IBM

developerWorks®

Hyperledger Composer basics, Part 1: Model and test your blockchain network

Start blockchaining the fast, easy way in the Hyperledger Composer Playground

J Steven Perry

April 13, 2018

Hyperledger Composer is a tool for quickly building blockchain business networks and prototyping blockchain applications. This tutorial gets you started using it.

This tutorial gets you started developing a blockchain network. I'll introduce you to **Hyperledger Composer** and its user interface, **Hyperledger Composer Playground**, where you can model and test your network with nothing more than Docker and your web browser.

In **Part 2**, you'll learn how to refine and deploy your blockchain network, and in **Part 3**, you'll see how to install Hyperledger Fabric on your computer, deploy your business network to your local instance, and interact with the sample network blockchain application.

Hyperledger Composer is now integrated into the IBM Blockchain Platform! The new Starter Plan (free beta!) can help you kick-start your blockchain network.

Prerequisites

To follow along with this tutorial, you should have the following software installed on your computer:

- Docker Engine 17.03 or greater
- Web browser

What is Hyperledger Composer?

One of the Hyperledger projects hosted by The Linux Foundation, Hyperledger Composer is a set of tools that makes building blockchain applications easier, and consists of:

What does CTO stand for?

It doesn't correlate to anything nowadays. The files were given the CTO extension because the original project was called "Concerto," according to project committer Simon Stone's explanation on Rocket.Chat. (You'll need your Linux Foundation ID to log in.)

- A **modeling language called CTO** (an homage to the original project name, *Concerto*)
- A **user interface called Hyperledger Composer Playground** for rapid configuration, deployment, and testing of a business network
- **Command-Line Interface (CLI) tools** for integrating business networks modeled using Hyperledger Composer with a running instance of the Hyperledger Fabric blockchain network

In this part of the tutorial series, I'm going to introduce you to CTO modeling language and Hyperledger Composer Playground. I'll save the CLI for Part 2 of this series, where I will show you all about the CLI (and lots more).

Hyperledger Composer modeling language (CTO)

Hyperledger Composer has its own modeling language (called CTO) used to model your business network. In Part 1, I'll show you how to use CTO to model the sample Perishable Goods network. This sample business network demonstrates how growers, shippers, and importers define contracts for the price of perishable goods, based on temperature readings received for shipping containers.

Hyperledger Composer Playground

Hyperledger Composer Playground is a browser-based interface that you can use to model your business network: what items of value (assets) are exchanged, who participates (participants) in their exchange, how access is secured (access control), what business logic (transactions) is involved in the process, and more.

Hyperledger Composer Playground (Playground from now on) uses your browser's local storage to simulate the blockchain network's state storage, which means you don't need to run a real validating peer network to use Playground.

What you'll do in this tutorial:

- Learn about business network concepts
- Run Playground on your computer using Docker
- Get familiar with the modeling language
- Use the modeling language to model, or describe, the business network
- Test the business network

In Part 2, I'll show you how to install the full set of Hyperledger Composer development utilities, how to work with more advanced features of the CTO language (including events), how to unit test your JavaScript smart contracts, and how to interact with a real Hyperledger Fabric blockchain network using the Command-Line Interface (CLI).

In Part 3, I'll give you an advanced look at Composer: how to generate a REST interface and GUI using Yeoman, and how to deploy your blockchain network application to the IBM Cloud.

Business network concepts

At a high level, a business network is a group of entities who work together to accomplish certain goals. In order to achieve these goals, there must be agreement among the members of the business network regarding:

- The goods and services that are exchanged
- How the exchange is to take place (including business rules governing payment and penalties)
- Which members within the group are allowed to participate, and when

In the next section, I'll introduce you to common business network terminology. First, I want to tell you about the business problem that your first blockchain business network will solve: shipment of perishable goods. More importantly, how the Internet of Things, temperature sensors, and the Cloud are used to ensure the perishables are shipped in ideal conditions (and what happens if not).

The Perishable Goods network

What's IoT got to do with it?

Seems like the Internet of Things is everywhere. The Perishable Goods network gets temperature readings from an array of IoT sensors that transmit the temperature within cargo containers to the IBM Cloud, where that sensor data is stored in the blockchain.

Learn more about the convergence of IoT and Blockchain.

The Internet of Things (IoT)-based Perishable Goods network is a business network involving:

- Perishable items such as bananas, pears, and coffee
- Business partners such as growers, shippers, and importers
- Shipments of perishable goods
- Agreements between business parties that stipulate conditions of the agreements
- Acknowledgement of receipt of goods and services

I'll be using this business network as the example throughout this tutorial series. As you progress through the series, you will notice the application growing in complexity as I introduce more Hyperledger Composer concepts, and show how they relate to blockchain and the IBM Cloud.

Assets

An *asset* is anything of value that can be exchanged between parties in a business agreement. That means an asset can be pretty much anything. Examples include:

- A boat
- A quantity of stock
- A house
- A crate of bananas
- A shipment of bananas
- A contract for shipment of 1000 crates of bananas for X price based on conditions {X, Y, Z}

You name it. If it has perceived value, and can be exchanged between parties, it's an asset. In the Perishable Goods network, assets include the perishable goods themselves, shipments of those goods, and the contracts that govern the activities performed during the exchange.

Participants

A *participant* is a member of a business network. For the Perishable Goods network, this includes the growers who produce the perishable goods, the shippers who transport them from the growers to the ports, and the importers who take delivery of the goods at the ports. Obviously, this model is oversimplified, but it should give you a sense of how real-world applications are modeled using business network terminology.

Access control

In a business network, not every participant has access to everything. For example, a grower does not have access to the contract between the shipper and the importer. *Access control* is used to limit who has access to what (and under what conditions).

Transactions

When an asset is "touched," that interaction potentially affects the state of the blockchain ledger. The interaction is modeled in Hyperledger Composer as a *transaction*.

Examples of transactions in the Perishable Goods network include:

- An IoT sensor in the shipping container records a temperature reading
- An Importer receives a shipment of perishable goods
- An IoT GPS sensor records the shipping container's current location

Transactions are the business logic (or smart contracts) of the system.

Events

An *event* is a notification produced by the blockchain application, and consumed by an external entity (such as an application) in publish/subscribe fashion.

