be members of.



The Fabric test network is comprised of two peer organizations, Org1 and Org2, and one ordering organization. Each component runs as a Docker container.

Each organization runs their own Certificate Authority. The two peers, the state databases, the ordering service node, and each organization CA each run in their own Docker container. In production environments, organizations typically use existing CAs that are shared with other systems; they're not dedicated to the Fabric network.

The two organizations of the test network allow us to interact with a blockchain ledger as two organizations that operate separate peers. In this tutorial, we will operate Org1 of the test network as DigiBank and Org2 as MagnetoCorp.

You can start the test network and create the channel with a script provided in the commercial paper directory. Change to the `commercial-paper` directory in the `fabric-samples`:

```
cd fabric-samples/commercial-paper
```

Then use the script to start the test network:

```
./network-starter.sh
```

While the script is running, you will see logs of the test network being deployed. When the script is complete, you can use the `docker ps` command to see the Fabric nodes running on your local machine:

```
$ docker ps

CONTAINER ID        IMAGE                               COMMAND                  ␣
↪CREATED             STATUS              PORTS                                    ␣
↪  NAMES
a86f50ca1907        hyperledger/fabric-peer:latest      "peer node start"        ␣
↪About a minute ago   Up About a minute   7051/tcp, 0.0.0.0:9051->9051/tcp         ␣
↪  peer0.org2.example.com
77d0fcaee61b        hyperledger/fabric-peer:latest      "peer node start"        ␣
↪About a minute ago   Up About a minute   0.0.0.0:7051->7051/tcp                   ␣
↪  peer0.org1.example.com
7eb5f64bfe5f        hyperledger/fabric-couchdb          "tini -- /docker-ent..."  ␣
↪About a minute ago   Up About a minute   4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp␣
↪  couchdb0
2438df719f57        hyperledger/fabric-couchdb          "tini -- /docker-ent..."  ␣
↪About a minute ago   Up About a minute   4369/tcp, 9100/tcp, 0.0.0.0:7984->5984/tcp␣
↪  couchdb1
03373d116c5a        hyperledger/fabric-orderer:latest   "orderer"                ␣
↪About a minute ago   Up About a minute   0.0.0.0:7050->7050/tcp                   ␣
↪  orderer.example.com
6b4d87f65909        hyperledger/fabric-ca:latest        "sh -c 'fabric-ca-se..."  ␣
↪About a minute ago   Up About a minute   7054/tcp, 0.0.0.0:8054->8054/tcp         ␣
↪  ca_org2
7b01f5454832        hyperledger/fabric-ca:latest        "sh -c 'fabric-ca-se..."  ␣
↪About a minute ago   Up About a minute   7054/tcp, 0.0.0.0:9054->9054/tcp         ␣
↪  ca_orderer
87aef6062f23        hyperledger/fabric-ca:latest        "sh -c 'fabric-ca-se..."  ␣
↪About a minute ago   Up About a minute   0.0.0.0:7054->7054/tcp                   ␣
↪  ca_org1
```

See if you can map these containers to the nodes of the test network (you may need to horizontally scroll to locate the information):

- The Org1 peer, `peer0.org1.example.com`, is running in container `a86f50ca1907`
- The Org2 peer, `peer0.org2.example.com`, is running in container `77d0fcaee61b`
- The CouchDB database for the Org1 peer, `couchdb0`, is running in container `7eb5f64bfe5f`
- The CouchDB database for the Org2 peer, `couchdb1`, is running in container `2438df719f57`
- The Ordering node, `orderer.example.com`, is running in container `03373d116c5a`
- The Org1 CA, `ca_org1`, is running in container `87aef6062f23`
- The Org2 CA, `ca_org2`, is running in container `6b4d87f65909`
- The Ordering Org CA, `ca_orderer`, is running in container `7b01f5454832`

These containers all form a Docker network called `net_test`. You can view the network with the `docker network` command:

```
$ docker network inspect net_test

  [
      {
          "Name": "net_test",
          "Id": "f4c9712139311004b8f7acc14e9f90170c5dcfd8cdd06303c7b074624b44dc9f",
          "Created": "2020-04-28T22:45:38.525016Z",
          "Containers": {
```

(continues on next page)

(continued from previous page)

```
            "03373d116c5abf2ca94f6f00df98bb74f89037f511d6490de4a217ed8b6fbcd0": {
                "Name": "orderer.example.com",
                "EndpointID":
↪"0eed871a2aaf9a5dbcf7896aa3c0f53cc61f57b3417d36c56747033fd9f81972",
                "MacAddress": "02:42:c0:a8:70:05",
                "IPv4Address": "192.168.112.5/20",
                "IPv6Address": ""
            },
            "2438df719f57a597de592cfc76db30013adfdcfa0cec5b375f6b7259f67baff8": {
                "Name": "couchdb1",
                "EndpointID":
↪"52527fb450a7c80ea509cb571d18e2196a95c630d0f41913de8ed5abbd68993d",
                "MacAddress": "02:42:c0:a8:70:06",
                "IPv4Address": "192.168.112.6/20",
                "IPv6Address": ""
            },
            "6b4d87f65909afd335d7acfe6d79308d6e4b27441b25a829379516e4c7335b88": {
                "Name": "ca_org2",
                "EndpointID":
↪"1cc322a995880d76e1dd1f37ddf9c43f86997156124d4ecbb0eba9f833218407",
                "MacAddress": "02:42:c0:a8:70:04",
                "IPv4Address": "192.168.112.4/20",
                "IPv6Address": ""
            },
            "77d0fcaee61b8fff43d33331073ab9ce36561a90370b9ef3f77c663c8434e642": {
                "Name": "peer0.org1.example.com",
                "EndpointID":
↪"05d0d34569eee412e28313ba7ee06875a68408257dc47e64c0f4f5ef4a9dc491",
                "MacAddress": "02:42:c0:a8:70:08",
                "IPv4Address": "192.168.112.8/20",
                "IPv6Address": ""
            },
            "7b01f5454832984fcd9650f05b4affce97319f661710705e6381dfb76cd99fdb": {
                "Name": "ca_orderer",
                "EndpointID":
↪"057390288a424f49d6e9d6f788049b1e18aa28bccd56d860b2be8ceb8173ef74",
                "MacAddress": "02:42:c0:a8:70:02",
                "IPv4Address": "192.168.112.2/20",
                "IPv6Address": ""
            },
            "7eb5f64bfe5f20701aae8a6660815c4e3a81c3834b71f9e59a62fb99bed1afc7": {
                "Name": "couchdb0",
                "EndpointID":
↪"bfe740be15ec9dab7baf3806964e6b1f0b67032ce1b7ae26ac7844a1b422ddc4",
                "MacAddress": "02:42:c0:a8:70:07",
                "IPv4Address": "192.168.112.7/20",
                "IPv6Address": ""
            },
            "87aef6062f2324889074cda80fec8fe014d844e10085827f380a91eea4ccdd74": {
                "Name": "ca_org1",
                "EndpointID":
↪"a740090d33ca94dd7c6aaf14a79e1cb35109b549ee291c80195beccc901b16b7",
                "MacAddress": "02:42:c0:a8:70:03",
                "IPv4Address": "192.168.112.3/20",
                "IPv6Address": ""
            },
            "a86f50ca19079f59552e8674932edd02f7f9af93ded14db3b4c404fd6b1abe9c": {
```

(continues on next page)

(continued from previous page)

```
                "Name": "peer0.org2.example.com",
                "EndpointID":
↪"6e56772b4783b1879a06f86901786fed1c307966b72475ce4631405ba8bca79a",
                "MacAddress": "02:42:c0:a8:70:09",
                "IPv4Address": "192.168.112.9/20",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

See how the eight containers use different IP addresses, while being part of a single Docker network. (We've abbreviated the output for clarity.)

Because we are operating the test network as DigiBank and MagnetoCorp, `peer0.org1.example.com` will belong to the DigiBank organization while `peer0.org2.example.com` will be operated by MagnetoCorp. Now that the test network is up and running, we can refer to our network as PaperNet from this point forward.

To recap: you've downloaded the Hyperledger Fabric samples repository from GitHub and you've got a Fabric network running on your local machine. Let's now start to play the role of MagnetoCorp, who wishes to issue and trade commercial paper.

7.3.4 Monitor the network as MagnetoCorp

The commercial paper tutorial allows you to act as two organizations by providing two separate folders for DigiBank and MagnetoCorp. The two folders contain the smart contracts and application files for each organization. Because the two organizations have different roles in the trading of the commercial paper, the application files are different for each organization. Open a new window in the `fabric-samples` repository and use the following command to change into the MagnetoCorp directory:

```
cd commercial-paper/organization/magnetocorp
```

The first thing we are going to do as MagnetoCorp is monitor the components of PaperNet. An administrator can view the aggregated output from a set of Docker containers using the `logspout` tool. The tool collects the different output streams into one place, making it easy to see what's happening from a single window. This can be really helpful for administrators when installing smart contracts or for developers when invoking smart contracts, for example.

In the MagnetoCorp directory, run the following command to run the `monitordocker.sh` script and start the `logspout` tool for the containers associated with PaperNet running on `net_test`:

```
(magnetocorp admin)$ ./configuration/cli/monitordocker.sh net_test
...
latest: Pulling from gliderlabs/logspout
4fe2ade4980c: Pull complete
decca452f519: Pull complete
(...)
Starting monitoring on all containers on the network net_test
b7f3586e5d0233de5a454df369b8eadab0613886fc9877529587345fc01a3582
```

Note that you can pass a port number to the above command if the default port in `monitordocker.sh` is already in use.

```
(magnetocorp admin)$ ./monitordocker.sh net_test <port_number>
```

This window will now show output from the Docker containers for the remainder of the tutorial, so go ahead and open another command window. The next thing we will do is examine the smart contract that MagnetoCorp will use to issue to the commercial paper.

7.3.5 Examine the commercial paper smart contract

`issue`, `buy` and `redeem` are the three functions at the heart of the commercial paper smart contract. It is used by applications to submit transactions which correspondingly issue, buy and redeem commercial paper on the ledger. Our next task is to examine this smart contract.

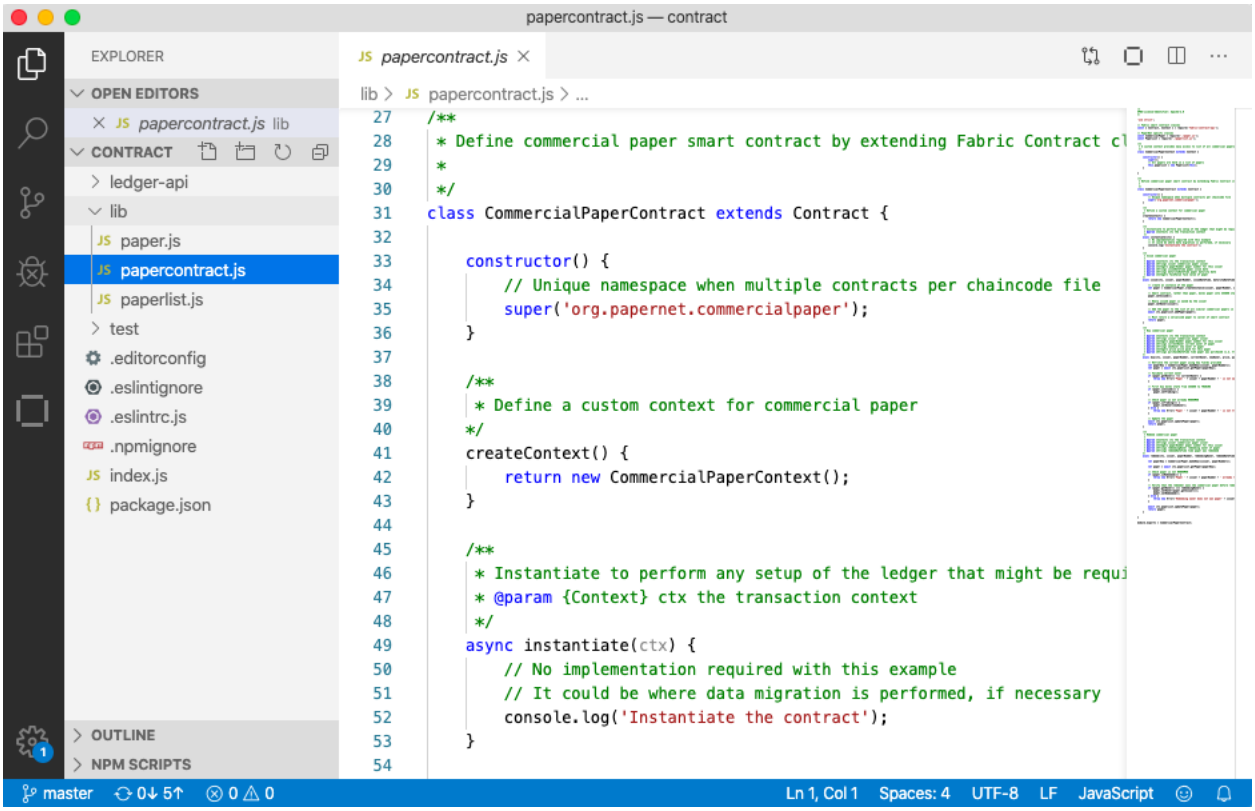
Open a new terminal in the `fabric-samples` directory and change into the MagnetoCorp folder to act as the MagnetoCorp developer.

