

IBM Cloud

Introduction to Blockchain

Building your first Blockchain Network



Lab Guide





Notices and Disclaimers

© Copyright IBM Corporation 2017.

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. This information is based on current IBM product plans and strategy, which are subject to change by IBM without notice. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

This document is current as of the initial date of publication and may be changed by IBM at any time. Not all offerings are available in every country in which IBM operates.

IBM, the IBM logo and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at ibm.com/legal/copytrade.shtml

Other company, product and service names may be trademarks or service marks of others

Document Revision History

Rev #	File Name	Date
1.0	Initial Course Development	02/08/2018
1.1		

Prepared & Revised by:

Chris Tyler – ctyler@us.ibm.com

Table of Contents

Lab Environment Overview.....	5
Module 1: Building your Model.....	6
Module 1: Lab Workflow Overview	7
Module 1: Lab Instructions.....	8
Module 1: Lab Summary.....	36

Lab Environment Overview

Installed Software and Tools

Software	Link
Hyperledger Composer Playground on the IBM Cloud	https://composer-playground.mybluemix.net/test

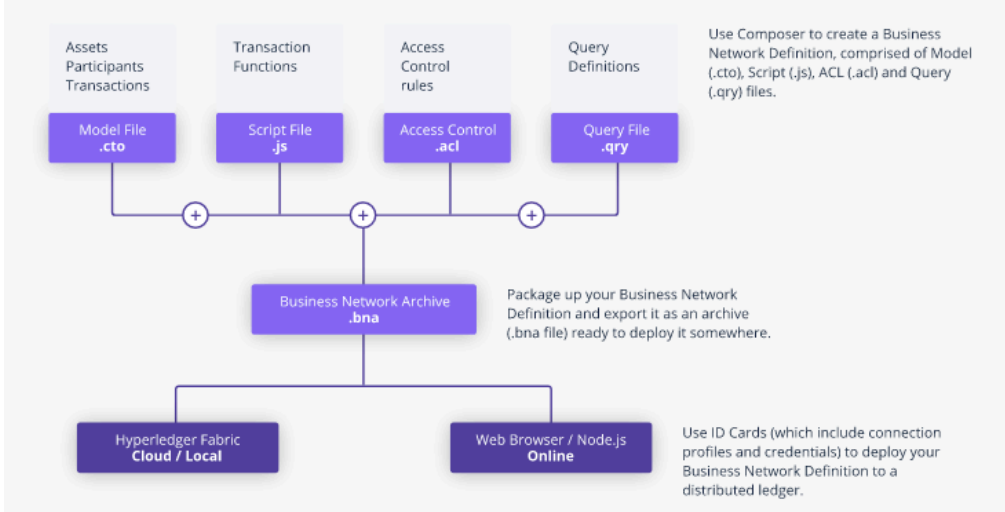
Module 1: Building your Model

Purpose:	<p>This lab introduces the subject of using Hyperledger Composer to build a network model. After completing the lab, you should be able to:</p> <ul style="list-style-type: none">• Learn the concepts of a Hyperledger Business Network• Understand the Hyperledger Composer modeling language• Use the modeling language to model the business network• Test the business network
Tasks:	<p>Tasks you will complete in this lab exercise include:</p> <ul style="list-style-type: none">• Learn Business Network Concepts• Access the Hyperledger Composer Playground• Learn the basics of the Hyperledger Composer modeling language• Model a Business Network• Test the Business Network• Export your Business Network for deployment

Module 1: Lab Workflow Overview

- 1 • Hyperledger Composer Concepts
- 2 • Access the Playground
- 3 • Modeling language basics
- 4 • Build the network
- 5 • Test the network
- 6 • Export the network

Module 1: Lab Instructions

Step	Action
1	<p><u>Understand the Hyperledger Composer concepts</u></p> <p>a. Hyperledger Composer Components</p>  <p>Hyperledger Composer is a web-based UI for building and deploying Hyperledger Fabric Business Networks. A Business Network is the definition of the parties and events involved in tracking transactions which can include trading of assets. There are multiple components to a Business Network which we will discuss below and create in some cases. In the Business Network definition, there are multiple files which are used to define and execute the Network.</p> <ul style="list-style-type: none"> • Model File – defines the parties involved, the assets that are to be tracked and the transactions that can occur • Script File – defines what happens when transactions are submitted • Access Control – defines who can access what components of the Business Network • Query File – defines the types of queries that will be accessible to retrieve information from the Business Network • Business Network Archive – the packaging of the Business Network for deployment into another environment • Business Network Card – a file containing the identity and credentials of an authorized user of the Business Network <p>b. Identities – Users who are authorized to participate in the Business Network and are allowed or required to endorse transactions need an Identity which is handled by a Business Network Card. Some participants in</p>