While the blockchain ledger is updated as assets are exchanged in the system, this is an internal (system) process. However, there are times when an external entity needs to be notified that the state of the ledger has changed, or that maybe something else of note has occurred (or not occurred but should have) in the system. Your blockchain application could use an event in this case.

Examples of events in the Perishable Goods network might include:

- A temperature reading has exceeded an upper or lower boundary X number of times (this might indicate a problem with the shipping container itself, for example).
- A shipment has been received.
- A shipment has arrived at the port (an IoT GPS sensor could report this event, for example).

In the next section, I'll show you how to model the Perishable Goods network of assets, participants, and transactions. I'll touch on access control, and save events for Part 2 of this series.

Run Playground on your computer using Docker

What is Playground?

Playground is an environment that lets you quickly build and test blockchain business networks. It doesn't need a running blockchain network, and so it reduces the complexity of getting a business network defined, validated, and tested.

Playground runs in a Docker container, and can be installed to your computer in either of two modes:

- With a Hyperledger Fabric validating peer network
- In browser-only mode

With a Hyperledger Fabric validating peer network

This mode installs Playground with the Hyperledger Fabric validating peer network, and includes the Docker container for Playground, along with all the other Docker containers to run a Hyperledger Fabric validating peer network.

As you get deeper into the tutorial in Parts 2 and 3, you will need a full Hyperledger Fabric validating peer network. For now, it is overkill, as none of the activities you'll perform in Part 1 require this.

Browser-only mode

Using browser-only mode, you can model and test the business network using a mock blockchain ledger that resides in your browser's local storage.

This is the approach I'll use for Part 1.

Run Playground

From a terminal window (Mac/Linux) or command prompt (Windows), execute this command:

```
                docker run --name composer-playground --publish 8080:8080 hyperledger/composer-playground
```

This starts Docker interactively. I like to run Playground like this so I can see what gets logged to STDOUT:

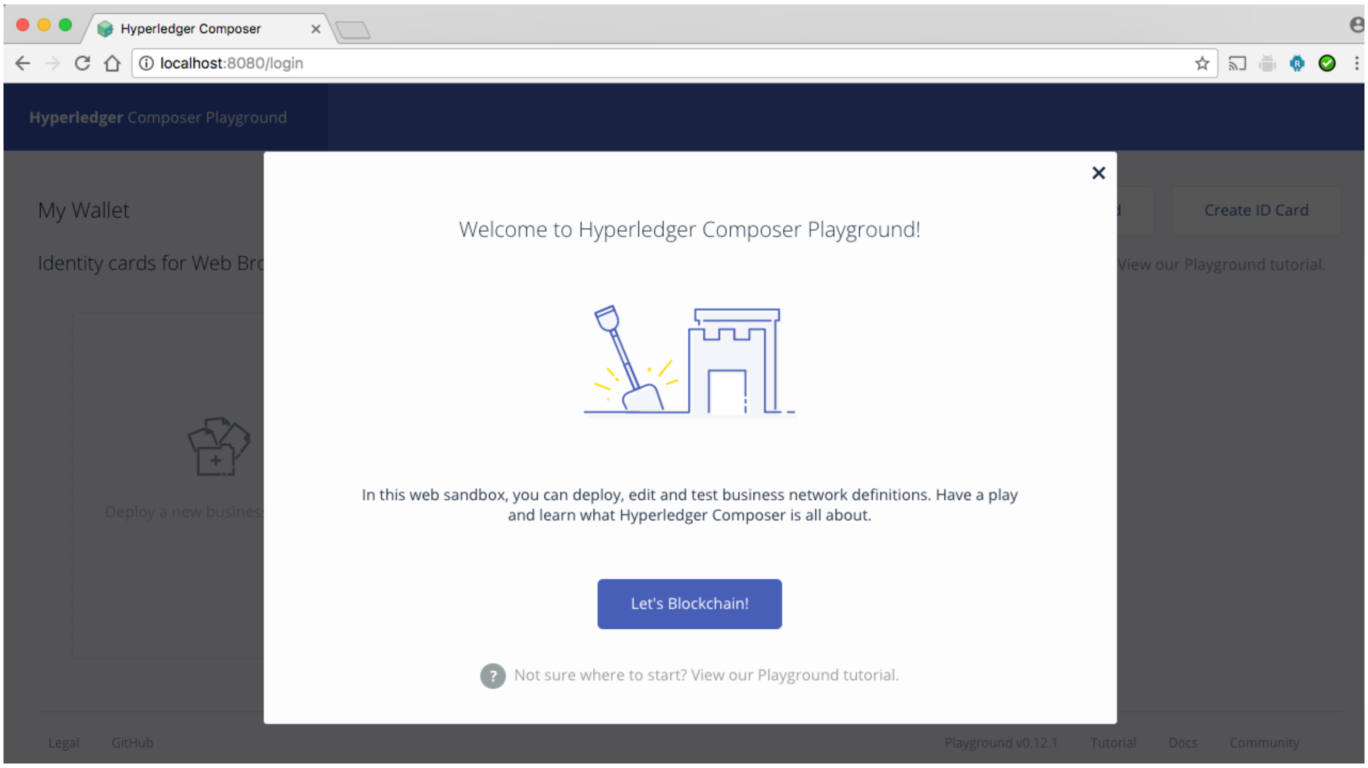
```
$ docker run --name composer-playground --publish 8080:8080 hyperledger/composer-playground
0|composer | PlaygroundAPI            :createServer()           > 8080
0|composer | ConnectionProfileManager :constructor()            Created a new ConnectionProfileManager
 {"fs":{"constants":
{"O_RDONLY":0,"O_WRONLY":1,"O_RDWR":2,"S_IFMT":61440,"S_IFREG":32768,"S_IFDIR":16384,"S_IFCHR":8192,"S_IFBLK":24576,"
0|composer | PlaygroundAPI            :createServer()           Playground API started on port 8080
0|composer | PlaygroundAPI            :createServer()           <
0|composer | Composer                 :main()                   >
```

If you want to run Playground detached, just add `--detach`:

```
                docker run --name composer-playground --publish 8080:8080 --detach hyperledger/composer-
playground
```

Now open a browser, and go to `http://localhost:8080` and you will see a screen like Figure 1:

Figure 1. Playground welcome screen


Hyperledger Composer
localhost:8080/login
Hyperledger Composer Playground
My Wallet
Identity cards for Web Bro
Create ID Card
View our Playground tutorial.
Deploy a new busines
Welcome to Hyperledger Composer Playground!
Let's Blockchain!
Not sure where to start? View our Playground tutorial.
Legal
GitHub
Playground v0.12.1
Tutorial
Docs
Community

In this web sandbox, you can deploy, edit and test business network definitions. Have a play and learn what Hyperledger Composer is all about.