```
cd commercial-paper/organization/magnetocorp
```

You can then view the smart contract in the `contract` directory using your chosen editor (VS Code in this tutorial):

```
(magnetocorp developer)$ code contract
```

In the `lib` directory of the folder, you'll see `papercontract.js` file – this contains the commercial paper smart contract!



An example code editor displaying the commercial paper smart contract in `papercontract.js`

`papercontract.js` is a JavaScript program designed to run in the Node.js environment. Note the following key program lines:

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

  This statement brings into scope two key Hyperledger Fabric classes that will be used extensively by the smart contract – `Contract` and `Context`. You can learn more about these classes in the `fabric-shim` JSDOCS.

- `class CommercialPaperContract extends Contract {`

  This defines the smart contract class `CommercialPaperContract` based on the built-in Fabric `Contract` class. The methods which implement the key transactions to `issue`, `buy` and `redeem` commercial paper are defined within this class.

- `async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime...) {`

  This method defines the commercial paper `issue` transaction for PaperNet. The parameters that are passed to this method will be used to create the new commercial paper.

  Locate and examine the `buy` and `redeem` transactions within the smart contract.

- `let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime...);`

  Within the `issue` transaction, this statement creates a new commercial paper in memory using the `CommercialPaper` class with the supplied transaction inputs. Examine the `buy` and `redeem` transactions to see how they similarly use this class.

- `await ctx.paperList.addPaper(paper);`

  This statement adds the new commercial paper to the ledger using `ctx.paperList`, an instance of a `PaperList` class that was created when the smart contract context `CommercialPaperContext` was initialized. Again, examine the `buy` and `redeem` methods to see how they use this class.

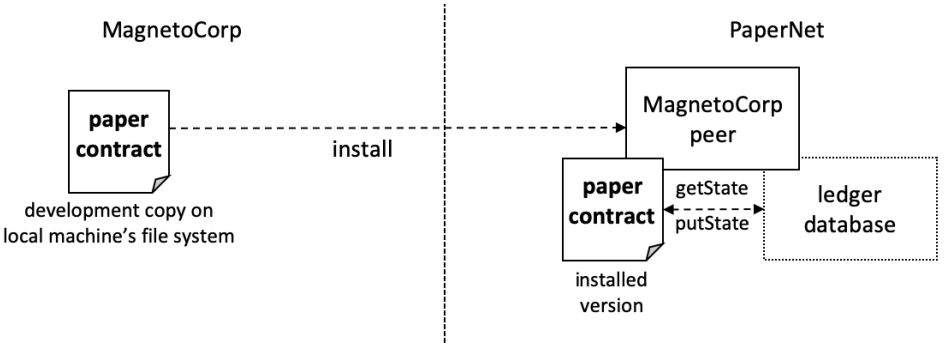
- `return paper;`

  This statement returns a binary buffer as response from the `issue` transaction for processing by the caller of the smart contract.

Feel free to examine other files in the `contract` directory to understand how the smart contract works, and read in detail how `papercontract.js` is designed in the smart contract processing topic.

7.3.6 Deploy the smart contract to the channel

Before `papercontract` can be invoked by applications, it must be installed onto the appropriate peer nodes of the test network and then defined on the channel using the Fabric chaincode lifecycle. The Fabric chaincode lifecycle allows multiple organizations to agree to the parameters of a chaincode before the chaincode is deployed to a channel. As a result, we need to install and approve the chaincode as administrators of both MagnetoCorp and DigiBank.



A MagnetoCorp administrator installs a copy of the `papercontract` onto a MagnetoCorp peer.

Smart contracts are the focus of application development, and are contained within a Hyperledger Fabric artifact called chaincode. One or more smart contracts can be defined within a single chaincode, and installing a chaincode will allow them to be consumed by the different organizations in PaperNet. It means that only administrators need to worry about chaincode; everyone else can think in terms of smart contracts.

Install and approve the smart contract as MagnetoCorp

We will first install and approve the smart contract as the MagnetoCorp admin. Make sure that you are operating from the `magnetocorp` folder, or navigate back to that folder using the following command:

```
cd commercial-paper/organization/magnetocorp
```

A MagnetoCorp administrator can interact with PaperNet using the `peer` CLI. However, the administrator needs to set certain environment variables in their command window to use the correct set of `peer` binaries, send commands to the address of the MagnetoCorp peer, and sign requests with the correct cryptographic material.

You can use a script provided by the sample to set the environment variables in your command window. Run the following command in the `magnetocorp` directory:

```
source magnetocorp.sh
```

You will see the full list of environment variables printed in your window. We can now use this command window to interact with PaperNet as the MagnetoCorp administrator.

The first step is to install the `papercontract` smart contract. The smart contract can be packaged into a chaincode using the `peer lifecycle chaincode package` command. In the MagnetoCorp administrator's command window, run the following command to create the chaincode package:

```
(magnetocorp admin)$ peer lifecycle chaincode package cp.tar.gz --lang node --path ./
↪contract --label cp_0
```

The MagnetoCorp admin can now install the chaincode on the MagnetoCorp peer using the `peer lifecycle chaincode install` command:

```
(magnetocorp admin)$ peer lifecycle chaincode install cp.tar.gz
```

When the chaincode package is installed, you will see messages similar to the following printed in your terminal:

```
2020-01-30 18:32:33.762 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪001 Installed remotely: response:<status:200 payload:"\nEcp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c\022\004cp_0" >
2020-01-30 18:32:33.762 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪002 Chaincode code package identifier: cp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

Because the MagnetoCorp admin has set `CORE_PEER_ADDRESS=localhost:9051` to target its commands to `peer0.org2.example.com`, the `INFO 001 Installed remotely...` indicates that `papercontract` has been successfully installed on this peer.

After we install the smart contract, we need to approve the chaincode definition for `papercontract` as MagnetoCorp. The first step is to find the packageID of the chaincode we installed on our peer. We can query the packageID using the `peer lifecycle chaincode queryinstalled` command:

```
peer lifecycle chaincode queryinstalled
```

The command will return the same package identifier as the install command. You should see output similar to the following:

```
Installed chaincodes on peer:
Package ID: cp_0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c,␣
↪Label: cp_0
```

We will need the package ID in the next step, so we will save it as an environment variable. The package ID may not be the same for all users, so you need to complete this step using the package ID returned from your command window.

```
export PACKAGE_ID=cp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

The admin can now approve the chaincode definition for MagnetoCorp using the `peer lifecycle chaincode approveformyorg` command:

```
(magnetocorp admin)$ peer lifecycle chaincode approveformyorg --orderer␣
↪localhost:7050 --ordererTLSHostnameOverride orderer.example.com --channelID␣
↪mychannel --name papercontract -v 0 --package-id $PACKAGE_ID --sequence 1 --tls --
↪cafile $ORDERER_CA
```

One of the most important chaincode parameters that channel members need to agree to using the chaincode definition is the chaincode endorsement policy. The endorsement policy describes the set of organizations that must endorse (execute and sign) a transaction before it can be determined to be valid. By approving the `papercontract` chaincode without the `--policy` flag, the MagnetoCorp admin agrees to using the channel's default `Endorsement` policy, which in the case of the `mychannel` test channel requires a majority of organizations on the channel to endorse a transaction. All transactions, whether valid or invalid, will be recorded on the ledger blockchain, but only valid transactions will update the world state.

Install and approve the smart contract as DigiBank

Based on the `mychannel` `LifecycleEndorsement` policy, the Fabric Chaincode lifecycle will require a majority of organizations on the channel to agree to the chaincode definition before the chaincode can be committed to the channel. This implies that we need to approve the `papernet` chaincode as both MagnetoCorp and DigiBank to get the required majority of 2 out of 2. Open a new terminal window in the `fabric-samples` and navigate to the folder that contains the DigiBank smart contract and application files:

```
(digibank admin)$ cd commercial-paper/organization/digibank/
```

Use the script in the DigiBank folder to set the environment variables that will allow you to act as the DigiBank admin:

```
source digibank.sh
```

We can now install and approve `papercontract` as the DigiBank. Run the following command to package the chaincode:

```
(digibank admin)$ peer lifecycle chaincode package cp.tar.gz --lang node --path ./
↪contract --label cp_0
```

The admin can now install the chaincode on the DigiBank peer:

```
(digibank admin)$ peer lifecycle chaincode install cp.tar.gz
```

We then need to query and save the packageID of the chaincode that was just installed:

```
(digibank admin)$ peer lifecycle chaincode queryinstalled
```

Save the package ID as an environment variable. Complete this step using the package ID returned from your console.

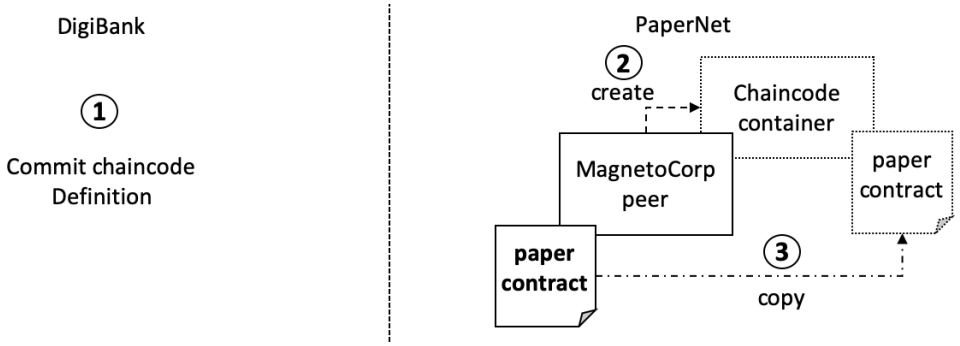
```
export PACKAGE_ID=cp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

The Digibank admin can now approve the chaincode definition of `papercontract`:

```
(digibank admin)$ peer lifecycle chaincode approveformyorg --orderer localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name␣
↪papercontract -v 0 --package-id $PACKAGE_ID --sequence 1 --tls --cafile $ORDERER_CA
```

Commit the chaincode definition to the channel

Now that DigiBank and MagnetoCorp have both approved the `papernet` chaincode, we have the majority we need (2 out of 2) to commit the chaincode definition to the channel. Once the chaincode is successfully defined on the channel, the `CommercialPaper` smart contract inside the `papercontract` chaincode can be invoked by client applications on the channel. Since either organization can commit the chaincode to the channel, we will continue operating as the DigiBank admin:



After the DigiBank administrator commits the definition of the `papercontract` chaincode to the channel, a new Docker chaincode container will be created to run `papercontract` on both PaperNet peers

The DigiBank administrator uses the `peer lifecycle chaincode commit` command to commit the chaincode definition of `papercontract` to `mychannel`:

```
(digibank admin)$ peer lifecycle chaincode commit -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --peerAddresses localhost:7051 --
↪tlsRootCertFiles ${PEER0_ORG1_CA} --peerAddresses localhost:9051 --tlsRootCertFiles
↪${PEER0_ORG2_CA} --channelID mychannel --name papercontract -v 0 --sequence 1 --tls␣
↪--cafile $ORDERER_CA --waitForEvent
```

The chaincode container will start after the chaincode definition has been committed to the channel. You can use the `docker ps` command to see `papercontract` container starting on both peers.

```
(digibank admin)$ docker ps

CONTAINER ID        IMAGE
↪                                                                                  ␣
↪               COMMAND                  CREATED             STATUS              ␣
↪PORTS                                    NAMES
d4ba9dc9c55f        dev-peer0.org1.example.com-cp_0-
↪ebef35e7f1f25eea1dcc6fcad5019477cd7f434c6a5dcaf4e81744e282903535-
↪05cf67c20543ee1c24cf7dfe74abce99785374db15b3bc1de2da372700c25608   "docker-
↪entrypoint.s..."   30 seconds ago      Up 28 seconds                           ␣
↪                    dev-peer0.org1.example.com-cp_0-
↪ebef35e7f1f25eea1dcc6fcad5019477cd7f434c6a5dcaf4e81744e282903535
a944c0f8b6d6        dev-peer0.org2.example.com-cp_0-
↪1487670371e56d107b5e980ce7f66172c89251ab21d484c7f988c02912ddeaec-
↪1a147b6fd2a8bd2ae12db824fad8d08a811c30cc70bc5b6bc49a2cbebc2e71ee   "docker-
↪entrypoint.s..."   31 seconds ago      Up 28 seconds                           ␣
↪                    dev-peer0.org2.example.com-cp_0-
↪1487670371e56d107b5e980ce7f66172c89251ab21d484c7f988c02912ddeaec
```

(continues on next page)

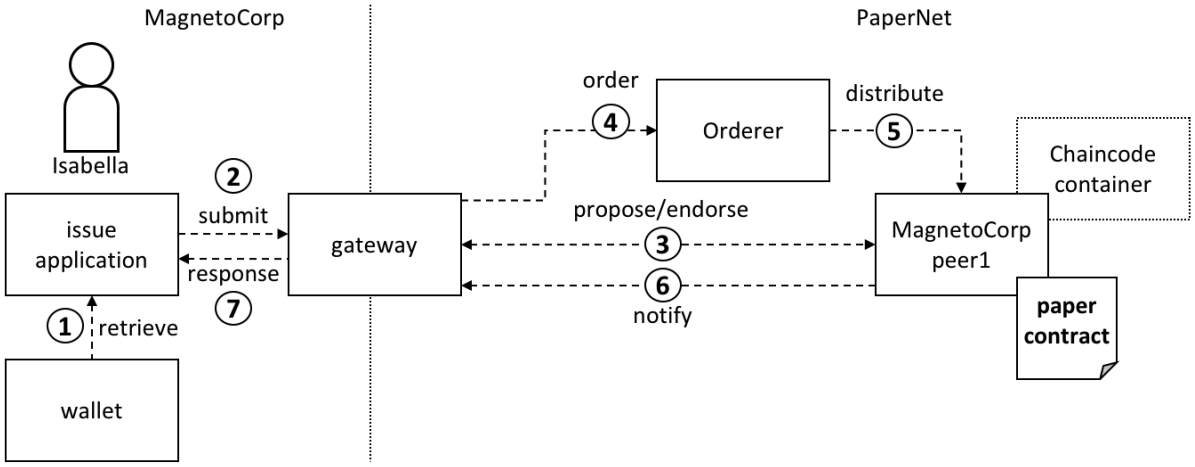
(continued from previous page)

Notice that the containers are named to indicate the peer that started it, and the fact that it's running `papercontract` version `0`.

Now that we have deployed the `papercontract` chaincode to the channel, we can use the MagnetoCorp application to issue the commercial paper. Let's take a moment to examine the application structure.

7.3.7 Application structure

The smart contract contained in `papercontract` is called by MagnetoCorp's application `issue.js`. Isabella uses this application to submit a transaction to the ledger which issues commercial paper `00001`. Let's quickly examine how the `issue` application works.



A gateway allows an application to focus on transaction generation, submission and response. It coordinates transaction proposal, ordering and notification processing between the different network components.

Because the `issue` application submits transactions on behalf of Isabella, it starts by retrieving Isabella's X.509 certificate from her wallet, which might be stored on the local file system or a Hardware Security Module HSM. The `issue` application is then able to utilize the gateway to submit transactions on the channel. The Hyperledger Fabric SDK provides a gateway abstraction so that applications can focus on application logic while delegating network interaction to the gateway. Gateways and wallets make it straightforward to write Hyperledger Fabric applications.