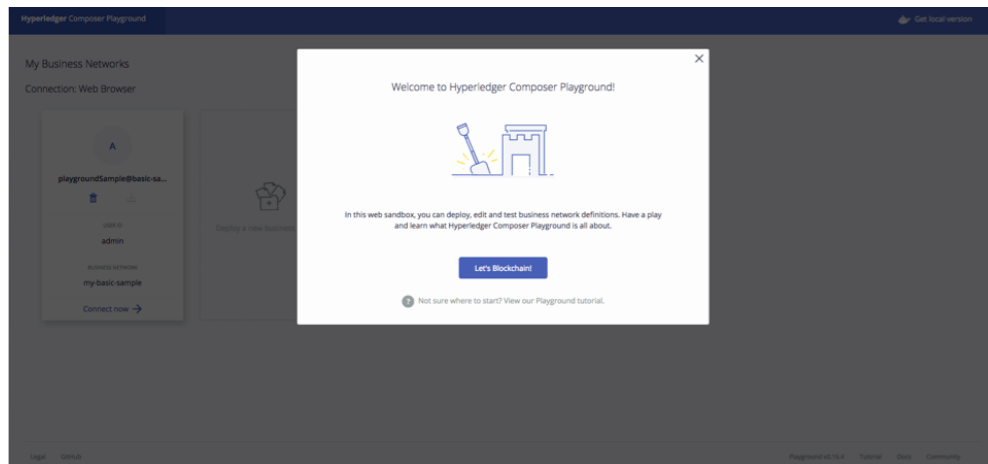
Step	Action
	<p>the Business Network may not be endorsing transactions and are simply involved in a transaction. For example, I may be buying a car that is tracked on a Blockchain, but I am not necessarily the endorser of a transaction and may have no need to ever interact directly with the Blockchain. Therefore, I would not need an Identity in the Business Network.</p> <p>However, the Sales Manager and Manufacturing Plant would be involved in endorsing the sale of the vehicle and the request for manufacturing as well as the delivery to the customer. Those participants directly involved with endorsing that those things occurred, would need identities.</p> <p>c. Model File</p> <p>The model file uses the Hyperledger Composer modeling language to define components such as:</p> <ul style="list-style-type: none"> • Assets – the entities which can be owned, traded, shipped, serviced, etc. • Participants – the people or things that will act on assets • Transactions – the actions that can occur between participants and their assets <p>All of the attributes for each type of entity are defined in the model file as well as the relationships between them</p> <p>d. Script File</p> <p>This is the code that is executed when transactions are submitted for endorsement. In the Blockchain world, this is commonly referred to as “Chaincode” and also is thought of as the vehicle for providing “Smart Contracts”. Before a transaction is endorsed, it must pass the validation of the Chaincode and then it can be added to the Blockchain. This Chaincode can be as sophisticated or as simple as you would like.</p> <p>e. Events</p> <p>These can be thought of as alerts or notifications when certain situations arise. Users can subscribe to listen for events and then take action.</p> <p>f. Access Control</p> <p>This defines which Identities can Create, Read, Update or Delete</p>

Step	Action
	<p>components of the Business Network. Of course, the immutability of the Blockchain prohibits any Updates or Deletes of Transactions or Events, but Assets and Participants may come and go.</p> <p>g. Query Definitions</p> <p>This defines what data elements will be exposed with a query.</p>