When you're finished running Playground, `ctrl+c` to kill the container in interactive mode. If running in detached mode, execute this command:

```
                docker stop composer-playground
```

Now clean up the Docker container (or Docker will complain if you try and run it again):

```
                    docker rm --force composer-playground
```

Video: Running Playground, tour of UI

In this video, I'll show you how to run Playground using Docker, and give you a tour of the Playground UI.

To view this video, **Hyperledger Composer Playground tour** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

The Hyperledger Composer modeling language

Before you can use Hyperledger Composer to test and deploy a blockchain business network, you have to build a model of it. Hyperledger Composer has its own modeling language for doing this.

Oh, great, another language to learn, right? Fortunately, the CTO modeling language is simple (and intuitive if you have worked with object-oriented concepts).

The CTO modeling language is tightly focused (for modeling business networks) with just a few keywords, so there is not a lot to learn. The model for your business network resides in a file that has a `.cto` file extension, and contains definitions for the following elements:

- Namespace
- Resources
- Imports from other namespaces, as required

If your model is very large, you can have multiple `.cto` model files, as necessary. Every `.cto` model file must include a single namespace and at least one resource definition.

Namespace

A namespace creates a boundary within whose scope names are considered to be unique. Every `.cto` model file requires a namespace, which means every name within a `.cto` model file must be unique.

You're already familiar with the concept of a namespace, even if you didn't realize it. File systems use the directory as the namespace, so that two files within the same directory cannot have the same name. However, two files in different directories are allowed to have the same name, since they are in different *namespaces*.

To sum up: Two resources in a CTO model file (the namespace boundary) cannot have the same name.

Resource

A resource is one of the following:

- asset - a business network Asset
- participant - a business network Participant
- transaction - business logic

- event - a notification of something interesting happening in the system
- enumerated type - a set of named values
- concept - any object you want to model that is not one of the other types

Each resource type corresponds to its model type of the same name (for example, `asset` is used to model an Asset, `participant` models a Participant, etc.).

A resource has the following properties:

- A namespace in which it is defined
- A name, which must be unique within the namespace
    - If the resource is an `asset` or `participant`, it must have an identifier field indicated by `identified by` followed by the name of the field
- (optional) Its parent type (super type), if applicable, indicated by `extends` followed by the name of the parent type
- (optional) The `abstract` keyword, if you do not want the resource to be instantiated, but rather used as a super type for other resources of that type

In the Perishable Goods network, a Shipment asset is modeled like this:

```
/**
  * A business network for shipping perishable goods
  * The cargo is temperature controlled and contracts
  * can be negociated based on the temperature
  * readings received for the cargo
  */
namespace org.acme.shipping.perishable
.
.
/**
 * A shipment being tracked as an asset on the ledger
 */
asset Shipment identified by shipmentId {
  o String shipmentId
  o ProductType type
  o ShipmentStatus status
  o Long unitCount
  o TemperatureReading[] temperatureReadings optional
  --> Contract contract
}
```

Let's look at the example above, and I'll point out a few things.

The namespace is `org.acme.shipping.perishable`.

The `asset` keyword indicates `Shipment` is an asset. The property (indicated by lowercase "o") `shipmentId` is of type `String`, and uniquely identifies a `Shipment` (as indicated by `identified by`).

Its properties each have a type, which can be a fundamental type (like `String`) or an enumerated type (like `ProductType`), or transaction (like an array of `TemperatureReading`).

The reference (indicated by `-->`) to `Contract` is called a *relationship*, and is one-way (unidirectional.)

Enumerated types

When the set of values a particular property can have is known, you should model that as an enumerated type. This makes it easier to constrain the values, which makes validation simpler.

An enumerated type is declared like this:

```
/**
  * The type of perishable product being shipped
  */
 enum ProductType {
   o BANANAS
   o APPLES
   o PEARS
   o PEACHES
   o COFFEE
 }
```

Concepts

When an entity exists in your business model, but is not an asset, participant, transaction, or event, you model the entity as a *concept.*

```
/**
 * A concept for a simple street address
 */
concept Address {
  o String city optional
  o String country
  o String street optional
  o String zip optional
}
```

The concept of Address is an important one in a business model, but doesn't exactly fit one of the other categories, which is why you model it as a `concept`.

Imports

Imports are used in a `.cto` model file to indicate a relationship between an entity in that model file and an entity from another model file.

I won't talk about imports in this tutorial, since the model you'll be working with is so small. The concept is similar to `import` in the Java™ language, and `#include` in C++.

CTO reference

Once you learn the basic syntax (which in most cases is obvious by context), the modeling language should be easy to pick up. If you have worked with object-oriented concepts, it will be easier still.

If you want to learn more, I recommend you read the full documentation for the CTO modeling language.

Model the business network

In the previous video, I showed you how to create an empty business model using Playground. In this section, you're going to model the Perishable Goods network in Playground (don't worry, you'll have help from the built-in `perishable-network` template).

Delete localhost browser storage

If you see a welcome screen like Figure 1, you're good to go, and you can skip right to "Create a new model in Playground."

In browser-only mode, Hyperledger Composer only allows working with one model at a time. If you have another model loaded, you may need to delete your browser's local storage before loading another model.

Playground should be able to replace the current model with a new one. However, if you run into errors, you can start "from scratch" by deleting your browser's local storage. The procedure varies from browser to browser.

In Chrome, for example, under `Settings > Advanced > Content Settings > Cookies > All cookies and site data > localhost`, click the trashcan icon to remove local storage. If you're using a different browser, follow the instructions specific to that browser, and delete all local storage.