So let's examine the `issue` application that Isabella is going to use. Open a separate terminal window for her, and in `fabric-samples` locate the MagnetoCorp `/application` folder:

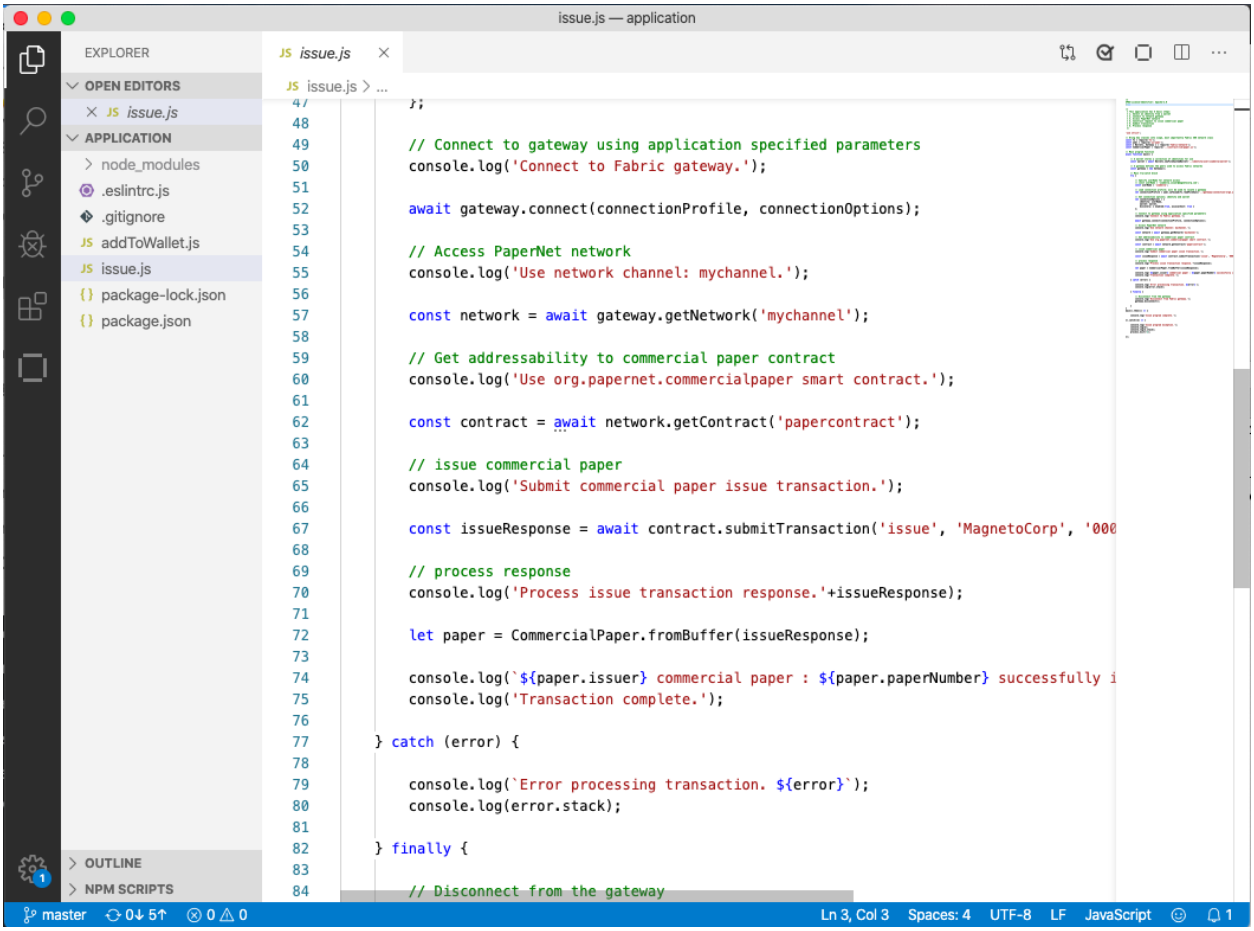
```
(isabella)$ cd commercial-paper/organization/magnetocorp/application/
(isabella)$ ls

addToWallet.js      enrollUser.js       issue.js        package.json
```

`addToWallet.js` is the program that Isabella is going to use to load her identity into her wallet, and `issue.js` will use this identity to create commercial paper `00001` on behalf of MagnetoCorp by invoking `papercontract`.

Change to the directory that contains MagnetoCorp's copy of the application `issue.js`, and use your code editor to examine it:

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application
(isabella)$ code issue.js
```

Examine this directory; it contains the issue application and all its dependencies.



A code editor displaying the contents of the commercial paper application directory.

Note the following key program lines in `issue.js`:

- `const { Wallets, Gateway } = require('fabric-network');`

  This statement brings two key Hyperledger Fabric SDK classes into scope – `Wallet` and `Gateway`.

- `const wallet = await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');`

  This statement identifies that the application will use `isabella` wallet when it connects to the blockchain network channel. Because Isabella's X.509 certificate is in the local file system, the application creates a new `FileSystemWallet`. The application will select a particular identity within `isabella` wallet.

- `await gateway.connect(connectionProfile, connectionOptions);`

  This line of code connects to the network using the gateway identified by `connectionProfile`, using the identity referred to in `ConnectionOptions`.

  See how `../gateway/networkConnection.yaml` and `User1@org1.example.com` are used for these values respectively.

- `const network = await gateway.getNetwork('mychannel');`

  This connects the application to the network channel `mychannel`, where the `papercontract` was previously deployed.

- `const contract = await network.getContract('papercontract');`

This statement gives the application access to the `papercontract` chaincode. Once an application has issued getContract, it can submit to any smart contract transaction implemented within the chaincode.

- `const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001', ...);`

  This line of code submits the a transaction to the network using the `issue` transaction defined within the smart contract. `MagnetoCorp`, `00001...` are the values to be used by the `issue` transaction to create a new commercial paper.

- `let paper = CommercialPaper.fromBuffer(issueResponse);`

  This statement processes the response from the `issue` transaction. The response needs to deserialized from a buffer into `paper`, a `CommercialPaper` object which can interpreted correctly by the application.

Feel free to examine other files in the `/application` directory to understand how `issue.js` works, and read in detail how it is implemented in the application topic.

7.3.8 Application dependencies

The `issue.js` application is written in JavaScript and designed to run in the Node.js environment that acts as a client to the PaperNet network. As is common practice, MagnetoCorp's application is built on many external node packages — to improve quality and speed of development. Consider how `issue.js` includes the `js-yaml` package to process the YAML gateway connection profile, or the `fabric-network` package to access the `Gateway` and `Wallet` classes:

```
const yaml = require('js-yaml');
const { Wallets, Gateway } = require('fabric-network');
```

These packages have to be downloaded from npm to the local file system using the `npm install` command. By convention, packages must be installed into an application-relative `/node_modules` directory for use at runtime.

Open the `package.json` file to see how `issue.js` identifies the packages to download and their exact versions by examining the "dependencies" section of the file.

npm versioning is very powerful; you can read more about it here.

Let's install these packages with the `npm install` command – this may take up to a minute to complete:

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application/
(isabella)$ npm install

(           ) extract:lodash: sill extract ansi-styles@3.2.1
(...)
added 738 packages in 46.701s
```

See how this command has updated the directory:

```
(isabella)$ ls

enrollUser.js       node_modules            package.json
issue.js            package-lock.json
```

Examine the `node_modules` directory to see the packages that have been installed. There are lots, because `js-yaml` and `fabric-network` are themselves built on other npm packages! Helpfully, the `package-lock.json` file identifies the exact versions installed, which can prove invaluable if you want to exactly reproduce environments; to test, diagnose problems or deliver proven applications for example.

7.3.9 Wallet

Isabella is almost ready to run `issue.js` to issue MagnetoCorp commercial paper `00001`; there's just one remaining task to perform! As `issue.js` acts on behalf of Isabella, and therefore MagnetoCorp, it will use identity from her wallet that reflects these facts. We now need to perform this one-time activity of generating the appropriate X.509 credentials to her wallet.

The MagnetoCorp Certificate Authority running on PaperNet, `ca_org2`, has an application user that was registered when the network was deployed. Isabella can use the identity name and secret to generate the X.509 cryptographic material for the `issue.js` application. The process of using a CA to generate client side cryptographic material is referred to as **enrollment**. In a real word scenario, a network operator would provide the name and secret of a client identity that was registered with the CA to an application developer. The developer would then use the credentials to enroll their application and interact with the network.

The `enrollUser.js` program uses the `fabric-ca-client` class to generate a private and public key pair, and then issues a **Certificate Signing Request** to the CA. If the identiy name and secret submitted by Isabella match the credentials registered with the CA, the CA will issue and sign a certificate that encodes the public key, establishing that Isabella belongs to MagnetoCorp. When the signing request is complete, `enrollUser.js` stores the private key and signing certificate in Isabella's wallet. You can examine the `enrollUser.js` file to learn more about how the Node SDK uses the `fabric-ca-client` class to complete these tasks.

In Isabella's terminal window, run the `enrollUser.js` program to add identity information to her wallet:

```
(isabella)$ node enrollUser.js

Wallet path: /Users/nikhilgupta/fabric-samples/commercial-paper/organization/
↪magnetocorp/identity/user/isabella/wallet
Successfully enrolled client user "isabella" and imported it into the wallet
```

We can now turn our focus to the result of this program — the contents of the wallet which will be used to submit transactions to PaperNet:

```
(isabella)$ ls ../identity/user/isabella/wallet/

isabella.id
```

Isabella can store multiple identities in her wallet, though in our example, she only uses one. The `wallet` folder contains an `isabella.id` file that provides the information that Isabella needs to connect to the network. Other identities used by Isabella would have their own file. You can open this file to see the identity information that `issue.js` will use on behalf of Isabella inside a JSON file. The output has been formatted for clarity.

```
(isabella)$  cat ../identity/user/isabella/wallet/*

{
  "credentials": {
    "certificate": "-----BEGIN CERTIFICATE-----
↪\nMIICKTCCAdCgAwIBAgIQWKwvLG+sqeO3LwwQK6avZDAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMKQ2
↪53dbo00wSzAOBgNVHQ8BAf8EBAMCB4AwDAYDVR0TAQH/
↪BAIwADArBgNV\nHSMEJDAigCDOCdm4irsZFU3D6Hak4+84QRg1N43iwg8w1V6DRhgLyDAKBggqhkjO\nPQQDAgNHADBEAiBhzK
↪mRtUdaJagIgIgYpbZ\nXf0CSiTXIWOJIsswN4Jp+ZxkJfFVmXndqKqz+VM=\n-----END CERTIFICATE---
↪--\n",
    "privateKey": "-----BEGIN PRIVATE KEY-----
↪\nMIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQggs55vQg2oXi8gNi8\nNidE8Fy5zenohArDq3FGJD8cKU2hRAN
↪53db\n-----END PRIVATE KEY-----\n"
  },
  "mspId": "Org2MSP",
  "type": "X.509",
```

(continues on next page)

(continued from previous page)

```
  "version": 1
}
```

In the file you can notice the following:

- a `"privateKey":` used to sign transactions on Isabella's behalf, but not distributed outside of her immediate control.
- a `"certificate":` which contains Isabella's public key and other X.509 attributes added by the Certificate Authority at certificate creation. This certificate is distributed to the network so that different actors at different times can cryptographically verify information created by Isabella's private key.

You can Learn more about certificates here. In practice, the certificate file also contains some Fabric-specific metadata such as Isabella's organization and role – read more in the wallet topic.

7.3.10 Issue application

Isabella can now use `issue.js` to submit a transaction that will issue MagnetoCorp commercial paper `00001`:

```
(isabella)$ node issue.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper issue transaction.
Process issue transaction response.{"class":"org.papernet.commercialpaper","key":"\
↪"MagnetoCorp\":\"00001\"","currentState":1,"issuer":"MagnetoCorp","paperNumber":
↪"00001","issueDateTime":"2020-05-31","maturityDateTime":"2020-11-30","faceValue":
↪"5000000","owner":"MagnetoCorp"}
MagnetoCorp commercial paper : 00001 successfully issued for value 5000000
Transaction complete.
Disconnect from Fabric gateway.
Issue program complete.
```

The `node` command initializes a Node.js environment, and runs `issue.js`. We can see from the program output that MagnetoCorp commercial paper 00001 was issued with a face value of 5M USD.

As you've seen, to achieve this, the application invokes the `issue` transaction defined in the `CommercialPaper` smart contract within `papercontract.js`. The smart contract interacts with the ledger via the Fabric APIs, most notably `putState()` and `getState()`, to represent the new commercial paper as a vector state within the world state. We'll see how this vector state is subsequently manipulated by the `buy` and `redeem` transactions also defined within the smart contract.

All the time, the underlying Fabric SDK handles the transaction endorsement, ordering and notification process, making the application's logic straightforward; the SDK uses a gateway to abstract away network details and connectionOptions to declare more advanced processing strategies such as transaction retry.

Let's now follow the lifecycle of MagnetoCorp 00001 by switching our emphasis to an employee of DigiBank, Balaji, who will buy the commercial paper using a DigiBank application.