2 Access the Hyperledger Composer Playground

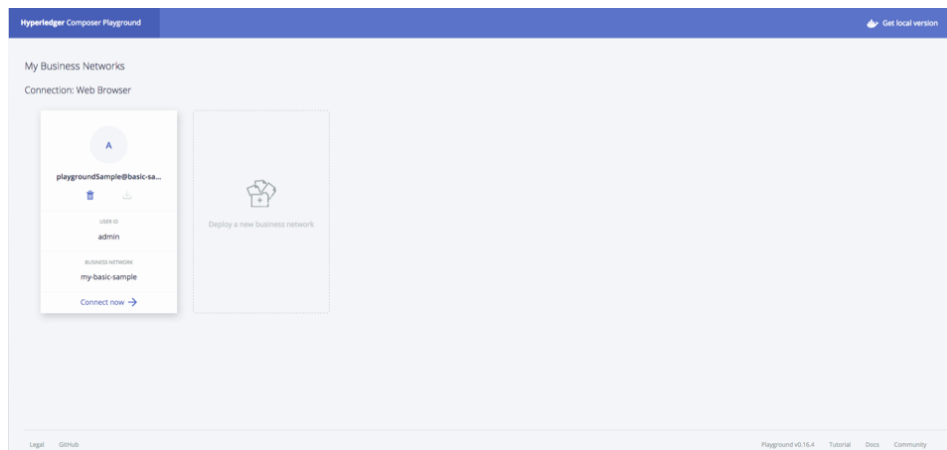
a. Access the web UI on the IBM Cloud

Open your web browser, preferably Chrome or Firefox, and navigate to the Hyperledger Composer Playground on IBM Cloud using the URL <https://composer-playground.mybluemix.net/test>. You will be presented with this:



If you receive a message stating that you have an older version, only if you've used this in the past, click the Clear State button.

On this page, click the **"Let's Blockchain!"** button. After that, you will see a mostly black canvas that looks like this:

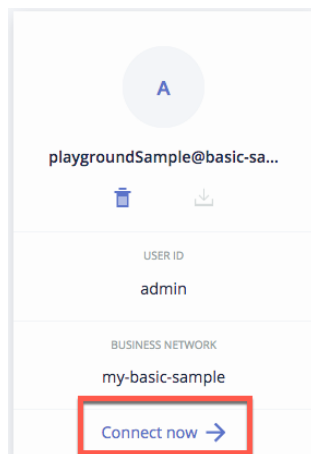


This is the main Composer page where you can see all available Business Networks. To start with, you have one, the my-basic-sample Business

Network. We will use this to navigate and explore the Composer user interface.

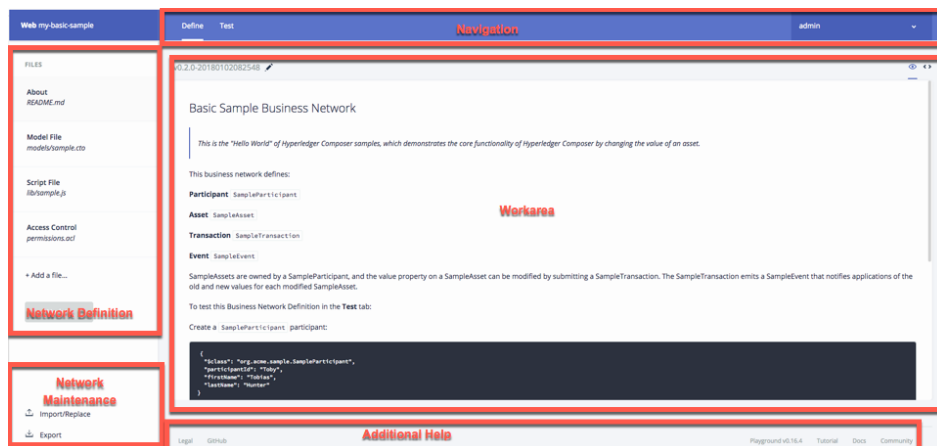
b. Explore the UI

Click on the [Connect now](#) -> link at the bottom of the my-basic-sample network card.



This will open the Composer UI and display the README.md file which gives an overview of what is in the Basic Sample Business Network. In short, there is one type of Participant called SampleParticipant, one type of Asset called SampleAsset, one type of Transaction called SampleTransaction and one Event type called SampleEvent.

The Hyperledger Composer is broken down into 5 main areas.



