Hyperledger Composer basics, Part 3: Deploy locally, interact with, and extend your blockchain network

Exploring more Hyperledger Composer tools, UIs, and security

J Steven Perry

December 14, 2017

In this tutorial, the third and final installment in this series, learn how to modify a sample blockchain network definition and transform it into a real-world blockchain application, write Cucumber feature tests, issue IDs for all participants, and execute every transaction through the CLI.

This tutorial builds on **Part 1**, where you learned how to model and test a simple business network in a local version of the Hyperledger Composer Playground. In **Part 2**, you saw how to model an IoT GPS sensor in a shipping container by adding GPS readings to the Shipment asset, and modify the chaincode to send an alert when the Shipment reaches its destination port. Then you deployed the sample Perishable Goods network to the IBM Cloud in the Composer Online Playground.

Now, in this third and final part, you'll install Hyperledger Fabric on your computer, and deploy the business network archive (BNA) to an instance of Hyperledger Fabric running on your machine (referred to as your *local Hyperledger Fabric*). You'll install more tools and generate a Loopback-based REST interface that you can use to interact with the sample network blockchain application.

Part 3 also includes a detailed discussion of Hyperledger Composer security concepts. This part concludes with steps to pull it all together and extend the `iot-perishable-network` to create a more "real world" version of the Perishable Goods network.

Prerequisites

I'll assume that you've worked through Part 2, and that you have the following software installed on your computer:

- Docker 17.03 or higher
- Web browser
- Composer Command Line Interface (`composer-cli`)

1. Deploy the network to your local Hyperledger Fabric

Hyperledger Fabric is a framework for building blockchain applications for business purposes, and as you have already learned, Hyperledger Composer is a companion tool that makes building blockchain applications that run on Hyperledger Fabric easier. Up to now, you have used Composer Playground in "browser-only" mode (in Part 1) and the Composer Online playground in the IBM Cloud (Part 2).

Now, you'll install and run the Hyperledger Fabric on your computer, and use the Composer Command Line Interface (CLI) to interact with it.

1a. Get Hyperledger Fabric

First, create a directory on your computer where you want to do your local Hyperledger Composer development. For this tutorial, I'll refer to this location as `$COMPOSER_ROOT` in code, and in prose, I'll refer to it as your *Composer root directory*. This directory can be anywhere you want, but I recommend you create it just off your home folder, and always set the `COMPOSER_ROOT` environment variable to that directory in the current shell, since that's how I'll refer to that location throughout this tutorial.

Now download the `fabric-dev-servers.zip` distribution file to your Composer root directory using the `curl` command, and unzip it. The sequence of commands looks like this (assuming ~/HyperledgerComposer as the root directory for this example):

```
mkdir ~/HyperledgerComposer
export COMPOSER_ROOT=~/HyperledgerComposer
cd $COMPOSER_ROOT
mkdir fabric-dev-servers && cd fabric-dev-servers
curl -O https://raw.githubusercontent.com/hyperledger/composer-tools/master/packages/fabric-dev-servers/fabric-dev-servers.zip
unzip fabric-dev-servers.zip
```

1b. Start Hyperledger Fabric

Note: before you continue, make sure Docker is running.

The first time you run Hyperledger Fabric, execute this sequence of commands:

```
cd $COMPOSER_ROOT/fabric-dev-servers
./downloadFabric.sh
./startFabric.sh
./createPeerAdminCard.sh
```

It will take several minutes to run the first command, which pulls all of the necessary Docker images. The second command starts the local Hyperledger Fabric. The last command issues an *ID card* for the fabric admin, which is `PeerAdmin`. Don't worry about that for now; I'll cover ID cards later in the tutorial. The output of the `createPeerAdminCard.sh` script looks like this:

```
$ ./createPeerAdminCard.sh
Development only script for Hyperledger Fabric control
Running 'createPeerAdminCard.sh'
FABRIC_VERSION is unset, assuming hlfv1
FABRIC_START_TIMEOUT is unset, assuming 15 (seconds)

Using composer-cli at v0.15.0
Successfully created business network card to /tmp/PeerAdmin@hlfv1.card

Command succeeded

Successfully imported business network card: PeerAdmin@hlfv1

Command succeeded

Hyperledger Composer PeerAdmin card has been imported
The following Business Network Cards are available:


┌─────────────────┬────────────┬─────────┐
│ CardName        │ UserId     │ Network │
├─────────────────┼────────────┼─────────┤
│ PeerAdmin@hlfv1 │ PeerAdmin  │         │
└─────────────────┴────────────┴─────────┘

Issue composer card list --name <CardName>  to get details of the card

Command succeeded
```

Make a note of the card name, because you will need it to execute all of the CLI commands in this tutorial.

You only need to run the `createPeerAdminCard.sh` script one time. From now on, whenever you start Hyperledger Fabric, you just run the `startFabric.sh` script like this:

```
cd $COMPOSER_ROOT/fabric-dev-servers
./startFabric.sh
```

When the script finishes, Hyperledger Fabric is running on your computer. To shut down the local Hyperledger Fabric, run the `stopFabric.sh` script.

For the purposes of the tutorial, I suggest you leave Hyperledger Fabric up for now.

1c. Deploy to Hyperledger Fabric

In Part 2, you cloned the `developerWorks` project from GitHub and modified the Perishable Goods network. I've provided a finished version of that network in the `developerWorks/iot-perishable-network` directory. You'll use that network for the rest of this tutorial.

To deploy the `iot-perishable-network` to your local Fabric, use the Composer CLI and execute this sequence of commands:

```
cd $COMPOSER_ROOT/developerWorks/iot-perishable-network
composer network deploy -a dist/iot-perishable-network.bna -A admin -S adminpw -c
PeerAdmin@hlfv1 -f networkadmin.card
composer card import --file networkadmin.card
```

The `composer network deploy` command deploys the specified network archive (`dist/iot-perishable-network.bna`) to the local Hyperledger Fabric, using the PeerAdmin card you created earlier when you ran the `createPeerAdminCard.sh` script to authenticate. When the deployment is finished, an ID card is *issued* for the network administrator, whose credentials — that is, userid and password (or secret) — are stored in the `networkadmin.card` file.

The output looks like this:

```
$ composer network deploy -a dist/iot-perishable-network.bna -A admin -S adminpw -c
PeerAdmin@hlfv1 -f networkadmin.card
Deploying business network from archive: dist/iot-perishable-network.bna
Business network definition:
 Identifier: iot-perishable-network@0.1.12
 Description: Shipping Perishable Goods Business Network

# Deploying business network definition. This may take a minute...
Successfully created business network card to networkadmin.card

Command succeeded
```

The `composer card import` command tells Hyperledger Composer to import the specified card file so it can be used later to authenticate the user whose credentials are stored in the card. In this case, the card is for the network admin. I'll talk more about cards later on. The output looks like this:

```
$ composer card import --file networkadmin.card
Successfully imported business network card: admin@iot-perishable-network

Command succeeded
```

2. Interact with the network via the Composer REST interface

You've deployed the `iot-perishable-network` to the local Hyperledger Fabric, but how do you see what's there? How do you interact with it? Hyperledger Composer provides a tool called `composer-rest-server` that generates a Loopback-based REST interface to access your network.

2a. Install the Composer REST interface generator

First, you need to install the `composer-rest-server` loopback generator. Go to a command line (Ubuntu) or open a Terminal window (MacOS) and enter this command:

```
npm install -g composer-rest-server
```

When the installation finishes, verify that `composer-rest-server` was installed correctly by running `composer-rest-server -v`:

```
$ composer-rest-server -v
v0.15.0
```

2b. Generate the REST interface

Make sure the Hyperledger Fabric is running and the `iot-perishable-network` is deployed before generating the REST interface (or there will be nothing to connect to). From the command line, run the `composer-rest-server` command. It will prompt you for input. You'll see something like this:

Listing 1. Starting the REST server

```
$ composer-rest-server
? Enter the name of the business network card to use: admin@iot-perishable-network
? Specify if you want namespaces in the generated REST API: always use namespaces
? Specify if you want to enable authentication for the REST API using Passport: No
? Specify if you want to enable event publication over WebSockets: No
? Specify if you want to enable TLS security for the REST API: No

To restart the REST server using the same options, issue the following command:
   composer-rest-server -c admin@iot-perishable-network -n always

Discovering types from business network definition ...
Discovered types from business network definition
Generating schemas for all types in business network definition ...
Generated schemas for all types in business network definition
Adding schemas for all types to Loopback ...
Added schemas for all types to Loopback
Web server listening at: http://localhost:3000
Browse your REST API at http://localhost:3000/explorer
```

The REST server needs to know how to authenticate with Hyperledger Fabric so it can communicate with the business network. You provide an ID card for that (line 2) such as the `admin@iot-perishable-network` ID card from earlier in the tutorial.

Namespaces help avoid name collisions. In the `iot-perishable-network`, this is not a big deal since there's only a handful of assets, participants, and transactions. I think it's a good idea to always use namespaces (line 3). Testing real-world networks with lots of participants, assets, and transactions is hard enough; using namespaces will help you avoid headaches related to name collisions.

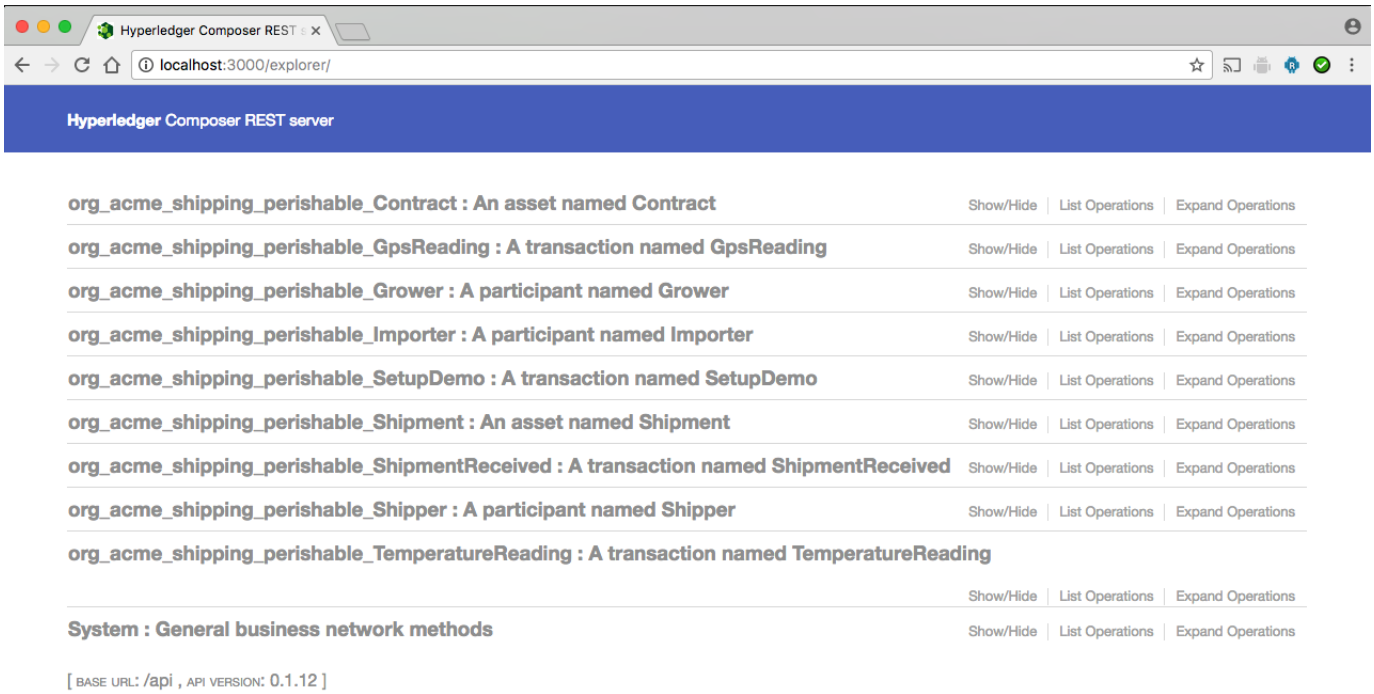
When using the REST interface for testing, don't worry about authentication (line 4), events and WebSockets (line 5), or TLS security (line 6) for now. However, if you decide to deploy the REST server as part of your production blockchain solution, make sure to check out the links here or in the Related topics at the end of this tutorial.

The next time you want to start the REST server with those same options, just use the command shown (line 9) and skip the interview!