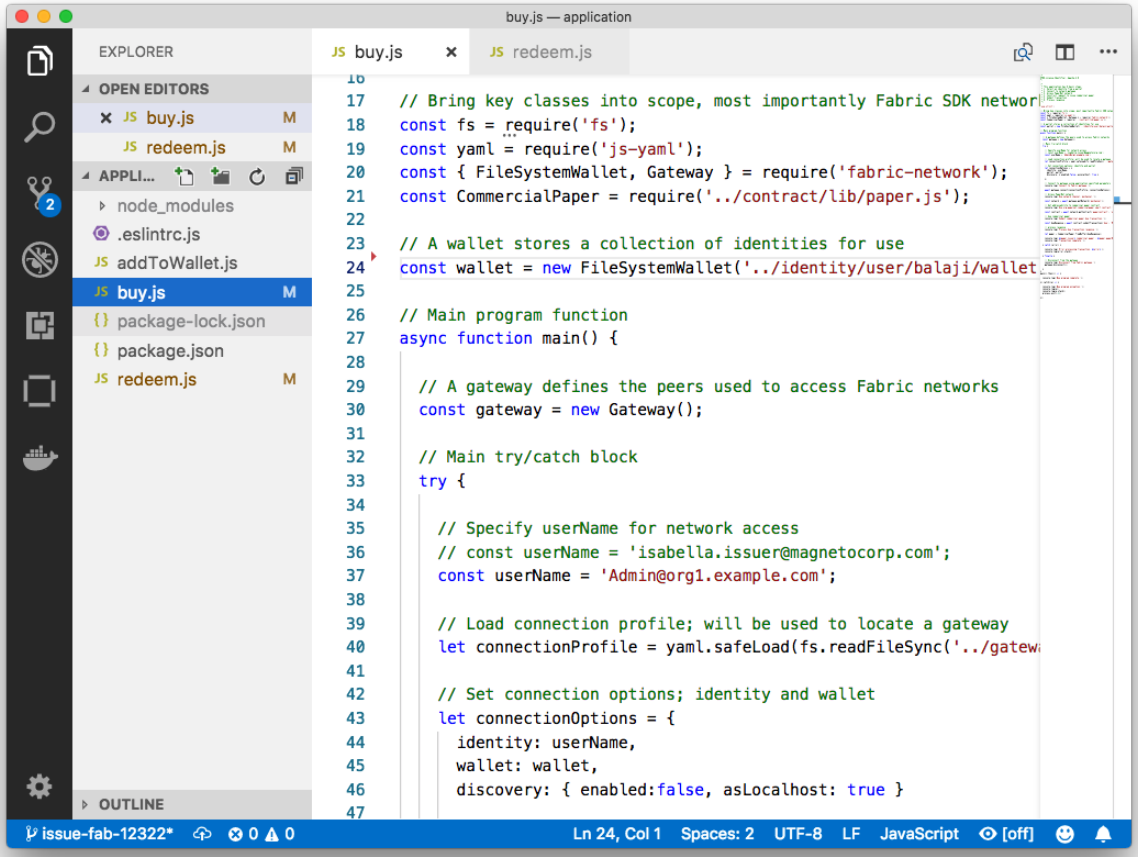
7.3.11 Digibank applications

Balaji uses DigiBank's `buy` application to submit a transaction to the ledger which transfers ownership of commercial paper `00001` from MagnetoCorp to DigiBank. The `CommercialPaper` smart contract is the same as that used by MagnetoCorp's application, however the transaction is different this time – it's `buy` rather than `issue`. Let's examine how DigiBank's application works.

Open a separate terminal window for Balaji. In `fabric-samples`, change to the DigiBank application directory that contains the application, `buy.js`, and open it with your editor:

```
(balaji)$ cd commercial-paper/organization/digibank/application/
(balaji)$ code buy.js
```

As you can see, this directory contains both the `buy` and `redeem` applications that will be used by Balaji.



DigiBank's commercial paper directory containing the `buy.js` and `redeem.js` applications.

DigiBank's `buy.js` application is very similar in structure to MagnetoCorp's `issue.js` with two important differences:

- **Identity**: the user is a DigiBank user `Balaji` rather than MagnetoCorp's `Isabella`

```
const wallet = await Wallets.newFileSystemWallet('../identity/user/balaji/wallet
↪');
```

  See how the application uses the `balaji` wallet when it connects to the PaperNet network channel. `buy.js` selects a particular identity within `balaji` wallet.

- **Transaction**: the invoked transaction is `buy` rather than `issue`

```
const buyResponse = await contract.submitTransaction('buy', 'MagnetoCorp', '00001
↪', ...);
```

A `buy` transaction is submitted with the values `MagnetoCorp`, `00001`, ..., that are used by the `CommercialPaper` smart contract class to transfer ownership of commercial paper `00001` to DigiBank.

Feel free to examine other files in the `application` directory to understand how the application works, and read in detail how `buy.js` is implemented in the application topic.

7.3.12 Run as DigiBank

The DigiBank applications which buy and redeem commercial paper have a very similar structure to MagnetoCorp's issue application. Therefore, let's install their dependencies and set up Balaji's wallet so that he can use these applications to buy and redeem commercial paper.

Like MagnetoCorp, Digibank must the install the required application packages using the `npm install` command, and again, this make take a short time to complete.

In the DigiBank administrator window, install the application dependencies:

```
(digibank admin)$ cd commercial-paper/organization/digibank/application/
(digibank admin)$ npm install

(            ) extract:lodash: sill extract ansi-styles@3.2.1
(...)
added 738 packages in 46.701s
```

In Balaji's command window, run the `enrollUser.js` program to generate a certificate and private key and them to his wallet:

```
(balaji)$ node enrollUser.js

Wallet path: /Users/nikhilgupta/fabric-samples/commercial-paper/organization/digibank/
↪identity/user/balaji/wallet
Successfully enrolled client user "balaji" and imported it into the wallet
```

The `addToWallet.js` program has added identity information for `balaji`, to his wallet, which will be used by `buy.js` and `redeem.js` to submit transactions to `PaperNet`.

Like Isabella, Balaji can store multiple identities in his wallet, though in our example, he only uses one. His corresponding id file at `digibank/identity/user/balaji/wallet/balaji.id` is very similar Isabella's — feel free to examine it.

7.3.13 Buy application

Balaji can now use `buy.js` to submit a transaction that will transfer ownership of MagnetoCorp commercial paper 00001 to DigiBank.

Run the `buy` application in Balaji's window:

```
(balaji)$ node buy.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper buy transaction.
Process buy transaction response.
MagnetoCorp commercial paper : 00001 successfully purchased by DigiBank
Transaction complete.
```

(continues on next page)

(continued from previous page)

```
Disconnect from Fabric gateway.
Buy program complete.
```

You can see the program output that MagnetoCorp commercial paper 00001 was successfully purchased by Balaji on behalf of DigiBank. `buy.js` invoked the `buy` transaction defined in the `CommercialPaper` smart contract which updated commercial paper `00001` within the world state using the `putState()` and `getState()` Fabric APIs. As you've seen, the application logic to buy and issue commercial paper is very similar, as is the smart contract logic.

7.3.14 Redeem application

The final transaction in the lifecycle of commercial paper 00001 is for DigiBank to redeem it with MagnetoCorp. Balaji uses `redeem.js` to submit a transaction to perform the redeem logic within the smart contract.

Run the `redeem` transaction in Balaji's window:

```
(balaji)$ node redeem.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper redeem transaction.
Process redeem transaction response.
MagnetoCorp commercial paper : 00001 successfully redeemed with MagnetoCorp
Transaction complete.
Disconnect from Fabric gateway.
Redeem program complete.
```

Again, see how the commercial paper 00001 was successfully redeemed when `redeem.js` invoked the `redeem` transaction defined in `CommercialPaper`. Again, it updated commercial paper `00001` within the world state to reflect that the ownership returned to MagnetoCorp, the issuer of the paper.

7.3.15 Clean up

When you are finished using the Commercial Paper tutorial, you can use a script to clean up your environment. Use a command window to navigate back to the root directory of the commercial paper sample:

```
cd fabric-samples/commercial-paper
```

You can then bring down the network with the following command:

```
./network-clean.sh
```

This command will bring down the peers, CouchDB containers, and ordering node of the network, in addition to the logspout tool. It will also remove the identities that we created for Isabella and Balaji. Note that all of the data on the ledger will be lost. If you want to go through the tutorial again, you will start from a clean initial state.

7.3.16 Further reading

To understand how applications and smart contracts shown in this tutorial work in more detail, you'll find it helpful to read Developing Applications. This topic will give you a fuller explanation of the commercial paper scenario, the PaperNet business network, its actors, and how the applications and smart contracts they use work in detail.

Also feel free to use this sample to start creating your own applications and smart contracts!

7.4 Using Private Data in Fabric

This tutorial will demonstrate the use of Private Data Collections (PDC) to provide storage and retrieval of private data on the blockchain network for authorized peers of organizations. The collection is specified using a collection definition file containing the policies governing that collection.

The information in this tutorial assumes knowledge of private data stores and their use cases. For more information, check out *Private data*.

Note: These instructions use the new Fabric chaincode lifecycle introduced in the Fabric v2.0 release. If you would like to use the previous lifecycle model to use private data with chaincode, visit the v1.4 version of the Using Private Data in Fabric tutorial.

The tutorial will take you through the following steps to practice defining, configuring and using private data with Fabric:

1. *Asset transfer private data sample use case*
2. *Build a collection definition JSON file*
3. *Read and Write private data using chaincode APIs*
4. *Deploy the private data smart contract to the channel*
5. *Register identities*
6. *Create an asset in private data*
7. *Query the private data as an authorized peer*
8. *Query the private data as an unauthorized peer*
9. *Transfer the Asset*
10. *Purge Private Data*
11. *Using indexes with private data*
12. *Additional resources*

This tutorial will deploy the asset transfer private data sample to the Fabric test network to demonstrate how to create, deploy, and use a collection of private data. You should have completed the task *Install Samples, Binaries, and Docker Images*.

7.4.1 Asset transfer private data sample use case

This sample demonstrates the use of three private data collections, `assetCollection`, `Org1MSPPrivateCollection` & `Org2MSPPrivateCollection` to transfer an asset between Org1 and Org2, using following use case:

A member of Org1 creates a new asset, henceforth referred as owner. The public details of the asset, including the identity of the owner, are stored in the private data collection named `assetCollection`. The asset is also created with an appraised value supplied by the owner. The appraised value is used by each participant to agree to the transfer of the asset, and is only stored in owner organization's collection. In our case, the initial appraisal value agreed by the owner is stored in the `Org1MSPPrivateCollection`.

To purchase the asset, the buyer needs to agree to the same appraised value as the asset owner. In this step, the buyer (a member of Org2) creates an agreement to trade and agree to an appraisal value using smart contract function `'AgreeToTransfer'`. This value is stored in `Org2MSPPrivateCollection` collection. Now, the asset owner

can transfer the asset to the buyer using smart contract function `'TransferAsset'`. The `'TransferAsset'` function uses the hash on the channel ledger to confirm that the owner and the buyer have agreed to the same appraised value before transferring the asset.

Before we go through the transfer scenario, we will discuss how organizations can use private data collections in Fabric.

7.4.2 Build a collection definition JSON file

Before a set of organizations can transact using private data, all organizations on channel need to build a collection definition file that defines the private data collections associated with each chaincode. Data that is stored in a private data collection is only distributed to the peers of certain organizations instead of all members of the channel. The collection definition file describes all of the private data collections that organizations can read and write to from a chaincode.

Each collection is defined by the following properties:

- `name`: Name of the collection.
- `policy`: Defines the organization peers allowed to persist the collection data.
- `requiredPeerCount`: Number of peers required to disseminate the private data as a condition of the endorsement of the chaincode
- `maxPeerCount`: For data redundancy purposes, the number of other peers that the current endorsing peer will attempt to distribute the data to. If an endorsing peer goes down, these other peers are available at commit time if there are requests to pull the private data.
- `blockToLive`: For very sensitive information such as pricing or personal information, this value represents how long the data should live on the private database in terms of blocks. The data will live for this specified number of blocks on the private database and after that it will get purged, making this data obsolete from the network. To keep private data indefinitely, that is, to never purge private data, set the `blockToLive` property to `0`.
- `memberOnlyRead`: a value of `true` indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed read access to private data.
- `memberOnlyWrite`: a value of `true` indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed write access to private data.
- `endorsementPolicy`: defines the endorsement policy that needs to be met in order to write to the private data collection. The collection level endorsement policy overrides to chaincode level policy. For more information on building a policy definition refer to the *Endorsement policies* topic.

The same collection definition file needs to be deployed by all organizations that use the chaincode, even if the organization does not belong to any collections. In addition to the collections that are explicitly defined in a collection file, each organization has access to an implicit collection on their peers that can only be read by their organization. For an example that uses implicit data collections, see the *Secured asset transfer in Fabric*.

The asset transfer private data example contains a *collections_config.json* file that defines three private data collection definitions: `assetCollection`, `Org1MSPPrivateCollection`, and `Org2MSPPrivateCollection`.

```
// collections_config.json

[
   {
   "name": "assetCollection",
   "policy": "OR('Org1MSP.member', 'Org2MSP.member')",
   "requiredPeerCount": 1,
```

(continues on next page)

(continued from previous page)

```
    "maxPeerCount": 1,
    "blockToLive":1000000,
    "memberOnlyRead": true,
    "memberOnlyWrite": true
    },
    {
    "name": "Org1MSPPrivateCollection",
    "policy": "OR('Org1MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 1,
    "blockToLive":3,
    "memberOnlyRead": true,
    "memberOnlyWrite": false,
    "endorsementPolicy": {
        "signaturePolicy": "OR('Org1MSP.member')"
    }
    },
    {
    "name": "Org2MSPPrivateCollection",
    "policy": "OR('Org2MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 1,
    "blockToLive":3,
    "memberOnlyRead": true,
    "memberOnlyWrite": false,
    "endorsementPolicy": {
        "signaturePolicy": "OR('Org2MSP.member')"
    }
    }
]
```

The `policy` property in the `assetCollection` definition specifies that both Org1 and Org2 can store the collection on their peers. The `memberOnlyRead` and `memberOnlyWrite` parameters are used to specify that only Org1 and Org2 clients can read and write to this collection.

The `Org1MSPPrivateCollection` collection allows only peers of Org1 to have the private data in their private database, while the `Org2MSPPrivateCollection` collection can only be stored by the peers of Org2. The `endorsementPolicy` parameter is used to create a collection specific endorsement policy. Each update to `Org1MSPPrivateCollection` or `Org2MSPPrivateCollection` needs to be endorsed by the organization that stores the collection on their peers. We will see how these collections are used to transfer the asset in the course of the tutorial.

This collection definition file is deployed when the chaincode definition is committed to the channel using the peer lifecycle chaincode commit command. More details on this process are provided in Section 3 below.