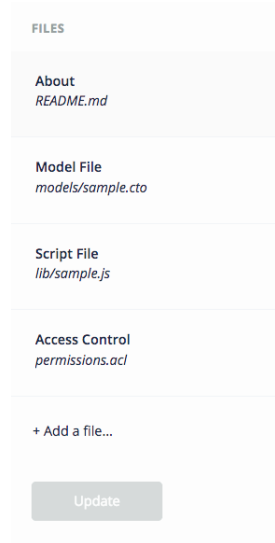
- Navigation – Move between defining and testing and jump between

business networks and identities

- Network Definition – The various files that are used to define the Business Network
- Network Maintenance – Tools to export and import the Business Network
- Workarea – The main area for editing files and testing the network
- Additional Help – Links to additional resources such as the GitHub repository, Documentation, Tutorial and Help from the Community

c. Review the Basic Network Sample in the Playground

Let's start out by navigating around the files within this Business Network. On the left hand side of the page is the Files section. You should see this:



You are currently looking at the README.md file which gives the overview of the Business Network.

Click on the Model File models/sample.cto. In the workarea, you should now see an editor with the models/sample.cto file.

```
Model File models/sample.cto
1 /**
2  * Sample business network definition.
3  */
4 namespace org.acme.sample
5
6 asset SampleAsset identified by assetId {
7   o String assetId
8   --> SampleParticipant owner
9   o String value
10 }
11
12 participant SampleParticipant identified by participantId {
13   o String participantId
14   o String firstName
15   o String lastName
16 }
17
18 transaction SampleTransaction {
19   --> SampleAsset asset
20   o String newValue
21 }
22
23 event SampleEvent {
24   --> SampleAsset asset
25   o String oldValue
26   o String newValue
27 }
28
```

We will cover the modeling language basics in the next section. In this model file, you'll see the following defined:

- Asset – SampleAsset with an Identifier attribute and a value attribute. Additionally, the SampleAsset can be related to a SampleParticipant who is the owner of this SampleAsset.
- Participant – SampleParticipant with an identifier attribute and a firstName and lastName attribute.
- Transaction – SampleTransaction with a newValue attribute which will be used to update the related SampleAsset's value attribute.
- Event – SampleEvent which shows the old and new value attributes for the related SampleAsset

So, you can see that you define the players in the network along with the attributes which describe each as well as the relationships among them.

Now, click on the Script File lib/sample.js and you will see the code that is executed when transactions are submitted. Remember, we have only one type of Transaction, SampleTransaction.

```
Script File lib/sample.js
1
2 /**
3  * Sample transaction processor function.
4  * @param {org.acme.sample.SampleTransaction} tx The sample transaction instance.
5  */
6
7 function sampleTransaction(tx) {
8
9   // Save the old value of the asset.
10   var oldValue = tx.asset.value;
11
12   // Update the asset with the new value.
13   tx.asset.value = tx.newValue;
14
15   // Get the asset registry for the asset.
16   return getAssetRegistry('org.acme.sample.SampleAsset')
17     .then(function (assetRegistry) {
18
19       // Update the asset in the asset registry.
20       return assetRegistry.update(tx.asset);
21
22     })
23     .then(function () {
24
25       // Emit an event for the modified asset.
26       var event = getFactory().newEvent('org.acme.sample', 'SampleEvent');
27       event.asset = tx.asset;
28       event.oldValue = oldValue;
29       event.newValue = tx.newValue;
30       emit(event);
31
32     });
33 }
34
```

The code in Composer is written in javascript. The sampleTransaction function takes a SampleTransaction structure which includes the new asset Value and the SampleAsset to be updated. You will see in the code that it saves off the old value into a local variable of oldValue in the statement

```
var oldValue = tx.asset.value;
```

tx is the SampleTransaction structure, which from the model file looks like this:

```
transaction SampleTransaction {
  --> SampleAsset asset
  o String newValue
}
```

So, there is a field called newValue and a relationship to the SampleAsset called asset. Since tx is the SampleTransaction object coming in, tx.asset is the related SampleAsset. And, remember, a SampleAsset looks like this in the model file.

```
asset SampleAsset identified by assetId {
  o String assetId
  --> SampleParticipant owner
  o String value
}
```

A SampleAsset has a field called assetId which is the identifier, a field called value which is the SampleAsset's value and a relationship to a SampleParticipant which is the owner of this SampleAsset.