2c. Use the REST interface

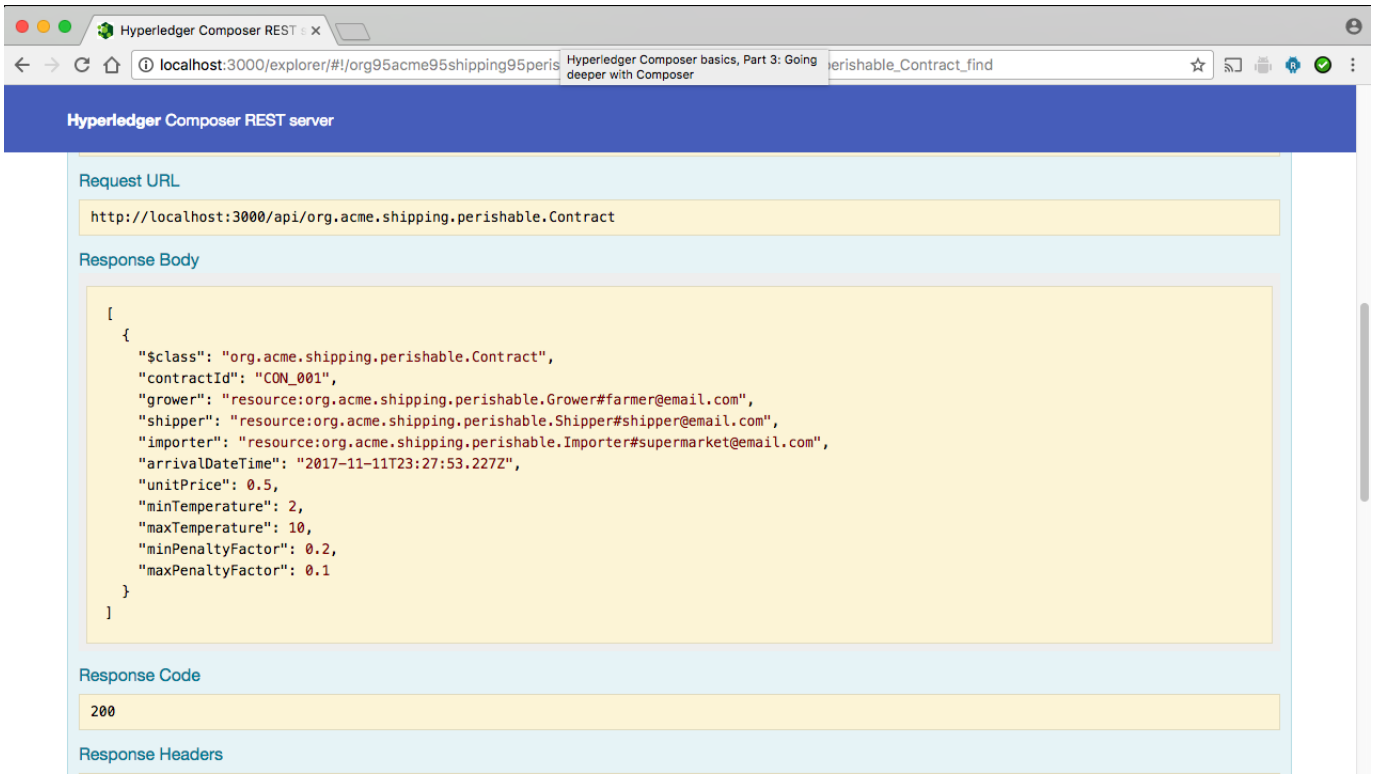
To use the REST interface, open a browser and point it to the address shown on line 18 in Listing 1. You'll see something like this:

Figure 1. The REST interface



The interface is fairly intuitive. To work with an object, click it, and when it expands, you'll see the REST methods you can invoke (for example, `/GET`, `/POST`, and so on). To invoke the `SetupDemo` transaction (which instantiates the business model), click the line that contains `SetupDemo`, which then expands to the `POST` method. Click the **Try it out!** button to invoke the transaction. If it succeeds, you'll see an HTTP 200 response code. Then you can navigate through the model and see the various objects like the Contract, shown in Figure 2.

Figure 2. The REST interface showing Contract object



In the video I'll show you how to use the REST interface in detail, so be sure to check that out.

2d. Secure the REST interface

This is beyond the scope of this tutorial, but if you plan to run the REST interface in production, you need a strategy to deal with it. An entire tutorial could be done on this topic alone! Fortunately there are resources on the Hyperledger Composer website to help. Here are some links to the Composer docs:

- REST interface authentication
- Events and WebSockets
- TLS security

For your convenience, these links are also listed in the Related topics at the end of the tutorial as well.

Video: Using the Composer REST interface

To view this video, **Using composer-rest-server** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

3. Set up Hyperledger Composer security

There are two levels of security with Hyperledger Composer:

- Hyperledger Fabric administrator
- Business network administrator

The administrator for your local Hyperledger Fabric is the Peer Administrator (or PeerAdmin for short), which you created when you installed the local Hyperledger Fabric. Every business network should have an administrator as well, which is created when the network is deployed by the Hyperledger Fabric administrator. Authentication for both is handled using ID Cards, which you'll learn about next.

3a. ID Cards

An ID card (or *card* for short) is a collection of files that contains all the information necessary to allow a participant to connect to a business network. The card is referred to as an *identity*. Before it can be used, it must be *issued* to the user, allowing him or her to be authenticated and authorized to use the network. Cards are a very handy way of securing access to a Hyperledger Fabric network. Rather than keeping up with a password (called a *secret* in Hyperledger Composer terminology), you import the card into a collection of cards in the Hyperledger Fabric called a *wallet*. From that point on, you can just reference the card to authenticate that identity.

The general form for specifying a card is: `userid@network`, where `userid` is the user's unique id, and the `network` is the network to which the user is authenticated.

To handle the two levels of Hyperledger Composer security, you need a card for at least: (1) the PeerAdmin and (2) the business network admin.

PeerAdmin

The `PeerAdmin` card is a special ID card used to administer the local Hyperledger Fabric. In a development installation, such as the one on your computer, the PeerAdmin ID card is created when you install the local Hyperledger Fabric.

The form for a `PeerAdmin` card for a Hyperledger Fabric v1.0 network is `PeerAdmin@hlfv1`. You already used this card earlier in the tutorial when you deployed the `iot-perishable-network` to your local Hyperledger Fabric.

In general, the PeerAdmin is a special role reserved for functions such as:

- Deploying business networks
- Creating, issuing, and revoking ID cards for business network admins

As a developer, in a production Hyperledger Fabric installation, you would not have access to the `PeerAdmin` card. Instead the Hyperledger Fabric administrator would deploy your business network, create ID cards, and so on. When developing and testing blockchain networks using your local Hyperledger Fabric, you will use the PeerAdmin ID card to perform these functions.

Business network admin

When the PeerAdmin deploys your network to the Hyperledger Fabric, an ID card is issued to the business network administrator, and then this card is used whenever the business network administrator needs to do anything with the business network, such as using the Composer Command Line Interface (which you'll use shortly).

Remember the `admin@iot-perishable-network` card from earlier? That was the business network admin card issued by the `PeerAdmin` (that is, the local Hyperledger Fabric administrator) when the `iot-perishable-network` was deployed.

In general, the business network admin is a special role reserved for functions such as:

- Updating the running business network
- Querying the various registries (participant, identity, and so forth)
- Creating, issuing, and revoking ID cards for participants in the business network

That's right, the admin ID card can also be used to issue other ID cards for specific participants (I'll show you how to do that later in the tutorial), so that all participants have their own ID cards. Access to the network can be controlled by these cards.

3b. Access control

Speaking of access control, Composer implements the concept of role-based security through permissions baked right into its architecture to handle both authentication and authorization. A participant's access to resources is controlled based on the identity that has been issued to that participant.

In this section, you'll see how to set up access control rules to lock down resources in your network by participant through the Access Control List (ACL) file called `permissions.acl` (at the time of this writing, the file **must** have this name). Here are the highlights:

- Access is *applied* to either *grant* or *deny* based on resources that *match* the rule. By default, if a resource matches no rule, then access to it is denied.
- The file is processed top-down so that the first rule that either grants or denies access to a particular resource is in effect and cannot be overridden by a subsequent rule.
- The format of a rule is fairly intuitive. The `rule` keyword indicates the start of a rule, followed by a rule name, which must be unique.
- The rule consists of a set of name/value pairs that define the rule's properties.

Listing 2 shows the `permissions.acl` file from the `iot-perishable-network` business network.

Listing 2. `permissions.acl` from the `iot-perishable-network`

```
/**
 * Sample access control list.
 */
rule Default {
    description: "Allow all participants access to all resources"
    participant: "ANY"
    operation: ALL
    resource: "org.acme.shipping.perishable.*"
    action: ALLOW
}

rule SystemACL {
  description:  "System ACL to permit all access"
  participant: "org.hyperledger.composer.system.Participant"
  operation: ALL
  resource: "org.hyperledger.composer.system.**"
  action: ALLOW
```

```
}

rule NetworkAdminUser {
    description: "Grant business network administrators full access to user resources"
    participant: "org.hyperledger.composer.system.NetworkAdmin"
    operation: ALL
    resource: "**"
    action: ALLOW
}

rule NetworkAdminSystem {
    description: "Grant business network administrators full access to system resources"
    participant: "org.hyperledger.composer.system.NetworkAdmin"
    operation: ALL
    resource: "org.hyperledger.composer.system.**"
    action: ALLOW
}
```

Stay tuned!

This explanation of how rules are processed represents how access control works at the time of this writing. Hyperledger Composer has not reached release 1.0 yet, so this may change. The final authority on how things work, of course, is in the code, and ideally the documentation.

The general format for each property is `property: "MATCH_EXPRESSION"`. The properties are:

description— a human-readable name for the rule in double quotes. Example: `description: "This is a description"`

participant— fully qualified name of the Participant to which access is granted or denied, surrounded by double quotes. Multiple participants may be specified in the `MATCH_EXPRESSION` through the use of the single asterisk (`*`) wildcard to indicate "all", double asterisk (`**`) to indicate recursion within a namespace, or `ANY`, which matches all participants in all namespaces.

Examples:

- `participant: "org.acme.shipping.perishable.Grower"`— Apply the rule to `org.acme.shipping.perishable.Grower` only.
- `participant: "org.acme.shipping.perishable.*"`— Apply the rule to all participants in the `org.acme.shipping.perishable` namespace.
- `participant: "org.acme.shipping.**"`— Apply the rule to all participants in the `org.acme.shipping` namespace, and recursively any namespaces beneath it.
- `participant: "ANY"`— Apply the rule to all participants in all namespaces.

operation— the `MATCH_EXPRESSION` may be one or more of `CREATE`, `READ`, `UPDATE`, `DELETE`, or `ALL`. Multiple values may be separated by commas (for example, `CREATE,READ` will grant both `CREATE` and `READ` access to the resource). Use `ALL` by itself to indicate the rule applies to all operations.

Examples:

- `operation: ALL`— Apply the rule to all CRUD operations.
- `operation: CREATE`— Apply the rule only to the `CREATE` operation
- `operation: READ,UPDATE`— Apply the rule to `READ` and `UPDATE` operations.

resource— defines the "thing" to which the rule applies. A resource can be any class (that is, an asset, participant, or transaction) from the business model. Through the use of wildcards, multiple classes can also be specified, in the same way as for participant above.

Examples:

- `resource: "org.acme.shipping.perishable.TemperatureReading"`— The rule applies only to the `org.acme.shipping.perishable.TemperatureReading` transaction.
- `resource: "org.acme.shipping.perishable.*"`— The rule applies to all classes in the `org.acme.shipping.perishable` namespace.
- `resource: "org.acme.shipping.**"`— Apply the rule to all resources in the `org.acme.shipping` namespace, and recursively to any namespaces beneath it.

action— the action that applies when the rule fires. One of: `ALLOW` to grant access, or `DENY` to deny the specified participant(s) access to the resource(s).

Examples:

- `action: ALLOW`— Allow access to the specified resource(s).
- `action: DENY`— Deny access to the specified resource(s).

Note: If your network has no `permissions.acl`, then access is granted to all participants (that is, access is wide open — there is no resource-level security).

You will see how to use rules later in the tutorial. Also, the Composer ACL documentation has lots of examples, so be sure to check that out.