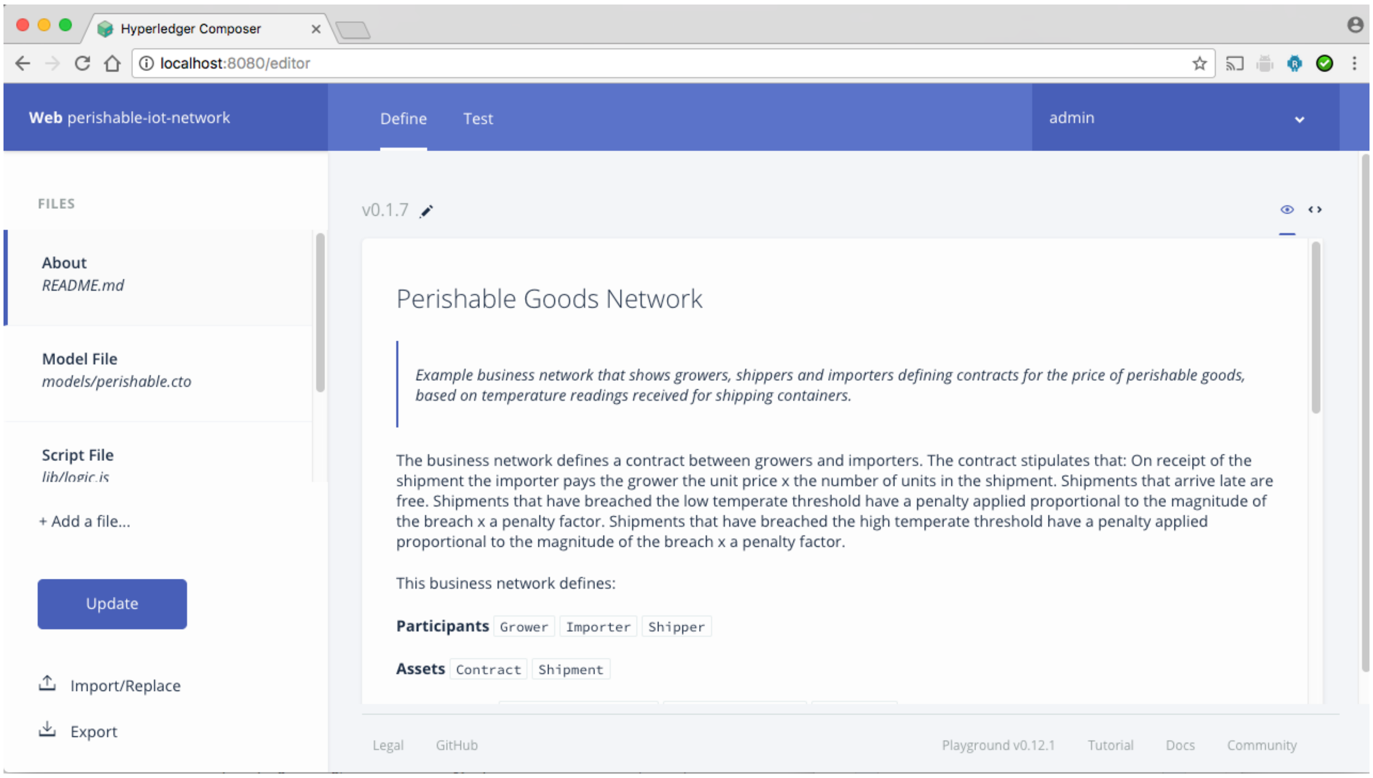
Create a new model in Playground

In the previous video, I showed you how to create a new, empty business network in Playground, as well as the basics of how to get around in Playground, so I won't go over those again here. If you didn't get a chance to watch the video, you should check it out before trying to follow along.

Click the **Let's Blockchain** button to get started (see Figure 1). Next, create a new business network from the perishable-network template. Call it `perishable-iot-network` and click **Deploy**.

On the Admin ID card, click the **Connect now** link. You should see something like Figure 2.

Figure 2. The Perishable Goods network



Perishable Goods Network

> *Example business network that shows growers, shippers and importers defining contracts for the price of perishable goods, based on temperature readings received for shipping containers.*

The business network defines a contract between growers and importers. The contract stipulates that: On receipt of the shipment the importer pays the grower the unit price x the number of units in the shipment. Shipments that arrive late are free. Shipments that have breached the low temperate threshold have a penalty applied proportional to the magnitude of the breach x a penalty factor. Shipments that have breached the high temperate threshold have a penalty applied proportional to the magnitude of the breach x a penalty factor.

This business network defines:

Participants `Grower` `Importer` `Shipper`

Assets `Contract` `Shipment`

Under FILES, notice the following:

- `README.md` - this Markdown file provides a quick overview of the Perishable Goods network
- `models/perishable.cto` - contains the business model
- `lib/logic.js` - contains the business logic (smart contract) code, including transaction implementation

When you select one of the files under FILES, it opens in the editor window on the right side. Go ahead and open the model file (perishable.cto), which contains the model.

A `Grower` asset is modeled like this:

```
/**
 * An abstract participant type in this business network
 */
abstract participant Business identified by email {
  o String email
  o Address address
  o Double accountBalance
}

/**
 * A Grower is a type of participant in the network
 */
participant Grower extends Business {
}
```

And a `Shipper` like this:

```
/**
 * A Shipper is a type of participant in the network
 */
participant Shipper extends Business {
}
```

A `Contract` asset like this:

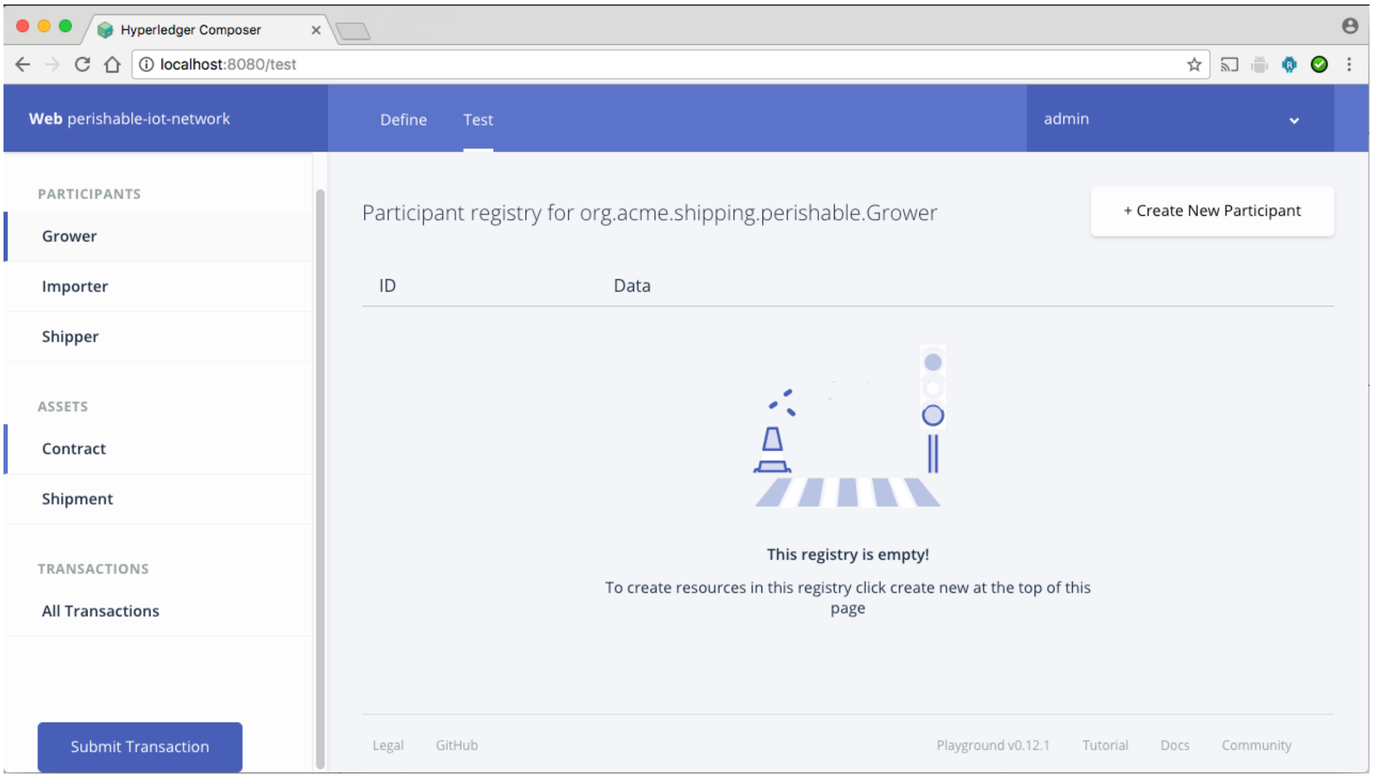
```
/**
 * Defines a contract between a Grower and an Importer to ship using
 * a Shipper, paying a set unit price. The unit price is multiplied by
 * a penality factor proportional to the deviation from the min and max
 * negociated temperatures for the shipment.
 */
asset Contract identified by contractId {
  o String contractId
  --> Grower grower
  --> Shipper shipper
  --> Importer importer
  o DateTime arrivalDateTime
  o Double unitPrice
  o Double minTemperature
  o Double maxTemperature
  o Double minPenaltyFactor
  o Double maxPenaltyFactor
}
```

I encourage you to get acquainted with the model, and how the various resources look inside the editor. Do the same for `lib/logic.js` and get familiar with the JavaScript code as well.

Instantiate the model

Click the **Test** tab at the top of the screen and you will see something like Figure 3.

Figure 3. perishable-network - Test Tab


Hyperledger Composer
localhost:8080/test
Web perishable-iot-network
Define
Test
admin
PARTICIPANTS
Grower
Importer
Shipper
ASSETS
Contract
Shipment
TRANSACTIONS
All Transactions
Submit Transaction
Participant registry for org.acme.shipping.perishable.Grower
+ Create New Participant

ID	Data

This registry is empty!
To create resources in this registry click create new at the top of this page

Notice the Assets and Participants from the model appear on the left side of the screen, but in the center of the screen is a message saying the registry is empty. What's going on?

As I showed you in the video, when the business network is first created, both the Asset and Participant registries are empty. You need to create Asset and Participant *instances*, and those instances will reside in the registry.

In the next section, I'll show you how to instantiate and test the model.

Test the business network

Models are great at acting as a sort of blueprint for the application you're building, but a model of a thing is not much good unless it results (at some point) in an *actual* thing. For example, a set of blueprints for a skyscraper is critical if you're going to construct a building, but not much good unless at some point they are actually used to build (instantiate) an actual building!

Back to the business model, which, if it is going to be useful, needs to be *instantiated*. But what does that mean for a blockchain application?

The Asset and Participant registries

Earlier you saw the model for a `Grower` participant, `Shipment` asset, and so forth. Now it's time to instantiate those resources, and their instances will live in their respective registries. So asset instances go in the asset registry, and participant instances go in the participant registry.

The `perishable-network` model includes a transaction implemented as a JavaScript function in the `lib/logic.js` module called `setupDemo()` that you can use to instantiate the model and create entries in the Asset and Participant registries. It is provided as a way to get the business network from the template up and running more quickly than if you entered the model by hand.

I won't show the `setupDemo()` function here, but I would like to point out that it does three things:

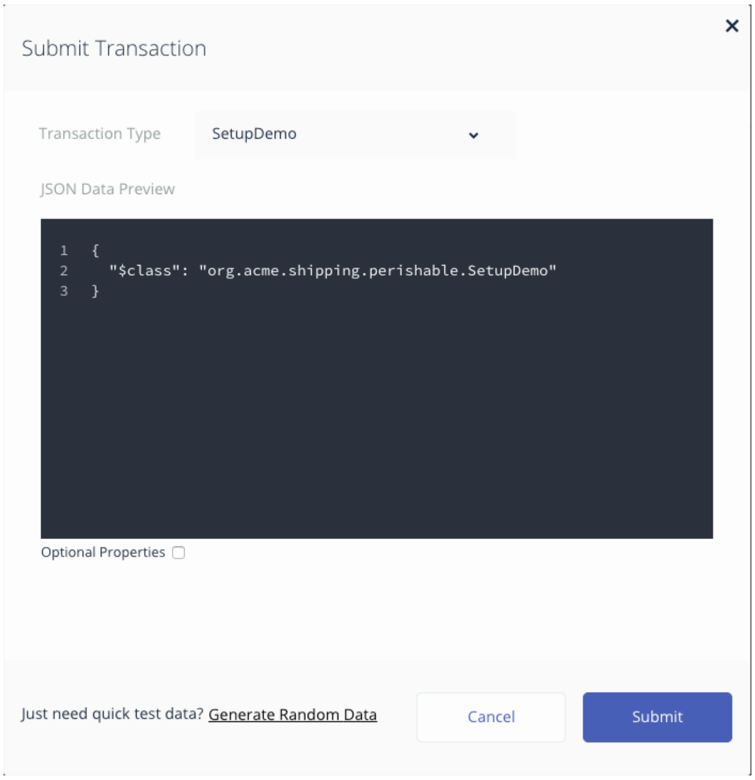
1. Creates instances of all the assets and participants from the model
2. Sets property values on those instances
3. Stores the instances in their respective registries

I encourage you to open the `lib/logic.js` file in the editor and look at it for yourself.