So, tx.asset is the SampleAsset related to the SampleTransaction and tx.asset.value is the SampleAsset's value.

After saving off the old value, it sets the value to the value passed in on the transaction with this statement:

```
tx.asset.value = tx.newValue;
```

Once the tx.asset SampleAsset is updated, it needs to update the Asset Registry. There are Registries for the main components of Hyperledger Fabric, including Assets, Participants, Transactions, Events and Identities.

To update the Asset Registry, you use a function called getAssetRegistry()

and you pass it the Asset Type you are looking for in the Registry.

```
getAssetRegistry('org.acme.sample.SampleAsset')
```

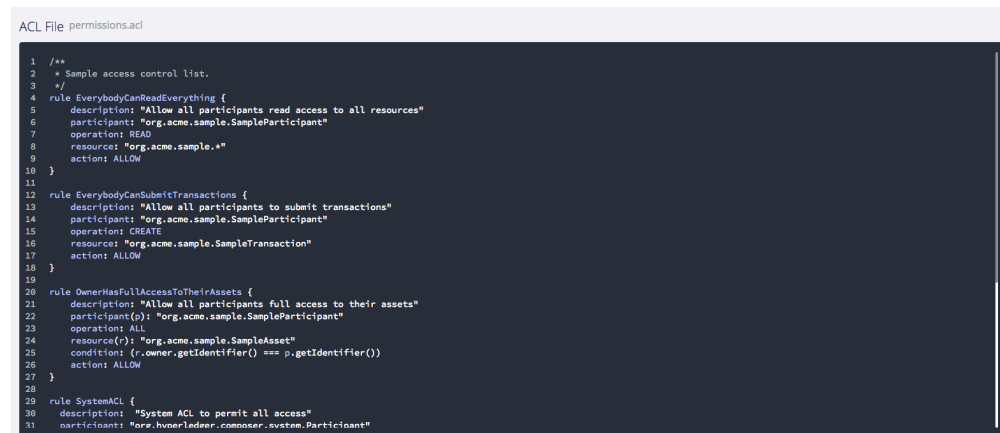
Upon a successful retrieval of the SampleAsset Registry, you simply update the Registry with the updated SampleAsset.

```
assetRegistry.update(tx.asset);
```

After updating the Registry, this code then triggers a SampleEvent event to notify listeners of the SampleAsset being updated and what the old and new values were.

```
var event = getFactory().newEvent('org.acme.sample',
'SampleEvent');
event.asset = tx.asset;
event.oldValue = oldValue;
event.newValue = tx.newValue;
emit(event);
```

Click on the Access Control permissions.acl file. You will see an editor window with the following.



```
ACL File permissions.acl
1 /**
2  * Sample access control list.
3  */
4  rule EverybodyCanReadEverything {
5    description: "Allow all participants read access to all resources"
6    participant: "org.acme.sample.SampleParticipant"
7    operation: READ
8    resource: "org.acme.sample.*"
9    action: ALLOW
10 }
11
12 rule EverybodyCanSubmitTransactions {
13   description: "Allow all participants to submit transactions"
14   participant: "org.acme.sample.SampleParticipant"
15   operation: CREATE
16   resource: "org.acme.sample.SampleTransaction"
17   action: ALLOW
18 }
19
20 rule OwnerHasFullAccessToTheirAssets {
21   description: "Allow all participants full access to their assets"
22   participant(p): "org.acme.sample.SampleParticipant"
23   operation: ALL
24   resource(r): "org.acme.sample.SampleAsset"
25   condition: (r.owner.getIdentifier() === p.getIdentifier())
26   action: ALLOW
27 }
28
29 rule SystemACL {
30   description: "System ACL to permit all access"
31   participant: "org.humanelem.composer.system.Participant"
```

The Access Control file allows you to set the permissions on who can Create, Read, Update or Delete components within the Network. Permissions are created as a set of Rules. One of these rules states that an Owner of a SampleAsset has full access to their SampleAssets.

```
rule OwnerHasFullAccessToTheirAssets {
  description: "Allow all participants full access to
their assets"
```