Issue a new ID - Playground

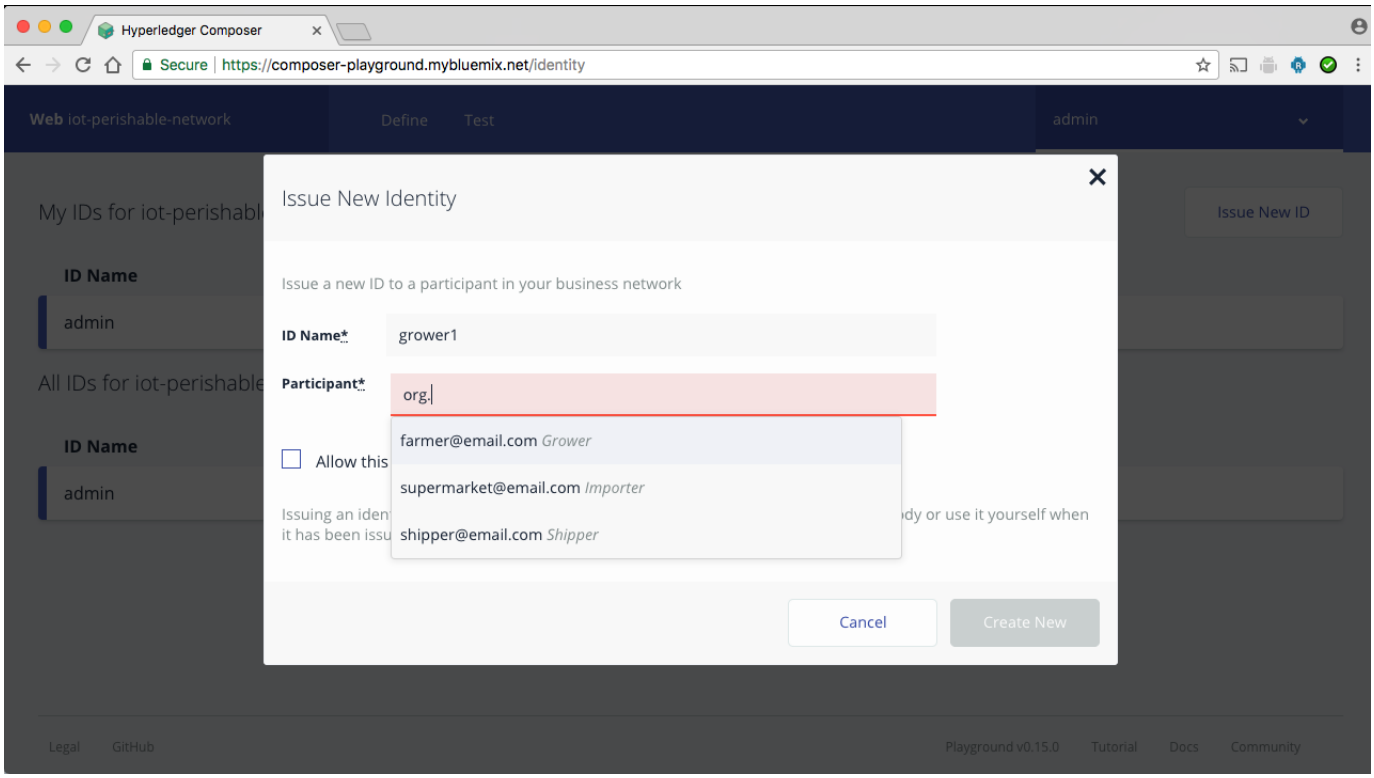
ID cards can be kind of abstract, and difficult to visualize. So let's look at one in Playground (in fact, you already have, you just might not have realized it).

Start the Hyperledger Composer Playground and import the `iot-perishable-network` (don't forget to create the BNA by running `npm install` if you haven't already). You imported a network in Part 2, so check that out if you need a refresher.

Make sure to invoke the `SetupDemo` transaction to create the participants and store them in the participant registry. Why? Because you cannot issue an ID card for a participant that is not in the participant registry. In order to work with the classes from the business model, you must *instantiate* the model, and that's what `SetupDemo` does for you. Please review the "Test the business network" section from Part 1 of this tutorial series if you want more information on working with business models.

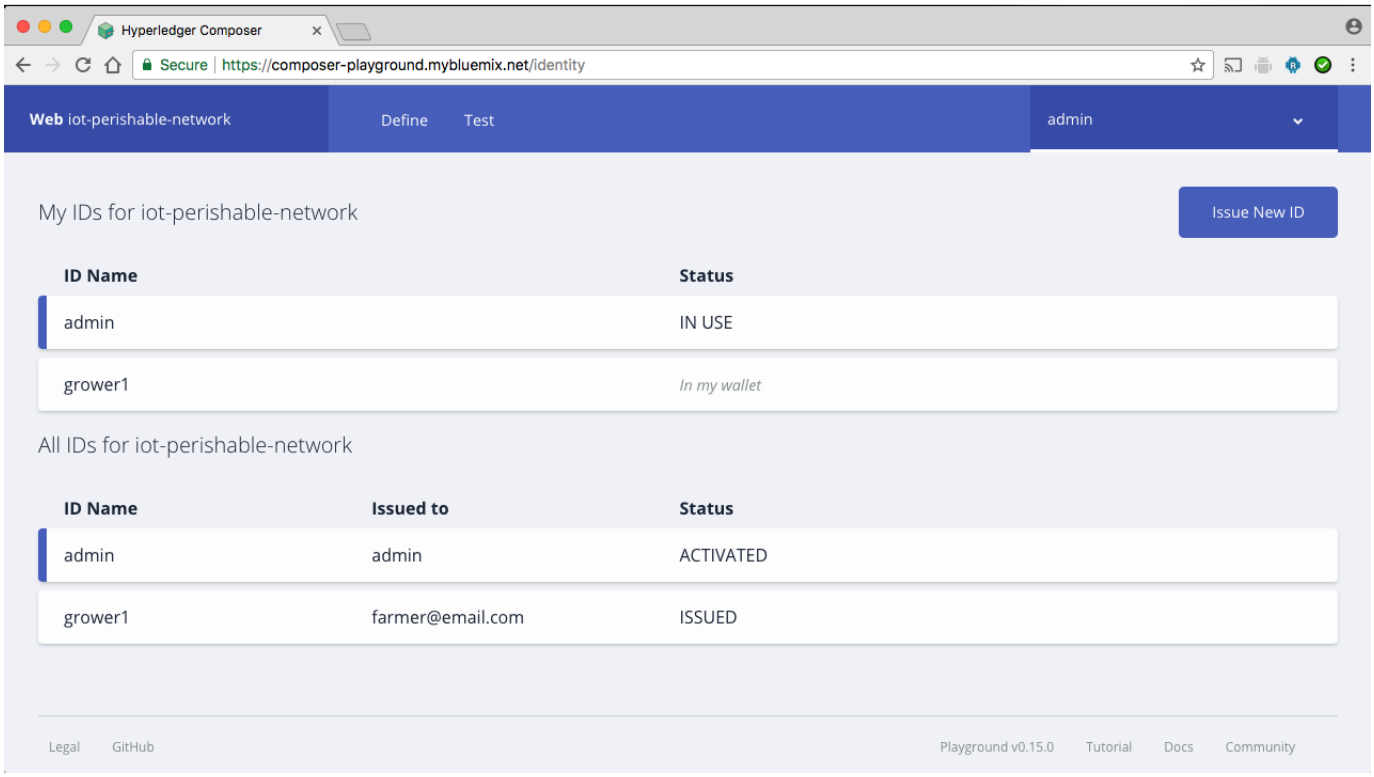
Once the model is instantiated, in the upper right corner where you see **admin**, click to expand the drop-down and select **ID Registry**. On the next screen, choose **Issue new ID** and then enter **grower1** as the ID Name, and select **farmer@email.com** from the drop-down. See Figure 3.

Figure 3. Issue a new ID card in Playground



Click the **Create New** button to issue the ID. Once the **grower1** ID card has been issued, it will show up in the ID registry. See Figure 4.

Figure 4. New grower1 ID card in Playground



I'll show you how to issue ID cards using the Composer CLI later in the tutorial.

4. Interact with the network via the Composer CLI

I showed you how to install the Composer Command Line Interface (CLI) in Part 2, and you used it earlier in this tutorial to deploy the `iot-perishable-network` and import the `admin@iot-perishable-network` card. Now you will use the CLI to interact with the `iot-perishable-network` business network. The CLI is really handy because it lends itself to use in scripts, and it's easy to use, as you'll see. To use the CLI, you'll need to go to the command line (Ubuntu) or open a terminal window (MacOS).

Ping the network

Once you've deployed the `iot-perishable-network` network, you can send it a *ping*, which doesn't have any side effects. It's a safe way to just see if your network is up and running. Enter this command:

```
composer network ping --card admin@iot-perishable-network
```

If the `ping` subcommand succeeds, you will see output like this:

```
$ composer network ping --card admin@iot-perishable-network
The connection to the network was successfully tested: iot-perishable-network
 version: 0.15.0
 participant: org.hyperledger.composer.system.NetworkAdmin#admin

Command succeeded
```

Invoke the SetupDemo transaction

The `SetupDemo` transaction is used to instantiate the model. Enter this command:

```
                composer transaction submit --card admin@iot-perishable-network -d '{"$class":
"org.acme.shipping.perishable.SetupDemo"}'
```

The output looks like this:

```
                $ composer transaction submit --card admin@iot-perishable-network -d '{"$class":
"org.acme.shipping.perishable.SetupDemo"}'
                Transaction Submitted.

                Command succeeded
```

Invoke the TemperatureReading transaction

The `TemperatureReading` transaction is invoked by the IoT sensor in the shipping container each time a temperature reading is taken and needs to be recorded in the blockchain. Enter this command:

```
                composer transaction submit --card admin@iot-perishable-network -d
'{ "$class": "org.acme.shipping.perishable.TemperatureReading", "centigrade": 0, "shipment":
"resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
```

The output looks like this:

```
                $ composer transaction submit --card admin@iot-perishable-network -d
'{ "$class": "org.acme.shipping.perishable.TemperatureReading", "centigrade": 0, "shipment":
"resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Transaction Submitted.

                Command succeeded
```

Invoke the ShipmentReceived transaction

The `ShipmentReceived` transaction is invoked by the Importer when the shipment is received. Enter this command:

```
                composer transaction submit --card admin@iot-perishable-network
-d '{ "$class": "org.acme.shipping.perishable.ShipmentReceived", "shipment":
"resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
```

The output looks like this:

```
                $ composer transaction submit --card admin@iot-perishable-network
-d '{ "$class": "org.acme.shipping.perishable.ShipmentReceived", "shipment":
"resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Transaction Submitted.

                Command succeeded
```

Update the network

This is handy whenever you make a change during development and you need to update the deployed network. After you run `npm install` to rebuild the BNA, enter this command:

```
                composer network update -a dist/iot-perishable-network.bna --card admin@iot-perishable-
network
```

The output looks like this:

```
                $ composer network update -a dist/iot-perishable-network.bna --card admin@iot-perishable-
network
                Deploying business network from archive: dist/iot-perishable-network.bna
                Business network definition:
                 Identifier: iot-perishable-network@0.1.12
                 Description: Shipping Perishable Goods Business Network

                # Updating business network definition. This may take a few seconds...
                Successfully created business network card to undefined

                Command succeeded
```

Be sure to check out the CLI documentation for a complete reference of commands, examples, and lots more.

5. Put it all together

Everything you've learned so far has been building to this point: You can now modify the `iot-perishable-network` into a more realistic blockchain application. Here's what you will do in this section:

- Make changes to the network model:
  - Add new participants to represent the IoT sensors in the shipping container
  - Add new transactions that more realistically model a shipping workflow
  - Add new events that are broadcast when certain points in the workflow are reached
- Add chaincode for the new transactions
- Modify the network's permissions
- Add Cucumber feature tests to unit test your changes
- Build the network and run the unit tests
- Deploy the network to your local Hyperledger Fabric
- Issue IDs for the Grower, Temperature and GPS sensors, Shipper, and Importer
- Run transactions through the CLI

When you're finished modifying the `iot-perishable-network`, you will have learned the basic skills needed to use Hyperledger Composer, you will be ready to tackle a real-world blockchain application project, you will be the envy of all your friends, and tales of your Hyperledger Composer and Fabric prowess will spread far and wide (okay, I may be embellishing a bit with those last two).

If you get stuck at any point and need a little help, be sure to check out the solution code, which is in a project called `iot-perishable-network-advanced` that you have already cloned to your computer when you cloned the `developerWorks` project from GitHub.

Note: You are welcome to use whatever editor you're comfortable with, but I will use VSCode (which you installed in Part 2) for all the changes in this section, and the instructions below will reflect that. If you're not using VSCode, please adapt accordingly.