7.4.3 Read and Write private data using chaincode APIs

The next step in understanding how to privatize data on a channel is to build the data definition in the chaincode. The asset transfer private data sample divides the private data into three separate data definitions according to how the data will be accessed.

```
// Peers in Org1 and Org2 will have this private data in a side database
type Asset struct {
       Type  string `json:"objectType"` //Type is used to distinguish the various␣
↪types of objects in state database
```

(continues on next page)

(continued from previous page)

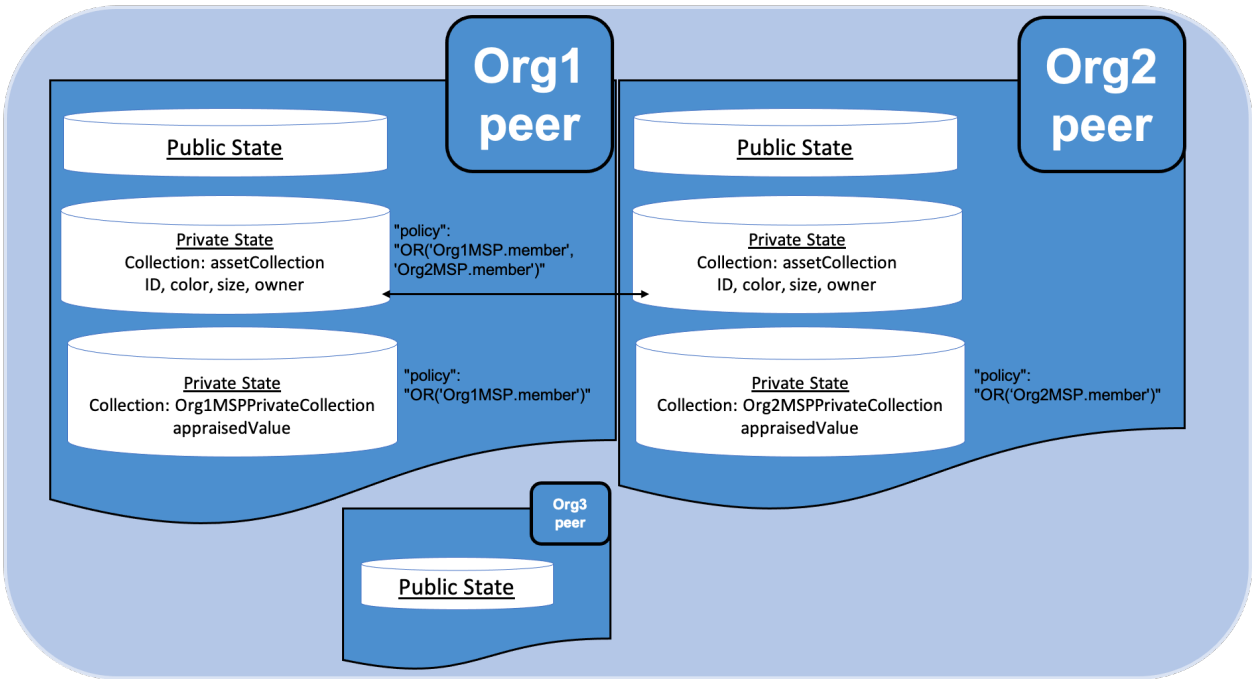
```
        ID    string `json:"assetID"`
        Color string `json:"color"`
        Size  int    `json:"size"`
        Owner string `json:"owner"`
}

// AssetPrivateDetails describes details that are private to owners

// Only peers in Org1 will have this private data in a side database
type AssetPrivateDetails struct {
        ID             string `json:"assetID"`
        AppraisedValue int    `json:"appraisedValue"`
}

// Only peers in Org2 will have this private data in a side database
type AssetPrivateDetails struct {
        ID             string `json:"assetID"`
        AppraisedValue int    `json:"appraisedValue"`
}
```

Specifically, access to the private data will be restricted as follows:

- `objectType, color, size, and owner` are stored in `assetCollection` and hence will be visible to members of the channel per the definition in the collection policy (Org1 and Org2).
- `AppraisedValue` of an asset is stored in collection `Org1MSPPrivateCollection` or `Org2MSPPrivateCollection` , depending on the owner of the asset. The value is only accessible to the users who belong to the organization that can store the collection.

All of the data that is created by the asset transfer private data sample smart contract is stored in PDC. The smart contract uses the Fabric chaincode API to read and write private data to private data collections using the `GetPrivateData()` and `PutPrivateData()` functions. You can find more information about those functions here. This private data is stored in private state db on the peer (separate from public state db), and is disseminated between authorized peers via gossip protocol.

The following diagram illustrates the private data model used by the private data sample. Note that Org3 is only shown in the diagram to illustrate that if there were any other organizations on the channel, they would not have access to *any* of the private data collections that were defined in the configuration.



Reading collection data

The smart contract uses the chaincode API `GetPrivateData()` to query private data in the database. `GetPrivateData()` takes two arguments, the **collection name** and the data key. Recall the collection `assetCollection` allows peers of Org1 and Org2 to have the private data in a side database, and the collection `Org1MSPPrivateCollection` allows only peers of Org1 to have their private data in a side database and `Org2MSPPrivateCollection` allows peers of Org2 to have their private data in a side database. For implementation details refer to the following two asset transfer private data functions:

- **ReadAsset** for querying the values of the `assetID, color, size and owner` attributes.
- **ReadAssetPrivateDetails** for querying the values of the `appraisedValue` attribute.

When we issue the database queries using the peer commands later in this tutorial, we will call these two functions.

Writing private data

The smart contract uses the chaincode API `PutPrivateData()` to store the private data into the private database. The API also requires the name of the collection. Note that the asset transfer private data sample includes three different private data collections, but it is called twice in the chaincode (in this scenario acting as Org1).

1. Write the private data `assetID, color, size and owner` using the collection named `assetCollection`.
2. Write the private data `appraisedValue` using the collection named `Org1MSPPrivateCollection`.

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

For example, in the following snippet of the `CreateAsset` function, `PutPrivateData()` is called twice, once for each set of private data.

```
// CreateAsset creates a new asset by placing the main asset details in the␣
↪assetCollection
// that can be read by both organizations. The appraisal value is stored in the␣
↪owners org specific collection.
func (s *SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface)␣
↪error {

    // Get new asset from transient map
    transientMap, err := ctx.GetStub().GetTransient()
    if err != nil {
        return fmt.Errorf("error getting transient: %v", err)
    }

    // Asset properties are private, therefore they get passed in transient field,␣
↪instead of func args
    transientAssetJSON, ok := transientMap["asset_properties"]
    if !ok {
        //log error to stdout
        return fmt.Errorf("asset not found in the transient map input")
    }

    type assetTransientInput struct {
        Type           string `json:"objectType"` //Type is used to distinguish the␣
↪various types of objects in state database
        ID             string `json:"assetID"`
        Color          string `json:"color"`
        Size           int    `json:"size"`
        AppraisedValue int    `json:"appraisedValue"`
    }

    var assetInput assetTransientInput
    err = json.Unmarshal(transientAssetJSON, &assetInput)
    if err != nil {
        return fmt.Errorf("failed to unmarshal JSON: %v", err)
    }

    if len(assetInput.Type) == 0 {
        return fmt.Errorf("objectType field must be a non-empty string")
    }
    if len(assetInput.ID) == 0 {
        return fmt.Errorf("assetID field must be a non-empty string")
    }
    if len(assetInput.Color) == 0 {
        return fmt.Errorf("color field must be a non-empty string")
    }
    if assetInput.Size <= 0 {
        return fmt.Errorf("size field must be a positive integer")
    }
    if assetInput.AppraisedValue <= 0 {
        return fmt.Errorf("appraisedValue field must be a positive integer")
    }

    // Check if asset already exists
    assetAsBytes, err := ctx.GetStub().GetPrivateData(assetCollection, assetInput.ID)
    if err != nil {
        return fmt.Errorf("failed to get asset: %v", err)
    } else if assetAsBytes != nil {
```

(continues on next page)

(continued from previous page)

```
        fmt.Println("Asset already exists: " + assetInput.ID)
        return fmt.Errorf("this asset already exists: " + assetInput.ID)
    }

    // Get ID of submitting client identity
    clientID, err := submittingClientIdentity(ctx)
    if err != nil {
        return err
    }

    // Verify that the client is submitting request to peer in their organization
    // This is to ensure that a client from another org doesn't attempt to read or
    // write private data from this peer.
    err = verifyClientOrgMatchesPeerOrg(ctx)
    if err != nil {
        return fmt.Errorf("CreateAsset cannot be performed: Error %v", err)
    }

    // Make submitting client the owner
    asset := Asset{
        Type:  assetInput.Type,
        ID:    assetInput.ID,
        Color: assetInput.Color,
        Size:  assetInput.Size,
        Owner: clientID,
    }
    assetJSONasBytes, err := json.Marshal(asset)
    if err != nil {
        return fmt.Errorf("failed to marshal asset into JSON: %v", err)
    }

    // Save asset to private data collection
    // Typical logger, logs to stdout/file in the fabric managed docker container,␣
↪running this chaincode
    // Look for container name like dev-peer0.org1.example.com-{chaincodename_version}
↪-xyz
    log.Printf("CreateAsset Put: collection %v, ID %v, owner %v", assetCollection,␣
↪assetInput.ID, clientID)

    err = ctx.GetStub().PutPrivateData(assetCollection, assetInput.ID,␣
↪assetJSONasBytes)
    if err != nil {
        return fmt.Errorf("failed to put asset into private data collecton: %v", err)
    }

    // Save asset details to collection visible to owning organization
    assetPrivateDetails := AssetPrivateDetails{
        ID:             assetInput.ID,
        AppraisedValue: assetInput.AppraisedValue,
    }

    assetPrivateDetailsAsBytes, err := json.Marshal(assetPrivateDetails) // marshal␣
↪asset details to JSON
    if err != nil {
        return fmt.Errorf("failed to marshal into JSON: %v", err)
    }
```

(continues on next page)

(continued from previous page)

```
    // Get collection name for this organization.
    orgCollection, err := getCollectionName(ctx)
    if err != nil {
        return fmt.Errorf("failed to infer private collection name for the org: %v",␣
↪err)
    }

    // Put asset appraised value into owners org specific private data collection
    log.Printf("Put: collection %v, ID %v", orgCollection, assetInput.ID)
    err = ctx.GetStub().PutPrivateData(orgCollection, assetInput.ID,␣
↪assetPrivateDetailsAsBytes)
    if err != nil {
        return fmt.Errorf("failed to put asset private details: %v", err)
    }
    return nil
}
```

To summarize, the policy definition above for our `collections_config.json` allows all peers in Org1 and Org2 to store and transact with the asset transfer private data `assetID, color, size, owner` in their private database. But only peers in Org1 can store and transact with the `appraisedValue` key data in the Org1 collection `Org1MSPPrivateCollection` and only peers in Org2 can store and transact with the `appraisedValue` key data in the Org2 collection `Org2MSPPrivateCollection`.

As an additional data privacy benefit, since a collection is being used, only the private data *hashes* go through orderer, not the private data itself, keeping private data confidential from orderer.

7.4.4 Start the network

Now we are ready to step through some commands which demonstrate how to use private data.

Try it yourself

Before installing, defining, and using the private data smart contract, we need to start the Fabric test network. For the sake of this tutorial, we want to operate from a known initial state. The following command will kill any active or stale Docker containers and remove previously generated artifacts. Therefore let's run the following command to clean up any previous environments:

```
cd fabric-samples/test-network
./network.sh down
```

From the `test-network` directory, you can use the following command to start up the Fabric test network with Certificate Authorities and CouchDB:

```
./network.sh up createChannel -ca -s couchdb
```

This command will deploy a Fabric network consisting of a single channel named `mychannel` with two organizations (each maintaining one peer node), certificate authorities, and an ordering service while using CouchDB as the state database. Either LevelDB or CouchDB may be used with collections. CouchDB was chosen to demonstrate how to use indexes with private data.

Note: For collections to work, it is important to have cross organizational gossip configured correctly. Refer to our documentation on *Gossip data dissemination protocol*, paying particular attention to the section on "anchor peers". Our tutorial does not focus on gossip given it is already configured in the test network, but when configuring a channel, the gossip anchors peers are critical to configure for collections to work properly.

7.4.5 Deploy the private data smart contract to the channel

We can now use the test network script to deploy the smart contract to the channel. Run the following command from the test network directory.

```
./network.sh deployCC -ccn private -ccep "OR('Org1MSP.peer','Org2MSP.peer')" -cccg ../
↪asset-transfer-private-data/chaincode-go/collections_config.json
```

Note that we need to pass the path to the private data collection definition file to the command. As part of deploying the chaincode to the channel, both organizations on the channel must pass identical private data collection definitions as part of the *Fabric chaincode lifecycle*. We are also deploying the smart contract with a chaincode level endorsement policy of `"OR('Org1MSP.peer','Org2MSP.peer')"`. This allows Org1 and Org2 to create an asset without receiving an endorsement from the other organization. You can see the steps required to deploy the chaincode printed in your logs after you issue the command above.

When both organizations approve the chaincode defition using the peer lifecycle chaincode approveformyorg command, the chaincode definition includes the path to the private data collection definition using the `--collections-config` flag. You can see the following *approveformyorg* command printed in your terminal:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name private␣
↪--version 1.0 --collections-config ../asset-transfer-private-data/chaincode-go/
↪collections_config.json --signature-policy "OR('Org1MSP.member','Org2MSP.member')" -
↪-package-id $CC_PACKAGE_ID --sequence 1 --tls --cafile $ORDERER_CA
```

After channel members agree to the private data collection as part of the chaincode definition, the data collection is committed to the channel using the peer lifecycle chaincode commit command. If you look for the commit command in your logs, you can see that it uses the same `--collections-config` flag to provide the path to the collection definition.

```
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride␣
↪orderer.example.com --channelID mychannel --name private --version 1.0 --sequence 1␣
↪--collections-config ../asset-transfer-private-data/chaincode-go/collections_config.
↪json --signature-policy "OR('Org1MSP.member','Org2MSP.member')" --tls --cafile
↪$ORDERER_CA --peerAddresses localhost:7051 --tlsRootCertFiles $ORG1_CA --
↪peerAddresses localhost:9051 --tlsRootCertFiles $ORG2_CA
```

7.4.6 Register identities

The private data transfer smart contract supports ownership by individual identities that belong to the network. In our scenario, the owner of the asset will be a member of Org1, while the buyer will belong to Org2. To highlight the connection between the `GetClientIdentity().GetID()` API and the information within a user's certificate, we will register two new identities using the Org1 and Org2 Certificate Authorities (CA's), and then use the CA's to generate each identity's certificate and private key.

First, we need to set the following environment variables to use the Fabric CA client:

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

We will use the Org1 CA to create the identity asset owner. Set the Fabric CA client home to the MSP of the Org1 CA admin (this identity was generated by the test network script):

```
export FABRIC_CA_CLIENT_HOME=${PWD}/organizations/peerOrganizations/org1.example.com/
```

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

```
fabric-ca-client register --caname ca-org1 --id.name owner --id.secret ownerpw --id.
↪type client --tls.certfiles ${PWD}/organizations/fabric-ca/org1/tls-cert.pem
```

You can now generate the identity certificates and MSP folder by providing the enroll name and secret to the enroll command:

```
fabric-ca-client enroll -u https://owner:ownerpw@localhost:7054 --caname ca-org1 -M $
↪{PWD}/organizations/peerOrganizations/org1.example.com/users/owner@org1.example.com/
↪msp --tls.certfiles ${PWD}/organizations/fabric-ca/org1/tls-cert.pem
```

Run the command below to copy the Node OU configuration file into the owner identity MSP folder.

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

We can now use the Org2 CA to create the buyer identity. Set the Fabric CA client home the Org2 CA admin:

```
export FABRIC_CA_CLIENT_HOME=${PWD}/organizations/peerOrganizations/org2.example.com/
```

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

```
fabric-ca-client register --caname ca-org2 --id.name buyer --id.secret buyerpw --id.
↪type client --tls.certfiles ${PWD}/organizations/fabric-ca/org2/tls-cert.pem
```

We can now enroll to generate the identity MSP folder:

```
fabric-ca-client enroll -u https://buyer:buyerpw@localhost:8054 --caname ca-org2 -M $
↪{PWD}/organizations/peerOrganizations/org2.example.com/users/buyer@org2.example.com/
↪msp --tls.certfiles ${PWD}/organizations/fabric-ca/org2/tls-cert.pem
```

Run the command below to copy the Node OU configuration file into the buyer identity MSP folder.