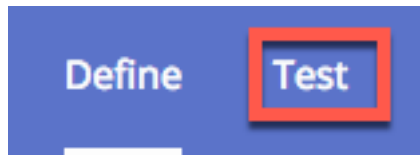


```

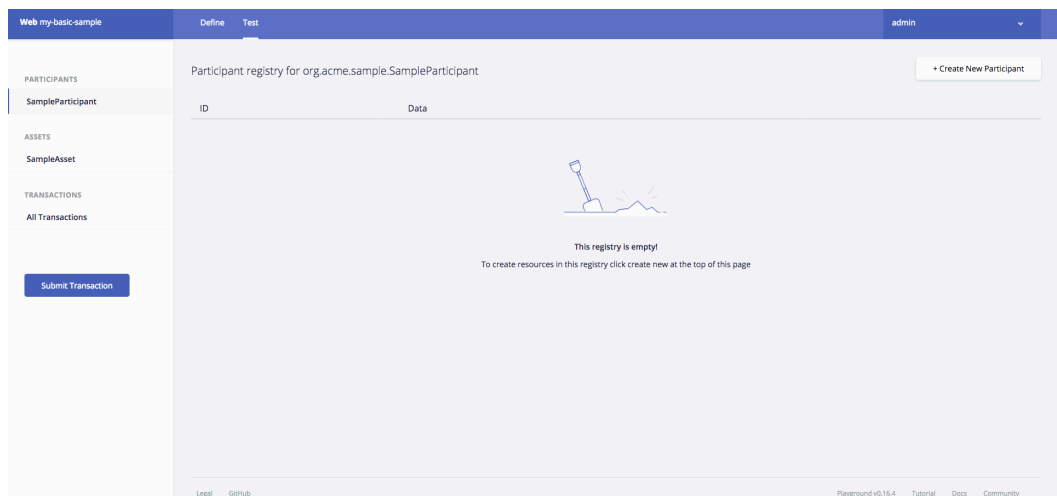
participant(p): "org.acme.sample.SampleParticipant"
operation: ALL
resource(r): "org.acme.sample.SampleAsset"
condition: (r.owner.getIdentifier() ==
p.getIdentifier())
action: ALLOW
}

```

Next, we will move from Defining our Business Network to Testing. In the Navigation section, click on Test.



Now, you will see an interface for testing the network.



In the left navigation, you'll see Participants, Assets and Transactions and button at the bottom for testing the Submission of Transactions. Click on SampleParticipant. You'll notice that there are no Participants in the registry. In the upper right corner, click on Create New Participant. You will see a json editor that allows you to create a new Participant record in the Registry. Copy and Paste the following and click Create New.

```

{
  "$class": "org.acme.sample.SampleParticipant",
  "participantId": "PARTICIPANT_001",
  "firstName": "John",

```

```
"lastName": "Doe"
}
```

You will then see your new participant in the SampleParticipant Registry.


Next, click SampleAsset. Again, you will notice there are no SampleAssets in the registry. Click Create New Asset. Replace the json in the editor with this and click Create New.

```
{
  "$class": "org.acme.sample.SampleAsset",
  "assetId": "ASSET_001",
  "owner": "PARTICIPANT_001",
  "value": "1234"
}
```

Now, if you click on All Transactions, you'll see all of the transactions since you deployed this Network, including adding the Administrator as a Participant, issuing the Administrator Identity and starting the Network. Plus, you will see your two transactions of adding your SampleParticipant and SampleAsset.

Let's create a transaction. Click Submit Transaction. In the editor, you will see a drop down of Transaction Type. In this Business Network, remember, there is only one type of Transaction, SampleTransaction. You see the structure of the Transaction which takes a value of the SampleAsset you want to update and a new Value that will update the value of the SampleAsset. Replace the json in the editor with this and click submit.

```
{
  "$class": "org.acme.sample.SampleTransaction",
  "asset": "ASSET_001",
  "newValue": "5678"
}
```

Once submitted, you will see the new SampleTransaction added to the Registry. Click the view record link to the right. You will see the transaction you submitted, plus a unique identifier and timestamp of the transaction. Click on the Events(1) link above it and you'll see that an event was fired because of this transaction. Click on the chevron icon  to the right to expose the details of the event.