5a. Make model changes

To make the changes listed above, you'll start with the model. Open the `models/perishable.cto` model file in VSCode and make the changes below. Note: I have omitted comments from the model snippets below only to save space in this tutorial. If you look at `iot-perishable-network-advanced`, you'll see that I have added comments for each model element I added, and I recommend you do the same.

First, modify the `ShipmentReceived` transaction to add an optional property called `receivedDateTime`, and then add three new transactions that extend `ShipmentTransaction` to update the blockchain (ledger) when a shipment (1) has been packed, (2) has been picked up for loading, and (3) has been loaded onto the container ship:

```
transaction ShipmentReceived extends ShipmentTransaction {
  o DateTime receivedDateTime optional
}

transaction ShipmentPacked extends ShipmentTransaction {
}

transaction ShipmentPickup extends ShipmentTransaction {
}

transaction ShipmentLoaded extends ShipmentTransaction {
}
```

Next, modify the `Shipment` asset to add four new properties for the shipment-related transactions so that when these transactions run, they are stored in the blockchain with the `Shipment`. The highlighted lines below are the ones you need to add (lines 9-12).

```
asset Shipment identified by shipmentId {
  o String shipmentId
  o ProductType type
  o ShipmentStatus status
  o Long unitCount
  --> Contract contract
  o TemperatureReading[] temperatureReadings optional
  o GpsReading[] gpsReadings optional
  o ShipmentPacked shipmentPacked optional
  o ShipmentPickup shipmentPickup optional
  o ShipmentLoaded shipmentLoaded optional
  o ShipmentReceived shipmentReceived optional
}
```

Now add an abstract participant to represent an IoT device that is identified by a String property called `deviceId`, and add two subclasses of it: one to represent a temperature sensor, and another to represent a GPS sensor. These participants will update the blockchain with their respective readings. Devices are network participants? That's right; a participant in a blockchain network does not have to be a human being.

```
abstract participant IoTDevice identified by deviceId {
  o String deviceId
}

participant TemperatureSensor extends IoTDevice {
}

participant GpsSensor extends IoTDevice {
}
```

Finally, in a real blockchain application, events would be emitted at each important point in the workflow: when a shipment has been packed, when it has been picked up, and so forth. Add four new events to represent these points in the shipment workflow.

```
event ShipmentPackedEvent {
  o String message
  --> Shipment shipment
}

event ShipmentPickupEvent {
  o String message
  --> Shipment shipment
}

event ShipmentLoadedEvent {
  o String message
  --> Shipment shipment
}

event ShipmentReceivedEvent {
  o String message
  --> Shipment shipment
}
```

That's it for the model changes. Next, you need to make a few modifications to the existing JavaScript transactions, and add chaincode for the new transactions.

5b. Add transaction chaincode

Until now, all of the chaincode for the networks you've worked with has been contained in `lib/logic.js`. Obviously, a real-world blockchain application could have lots of transactions, causing `logic.js` to become cluttered, and eventually difficult to maintain.

So long as all of your transaction code is in the `lib` directory of your model, Hyperledger Composer has no problem resolving the function calls. To illustrate this, you will make a few changes to `logic.js` and add a few more JavaScript source files.

In VSCode, right-click the `lib` directory and choose **New File**, and then enter `instantiateModelForTesting.js` as the file name. Now paste in the following header, or the "license check" step of the build will fail, giving you an error message when you build the code later:

Listing 3. The Apache 2.0 license header. ***Make sure to paste this into every new JavaScript file you create in your project*.**

```
/*
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

Now add the following lines of code to instantiate a TemperatureSensor and GpsSensor as part of the testing setup. Add just the highlighted lines. I've included surrounding lines of existing code for context, so you can locate them in the code:

```
.
.
// create the shipper
var shipper = factory.newResource(NS, 'Shipper', 'shipper@email.com');
var shipperAddress = factory.newConcept(NS, 'Address');
shipperAddress.country = 'Panama';
shipper.address = shipperAddress;
shipper.accountBalance = 0;

// create the Temperature sensor
var temperatureSensor = factory.newResource(NS, 'TemperatureSensor', 'SENSOR_TEMP001');
// create the GPS sensor
var gpsSensor = factory.newResource(NS, 'GpsSensor', 'SENSOR_GPS001');

// create the contract
var contract = factory.newResource(NS, 'Contract', 'CON_001');
contract.grower = factory.newRelationship(NS, 'Grower', 'farmer@email.com');
contract.importer = factory.newRelationship(NS, 'Importer', 'supermarket@email.com');
.
.
.then(function() {
    return getParticipantRegistry(NS + '.Shipper');
})
.then(function(shipperRegistry) {
    // add the shippers
    return shipperRegistry.addAll([shipper]);
})
.then(function() {
    return getParticipantRegistry(NS + '.TemperatureSensor');
})
.then(function(temperatureSensorRegistry) {
    // add the temperature sensors
    return temperatureSensorRegistry.addAll([temperatureSensor]);
})
.then(function() {
    return getParticipantRegistry(NS + '.GpsSensor');
})
.then(function(gpsSensorRegistry) {
    // add the GPS sensors
    return gpsSensorRegistry.addAll([gpsSensor]);
})
.then(function() {
    return getAssetRegistry(NS + '.Contract');
```

```
                })
                .then(function(contractRegistry) {
                    // add the contracts
                    return contractRegistry.addAll([contract]);
                })
                .
                .
```

Why the function renaming?

It's my firm belief that code names should mirror their model artifacts as closely as possible. The purpose of the `setupDemo` function is to instantiate the model for testing purposes, so let's just call it that. Then there's the `payOut` function, which does handle the participant payout when a shipment is received, but (IMO) it doesn't cleanly map to the model, so let's call it `receiveShipment` since it will be invoked once the shipment is received.

Now open `logic.js`, select the entire function body of `setupDemo`, and then cut it from `logic.js` and paste it into the new file. Finally, rename the function to `instantiateModelForTesting`.

Next, in `logic.js`, rename the `payOut` function to `receiveShipment` so that it more closely matches the transaction in the model. Then add the following code to emit the `ShipmentReceivedEvent` just after the block of code that computes the payout. The highlighted lines are the ones you should modify and/or add (modify line 6, add lines 17-27). I've included surrounding lines of existing code for context.

```
                /**
                 * A shipment has been received by an importer
                 * @param {org.acme.shipping.perishable.ShipmentReceived} shipmentReceived - the
ShipmentReceived transaction
                 * @transaction
                 */
                function receiveShipment(shipmentReceived) {
                    .
                    .
                    //console.log('Payout: ' + payOut);
                    contract.grower.accountBalance += payOut;
                    contract.importer.accountBalance -= payOut;

                    //console.log('Grower: ' + contract.grower.$identifier + ' new balance: ' +
contract.grower.accountBalance);
                    //console.log('Importer: ' + contract.importer.$identifier + ' new balance: ' +
contract.importer.accountBalance);

                    var NS = 'org.acme.shipping.perishable';
                    // Store the ShipmentReceived transaction with the Shipment asset it belongs to
                    shipment.shipmentReceived = shipmentReceived;

                    var factory = getFactory();
                    var shipmentReceivedEvent = factory.newEvent(NS, 'ShipmentReceivedEvent');
                    var message = 'Shipment ' + shipment.$identifier + ' received';
                    //console.log(message);
                    shipmentReceivedEvent.message = message;
                    shipmentReceivedEvent.shipment = shipment;
                    emit(shipmentReceivedEvent);

                    return getParticipantRegistry('org.acme.shipping.perishable.Grower')
                        .then(function (growerRegistry) {
                            // update the grower's balance
                            return growerRegistry.update(contract.grower);
                        })
                    .
                    .
```

Now you'll need to add JavaScript chaincode for the three new transactions you added to the model, and you'll create a new JavaScript source file for that. In VSCode, right-click the `lib` directory and choose **New File**, and then enter `shipment.js` as the file name. Add the functions below to this file. *Make sure to paste in the Apache 2.0 comment header at the top of the file. See Listing 3.*

Transaction: `ShipmentPacked` chaincode

First, add a function to handle the `ShipmentPacked` transaction. When the Grower participant (or one of its authorized agents) packs the shipment for pickup and transport, it invokes the `ShipmentPacked` blockchain transaction to record this fact in the ledger.

When this transaction is invoked, a `ShipmentPackedEvent` is emitted ,notifying any interested parties of this, and then the chaincode updates the ledger.

Name the new function `packShipment` using the "verb/object" function naming idiom. In JavaScript code, the comments shown below are mandatory for Hyperledger Composer to recognize the function as a transaction.

Copy *all* of the code below and paste it into `shipment.js`.

```
/**
 * ShipmentPacked transaction - invoked when the Shipment is packed and ready for pickup.
 *
 * @param {org.acme.shipping.perishable.ShipmentPacked} shipmentPacked - the ShipmentPacked
transaction
 * @transaction
 */
function packShipment(shipmentPacked) {
    var shipment = shipmentPacked.shipment;
    var NS = 'org.acme.shipping.perishable';
    var contract = shipment.contract;
    var factory = getFactory();

    // Add the ShipmentPacked transaction to the ledger (via the Shipment asset)
    shipment.shipmentPacked = shipmentPacked;

    // Create the message
    var message = 'Shipment packed for shipment ' + shipment.$identifier;

    // Log it to the JavaScript console
    //console.log(message);

    // Emit a notification telling subscribed listeners that the shipment has been packed
    var shipmentPackedEvent = factory.newEvent(NS, 'ShipmentPackedEvent');
    shipmentPackedEvent.shipment = shipment;
    shipmentPackedEvent.message = message;
    emit(shipmentPackedEvent);

    // Update the Asset Registry
    return getAssetRegistry(NS + '.Shipment')
        .then(function (shipmentRegistry) {
            // add the temp reading to the shipment
            return shipmentRegistry.update(shipment);
        });
}
```

Transaction: `ShipmentPickup` chaincode

Next, when the Shipper participant (or one of its authorized agents) picks up the packed shipment for transport, it invokes the `ShipmentPickup` blockchain transaction to record this fact in the ledger. Add a function to handle the `ShipmentPickup` transaction called `pickupShipment`.

When this transaction is invoked, a `ShipmentPickupEvent` is emitted, notifying any interested parties of this, and then the chaincode updates the ledger.

Copy all of the code below and paste it into `shipment.js`.

```
/**
 * ShipmentPickup - invoked when the Shipment has been picked up from the packer.
 *
 * @param {org.acme.shipping.perishable.ShipmentPickup} shipmentPickup - the ShipmentPickup
transaction
 * @transaction
 */
function pickupShipment(shipmentPickup) {
    var shipment = shipmentPickup.shipment;
    var NS = 'org.acme.shipping.perishable';
    var contract = shipment.contract;
    var factory = getFactory();

    // Add the ShipmentPacked transaction to the ledger (via the Shipment asset)
    shipment.shipmentPickup = shipmentPickup;

    // Create the message
    var message = 'Shipment picked up for shipment ' + shipment.$identifier;

    // Log it to the JavaScript console
    //console.log(message);

    // Emit a notification telling subscribed listeners that the shipment has been packed
    var shipmentPickupEvent = factory.newEvent(NS, 'ShipmentPickupEvent');
    shipmentPickupEvent.shipment = shipment;
    shipmentPickupEvent.message = message;
    emit(shipmentPickupEvent);

    // Update the Asset Registry
    return getAssetRegistry(NS + '.Shipment')
        .then(function (shipmentRegistry) {
            // add the temp reading to the shipment
            return shipmentRegistry.update(shipment);
        });
}
```

Transaction: `ShipmentLoaded` chaincode

Finally, when the Shipper participant (or one of its authorized agents) loads the shipment onto the container ship, it invokes the `ShipmentLoaded` blockchain transaction to record this fact in the ledger. Add a function to handle the `ShipmentLoaded` event called `loadShipment`.

When this transaction is invoked, a `ShipmentLoadedEvent` is emitted, notifying any interested parties of this, and then the chaincode updates the ledger.

Copy all of the code below and paste it into `shipment.js`.