Instantiate the model

To execute the SetupDemo transaction, click the **Submit Transaction** button, and a modal dialog appears that should resemble Figure 4.

Figure 4. SubmitTransaction - SetupDemo


Submit Transaction

Transaction Type: SetupDemo

JSON Data Preview

```
{
  "$class": "org.acme.shipping.perishable.SetupDemo"
}
```

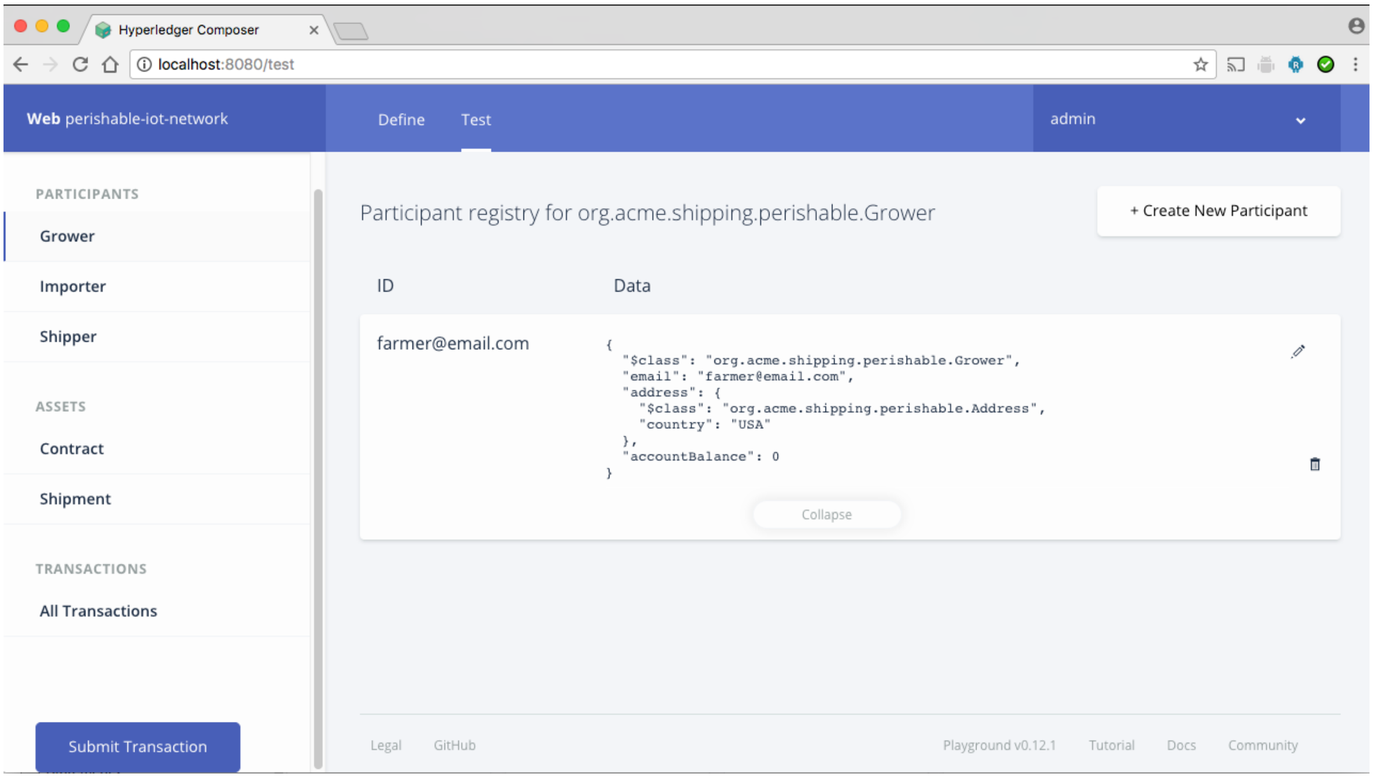
Optional Properties ☐

Just need quick test data? Generate Random Data

Make sure that SetupDemo appears in the **Transaction Type** drop-down, then click the **Submit** button. When the transaction executes successfully, you will see a brief notification message telling you so.

Select **Grower** in the ASSETS pane on the left side, and all of its instances appear on the right side (Figure 5). The same is true for the other resources (go ahead, try it!).

Figure 5. A Grower asset instance



farmer@email.com

```
{
  "$class": "org.acme.shipping.perishable.Grower",
  "email": "farmer@email.com",
  "address": {
    "$class": "org.acme.shipping.perishable.Address",
    "country": "USA"
  },
  "accountBalance": 0
}
```

Now that you have a business network defined, and assets and participants in their respective registries, you can test your network.

What about transactions?

So far in this section, I've talked about assets and participants, but what about the transactions in the business model? Where do they show up?