	<div data-bbox="418 197 1442 892"> <div>Historian Record</div> <div>Transaction Events (1)</div> <div>org.acme.sample.SampleEvent#685c0a3b-878f-42e2-9e95-bb6c6f42a1c1#0</div> <pre> 1 { 2 "\$class": "org.acme.sample.SampleEvent", 3 "asset": "resource:org.acme.sample.SampleAsset#ASSET_001", 4 "oldValue": "1234", 5 "newValue": "5678", 6 "eventId": "685c0a3b-878f-42e2-9e95-bb6c6f42a1c1#0", 7 "timestamp": "2018-02-12T21:36:32.760Z" 8 } </pre> </div> <p>You will notice the related SampleAsset that was updated, the unique identifier of the event and a timestamp the event was created. Also, you will see the old and new values. Remember, these were defined in the script file we viewed earlier.</p> <p>Some things to point out. When you were working with creating new SampleAssets and SampleParticipants, you could also delete them and update them. Note that with Transactions and Events, there is no way to Update or Delete them. This is one of the key features of Blockchain which gives it immutability. The Historian tracks everything and it is forever part of the Blockchain. You are logged in as System Administrator and you have no access to delete or update any record of what has happened in the Blockchain.</p> <p>Before we build our first network, let's take a look at some of the basics of modeling business networks.</p>
3	<p><u>Modeling Language Basics</u></p> <p>a. Namespace</p> <p>The Namespace is defined in your model file (.cto). All resources created are implicitly part of this namespace. In addition to your network</p>

namespace, there is a system namespace which contains the base classes for assets, events, participants, and transactions.

b. Data Types

Blockchain is often thought of as a type of database, which it is, albeit a purpose-built database. For simplicity and most common usage, Composer is limited to 6 data types.

- String – a UTF 8 encoded string
- Double – a double precision 64-bit numeric value
- Integer – a 32-bit signed whole number
- Long – a 64-bit signed whole number
- DateTime – an ISO-8601 compatible time instance with optional time zone and UTZ offset
- Boolean – a true/false value
- You will see below there are also Concepts and Enumerated data types which could be considered custom data types, but they're just groupings of the above core data types into logical representations of the data (for example Address which would include Address 1 and 2, City, State and Zip)

c. Assets and Participants

Assets are the entities that could be owned, traded, sold, etc.

Participants are the actors in a transaction such as buyer, seller, shipper, owner, physician, etc.

d. Transactions

Transactions are the definitions of an action, generally involving multiple participants and assets. Examples are Buying/Selling a vehicle, shipping a product, seeing a patient and providing services.

These are things that need to be tracked and auditable. As well, there should be requirements for a transaction to occur and be endorsed so that it can be entered into the ledger. For example, in order to deliver a vehicle to a buyer, funds need to be transferred from the buyer to the seller, the title/registration application needs to be submitted, and the sales manager needs to provide approval of the sales price. If financing is required, all credit checks and qualification has been done and meets the standards set by the seller.

These requirements would be established as part of the chaincode and SmartContract.

e. Relationships

You shouldn't be replicated information about Assets and Participants. Composer provides the mechanism to define relationships between them. So, once I've created an asset, with all of the attributes that define that asset, I can simply refer to it from any other Asset, Participant, Transaction or Event. I don't need to have an attribute like `assetName` on the Participant who is the Owner of the asset or the Transaction that tracks the service I've had on that Asset. I simply create the relationship and I can always get to the `assetName` through the relationship.

f. Abstract, Concepts and Enumerated Data Types

Abstracts define a base class of a type of Asset, Participant, Transaction or Event. Other types can then extend that. So, I might have an Abstract Participant called `Person` that has first and last name and then I can have a `Buyer` that extends `Person` and a `Seller` that extends it as well. Each adding their own unique attributes. Or, an Asset may be owned by a `Seller`, but can never be owned by a `Buyer` and wouldn't be owned by just a `Person`.

Concepts are a way of defining logical groupings of attributes. In the example I gave earlier, a Concept of Address could include the following:

```
abstract concept Address {
  o String street
  o String city default = "Winchester"
  o String country default = "UK"
  o Integer[] counts optional
}
```

I could then have Participant definition that has two attributes called:

```
Address billingAddress
Address shippingAddress
```

I would then refer to `billingAddress.street` or `shippingAddress.city` to get those values.