```
cp ${PWD}/organizations/peerOrganizations/org2.example.com/msp/config.yaml ${PWD}/
↪organizations/peerOrganizations/org2.example.com/users/buyer@org2.example.com/msp/
↪config.yaml
```

7.4.7 Create an asset in private data

Now that we have created the identity of the asset owner, we can invoke the private data smart contract to create a new asset. Copy and paste the following set of commands into your terminal in the *test-network* directory:

Try it yourself

```
export PATH=${PWD}/../bin:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/owner@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

We will use the `CreateAsset` function to create an asset that is stored in private data — assetID `asset1` with a color `green`, size `20` and appraisedValue of `100`. Recall that private data **appraisedValue** will be stored

separately from the private data **assetID, color, size**. For this reason, the `CreateAsset` function calls the `PutPrivateData()` API twice to persist the private data, once for each collection. Also note that the private data is passed using the `--transient` flag. Inputs passed as transient data will not be persisted in the transaction in order to keep the data private. Transient data is passed as binary data and therefore when using terminal it must be base64 encoded. We use an environment variable to capture the base64 encoded value, and use `tr` command to strip off the problematic newline characters that linux base64 command adds.

Run the following command to create the asset:

```
export ASSET_PROPERTIES=$(echo -n "{\"objectType\":\"asset\",\"assetID\":\"asset1\",\
↪"color\":\"green\",\"size\":20,\"appraisedValue\":100}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"CreateAsset","Args":[]}' --transient "{\"asset_properties\
↪":\"$ASSET_PROPERTIES\"}"
```

You should see results similar to:

```
[chaincodeCmd] chaincodeInvokeOrQuery->INFO 001 Chaincode invoke successful. result:␣
↪status:200
```

Note that command above only targets the Org1 peer. The `CreateAsset` transaction writes to two collections, `assetCollection` and `Org1MSPPrivateCollection`. The `Org1MSPPrivateCollection` requires an endorsement from the Org1 peer in order to write to the collection, while the `assetCollection` inherits the endorsement policy of the chaincode, `"OR('Org1MSP.peer','Org2MSP.peer')"`. An endorsement from the Org1 peer can meet both endorsement policies and is able to create an asset without an endorsement from Org2.

7.4.8 Query the private data as an authorized peer

Our collection definition allows all peers of Org1 and Org2 to have the `assetID, color, size, and owner` private data in their side database, but only peers in Org1 can have Org1's opinion of their `appraisedValue` private data in their side database. As an authorized peer in Org1, we will query both sets of private data.

The first `query` command calls the `ReadAsset` function which passes `assetCollection` as an argument.

```
// ReadAsset reads the information from collection
func (s *SmartContract) ReadAsset(ctx contractapi.TransactionContextInterface,␣
↪assetID string) (*Asset, error) {

    log.Printf("ReadAsset: collection %v, ID %v", assetCollection, assetID)
    assetJSON, err := ctx.GetStub().GetPrivateData(assetCollection, assetID) //get␣
↪the asset from chaincode state
    if err != nil {
        return nil, fmt.Errorf("failed to read asset: %v", err)
    }

    //No Asset found, return empty response
    if assetJSON == nil {
        log.Printf("%v does not exist in collection %v", assetID, assetCollection)
        return nil, nil
    }

    var asset *Asset
    err = json.Unmarshal(assetJSON, &asset)
    if err != nil {
```

(continues on next page)

(continued from previous page)

```
        return nil, fmt.Errorf("failed to unmarshal JSON: %v", err)
    }

    return asset, nil

}
```

The second `query` command calls the `ReadAssetPrivateDetails` function which passes `Org1MSPPrivateDetails` as an argument.

```
// ReadAssetPrivateDetails reads the asset private details in organization specific␣
↪collection
func (s *SmartContract) ReadAssetPrivateDetails(ctx contractapi.
↪TransactionContextInterface, collection string, assetID string)␣
↪(*AssetPrivateDetails, error) {
    log.Printf("ReadAssetPrivateDetails: collection %v, ID %v", collection, assetID)
    assetDetailsJSON, err := ctx.GetStub().GetPrivateData(collection, assetID) //␣
↪Get the asset from chaincode state
    if err != nil {
        return nil, fmt.Errorf("failed to read asset details: %v", err)
    }
    if assetDetailsJSON == nil {
        log.Printf("AssetPrivateDetails for %v does not exist in collection %v",␣
↪assetID, collection)
        return nil, nil
    }

    var assetDetails *AssetPrivateDetails
    err = json.Unmarshal(assetDetailsJSON, &assetDetails)
    if err != nil {
        return nil, fmt.Errorf("failed to unmarshal JSON: %v", err)
    }

    return assetDetails, nil
}
```

Now *Try it yourself*

We can read the main details of the asset that was created by using the *ReadAsset* function to query the *assetCollection* collection as Org1:

```
peer chaincode query -C mychannel -n private -c '{"function":"ReadAsset","Args":[
↪"asset1"]}'
```

When successful, the command will return the following result:

```
{"objectType":"asset","assetID":"asset1","color":"green","size":20,"owner":
↪"x509::CN=appUser1,OU=admin,O=Hyperledger,ST=North Carolina,C=US::CN=ca.org1.
↪example.com,O=org1.example.com,L=Durham,ST=North Carolina,C=US"}
```

The *"owner"* of the asset is the identity that created the asset by invoking the smart contract. The private data smart contract uses the `GetClientIdentity().GetID()` API to read the name and issuer of the identity certificate. You can see the name and issuer of the identity certificate, in the owner attribute.

Query for the `appraisedValue` private data of `asset1` as a member of Org1.

```
peer chaincode query -C mychannel -n private -c '{"function":"ReadAssetPrivateDetails
↪","Args":["Org1MSPPrivateCollection","asset1"]}'
```

You should see the following result:

```
{"assetID":"asset1","appraisedValue":100}
```

7.4.9 Query the private data as an unauthorized peer

Now we will operate a user from Org2. Org2 has the asset transfer private data `assetID, color, size, owner` in its side database as defined in the `assetCollection` policy, but does not store the asset `appraisedValue` data for Org1. We will query for both sets of private data.

Switch to a peer in Org2

Run the following commands to operate as an Org2 member and query the Org2 peer.

Try it yourself

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/buyer@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

Query private data Org2 is authorized to

Peers in Org2 should have the first set of asset transfer private data (`assetID, color, size and owner`) in their side database and can access it using the `ReadAsset()` function which is called with the `assetCollection` argument.

Try it yourself

```
peer chaincode query -C mychannel -n private -c '{"function":"ReadAsset","Args":[
↪"asset1"]}'
```

When successful, should see something similar to the following result:

```
{"objectType":"asset","assetID":"asset1","color":"green","size":20,
"owner":"x509::CN=appUser1,OU=admin,O=Hyperledger,ST=North Carolina,C=US::CN=ca.org1.
↪example.com,O=org1.example.com,L=Durham,ST=North Carolina,C=US" }
```

Query private data Org2 is not authorized to

Because the asset was created by Org1, the `appraisedValue` associated with `asset1` is stored in the `Org1MSPPrivateCollection` collection. The value is not stored by peers in Org2. Run the following command to demonstrate that the asset's `appraisedValue` is not stored in the `Org2MSPPrivateCollection` on the Org2 peer:

Try it yourself

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"ReadAssetPrivateDetails","Args":["Org2MSPPrivateCollection
↪","asset1"]}'
```

The empty response shows that the asset1 private details do not exist in buyer (Org2) private collection.

Nor can a user from Org2 read the Org1 private data collection:

```
peer chaincode query -C mychannel -n private -c '{"function":"ReadAssetPrivateDetails
↪","Args":["Org1MSPPrivateCollection","asset1"]}'
```

By setting `"memberOnlyRead": true` in the collection configuration file, we specify that only clients from Org1 can read data from the collection. An Org2 client who tries to read the collection would only get the following response:

```
Error: endorsement failure during query. response: status:500 message:"failed to
read asset details: GET_STATE failed: transaction ID:␣
↪d23e4bc0538c3abfb7a6bd4323fd5f52306e2723be56460fc6da0e5acaee6b23: tx
creator does not have read access permission on privatedata in chaincodeName:private␣
↪collectionName: Org1MSPPrivateCollection"
```

Users from Org2 will only be able to see the public hash of the private data.

7.4.10 Transfer the Asset

Let's see what it takes to transfer `asset1` to Org2. In this case, Org2 needs to agree to buy the asset from Org1, and they need to agree on the `appraisedValue`. You may be wondering how they can agree if Org1 keeps their opinion of the `appraisedValue` in their private side database. For the answer to this, lets continue.

Try it yourself

Switch back to the terminal with our peer CLI.

To transfer an asset, the buyer (recipient) needs to agree to the same `appraisedValue` as the asset owner, by calling chaincode function `AgreeToTransfer`. The agreed value will be stored in the `Org2MSPDetailsCollection` collection on the Org2 peer. Run the following commands to agree to the appraised value of 100 as Org2:

```
export ASSET_VALUE=$(echo -n "{\"assetID\":\"asset1\",\"appraisedValue\":100}" |␣
↪base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"AgreeToTransfer","Args":[]}' --transient "{\"asset_value\
↪":\"$ASSET_VALUE\"}"
```

The buyer can now query the value they agreed to in the Org2 private data collection:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"ReadAssetPrivateDetails","Args":["Org2MSPPrivateCollection
↪","asset1"]}'
```

The invoke will return the following value:

```
{"assetID":"asset1","appraisedValue":100}
```

Now that buyer has agreed to buy the asset for the appraised value, the owner can transfer the asset to Org2. The asset needs to be transferred by the identity that owns the asset, so lets go acting as Org1:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/owner@org1.example.com/msp
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_ADDRESS=localhost:7051
```

The owner from Org1 can read the data added by the *AgreeToTransfer* transaction to view the buyer identity:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"ReadTransferAgreement","Args":["asset1"]}'
```

```
{"assetID":"asset1","buyerID":
↪"eDUwOTo6Q049YnV5ZXIsT1U9Y2xpZW50LE89SHlwZXJsZWRnZXIsU1Q9Tm9ydGggQ2Fyb2xpbmEsQz1VUzo6Q049Y2Eub3JnM
↪"}
```

We now have all we need to transfer the asset. The smart contract uses the `GetPrivateDataHash()` function to check that the hash of the asset appraisal value in `Org1MSPPrivateCollection` matches the hash of the appraisal value in the `Org2MSPPrivateCollection`. If the hashes are the same, it confirms that the owner and the interested buyer have agreed to the same asset value. If the conditions are met, the transfer function will get the client ID of the buyer from the transfer agreement and make the buyer the new owner of the asset. The transfer function will also delete the asset appraisal value from the collection of the former owner, as well as remove the transfer agreement from the `assetCollection`.

Run the following commands to transfer the asset. The owner needs to provide the assetID and the organization MSP ID of the buyer to the transfer transaction:

```
export ASSET_OWNER=$(echo -n "{\"assetID\":\"asset1\",\"buyerMSP\":\"Org2MSP\"}" |␣
↪base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"TransferAsset","Args":[]}' --transient "{\"asset_owner\":\"
↪$ASSET_OWNER\"}" --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/
↪organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/
↪ca.crt
```

You can query `asset1` to see the results of the transfer:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"ReadAsset","Args":["asset1"]}'
```

The results will show that the buyer identity now owns the asset:

```
{"objectType":"asset","assetID":"asset1","color":"green","size":20,"owner":
↪"x509::CN=appUser2, OU=client + OU=org2 + OU=department1::CN=ca.org2.example.com,␣
↪O=org2.example.com, L=Hursley, ST=Hampshire, C=UK"}
```

The *"owner"* of the asset now has the buyer identity.

You can also confirm that transfer removed the private details from the Org1 collection:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"ReadAssetPrivateDetails","Args":["Org1MSPPrivateCollection
↪","asset1"]}'
```

Your query will return empty result, since the asset private data is removed from the Org1 private data collection.

7.4.11 Purge Private Data

For use cases where private data only needs to be persisted for a short period of time, it is possible to "purge" the data after a certain set number of blocks, leaving behind only a hash of the data that serves as immutable evidence of the transaction. An organization could decide to purge private data if the data contained sensitive information that was used by another transaction, but is not longer needed, or if the data is being replicated into an off-chain database.

The `appraisedValue` data in our example contains a private agreement that the organization may want to expire after a certain period of time. Thus, it has a limited lifespan, and can be purged after existing unchanged on the blockchain for a designated number of blocks using the `blockToLive` property in the collection definition.

The `Org2MSPPrivateCollection` definition has a `blockToLive` property value of `3`, meaning this data will live on the side database for three blocks and then after that it will get purged. If we create additional blocks on the channel, the `appraisedValue` agreed to by Org2 will eventually get purged. We can create 3 new blocks to demonstrate:

Try it yourself

Run the following commands in your terminal to switch back to operating as member of Org2 and target the Org2 peer:

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/buyer@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

We can still query the `appraisedValue` in the `Org2MSPPrivateCollection`:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"ReadAssetPrivateDetails","Args":["Org2MSPPrivateCollection
↪","asset1"]}'
```

You should see the value printed in your logs:

```
{"assetID":"asset1","appraisedValue":100}
```

Since we need to keep track of how many blocks we are adding before the private data gets purged, open a new terminal window and run the following command to view the private data logs for the Org2 peer. Note the highest block number.

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

Now return to the terminal where we are acting as a member of Org2 and run the following commands to create three new assets. Each command will create a new block.