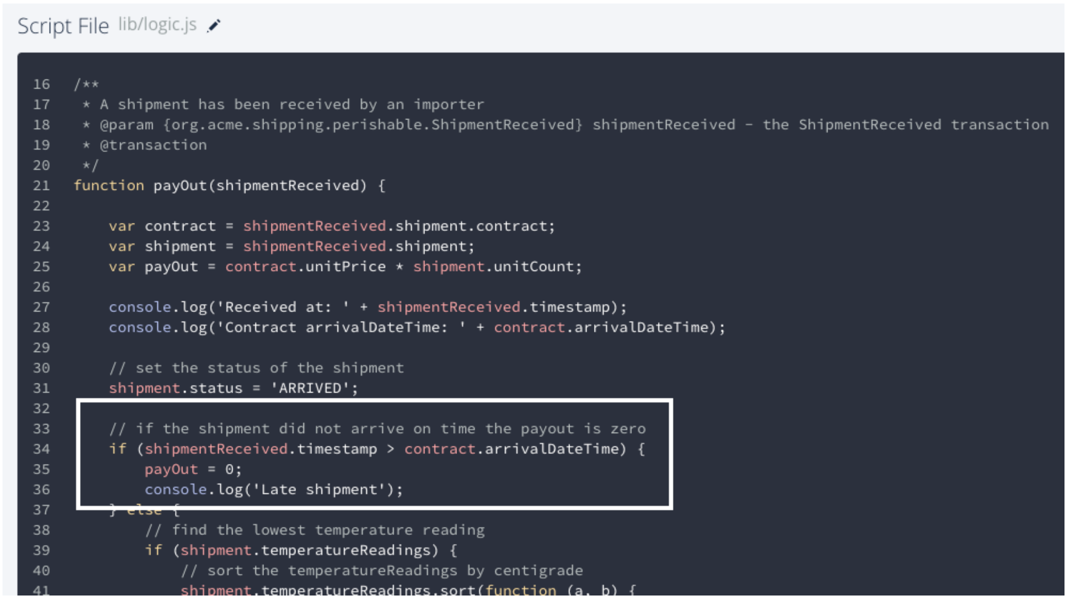
To answer the first question: the transactions represent the business logic of the application (smart contracts, or chaincode). The business logic enforced by the smart contract generated by `setupDemo()` stipulates the following conditions:

1. The temperature inside the shipping container is to be 6 degrees Celsius at all times. If the temperature of the shipment falls outside of the agreed-upon range (+/- 5 degrees), then the price of the shipment ($0.50/unit) is reduced by $0.20/unit for every degree below and $0.10 for every degree above.
2. If the shipment arrives late, the Grower receives no payment for the shipment.

Okay, so where do the transactions show up? A transaction is not *instantiated* per se, but rather shows up as JavaScript code in `lib/logic.js`.

Figure 6 shows the smart contract code (from lib/logic.js) that enforces the second stipulation (zero payout for a late shipment) from the contract.

Figure 6. Smart contract: Late shipment penalty


Script File lib/logic.js

```
/**
 * A shipment has been received by an importer
 * @param {org.acme.shipping.perishable.ShipmentReceived} shipmentReceived - the ShipmentReceived transaction
 * @transaction
 */
function payOut(shipmentReceived) {

    var contract = shipmentReceived.shipment.contract;
    var shipment = shipmentReceived.shipment;
    var payOut = contract.unitPrice * shipment.unitCount;

    console.log('Received at: ' + shipmentReceived.timestamp);
    console.log('Contract arrivalDateTime: ' + contract.arrivalDateTime);

    // set the status of the shipment
    shipment.status = 'ARRIVED';

    // if the shipment did not arrive on time the payout is zero
    if (shipmentReceived.timestamp > contract.arrivalDateTime) {
        payOut = 0;
        console.log('Late shipment');
    } else {
        // find the lowest temperature reading
        if (shipment.temperatureReadings) {
            // sort the temperatureReadings by centigrade
            shipment.temperatureReadings.sort(function (a, b) {
```

What about access control?

Access control is governed by a file in the model called `permissions.acl`. In the Business network concepts section, I introduced access control as one of the primary concepts. In Parts 2 and 3, I will cover more of this extremely important topic and how you control access to your blockchain application.

Let's have a look at `permissions.acl` in the **Define** tab of your model. The perishable-network template included an Access Control List (ACL) file that looks like this:

```
/**
 * 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
}
```

The ACL file contains rules that let you control access to the resources in your blockchain application. Suffice it to say, Hyperledger Composer has you covered when it comes to security, and I'll show you all about it later in this tutorial series.

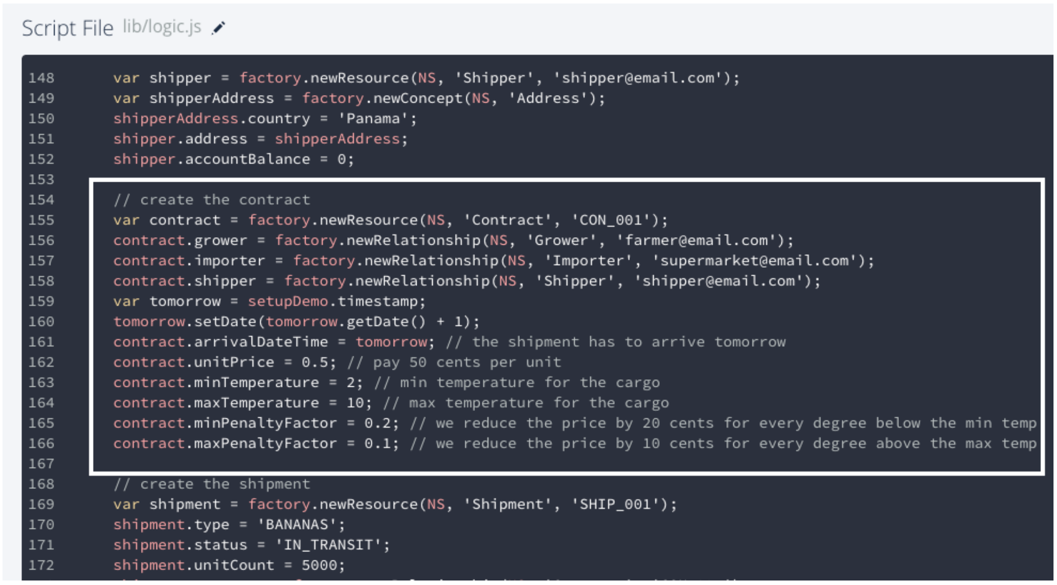
For now, the access control rules defined above pretty much grant wide-open access, which is fine for now since you're just starting out with Hyperledger Composer.

Test the model

Now that the model is instantiated, it's time to test it, and that means running code! In this case, that means running JavaScript code from `lib/logic.js`.