```
/**
 * ShipmentLoaded - invoked when the Shipment has been loaded onto the container ship.
```

```
                 *
                 * @param {org.acme.shipping.perishable.ShipmentLoaded} shipmentLoaded - the ShipmentLoaded
transaction
                 * @transaction
                 */
                function loadShipment(shipmentLoaded) {
                    var shipment = shipmentLoaded.shipment;
                    var NS = 'org.acme.shipping.perishable';
                    var contract = shipment.contract;
                    var factory = getFactory();

                    // Add the ShipmentPacked transaction to the ledger (via the Shipment asset)
                    shipment.shipmentLoaded = shipmentLoaded;

                    // Create the message
                    var message = 'Shipment loaded for shipment ' + shipment.$identifier;

                    // Log it to the JavaScript console
                    //console.log(message);

                    // Emit a notification telling subscribed listeners that the shipment has been packed
                    var shipmentLoadedEvent = factory.newEvent(NS, 'ShipmentLoadedEvent');
                    shipmentLoadedEvent.shipment = shipment;
                    shipmentLoadedEvent.message = message;
                    emit(shipmentLoadedEvent);

                    // Update the Asset Registry
                    return getAssetRegistry(NS + '.Shipment')
                        .then(function (shipmentRegistry) {
                            // add the temp reading to the shipment
                            return shipmentRegistry.update(shipment);
                        });
                }
```

5c. Change access control permissions

In a real blockchain application, it is not ideal to grant any participant permission to do anything they want to. For example, a Grower shouldn't be able to invoke transactions that make sense only for a Shipper, and a GPS sensor shouldn't be allowed to record temperature readings in the blockchain.

So how do you prevent that from happening? In other words, how do you build your network such that you control access to its resources? Through the Access Control List (ACL) file.

You've already seen it in Parts 1 and 2 of this series, but I just sort of glossed over it, and promised to cover it in more detail at a later point. Well, that time has come.

It's time to talk about access control, which begins with `permissions.acl`. You saw this file earlier, but it's time to put that knowledge to use. You'll modify the `iot-perishable-network` to ensure that resources can be accessed only by certain participants that make sense to access that resource. As an illustration, I've summarized the access control settings that make sense for the `iot-perishable-network` in Table 1.

You may notice the rule names and think they look a bit odd. I like to encode as much information in a name as possible without making it ridiculously long, or completely nonsensical. Here's my thinking: the rule is named `Participant_Operation_Resource`. So `Grower_R_Grower` means, "Grant the Grower participant READ access to Grower instances".

Table 1. Participant access control — The type of access is in parentheses: R = READ, U = UPDATE, C = CREATE

Participant	Participants accessed	Assets accessed	Transactions accessed
Grower	Grower (R)	Shipment (RU), Contract (RU)	ShipmentPacked (C)
Shipper	Shipper (R)	Shipment (RU), Contract (RU)	ShipmentPickup, ShipmentLoaded (C)
Importer	Importer (R), Grower (RU)	Shipment (RU), Contract	ShipmentReceived (C)
TemperatureSensor	NONE	Shipment (RU), Contract (RU)	TemperatureReading (C)
GpsSensor	NONE	Shipment (RU), Contract (RU)	GpsReading (C)

The rules are codified below. Open `permissions.acl` (in the root of the network project), delete its current contents, and replace then with the listings that follow. The entire listing is very long, so I'll talk through it one section at a time.

The first two rules grant permission to the Hyperledger Composer system `Participant` and `NetworkAdmin` participants to access everything in the `org.hyperledger.composer.system` namespace, and every class in the network, respectively.

Copy all of the code below and paste it into `permissions.acl`.

```
/**
 * System and Network Admin access rules
 */
rule SystemACL {
  description:  "System ACL to permit all access"
  participant: "org.hyperledger.composer.system.Participant"
  operation: ALL
  resource: "org.hyperledger.composer.system.**"
  action: ALLOW
}

rule NetworkAdminUser {
    description: "Grant business network administrators full access to user resources"
    participant: "org.hyperledger.composer.system.NetworkAdmin"
    operation: ALL
    resource: "**"
    action: ALLOW
}
```

The next set of rules grants permission for the participants in the `iot-perishable-network` to access other participants (including themselves). This access varies as it makes sense for the business functions of the network. For example, the Grower can only access the Grower class, but since the Importer executes the `ShipmentReceived` transaction (which updates the Importer's and Grower's account balances), the Importer needs READ and UPDATE access to both Importer and Grower.

Copy all of the code below and paste it into `permissions.acl`.

```
/**
 * Rules for Participant registry access
 */
rule Grower_R_Grower {
```

```
                    description: "Grant Growers access to Grower resources"
                    participant: "org.acme.shipping.perishable.Grower"
                    operation: READ
                    resource: "org.acme.shipping.perishable.Grower"
                    action: ALLOW
                }

                rule Shipper_R_Shipper {
                    description: "Grant Shippers access to Shipper resources"
                    participant: "org.acme.shipping.perishable.Shipper"
                    operation: READ
                    resource: "org.acme.shipping.perishable.Shipper"
                    action: ALLOW
                }

                rule Importer_RU_Importer {
                    description: "Grant Importers access to Importer resources"
                    participant: "org.acme.shipping.perishable.Importer"
                    operation: READ,UPDATE
                    resource: "org.acme.shipping.perishable.Importer"
                    action: ALLOW
                }

                rule Importer_RU_Grower {
                    description: "Grant Importers access to Grower participant"
                    participant: "org.acme.shipping.perishable.Importer"
                    operation: READ,UPDATE
                    resource: "org.acme.shipping.perishable.Grower"
                    action: ALLOW
                }
```

Next are the rules for access to the network's assets. In this case, I want to grant access to the Shipment and Contract resources to all participants.

Copy all of the code below and paste it into `permissions.acl`.

```
                /**
                 * Rules for Asset registry access
                 */
                rule ALL_RU_Shipment {
                    description: "Grant All Participants in org.acme.shipping.perishable namespace READ/
UPDATE access to Shipment assets"
                    participant: "org.acme.shipping.perishable.*"
                    operation: READ,UPDATE
                    resource: "org.acme.shipping.perishable.Shipment"
                    action: ALLOW
                }

                rule ALL_RU_Contract {
                    description: "Grant All Participants in org.acme.shipping.perishable namespace READ/
UPDATE access to Contract assets"
                    participant: "org.acme.shipping.perishable.*"
                    operation: READ,UPDATE
                    resource: "org.acme.shipping.perishable.Contract"
                    action: ALLOW
                }
```

Next are the rules for invoking transactions (which require CREATE access). As you can see, these are on a case-by-case basis as it makes sense. For example, the Grower participant does not need to access the Shipper transactions (and vice versa).

Copy all of the code below and paste it into `permissions.acl`.

```
                /**
                 * Rules for Transaction invocations
                 */
                rule Grower_C_ShipmentPacked {
                    description: "Grant Growers access to invoke ShipmentPacked transaction"
                    participant: "org.acme.shipping.perishable.Grower"
                    operation: CREATE
                    resource: "org.acme.shipping.perishable.ShipmentPacked"
                    action: ALLOW
                }

                rule Shipper_C_ShipmentPickup {
                    description: "Grant Shippers access to invoke ShipmentPickup transaction"
                    participant: "org.acme.shipping.perishable.Shipper"
                    operation: CREATE
                    resource: "org.acme.shipping.perishable.ShipmentPickup"
                    action: ALLOW
                }

                rule Shipper_C_ShipmentLoaded {
                    description: "Grant Shippers access to invoke ShipmentLoaded transaction"
                    participant: "org.acme.shipping.perishable.Shipper"
                    operation: CREATE
                    resource: "org.acme.shipping.perishable.ShipmentLoaded"
                    action: ALLOW
                }

                rule GpsSensor_C_GpsReading {
                    description: "Grant IoT GPS Sensor devices full access to the appropriate transactions"
                    participant: "org.acme.shipping.perishable.GpsSensor"
                    operation: CREATE
                    resource: "org.acme.shipping.perishable.GpsReading"
                    action: ALLOW
                }

                rule TemperatureSensor_C_TemperatureReading {
                    description: "Grant IoT Temperature Sensor devices full access to the appropriate
transactions"
                    participant: "org.acme.shipping.perishable.TemperatureSensor"
                    operation: CREATE
                    resource: "org.acme.shipping.perishable.TemperatureReading"
                    action: ALLOW
                }

                rule Importer_C_ShipmentReceived {
                    description: "Grant Importers access to invoke the ShipmentReceived transaction"
                    participant: "org.acme.shipping.perishable.Importer"
                    operation: CREATE
                    resource: "org.acme.shipping.perishable.ShipmentReceived"
                    action: ALLOW
                }
```

The last rule says (in effect), "If there is some resource that we did not explicitly grant access to above, deny access to it." This is currently Hyperledger Composer's default behavior, but if that changes in a subsequent release, this rule makes sure the network behaves like this regardless of the default behavior.

Copy all of the code below and paste it into `permissions.acl`.

```
/**
 * Make sure all resources are locked down by default.
 * If permissions need to be granted to certain resources, that should happen
 * above this rule. Anything not explicitly specified gets locked down.
 */
rule Default {
    description: "Deny all participants access to all resources"
    participant: "ANY"
    operation: ALL
    resource: "org.acme.shipping.perishable.*"
    action: DENY
}
```

That's all for `permissions.acl`. Save the file, and get ready to unit test these permissions.

5d. Add feature tests

Wait, unit test permissions? That's right. All of the permissions in `permissions.acl` can be unit tested using Cucumber. I introduced you to Cucumber in Part 2, and now you'll use Cucumber to unit test the new transaction logic and the ACL rules you added to `permissions.acl` in the previous section.

The cool thing about Cucumber (pun intended) is that its syntax (Gherkin) is intuitive, so I don't need to explain a lot of the tests you're about to see.

Basic test scenarios

In VSCode, open `iot-perishable.feature`, then delete its contents, and replace them with the listings below as you work through this section.

First, you will set up the feature test as shown below. Notice on lines 14-16 that the test issues identities (that is, ID cards) for the specified participants. These identities will be used later in the unit test to test the security permissions you added to `permissions.acl` in the previous section.

Copy all of the code below and paste it into `iot-perishable.feature`.

```
Feature: Basic Test Scenarios
    Background:
        Given I have deployed the business network definition ..
        And I have added the following participants
        """
        [
        {"$class":"org.acme.shipping.perishable.Grower", "email":"grower@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"USA"}, "accountBalance":0},
        {"$class":"org.acme.shipping.perishable.Importer", "email":"supermarket@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"UK"}, "accountBalance":0}
        ]
        """
        And I have added the following participants of type
org.acme.shipping.perishable.TemperatureSensor
            | deviceId |
            | TEMP_001 |
        And I have issued the participant
org.acme.shipping.perishable.Grower#grower@email.com with the identity grower1
        And I have issued the participant
org.acme.shipping.perishable.Importer#supermarket@email.com with the identity importer1
        And I have issued the participant
org.acme.shipping.perishable.TemperatureSensor#TEMP_001 with the identity sensor_temp1
        And I have added the following asset of type org.acme.shipping.perishable.Contract
```

```
                        | contractId | grower           | shipper               | importer           |
arrivalDateTime  | unitPrice | minTemperature | maxTemperature | minPenaltyFactor | maxPenaltyFactor |
                        | CON_001    | grower@email.com | supermarket@email.com | supermarket@email.com |
10/26/2018 00:00 | 0.5       | 2              | 10             | 0.2              | 0.1              |
                    And I have added the following asset of type org.acme.shipping.perishable.Shipment
                        | shipmentId | type    | status     | unitCount | contract |
                        | SHIP_001   | BANANAS | IN_TRANSIT | 5000      | CON_001  |
                    And I submit the following transactions of type
org.acme.shipping.perishable.TemperatureReading
                        | shipment | centigrade |
                        | SHIP_001 | 4          |
                        | SHIP_001 | 5          |
                        | SHIP_001 | 10         |
                    When I use the identity importer1
```

No identity, no problem

You may be wondering how the Cucumber tests run if you remove the "When I use the identity xyz" (or noticed that when you do, the tests run just fine). That is because they are running as the default user, which has admin rights to your network. Keep in mind when running feature tests, you have to explicitly tell Cucumber to use a specific identity.

Notice line 28 in the listing above. I recommend you always put a "default identity" in your Cucumber tests to make sure your security permissions are always being tested (see the sidebar). You can always override this with a subsequent "When I use the identity xyz" in a specific scenario test if you like, as you'll see in this section.

The first scenario you'll run is when there are no temperature readings outside the agreed-upon range. Notice on line 2 that this test will be run as the identity **importer1**, which is an Importer.

Copy all of the code below and paste it into `iot-perishable.feature`.