```
export ASSET_PROPERTIES=$(echo -n "{\"objectType\":\"asset\",\"assetID\":\"asset2\",\
↪"color\":\"blue\",\"size\":30,\"appraisedValue\":100}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"CreateAsset","Args":[]}' --transient "{\"asset_properties\
↪":\"$ASSET_PROPERTIES\"}"
```

```
export ASSET_PROPERTIES=$(echo -n "{\"objectType\":\"asset\",\"assetID\":\"asset3\",\
↪"color\":\"red\",\"size\":25,\"appraisedValue\":100}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"CreateAsset","Args":[]}' --transient "{\"asset_properties\
↪":\"$ASSET_PROPERTIES\"}"
```

```
export ASSET_PROPERTIES=$(echo -n "{\"objectType\":\"asset\",\"assetID\":\"asset4\",\
↪"color\":\"orange\",\"size\":15,\"appraisedValue\":100}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"CreateAsset","Args":[]}' --transient "{\"asset_properties\
↪":\"$ASSET_PROPERTIES\"}"
```

Return to the other terminal and run the following command to confirm that the new assets resulted in the creation of three new blocks:

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

The `appraisedValue` has now been purged from the `Org2MSPDetailsCollection` private data collection. Issue the query again from the Org2 terminal to see that the response is empty.

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪private -c '{"function":"ReadAssetPrivateDetails","Args":["Org2MSPPrivateCollection
↪","asset1"]}'
```

7.4.12 Using indexes with private data

Indexes can also be applied to private data collections, by packaging indexes in the `META-INF/statedb/couchdb/collections/<collection_name>/indexes` directory alongside the chaincode. An example index is available here .

For deployment of chaincode to production environments, it is recommended to define any indexes alongside chaincode so that the chaincode and supporting indexes are deployed automatically as a unit, once the chaincode has been installed on a peer and instantiated on a channel. The associated indexes are automatically deployed upon chaincode instantiation on the channel when the `--collections-config` flag is specified pointing to the location of the collection JSON file.

7.4.13 Clean up

When you are finished using the private data smart contract, you can bring down the test network using `network.sh` script.

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

This command will bring down the CAs, peers, and ordering node of the network that we created. Note that all of the data on the ledger will be lost. If you want to go through the tutorial again, you will start from a clean initial state.

7.4.14 Additional resources

For additional private data education, a video tutorial has been created.

Note: The video uses the previous lifecycle model to install private data collections with chaincode.

7.5 Secured asset transfer in Fabric

This tutorial will demonstrate how an asset can be represented and traded between organizations in a Hyperledger Fabric blockchain channel, while keeping details of the asset and transaction private using private data. Each on-chain asset is a non-fungible token (NFT) that represents a specific asset having certain immutable metadata properties (such as size and color) with a unique owner. When the owner wants to sell the asset, both parties need to agree to the same price before the asset is transferred. The private asset transfer smart contract enforces that only the owner of the asset can transfer the asset. In the course of this tutorial, you will learn how Fabric features such as state based endorsement, private data, and access control come together to provide secured transactions that are both private and verifiable.

This tutorial will deploy the secured asset transfer sample to demonstrate how to transfer a private asset between two organizations without publicly sharing data. You should have completed the task Install Samples, Binaries, and Docker Images.

7.5.1 Scenario requirements

The private asset transfer scenario is bound by the following requirements:

- An asset may be issued by the first owner's organization (in the real world issuance may be restricted to some authority that certifies an asset's properties).
- Ownership is managed at the organization level (the Fabric permissioning scheme would equally support ownership at an individual identity level within an organization).
- The asset identifier and owner is stored as public channel data for all channel members to see.
- The asset metadata properties however are private information known only to the asset owner (and prior owners).
- An interested buyer will want to verify an asset's private properties.
- An interested buyer will want to verify an asset's provenance, specifically the asset's origin and chain of custody. They will also want to verify that the asset has not changed since issuance, and that all prior transfers have been legitimate.
- To transfer an asset, a buyer and seller must first agree on the sales price.
- Only the current owner may transfer their asset to another organization.

- The actual private asset transfer must verify that the legitimate asset is being transferred, and verify that the price has been agreed to. Both buyer and seller must endorse the transfer.

7.5.2 How privacy is maintained

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

- The asset metadata properties are stored in the current owning organization's implicit private data collection on the organization's peers only. Each organization on a Fabric channel has a private data collection that their own organization can use. This collection is *implicit* because it does not need to be explicitly defined in the chaincode.
- Although a hash of the private properties is automatically stored on-chain for all channel members to see, a random salt is included in the private properties so that other channel members cannot guess the private data pre-image through a dictionary attack.
- Smart contract requests utilize the transient field for private data so that private data does not get included in the final on-chain transaction.
- Private data queries must originate from a client whose org id matches the peer's org id, which must be the same as the asset owner's org id.

7.5.3 How the transfer is implemented

Before we start using the private asset transfer smart contract we will provide an overview of the transaction flow and how Fabric features are used to protect the asset created on the blockchain:

Creating the asset

The private asset transfer smart contract is deployed with an endorsement policy that requires an endorsement from any channel member. This allows any organization to create an asset that they own without requiring an endorsement from other channel members. The creation of the asset is the only transaction that uses the chaincode level endorsement policy. Transactions that update or transfer existing assets will be governed by state based endorsement policies or the endorsement policies of private data collections. Note that in other scenarios, you may want an issuing authority to also endorse create transactions.

The smart contract uses the following Fabric features to ensure that the asset can only be updated or transferred by the organization that owns the asset:

- When the asset is created, the smart contract gets the MSP ID of the organization that submitted the request, and stores the MSP ID as the owner in the asset key/value in the public chaincode world state. Subsequent smart contract requests to update or transfer the asset will use access control logic to verify that the requesting client is from the same organization. Note that in other scenarios, the ownership could be based on a specific client identity within an organization, rather than an organization itself.
- Also when the asset is created, the smart contract sets a state based endorsement policy for the asset key. The state based policy specifies that a peer from the organization that owns the asset must endorse a subsequent request to update or transfer the asset. This prevents any other organization from updating or transferring the asset using a smart contract that has been maliciously altered on their own peers.

Agreeing to the transfer

After a asset is created, channel members can use the smart contract to agree to transfer the asset:

- The owner of the asset can change the description in the public ownership record, for example to advertise that the asset is for sale. Smart contract access control enforces that this change needs to be submitted from a member of the asset owner organization. The state based endorsement policy enforces that this description change must be endorsed by a peer from the owner's organization.

The asset owner and the asset buyer agree to transfer the asset for a certain price:

- The price agreed to by the buyer and the seller is stored in each organization's implicit private data collection. The private data collection keeps the agreed price secret from other members of the channel. The endorsement policy of the private data collection ensures that the respective organization's peer endorsed the price agreement, and the smart contract access control logic ensures that the price agreement was submitted by a client of the associated organization.
- A hash of each price agreement is stored on the ledger. The two hashes will match only if the two organizations have agreed to the same price. This allows the organizations to verify that they have come to agreement on the transfer details before the transfer takes place. A random trade id is added to the price agreement, which serves as a *salt* to ensure that other channel members can not use the hash on the ledger to guess the price.

Transferring the asset

After the two organizations have agreed to the same price, the asset owner can use the transfer function to transfer the asset to the buyer:

- Smart contract access control ensures that the transfer must be initiated by a member of the organization that owns the asset.
- The transfer function verifies that the asset's private immutable properties passed to the transfer function matches the on chain hash of the asset data in private collection, to ensure that the asset owner is *selling* the same asset that they own.
- The transfer function uses the hash of the price agreement on the ledger to ensure that both organizations have agreed to the same price.
- If the transfer conditions are met, the transfer function adds the asset to the implicit private data collection of the buyer, and deletes the asset from the collection of the seller. The transfer also updates the owner in the public ownership record.
- Because of the endorsement policies of the seller and buyer implicit data collections, and the state based endorsement policy of the public record (requiring the seller to endorse), the transfer needs to be endorsed by peers from both buyer and seller.
- The state based endorsement policy of the public asset record is updated so that only a peer of the new owner of the asset can update or sell their new asset.
- The price agreements are also deleted from both the seller and buyer implicit private data collection, and a sales receipt is created in each private data collection.

7.5.4 Running the private asset transfer smart contract

You can use the Fabric test network to run the private asset transfer smart contract. The test network contains two peer organizations, Org1 and Org1, that operate one peer each. In this tutorial, we will deploy the smart contract to a channel of the test network joined by both organizations. We will first create an asset that is owned by Org1. After the two organizations agree on the price, we will transfer the asset from Org1 to Org2.

7.5.5 Deploy the test network

We are going to use the Fabric test network to run the secured asset transfer smart contract. Open a command terminal and navigate to test network directory in your local clone of fabric-samples. We will operate from the `test-network` directory for the remainder of the tutorial.

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

First, bring down any running instances of the test network:

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

You can then deploy a new instance the network with the following command:

```
./network.sh up createChannel -c mychannel
```

The script will deploy the nodes of the network and create a single channel named `mychannel` with Org1 and Org2 as channel members. We will use this channel to deploy the smart contract and trade our asset.

7.5.6 Deploy the smart contract

You can use the test network script to deploy the secured asset transfer smart contract to the channel. Run the following command to deploy the smart contract to `mychannel`:

```
./network.sh deployCC -ccn secured -ccep "OR('Org1MSP.peer','Org2MSP.peer')"
```

Note that we are using the `-ccep` flag to deploy the smart contract with an endorsement policy of `"OR('Org1MSP.peer','Org2MSP.peer')"`. This allows either organization to create an asset without receiving an endorsement from the other organization.

Set the environment variables to operate as Org1

In the course of running this sample, you need to interact with the network as both Org1 and Org2. To make the tutorial easier to use, we will use separate terminals for each organization. Open a new terminal and make sure that you are operating from the `test-network` directory. Set the following environment variables to operate the `peer` CLI as the Org1 admin:

```
export PATH=${PWD}/../bin:${PWD}:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_ADDRESS=localhost:7051
```

The environment variables also specify the endpoint information of the Org1 peer to submit requests.

Set the environment variables to operate as Org2

Now that we have one terminal that we can operate as Org1, open a new terminal for Org2. Make sure that this terminal is also operating from the `test-network` directory. Set the following environment variables to operate as the Org2 admin:

```
export PATH=${PWD}/../bin:${PWD}:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_ADDRESS=localhost:9051
```

You will need switch between the two terminals as you go through the tutorial.

7.5.7 Create an asset

Any channel member can use the smart contract to create an asset that is owned by their organization. The details of the asset will be stored in a private data collection, and can only accessed by the organization that owns the asset. A public record of the asset, its owner, and a public description is stored on the channel ledger. Any channel member can access the public ownership record to see who owns the asset, and can read the description to see if the asset is for sale.

Operate from the Org1 terminal

Before we create the asset, we need to specify the details of what our asset will be. Issue the following command to create a JSON that will describe the asset. The `"salt"` parameter is a random string that would prevent another member of the channel from guessing the asset using the hash on the ledger. If there was no salt, a user could theoretically guess asset parameters until the hash of the of the guess and the hash on the ledger matched (this is known as a dictionary attack). This string is encoded in Base64 format so that it can be passed to the creation transaction as transient data.

```
export ASSET_PROPERTIES=$(echo -n "{\"object_type\":\"asset_properties\",\"asset_id\
↪":\"asset1\",\"color\":\"blue\",\"size\":35,\"salt\":\
↪"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3\"}" | base64 | tr -d \\n)
```

We can now use the following command to create a asset that belongs to Org1:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"CreateAsset","Args":["asset1", "A new asset for Org1MSP"]}
↪' --transient "{\"asset_properties\":\"$ASSET_PROPERTIES\"}"
```

We can can query the Org1 implicit data collection to see the asset that was created:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"GetAssetPrivateProperties","Args":["asset1"]}'
```

When successful, the command will return the following result:

```
{"object_type":"asset_properties","asset_id":"asset1","color":"blue","size":35,"salt":
↪"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3"}
```

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

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"ReadAsset","Args":["asset1"]}'
```

The command will return the record that the asset1 is owned by Org1:

```
{"object_type":"asset","asset_id":"asset1","owner_org":"Org1MSP","public_description":
↪"A new asset for Org1MSP"}
```

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

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"ChangePublicDescription","Args":["asset1","This asset is␣
↪for sale"]}'
```

Query the ledger again to see the updated description:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"ReadAsset","Args":["asset1"]}'
```

We can now see that the asset is for sale:

```
{"object_type":"asset","asset_id":"asset1","owner_org":"Org1MSP","public_description":
↪"This asset is for sale"}
```

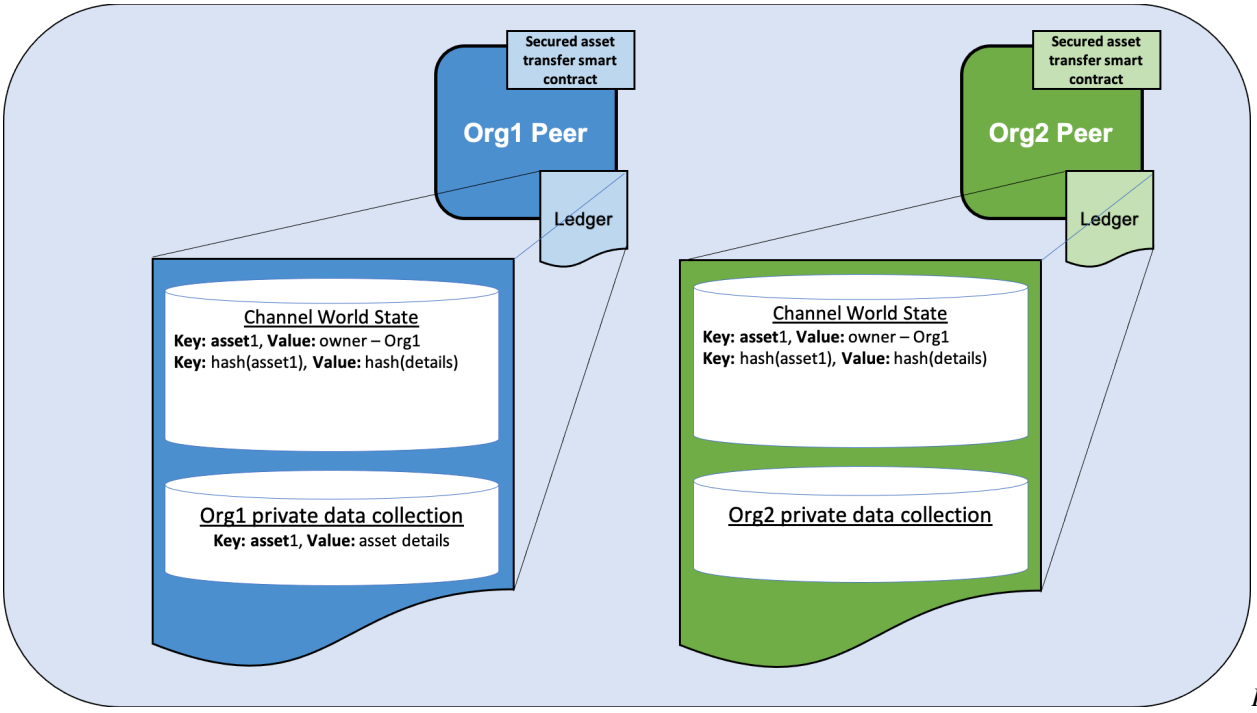


Figure 1: When Org1 creates an asset that they own, the asset details are stored in the Org1 implicit data collection on the Org1 peer. The public ownership record is stored in the channel world state, and is stored on both the Org1 and Org2

peers. A hash of the asset key and a hash the asset details are also visible in the channel world state and are stored on the peers of both organizations.

Operate from the Org2 terminal

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

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"ReadAsset","Args":["asset1"]}'
```

From this query, Org2 learns that asset1 is for sale:

```
{"object_type":"asset","asset_id":"asset1","owner_org":"Org1MSP","public_description":
↪"This asset is for sale"}
```

In a real chaincode you may want to query for all assets for sale, by using a JSON query, or by creating a different sale key and using a key range query to find the assets currently for sale. Any changes to the public description of the asset owned by Org1 needs to be endorsed by Org1. The endorsement policy is reinforced by an access control policy within the chaincode that any update needs to be submitted by the organization that owns the asset. Lets see what happens if Org2 tried to change the public description as a prank:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"ChangePublicDescription","Args":["asset1","the worst asset
↪"]}'
```

The smart contract does not allow Org2 to access the public description of the asset.

```
Error: endorsement failure during invoke. response: status:500 message:"a client from␣
↪Org2MSP cannot update the description of a asset owned by Org1MSP"
```

7.5.8 Agree to sell the asset

To sell an asset, both the buyer and the seller must agree on an asset price. Each party stores the price that they agree to in their own private data collection. The private asset transfer smart contract enforces that both parties need to agree to the same price before the asset can be transferred.

7.5.9 Agree to sell as Org1

Operate from the Org1 terminal. Org1 will agree to set the asset price as 110 dollars. The `trade_id` is used as salt to prevent a channel member that is not a buyer or a seller from guessing the price. This value needs to be passed out of band, through email or other communication, between the buyer and the seller. The buyer and the seller can also add salt to the asset key to prevent other members of the channel from guessing which asset is for sale.

```
export ASSET_PRICE=$(echo -n "{\"asset_id\":\"asset1\",\"trade_id\":\
↪"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\",\"price\":110}" | base64)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"AgreeToSell","Args":["asset1"]}' --transient "{\"asset_
↪price\":\"$ASSET_PRICE\"}"
```

We can query the Org1 private data collection to read the agreed to selling price:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"GetAssetSalesPrice","Args":["asset1"]}'
```

7.5.10 Agree to buy as Org2

Operate from the Org2 terminal. Run the following command to verify the asset properties before agreeing to buy. The asset properties and salt would be passed out of band, through email or other communication, between the buyer and seller.

```
export ASSET_PROPERTIES=$(echo -n "{\"object_type\":\"asset_properties\",\"asset_id\
↪":\"asset1\",\"color\":\"blue\",\"size\":35,\"salt\":\
↪"a94a8fe5ccb19ba61c4c0873d391e987982fbbd3\"}" | base64)
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"VerifyAssetProperties","Args":["asset1"]}' --transient "{\
↪"asset_properties\":\"$ASSET_PROPERTIES\"}"
```

Run the following command to agree to buy asset1 for 100 dollars. As of now, Org2 will agree to a different price than Org2. Don't worry, the two organizations will agree to the same price in a future step. However, we we can use this temporary disagreement as a test of what happens if the buyer and the seller agree to a different price. Org2 needs to use the same `trade_id` as Org1.

```
export ASSET_PRICE=$(echo -n "{\"asset_id\":\"asset1\",\"trade_id\":\
↪"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\",\"price\":100}" | base64)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"AgreeToBuy","Args":["asset1"]}' --transient "{\"asset_
↪price\":\"$ASSET_PRICE\"}"
```

You can read the agreed purchase price from the Org2 implicit data collection:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"GetAssetBidPrice","Args":["asset1"]}'
```

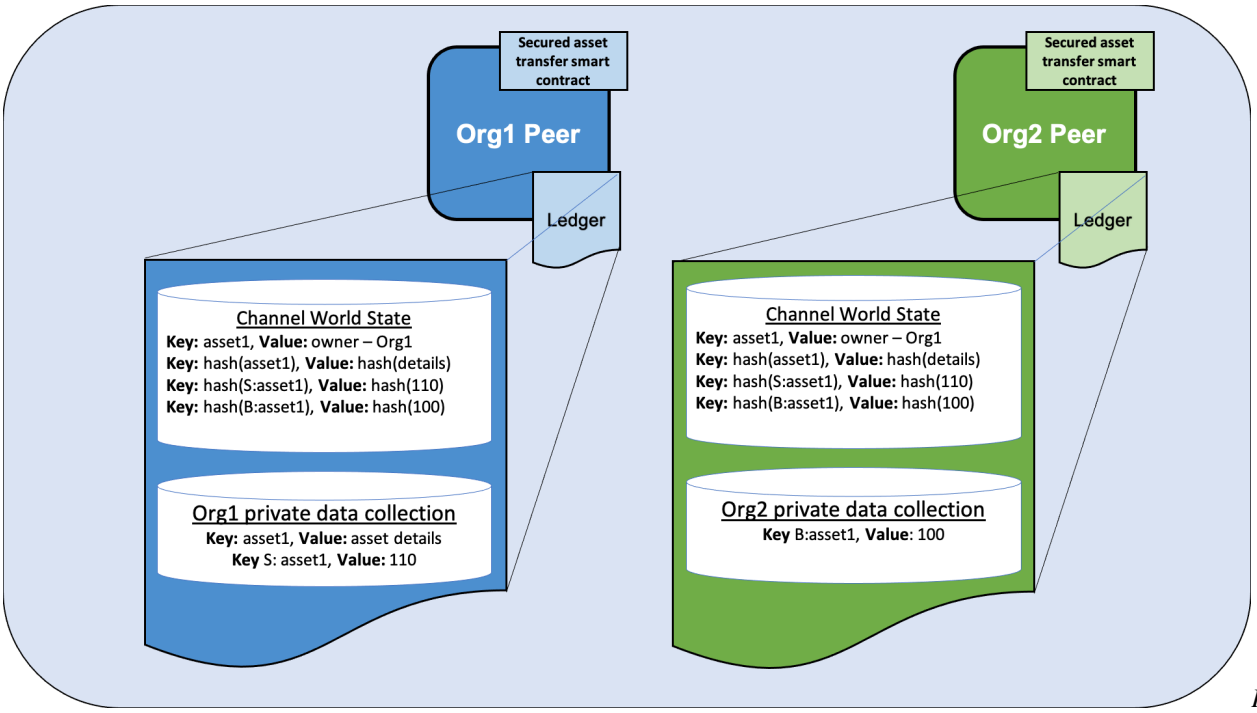


Figure 2: After Org1 and Org2 agree to transfer the asset, the price agreed to by each organization is stored in their private data collections. A composite key for the seller and the buyer is used to prevent a collision with the asset details and asset ownership record. The price that is agreed to is only stored on the peers of each organization. However, the hash of both agreements is stored in the channel world state on every peer joined to the channel.

7.5.11 Transfer the asset from Org1 to Org2

After both organizations have agreed to their price, Org1 can attempt to transfer the asset to Org2. The private asset transfer function in the smart contract uses the hash on the ledger to check that both organizations have agreed to the same price. The function will also use the hash of the private asset details to check that the asset that is transferred is the same asset that Org1 owns.

Transfer the asset as Org1

Operate from the Org1 terminal. The owner of the asset needs to initiate the transfer. Note that the command below uses the `--peerAddresses` flag to target the peers of both Org1 and Org2. Both organizations need to endorse the transfer. *Also note that the asset properties and price are passed in the transfer request as transient properties. These are passed so that the current owner can be sure that the correct asset is transferred for the correct price. These properties will be checked against the on-chain hashes by both endorsers.*

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"TransferAsset","Args":["asset1","Org2MSP"]}' --transient "
↪{\"asset_properties\":\"$ASSET_PROPERTIES\",\"asset_price\":\"$ASSET_PRICE\"}" --
↪peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

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

```
Error: endorsement failure during invoke. response: status:500 message:"failed␣
↪transfer verification: hash␣
↪cf74b8ce092b637bd28f98f7cdd490534c102a0665e7c985d4f2ab9810e30b1c for passed price␣
↪JSON {\"asset_id\":\"asset1\",\"trade_id\":\
↪"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\",\"price\":110} does not match on-chain␣
↪hash 09341dbb39e81fb50ccb3a81770254525318f777fad217ae49777487116cceb4, buyer hasn't␣
↪agreed to the passed trade id and price"
```

As a result, Org1 and Org2 come to a new agreement on the price at which the asset will be purchased. Org1 drops the price of the asset to 100:

```
export ASSET_PRICE=$(echo -n "{\"asset_id\":\"asset1\",\"trade_id\":\
↪"109f4b3c50d7b0df729d299bc6f8e9ef9066971f\",\"price\":100}" | base64)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"AgreeToSell","Args":["asset1"]}' --transient "{\"asset_
↪price\":\"$ASSET_PRICE\"}"
```

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

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"TransferAsset","Args":["asset1","Org2MSP"]}' --transient "
↪{\"asset_properties\":\"$ASSET_PROPERTIES\",\"asset_price\":\"$ASSET_PRICE\"}" --
↪peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

You can query the asset ownership record to verify that the transfer was successful.

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"ReadAsset","Args":["asset1"]}'
```

The record now lists Org2 as the asset owner:

```
{"object_type":"asset","asset_id":"asset1","owner_org":"Org2MSP","public_description":
↪"This asset is for sale"}
```

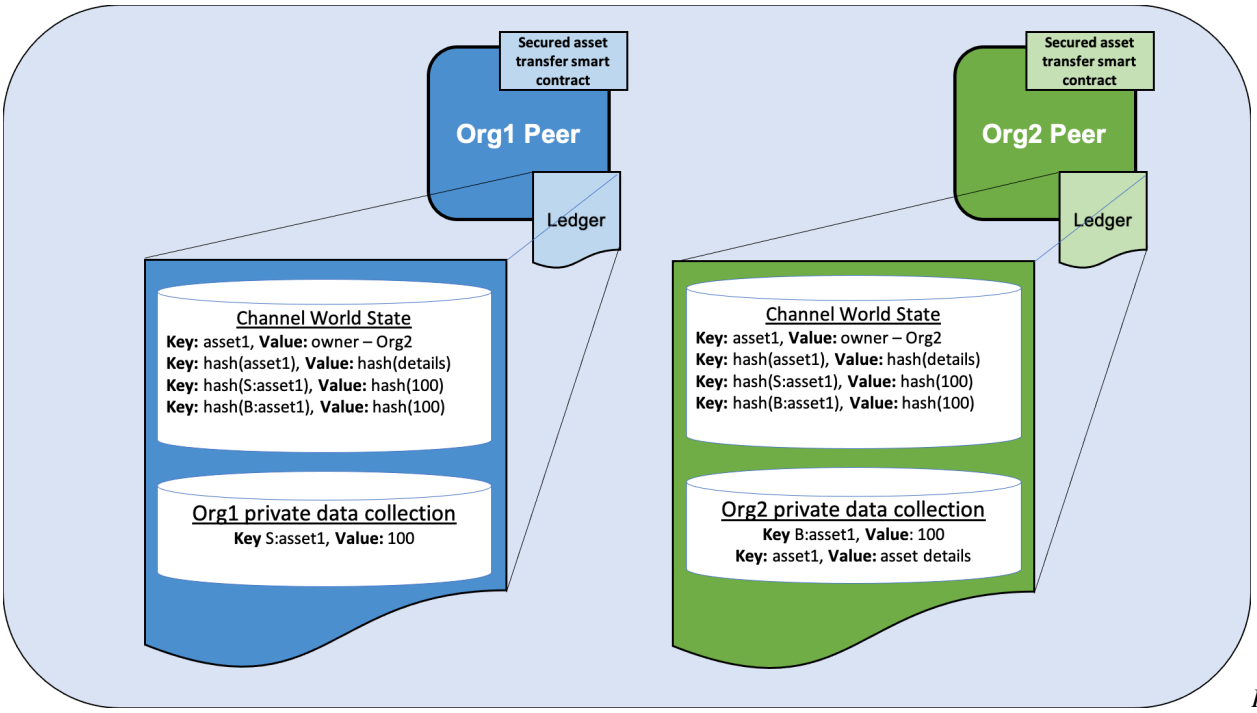


Figure 3: After the asset is transferred, the asset details are placed in the Org2 implicit data collection and deleted from the Org1 implicit data collection. As a result, the asset details are now only stored on the Org2 peer. The asset ownership record on the ledger is updated to reflect that the asset is owned by Org1.

Update the asset description as Org2

Operate from the Org2 terminal. Now that Org2 owns the asset, we can read the asset details from the Org2 implicit data collection:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"GetAssetPrivateProperties","Args":["asset1"]}'
```

Org2 can now update the asset public description:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"ChangePublicDescription","Args":["asset1","This asset is␣
↪not for sale"]}'
```

Query the ledger to verify that the asset is no longer for sale:

```
peer chaincode query -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪secured -c '{"function":"ReadAsset","Args":["asset1"]}'
```