Before you set up the test, let's review the contract (which was instantiated in the `setupDemo()` function) to see the terms. Figure 7 shows the JavaScript code used to instantiate the contract asset:

Figure 7. Smart contract: terms and conditions



```
    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 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');
    contract.shipper = factory.newRelationship(NS, 'Shipper', 'shipper@email.com');
    var tomorrow = setupDemo.timestamp;
    tomorrow.setDate(tomorrow.getDate() + 1);
    contract.arrivalDateTime = tomorrow; // the shipment has to arrive tomorrow
    contract.unitPrice = 0.5; // pay 50 cents per unit
    contract.minTemperature = 2; // min temperature for the cargo
    contract.maxTemperature = 10; // max temperature for the cargo
    contract.minPenaltyFactor = 0.2; // we reduce the price by 20 cents for every degree below the min temp
    contract.maxPenaltyFactor = 0.1; // we reduce the price by 10 cents for every degree above the max temp

    // create the shipment
    var shipment = factory.newResource(NS, 'Shipment', 'SHIP_001');
    shipment.type = 'BANANAS';
    shipment.status = 'IN_TRANSIT';
    shipment.unitCount = 5000;
```

So now you're ready to test the smart contract. With Playground running in your browser, click the Test tab at the top of the Playground UI.

Let's test the following scenario:

1. IoT temperature sensors provide the following readings (numbers are degrees Celsius):
    1. 5
    2. 7
    3. 1
    4. 4
2. The Shipment is received.

Let's look at the components of this scenario one at a time, starting with the temperature sensor data.

In a real-world application, the IoT temperature sensors could, say, send this data to the IBM Cloud, where the smart contract code would be invoked against the blockchain to record these transactions.

In Playground, the blockchain is maintained in your browser's local storage, but the transaction code that executes is the same regardless of where the blockchain resides (which makes Playground a perfect place to test, right?).

Check out the `temperatureReading()` function inside `lib/logic.js`:

```
                /**
                 * A temperature reading has been received for a shipment
                 * @param {org.acme.shipping.perishable.TemperatureReading} temperatureReading - the
TemperatureReading transaction
                 * @transaction
                 */
                function temperatureReading(temperatureReading) {

                    var shipment = temperatureReading.shipment;

                    console.log('Adding temperature ' + temperatureReading.centigrade + ' to shipment ' +
shipment.$identifier);

                    if (shipment.temperatureReadings) {
                        shipment.temperatureReadings.push(temperatureReading);
                    } else {
                        shipment.temperatureReadings = [temperatureReading];
                    }

                    return getAssetRegistry('org.acme.shipping.perishable.Shipment')
                        .then(function (shipmentRegistry) {
                            // add the temp reading to the shipment
                            return shipmentRegistry.update(shipment);
                        });
                }
```

In the real-world app, when an IoT sensor in the shipping container wants to send a reading, it sends it to the cloud (through the cargo ship's network), where it is picked up by, say, a serverless function running in OpenWhisk, which invokes the `temperatureReading()` function.

To simulate this in Playground:

1. Click the **Submit Transaction** button (just like you did to invoke the `setupDemo()` function).
2. Make sure **TemperatureReading** appears in the **Transaction Type** drop-down.
3. Change the "centigrade" reading from 0 to 5 (the first reading we want to send) in the **JSON Data Preview** window.
4. Make sure the shipment ID is set to SHIP_001.
5. Click **Submit**.
6. Repeat for the remaining three readings.

To receive the shipment in a real-world application, an app running on the importer's hand-held device could indicate to an application running in the IBM Cloud (or a serverless function running in OpenWhisk) that the shipment was received, which would then calculate payment to be remitted to the grower.

To simulate receipt of the shipment in Playground, run the **ShipmentReceived** transaction in Playground, make sure to provide the shipment's ID, and click **Submit**.

Video: Testing the model, seeing transactions

We've covered a lot of ground, I know. But don't worry: in this next video, I'll show you how to test the model, and see the results of all the transactions you run in the browser's JavaScript console.

To view this video, **Hyperledger Composer Playground - Create and test a blockchain network** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Working with models

So you have this great blockchain business model. Now what?

Export the model

Let's suppose you want to make the model available to other members of your team, or need to export the model to deploy it to a real blockchain network in production (I'll show you how to do this in Part 2).

Playground has an export feature that allows you to create a Business Network Archive (BNA) file that you can share. To export the model, in the Define tab, click the **Export** link at the bottom left of the screen, and Playground generates a Business Network Archive (BNA) file and then downloads it to your computer.

Import a model

Let's suppose you have a Business Model Archive (BNA) file — maybe it was given to you by a team member — and you want to play around with it in Playground.

How do you get a BNA file into Playground? First, make sure Playground is up and running. Then under the Define tab, choose **Import/Replace**. On the Import/Replace Network screen, click **Drop here to upload or browse**, use the browse dialog to locate and select the BNA file you want to import, and choose **Open**. Then click **Import** and confirm that you want to replace the current model with the one you want to import.

Note: If you run into trouble importing the BNA file, you may need to clear your browser's local storage, then try the import again. See the section "Delete localhost browser storage" earlier in this tutorial for more information.

Conclusion to Part 1

Part 1 of this tutorial series showed how to run Playground using Docker and introduced to the basics of the Hyperledger Composer modeling language (CTO). You also saw what a simple business network model looks like in Playground by creating a new business network that was built on the perishable-network template provided by the Playground Docker image.

Then the tutorial showed you how to test that model by storing various temperature reading transactions to the blockchain, and how the smart contract used those transactions to resolve the terms of the contract when the Shipment was received.

Finally, I showed you how to share models with colleagues and others by exporting and importing models out of and into Playground, respectively.

Now head on over to **Part 2**, where I'll show you how to build on and refine the existing business network from this part, as well as unit-test it and deploy it to the IBM Cloud.

This content was originally published on IBM developerWorks on October 5, 2017.

Related topics

- All parts of this series
- Hyperledger Composer documentation
- Perishable Goods network (GitHub)
- More sample business network definitions (GitHub)
- Get going quickly with blockchain using Hyperledger Composer
- Building a blockchain PoC in ten minutes using Hyperledger Composer
- Integrate device data with smart contracts in IBM Blockchain
- IBM Blockchain Developer Center
- IBM Blockchain Platform
- IBM Blockchain Platform for developers
- Blockchain courses for developers