```
                Scenario: When the temperature range is within the agreed-upon boundaries
                    When I use the identity importer1
                    And I submit the following transaction of type
org.acme.shipping.perishable.ShipmentReceived
                        | shipment |
                        | SHIP_001 |
                    Then I should have the following participants
                    """
                    [
                    {"$class":"org.acme.shipping.perishable.Grower", "email":"grower@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"USA"}, "accountBalance":2500},
                    {"$class":"org.acme.shipping.perishable.Importer", "email":"supermarket@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"UK"}, "accountBalance":-2500}
                    ]
                    """
```

The next scenario tests the case when there is a temperature reading that is below the agreed-upon threshold. The payout amount should be less (because of the low temperature penalty in the contract).

Notice that the test overrides the default identity, and is run first as **sensor_temp1**, which is a TemperatureSensor participant (line 2), then the current identity is switched to **importer1**, which is an Importer (line 6). If the test is not run this way, it will attempt to execute the `TemperatureReading` transaction as **importer1** (which was set in the Background section), which does not have permission to invoke that transaction. Then the identity must be switched back or

the `ShipmentReceived` transaction call will fail because a TemperatureSensor participant does not have permission to invoke that transaction.

Copy all of the code below and paste it into `iot-perishable.feature`.

```
                    Scenario: When the low/min temperature threshold is breached by 2 degrees C
                        When I use the identity sensor_temp1
                        And I submit the following transaction of type
org.acme.shipping.perishable.TemperatureReading
                            | shipment | centigrade |
                            | SHIP_001 | 0          |
                        Then I use the identity importer1
                        And I submit the following transaction of type
org.acme.shipping.perishable.ShipmentReceived
                            | shipment |
                            | SHIP_001 |
                        Then I should have the following participants
                        """
                        [
                        {"$class":"org.acme.shipping.perishable.Grower", "email":"grower@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"USA"}, "accountBalance":500},
                        {"$class":"org.acme.shipping.perishable.Importer", "email":"supermarket@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"UK"}, "accountBalance":-500}
                        ]
                        """
```

The next scenario tests the case when there is a temperature reading that is above the threshold. Like the previous scenario, this one must also be run using two separate identities.

Copy all of the code below and paste it into `iot-perishable.feature`.

```
                    Scenario: When the hi/max temperature threshold is breached by 2 degrees C
                        When I use the identity sensor_temp1
                        And I submit the following transaction of type
org.acme.shipping.perishable.TemperatureReading
                            | shipment | centigrade |
                            | SHIP_001 | 12         |
                        Then I use the identity importer1
                        When I submit the following transaction of type
org.acme.shipping.perishable.ShipmentReceived
                            | shipment |
                            | SHIP_001 |
                        Then I should have the following participants
                        """
                        [
                        {"$class":"org.acme.shipping.perishable.Grower", "email":"grower@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"USA"}, "accountBalance":1500},
                        {"$class":"org.acme.shipping.perishable.Importer", "email":"supermarket@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"UK"}, "accountBalance":-1500}
                        ]
                        """
```

Finally, the `ShipmentReceived` transaction must be run as an Importer (the only participant that has permission to invoke it), and when it does an event is emitted. The identity is set for this scenario in the Background section.

Copy all of the code below and paste it into `iot-perishable.feature`.

```
                    Scenario: When shipment is received a ShipmentReceivedEvent should be broadcast
                        When I submit the following transaction of type
org.acme.shipping.perishable.ShipmentReceived
                            | shipment |
                            | SHIP_001 |
                        Then I should have received the following event of type
org.acme.shipping.perishable.ShipmentReceivedEvent
                            | message                    | shipment |
                            | Shipment SHIP_001 received | SHIP_001 |
```

That's all for `iot-perishable.feature`. Go ahead and save the file.

Tests for IoT devices

The remaining feature tests work essentially the same as the ones you just saw, so I won't bore you with a repeated explanation. As you complete this section, just follow the directions, and paste the listings as a whole into the new `.feature` files.

In VSCode, click on the `features` directory and choose **New File**, and call the file `sensors.feature`. When the file opens in the editor window, copy and paste the following listing into the new empty file:

Copy all of the code below and paste it into `sensors.feature`.

```
                Feature: Tests related to IoT Devices

                    Background:
                        Given I have deployed the business network definition ..
                        And I have added the following participants
                        """
                        [
                        {"$class":"org.acme.shipping.perishable.Grower", "email":"grower@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"USA"}, "accountBalance":0},
                        {"$class":"org.acme.shipping.perishable.Shipper", "email":"shipper@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"Panama"}, "accountBalance":0},
                        {"$class":"org.acme.shipping.perishable.Importer", "email":"importer@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"UK"}, "accountBalance":0}
                        ]
                        """
                        And I have added the following participants of type
org.acme.shipping.perishable.TemperatureSensor
                            | deviceId |
                            | TEMP_001 |
                        And I have added the following participant of type
org.acme.shipping.perishable.GpsSensor
                            | deviceId |
                            | GPS_001  |
                        And I have issued the participant
org.acme.shipping.perishable.Grower#grower@email.com with the identity grower1
                        And I have issued the participant
org.acme.shipping.perishable.Shipper#shipper@email.com with the identity shipper1
                        And I have issued the participant
org.acme.shipping.perishable.TemperatureSensor#TEMP_001 with the identity sensor_temp1
                        And I have issued the participant org.acme.shipping.perishable.GpsSensor#GPS_001 with
the identity sensor_gps1
                        And I have added the following asset of type org.acme.shipping.perishable.Contract
                            | contractId | grower           | shipper               | importer              |
arrivalDateTime  | unitPrice | minTemperature | maxTemperature | minPenaltyFactor | maxPenaltyFactor |
                            | CON_001    | grower@email.com | shipper@email.com     | supermarket@email.com |
10/26/2018 00:00 | 0.5       | 2              | 10             | 0.2              | 0.1              |
                        And I have added the following asset of type org.acme.shipping.perishable.Shipment
                            | shipmentId | type    | status     | unitCount | contract |
```

```
                          | SHIP_001   | BANANAS | IN_TRANSIT | 5000      | CON_001  |

                  Scenario: Test TemperatureThresholdEvent is emitted when the max temperature threshold is
violated
                      When I use the identity sensor_temp1
                      When I submit the following transactions of type
org.acme.shipping.perishable.TemperatureReading
                          | shipment | centigrade |
                          | SHIP_001 | 11         |

                      Then I should have received the following event of type
org.acme.shipping.perishable.TemperatureThresholdEvent
                          | message                                                    | temperature | shipment |
                          | Temperature threshold violated! Emitting TemperatureEvent for shipment:
SHIP_001 | 11          | SHIP_001 |

                  Scenario: Test TemperatureThresholdEvent is emitted when the min temperature threshold is
violated
                      When I use the identity sensor_temp1
                      When I submit the following transactions of type
org.acme.shipping.perishable.TemperatureReading
                          | shipment | centigrade |
                          | SHIP_001 | 0          |

                      Then I should have received the following event of type
org.acme.shipping.perishable.TemperatureThresholdEvent
                          | message                                                    | temperature | shipment |
                          | Temperature threshold violated! Emitting TemperatureEvent for shipment:
SHIP_001 | 0           | SHIP_001 |

                  Scenario: Test ShipmentInPortEvent is emitted when GpsReading indicates arrival at
destination port
                      When I use the identity sensor_gps1
                      When I submit the following transaction of type
org.acme.shipping.perishable.GpsReading
                          | shipment | readingTime | readingDate | latitude | latitudeDir | longitude |
longitudeDir |
                          | SHIP_001 | 120000      | 20171025    | 40.6840  | N           | 74.0062   | W
           |

                      Then I should have received the following event of type
org.acme.shipping.perishable.ShipmentInPortEvent
                          | message                                                                    |
shipment |
                          | Shipment has reached the destination port of /LAT:40.6840N/LONG:74.0062W |
SHIP_001 |

                  Scenario: GpsSensor sensor_gps1 can invoke GpsReading transaction
                      When I use the identity sensor_gps1
                      When I submit the following transaction of type
org.acme.shipping.perishable.GpsReading
                          | shipment | readingTime | readingDate | latitude | latitudeDir | longitude |
longitudeDir |
                          | SHIP_001 | 120000      | 20171025    | 40.6840  | N           | 74.0062   | W
           |
                      Then I should have received the following event of type
org.acme.shipping.perishable.ShipmentInPortEvent
                          | message                                                                    |
shipment |
                          | Shipment has reached the destination port of /LAT:40.6840N/LONG:74.0062W |
SHIP_001 |

                  Scenario: Temperature Sensor cannot invoke GpsReading transaction
                      When I use the identity sensor_temp1
```

```
                        When I submit the following transaction of type
org.acme.shipping.perishable.GpsReading
                            | shipment | readingTime | readingDate | latitude | latitudeDir | longitude |
longitudeDir |
                            | SHIP_001 | 120000      | 20171025    | 40.6840  | N           | 74.0062   | W
           |
                        Then I should get an error matching /Participant .* does not have 'CREATE' access to
resource/

                    Scenario: Gps Sensor cannot invoke TemperatureReading transaction
                        When I use the identity sensor_gps1
                        When I submit the following transactions of type
org.acme.shipping.perishable.TemperatureReading
                            | shipment | centigrade |
                            | SHIP_001 | 11         |
                        Then I should get an error matching /Participant .* does not have 'CREATE' access to
resource/

                    Scenario: Grower cannot invoke TemperatureReading transaction
                        When I use the identity sensor_gps1
                        When I submit the following transactions of type
org.acme.shipping.perishable.TemperatureReading
                            | shipment | centigrade |
                            | SHIP_001 | 11         |
                        Then I should get an error matching /Participant .* does not have 'CREATE' access to
resource/
```

Now save this file.

Tests related to Growers

In VSCode, click the `features` directory, choose **New File**, and call the file `grower.feature`. When the file opens in the editor window, copy and paste the following listing into the new empty file:

Copy all of the code below and paste it into `grower.feature`.

```
                Feature: Tests related to Growers

                    Background:
                        Given I have deployed the business network definition ..
                        And I have added the following participants
                        """
                        [
                        {"$class":"org.acme.shipping.perishable.Grower", "email":"grower@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"USA"}, "accountBalance":0},
                        {"$class":"org.acme.shipping.perishable.Shipper", "email":"shipper@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"Paname"}, "accountBalance":0}
                        ]
                        """
                        And I have issued the participant
org.acme.shipping.perishable.Grower#grower@email.com with the identity grower1
                        And I have issued the participant
org.acme.shipping.perishable.Shipper#shipper@email.com with the identity shipper1
                        And I have added the following asset of type org.acme.shipping.perishable.Contract
                            | contractId | grower           | shipper               | importer              |
arrivalDateTime  | unitPrice | minTemperature | maxTemperature | minPenaltyFactor | maxPenaltyFactor |
                            | CON_001    | grower@email.com | shipper@email.com     | supermarket@email.com |
10/26/2018 00:00 | 0.5       | 2              | 10             | 0.2              | 0.1              |
                        And I have added the following asset of type org.acme.shipping.perishable.Shipment
                            | shipmentId | type    | status     | unitCount | contract |
                            | SHIP_001   | BANANAS | IN_TRANSIT | 5000      | CON_001  |
                        When I use the identity grower1

                    Scenario: grower1 can read Grower assets
```

```
                        Then I should have the following participants
                        """
                        [
                        {"$class":"org.acme.shipping.perishable.Grower", "email":"grower@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"USA"}, "accountBalance":0}
                        ]
                        """

                    Scenario: grower1 invokes the ShipmentPacked transaction
                        And I submit the following transaction of type
org.acme.shipping.perishable.ShipmentPacked
                            | shipment |
                            | SHIP_001 |
                        Then I should have received the following event of type
org.acme.shipping.perishable.ShipmentPackedEvent
                            | message                              | shipment |
                            | Shipment packed for shipment SHIP_001 | SHIP_001 |

                    Scenario: shipper1 cannot read Grower assets
                        When I use the identity shipper1
                        And I should have the following participants
                        """
                        [
                        {"$class":"org.acme.shipping.perishable.Grower", "email":"grower@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"USA"}, "accountBalance":0}
                        ]
                        """
                        Then I should get an error matching /Object with ID .* does not exist/

                    Scenario: shipper1 cannot invoke the ShipmentPacked transaction
                        When I use the identity shipper1
                        And I submit the following transaction of type
org.acme.shipping.perishable.ShipmentPacked
                            | shipment |
                            | SHIP_001 |
                        Then I should get an error matching /Participant .* does not have 'CREATE' access to
resource/
```

Now save this file. Notice the default identity in use (highlighted line). You need to set an identity for a specific scenario only if it needs a different identity.

In VSCode, click the `features` directory, choose **New File**, and call the file `shipper.feature`. When the file opens in the editor window, copy and paste the following listing into the new empty file:

Copy all of the code below and paste it into `shipper.feature`.

```
                Feature: Tests related to Shippers

                    Background:
                        Given I have deployed the business network definition ..
                        And I have added the following participants
                        """
                        [
                        {"$class":"org.acme.shipping.perishable.Grower", "email":"grower@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"USA"}, "accountBalance":0},
                        {"$class":"org.acme.shipping.perishable.Shipper", "email":"shipper@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"Paname"}, "accountBalance":0}
                        ]
                        """
                        And I have issued the participant
org.acme.shipping.perishable.Grower#grower@email.com with the identity grower1
                        And I have issued the participant
org.acme.shipping.perishable.Shipper#shipper@email.com with the identity shipper1
```

```
                        And I have added the following asset of type org.acme.shipping.perishable.Contract
                            | contractId | grower           | shipper               | importer              |
arrivalDateTime  | unitPrice | minTemperature | maxTemperature | minPenaltyFactor | maxPenaltyFactor |
                            | CON_001    | grower@email.com | shipper@email.com     | supermarket@email.com |
10/26/2018 00:00 | 0.5       | 2              | 10             | 0.2              | 0.1              |
                        And I have added the following asset of type org.acme.shipping.perishable.Shipment
                            | shipmentId | type    | status     | unitCount | contract |
                            | SHIP_001   | BANANAS | IN_TRANSIT | 5000      | CON_001  |
                        When I use the identity shipper1

                    Scenario: shipper1 can read Shipper assets
                        Then I should have the following participants
                        """
                        [
                        {"$class":"org.acme.shipping.perishable.Shipper", "email":"shipper@email.com",
"address":{"$class":"org.acme.shipping.perishable.Address", "country":"Paname"}, "accountBalance":0}
                        ]
                        """

                    Scenario: shipper1 invokes the ShipmentPickup transaction
                        And I submit the following transaction of type
org.acme.shipping.perishable.ShipmentPickup
                            | shipment |
                            | SHIP_001 |
                        Then I should have received the following event of type
org.acme.shipping.perishable.ShipmentPickupEvent
                            | message                                  | shipment |
                            | Shipment picked up for shipment SHIP_001 | SHIP_001 |

                    Scenario: shipper1 invokes the ShipmentLoaded transaction
                        And I submit the following transaction of type
org.acme.shipping.perishable.ShipmentLoaded
                            | shipment |
                            | SHIP_001 |
                        Then I should have received the following event of type
org.acme.shipping.perishable.ShipmentLoadedEvent
                            | message                                | shipment |
                            | Shipment loaded for shipment SHIP_001 | SHIP_001 |

                    Scenario: grower1 cannot invoke the ShipmentPickup transaction
                        When I use the identity grower1
                        And I submit the following transaction of type
org.acme.shipping.perishable.ShipmentPickup
                            | shipment |
                            | SHIP_001 |
                        Then I should get an error matching /Participant .* does not have 'CREATE' access to
resource/

                    Scenario: grower1 cannot invoke the ShipmentPickup transaction
                        When I use the identity grower1
                        And I submit the following transaction of type
org.acme.shipping.perishable.ShipmentLoaded
                            | shipment |
                            | SHIP_001 |
                        Then I should get an error matching /Participant .* does not have 'CREATE' access to
resource/
```

Now save this file.

That's it for the Cucumber feature tests.

5e. Build and unit test the network

Now it's time to build and unit test your network. Go to the command line (Ubuntu) or open a terminal window (MacOS) and enter this command:

```
npm install && npm test
```

You have run these commands before in Part 2 of this tutorial series. You'll see a lot of output, but when the unit tests have run, you should see this:

```
                # And I have issued the participant org.acme.shipping.perishable.Shipper#shipper@email.com
with the identity shipper1
                # And I have added the following asset of type org.acme.shipping.perishable.Contract
                    | contractId | grower           | shipper           | importer              |
arrivalDateTime  | unitPrice | minTemperature | maxTemperature | minPenaltyFactor | maxPenaltyFactor |
                    | CON_001    | grower@email.com | shipper@email.com | supermarket@email.com |
10/26/2018 00:00 | 0.5       | 2              | 10             | 0.2              | 0.1              |
                # And I have added the following asset of type org.acme.shipping.perishable.Shipment
                    | shipmentId | type    | status     | unitCount | contract |
                    | SHIP_001   | BANANAS | IN_TRANSIT | 5000      | CON_001  |
                # When I use the identity shipper1
                # When I use the identity grower1
                # And I submit the following transaction of type
org.acme.shipping.perishable.ShipmentLoaded
                    | shipment |
                    | SHIP_001 |
                # Then I should get an error matching /Participant .* does not have 'CREATE' access to
resource/

              20 scenarios (20 passed)
              229 steps (229 passed)
              0m09.497s
```

Lines 15-16 above show the 20 scenarios (consisting of 229 steps) have all run without error. You're now ready to deploy the network!

6. Deploy your changes to the network

If you've been following along with the tutorial, you have already deployed the `iot-perishable-network` to the local Hyperledger Fabric and instantiated the model. Unfortunately, you need to add a TemperatureSensor and GpsSensor participants to the participant registry. You can do this through the CLI, but I want to show you how to tear down and clean up your local Hyperledger Fabric, which is something you need to do from time to time.

When you tear down the Hyperledger Fabric using the procedure I'll outlined below, it removes the network and any crypto materials you've generated so far. Now, you never want to do this in production, for obvious reasons, but this is development, so it's something you're going to need to know how to do.

Navigate to your `$COMPOSER_ROOT` directory and enter the `./teardownFabric.sh` command. You'll see output like this:

```
              $ ./teardownFabric.sh
              Development only script for Hyperledger Fabric control
              Running 'teardownFabric.sh'
              .
```

```
                .
                # Shut down the Docker containers for the system tests.
                cd "${DIR}"/composer
                ARCH=$ARCH docker-compose -f docker-compose.yml kill && docker-compose -f docker-compose.yml
down
                Killing peer0.org1.example.com ... done
                Killing ca.org1.example.com    ... done
                Killing orderer.example.com    ... done
                Killing couchdb                ... done
                WARNING: The ARCH variable is not set. Defaulting to a blank string.
                Removing peer0.org1.example.com ... done
                Removing ca.org1.example.com    ... done
                Removing orderer.example.com    ... done
                Removing couchdb                ... done
                Removing network composer_default
                .
                .
                # Your system is now clean
```

Now wipe out the `~/.composer` directory, which will delete all the cards and crypto materials you've generated so far. Then run the `createPeerAdmin.sh` script again:

```
                $ rm -Rf ~/.composer
                $ ./createPeerAdminCard.sh
                Development only script for Hyperledger Fabric control
                Running 'createPeerAdminCard.sh'
                FABRIC_VERSION is unset, assuming hlfv1
                FABRIC_START_TIMEOUT is unset, assuming 15 (seconds)

                Using composer-cli at v0.15.0
                Successfully created business network card to /tmp/PeerAdmin@hlfv1.card

                Command succeeded

                Successfully imported business network card: PeerAdmin@hlfv1

                Command succeeded

                Hyperledger Composer PeerAdmin card has been imported
                The following Business Network Cards are available:


                ┌─────────────────┬────────────┬─────────┐
                │ CardName        │ UserId     │ Network │
                ├─────────────────┼────────────┼─────────┤
                │ PeerAdmin@hlfv1 │ PeerAdmin  │         │
                └─────────────────┴────────────┴─────────┘

                Issue composer card list --name <CardName>  to get details of the card

                Command succeeded
```

Next, start the local Hyperledger Fabric by running the `startFabric.sh` script, and then navigate to the `iot-perishable-network` directory, deploy the network, and import the `admin@iot-perishable-network` card again:

```
                $ cd $COMPOSER_ROOT/fabric-tools
                $./startFabric.sh
                .
                .
                $ composer network deploy -a dist/iot-perishable-network.bna -A admin -S adminpw -c
 PeerAdmin@hlfv1 -f networkadmin.card
                Deploying business network from archive: dist/iot-perishable-network.bna
                Business network definition:
                 Identifier: iot-perishable-network@0.1.12
                 Description: Shipping Perishable Goods Business Network

                # Deploying business network definition. This may take a minute...
                Successfully created business network card to networkadmin.card

                Command succeeded

                $ composer card import --file networkadmin.card
                Successfully imported business network card: admin@iot-perishable-network

                Command succeeded
```

Finally, you need to execute the `SetupDemo` transaction to instantiate the model, or there will be no participants for which to issue ID cards!

```
                $ composer transaction submit --card admin@iot-perishable-network -d '{"$class":
 "org.acme.shipping.perishable.SetupDemo"}'
                Transaction Submitted.

                Command succeeded
```

From now on, if you make changes to the network (so long as it does not involve modifications to any of the entries in the blockchain), you just need to update it:

```
                $ composer network update -a dist/iot-perishable-network.bna --card admin@iot-perishable-
network
                Deploying business network from archive: dist/iot-perishable-network.bna
                Business network definition:
                    Identifier: iot-perishable-network@0.1.12
                    Description: Shipping Perishable Goods Business Network

                # Updating business network definition. This may take a few seconds...
                Successfully created business network card to undefined

                Command succeeded
```

6a. Issue IDs

Now it's time to issue some IDs. You did this in Playground earlier. Now you will use the CLI to issue ID cards for the participants shown in Table 2.

Table 2. Participant Identities to be issued

Participant	Identity	ID card file name
Grower	grower1	grower1.card
Shipper	shipper1	shipper1.card
Importer	importer1	importer1.card
TemperatureSensor	sensor_temp1	sensor_temp1.card

GpsSensor	sensor_gps1	sensor_gps1.card

In the CLI section, you invoked the `SetupDemo` transaction, which instantiated the network. Remember, you cannot issue an ID for a participant that is not in the Participant Registry.

To issue an ID card for a participant, first execute the `composer identity issue` command, specifying the card file, and then import the card file into the local wallet. Use the `admin@iot-perishable-network` card to authenticate when you execute the `composer identity issue` command.

The general format for the command *for the `iot-perishable-network`* is this:

```
                composer identity issue --card admin@iot-perishable-network --file ID_CARD_FILE --newUserId
 IDENTITY --participantId 'resource:org.acme.shipping.perishable.PARTICIPANT#PARTICIPANT_ID'
```

Where:

`ID_CARD_FILE`— is the file name where the ID card will be stored (see Table 2).

`IDENTITY`— is the identity that is to be issued (see Table 2).

`PARTICIPANT_CLASS`— is the participant class (for example, Grower).

`PARTICIPANT_ID`— is the ID of the participant when it was instantiated in the registry (for example, farmer@email.com).

ID Card: Grower

```
                $ composer identity issue --card admin@iot-perishable-network --file grower1.card --newUserId
 grower1 --participantId 'resource:org.acme.shipping.perishable.Grower#farmer@email.com'

                Command succeeded

                $ composer card import --file grower1.card
                Successfully imported business network card: grower1@iot-perishable-network

                Command succeeded
```

ID Card: Shipper

```
                $ composer identity issue --card admin@iot-perishable-network --file shipper1.card --
newUserId shipper1 --participantId 'resource:org.acme.shipping.perishable.Shipper#shipper@email.com'

                Command succeeded

                Ix:~/HyperledgerComposer/developerWorks/iot-perishable-network sperry$ composer card import
 --file shipper1.card
                Successfully imported business network card: shipper1@iot-perishable-network

                Command succeeded
```

ID Card: Importer

```
                $ composer identity issue --card admin@iot-perishable-network --file importer1.card --
newUserId importer1 --participantId 'resource:org.acme.shipping.perishable.Importer#supermarket@email.com'

                Command succeeded

                Ix:~/HyperledgerComposer/developerWorks/iot-perishable-network sperry$ composer card import
 --file importer1.card
                Successfully imported business network card: importer1@iot-perishable-network

                Command succeeded
```

ID Card: TemperatureSensor

```
                $ composer identity issue --card admin@iot-perishable-
network --file sensor_temp1.card --newUserId sensor_temp1 --participantId
 'resource:org.acme.shipping.perishable.TemperatureSensor#SENSOR_TEMP001'

                Command succeeded

                $ composer card import --file sensor_temp1.card
                Successfully imported business network card: sensor_temp1@iot-perishable-network

                Command succeeded
```

ID Card: GpsSensor

```
                $ composer identity issue --card admin@iot-perishable-network --file sensor_gps1.card --
newUserId sensor_gps1 --participantId 'resource:org.acme.shipping.perishable.GpsSensor#SENSOR_GPS001'

                Command succeeded

                Ix:~/HyperledgerComposer/developerWorks/iot-perishable-network sperry$ composer card import
 --file sensor_gps1.card
                Successfully imported business network card: sensor_gps1@iot-perishable-network

                Command succeeded
```

Now that you've issued ID cards for all the participants in the network, and imported those cards into your local Hyperledger Fabric wallet, you can simulate the workflow of the IoT Perishable Goods business network from the command line using the CLI.

To see the cards you've issued, run the `composer card list` command:

```
                $ composer card list
                The following Business Network Cards are available:


                ┌──────────────────────────────────┬──────────────┬──────────────────────────┐
                │ CardName                         │ UserId       │ Network                  │
                ├──────────────────────────────────┼──────────────┼──────────────────────────┤
                │ admin@iot-perishable-network     │ admin        │ iot-perishable-network   │
                ├──────────────────────────────────┼──────────────┼──────────────────────────┤
                │ importer1@iot-perishable-network │ importer1    │ iot-perishable-network   │
                ├──────────────────────────────────┼──────────────┼──────────────────────────┤
                │ grower1@iot-perishable-network   │ grower1      │ iot-perishable-network   │
                ├──────────────────────────────────┼──────────────┼──────────────────────────┤
                │ sensor_temp1@iot-perishable-network │ sensor_temp1 │ iot-perishable-network │
                ├──────────────────────────────────┼──────────────┼──────────────────────────┤
                │ sensor_gps1@iot-perishable-network │ sensor_gps1 │ iot-perishable-network │
```

```
┌──────────────────────────────────────┬──────────────┬──────────────────────────┐
│ shipper1@iot-perishable-network      │ shipper1     │ iot-perishable-network   │
├──────────────────────────────────────┼──────────────┼──────────────────────────┤
│ PeerAdmin@hlfv1                      │ PeerAdmin    │                          │
└──────────────────────────────────────┴──────────────┴──────────────────────────┘

Issue composer card list --name <CardName>  to get details of the card

Command succeeded
```

6b. Submit transactions

The general format for the `composer transaction submit` command is:

```
composer transaction submit --card CARD_NAME -d 'DATA'
```

Where:

`CARD_NAME`— is the name of the card to use (see Table 2).

`DATA` is a JSON object containing the transaction data. You can copy the data format for any transaction from Playground, remove the newlines, and replace the specific data values (yes, that's how I came up with the data for the transactions you'll see in this section).

Submit transactions

In this section you'll submit the following transactions through the CLI:

- ShipmentPacked
- ShipmentPickup
- ShipmentLoaded
- TemperatureReading
- GpsReading
- ShipmentReceived

Each of these transactions corresponds to a step in the workflow of moving perishable goods from the Grower to the Importer.

Submit a ShipmentPacked transaction, using the **grower1** ID card to authenticate and authorize with the network:

```
$ composer transaction submit --card grower1@iot-perishable-
network -d '{"$class": "org.acme.shipping.perishable.ShipmentPacked", "shipment":
 "resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
Transaction Submitted.

Command succeeded
```

This transaction succeeded because the Grower participant has permission to execute the `ShipmentPacked` transaction. Just for fun, try to execute the transaction using the **shipper1** ID card:

```
                $ composer transaction submit --card shipper1@iot-perishable-
network -d '{"$class": "org.acme.shipping.perishable.ShipmentPacked", "shipment":
 "resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Error: Error trying invoke business network. Error: chaincode error (status: 500, message:
 Error: Participant 'org.acme.shipping.perishable.Shipper#shipper@email.com' does not have 'CREATE' access to
 resource
 'org.acme.shipping.perishable.ShipmentPacked#8db39906c73c0821021489834f9e5fb37f29bab4253840cb756038b77da4dc00')
                Command failed
```

The Shipper participant does not have access to the `ShipmentPacked` transaction, so the attempt fails, showing the permissions you coded earlier in the tutorial are working just as expected (of course, you knew that when you ran the unit test, but it's always nice to see it in action).

The second step in the workflow of getting a shipment of perishable goods from the Grower to the Importer is to pick up the packed shipment, which is handled by the Shipper:

```
                $ composer transaction submit --card shipper1@iot-perishable-
network -d '{"$class": "org.acme.shipping.perishable.ShipmentPickup", "shipment":
 "resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Transaction Submitted.

                Command succeeded
```

Once the shipment has been picked up, the Shipper loads it onto the container ship and executes the `ShipmentLoaded` transaction to record this in the ledger:

```
                $ composer transaction submit --card shipper1@iot-perishable-
network -d '{"$class": "org.acme.shipping.perishable.ShipmentLoaded", "shipment":
 "resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Transaction Submitted.

                Command succeeded
```

Along the way, the TemperatureSensor participant is taking readings inside the cargo container and recording them in the ledger. Simulate a few of those like this:

```
                $ composer transaction submit --card sensor_temp1@iot-perishable-network -d
 '{ "$class": "org.acme.shipping.perishable.TemperatureReading", "centigrade": 2, "shipment":
 "resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Transaction Submitted.

                Command succeeded
                $ composer transaction submit --card sensor_temp1@iot-perishable-network -d
 '{ "$class": "org.acme.shipping.perishable.TemperatureReading", "centigrade": 3, "shipment":
 "resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Transaction Submitted.

                Command succeeded

                Ix:~/HyperledgerComposer/developerWorks/iot-perishable-network sperry
$ composer transaction submit --card sensor_temp1@iot-perishable-network -d '{ "$class":
 "org.acme.shipping.perishable.TemperatureReading", "centigrade": 11, "shipment":
 "resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Transaction Submitted.

                Command succeeded
```

That was three `TemperatureReading` transactions, with the last one at 11C, which is 1 degree above the contract threshold, so the high temperature penalty will be in effect when the shipment is received.

Now submit a GPS transaction indicating the container ship has reached its destination, which is the Port of New York/New Jersey:

```
                $ composer transaction submit --card sensor_gps1@iot-perishable-network -d '{"$class":
 "org.acme.shipping.perishable.GpsReading", "readingTime": "2200", "readingDate": "20171118",
 "latitude": "40.6840", "latitudeDir": "N", "longitude": "74.0062", "longitudeDir": "W", "shipment":
 "resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Transaction Submitted.

                Command succeeded
```

The final step in the workflow is for the Importer to receive the shipment and invoke the `ShipmentReceived` transaction to record this in the blockchain:

```
                $ composer transaction submit --card importer1@iot-perishable-
network -d '{"$class": "org.acme.shipping.perishable.ShipmentReceived", "shipment":
 "resource:org.acme.shipping.perishable.Shipment#SHIP_001"}'
                Transaction Submitted.

                Command succeeded
```

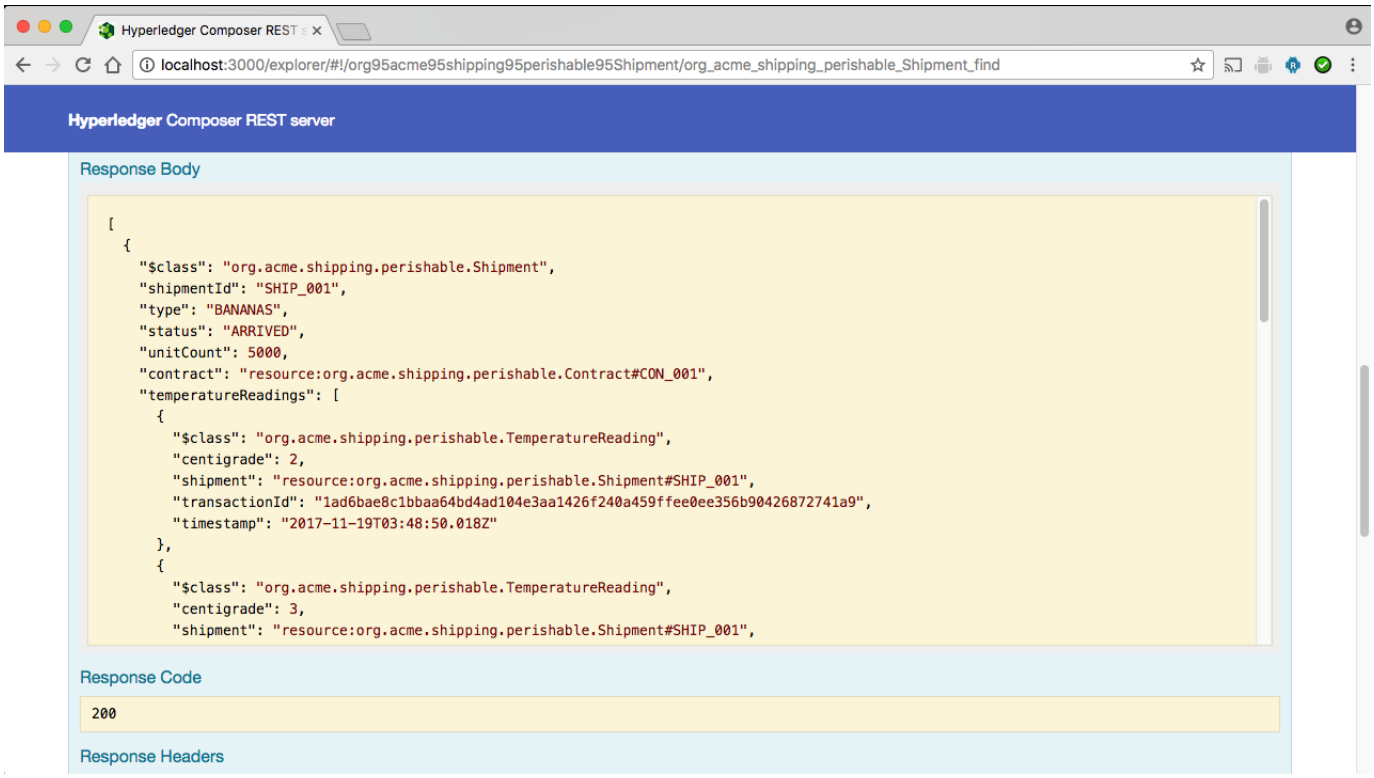
That doesn't seem very exciting, does it? Go ahead and fire up the REST server:

```
                $ composer-rest-server
                ? Enter the name of the business network card to use: admin@iot-perishable-network
                ? Specify if you want namespaces in the generated REST API: always use namespaces
                ? Specify if you want to enable authentication for the REST API using Passport: No
                ? Specify if you want to enable event publication over WebSockets: No
                ? Specify if you want to enable TLS security for the REST API: No
                Discovered types from business network definition
                Generating schemas for all types in business network definition ...
                Generated schemas for all types in business network definition
                Adding schemas for all types to Loopback ...
                Added schemas for all types to Loopback
                Web server listening at: http://localhost:3000
                Browse your REST API at http://localhost:3000/explorer
                To restart the REST server using the same options, issue the following command:
                   composer-rest-server -c admin@iot-perishable-network -n always

                Discovering types from business network definition ...
```

Now point your browser to `localhost:3000`, locate the Shipment asset, and execute the `/Get` method. You will see all of the transactions you submitted, recorded in the blockchain as part of the Shipment asset. Figure 5 shows what you should see in the response body of the `/Get` request.

Figure 5. Shipment asset showing transactions



There's quite a bit of data, so I could only show a little of it in Figure 5, but go ahead, fire up the REST server for yourself and scroll through the Shipment asset and see the results for yourself.

Video: Wrap-up demo

All the stuff from the previous section, plus Playground (where applicable), is wrapped up in this video:

To view this video, **Putting it all together** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Conclusion to Part 3

This tutorial covered a lot of ground. First, you installed, started, and then deployed the `iot-perishable-network` to a local instance of Hyperledger Fabric.

Then you installed and ran the REST interface, which you can use to access the network. I hope you had a chance to watch the video, where I walk through that in more detail. If not, make sure and check that out.

After that, you saw how access control works in Hyperledger Composer, including ACL rules and where they live in the network's source code.

Then you worked with the Composer Command Line Interface (CLI) to ping the running network, execute the `SetupDemo` and other transactions, and update the network as you make changes in development.

Finally, the *coup de grace*: you modified the `iot-perishable-network` to transform it into a more real-world blockchain application, write Cucumber feature tests, issue IDs for all participants, and execute every transaction through the CLI.

At this point, you should have everything you need to start developing your own blockchain applications using Hyperledger Composer. Good luck!

Related topics

- All parts of this series
- Enable REST authentication
- Events and WebSockets
- Securing the REST server
- Previous parts of this series
- Hyperledger Composer documentation
- Perishable Goods network (GitHub)
- More sample business network definitions (GitHub)
- IBM Blockchain Developer Center
- Blockchain courses for developers