Enumerated Data Types allow me to define the allowable values. For account status, I may only have `Active` or `Inactive`. I could define it as this:

```
enum AccountStatus {
  o Active
  o Inactive
}
```

I would then use it in my definition of my Participant as

```
o AccountStatus memberStatus
```

If any process tried to update the memberStatus to something like Closed, the transaction would fail because it is not a valid value.

g. Arrays

Arrays are not unlike other environments. I can define recurring groups of a specific data type.

If I had a field like rolling13monthSales defined as an array like this:

```
o Double[] rolling13monthSales
```

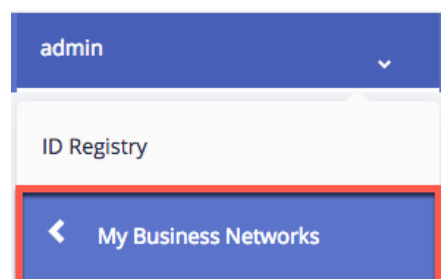
I could define rolling13monthSales[0]-rolling13monthSales[12] as the past 13 months of monthly sales figures. I could define the 0 offset as current and the 12 offset as this month last year.

So, now that you have a basic understanding of the modeling concepts, let's build our first network.

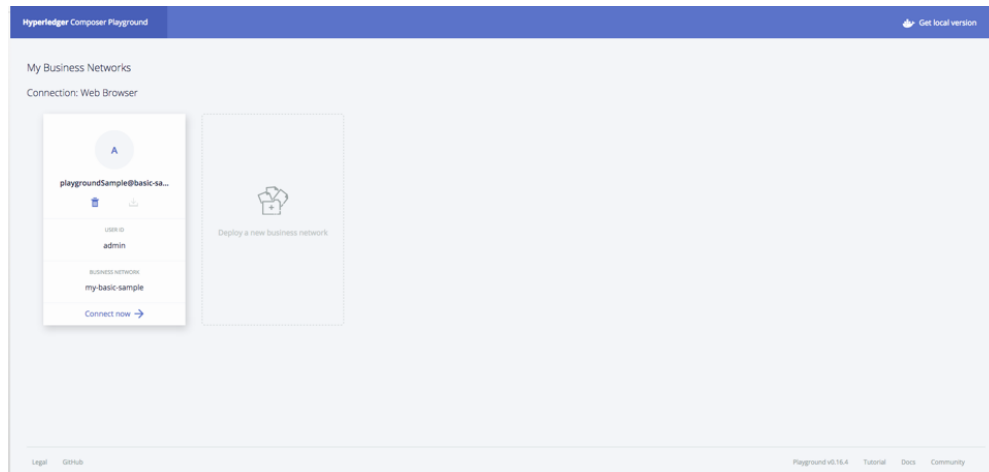
In the Navigation section at the top in the upper right-hand corner, click where it says admin.



Then, click on My Business Networks



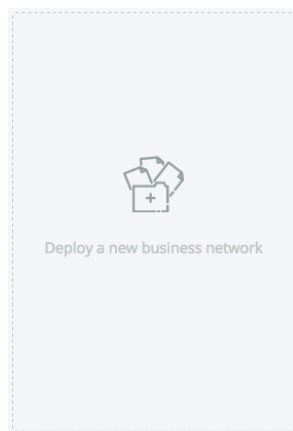
This will take you back to the main Composer page.



4 **Build a Network**

- a. Deploy a new network using sample network template

Back on the main Composer page, click on the Deploy a new business network card.



You will be prompted with the following page.

Hyperledger Composer Playground

Get local version

My Wallet

Not sure where to start? View our Playground tutorial.

Deploy New Business Network

1. BASIC INFORMATION

Give your new Business Network a name:

basic-sample-network

Describe what your Business Network will be used for:

The Hello World of Hyperledger Composer samples


Give the network admin card that will be created a name

eg. admin@basic-sample-network


2. MODEL NETWORK STARTER TEMPLATE

Choose a Business Network Definition to start with:


Choose a sample to play with, start a new project, or import your previous work



basic-sample-network




empty-business-network



Drop here to upload or browse

Samples on npm



basic-sample-network

The Hello World of Hyperledger Composer samples

CONNECTION PROFILE

BASED ON

basic-sample-network

The Hello World of Hyperledger Composer samples

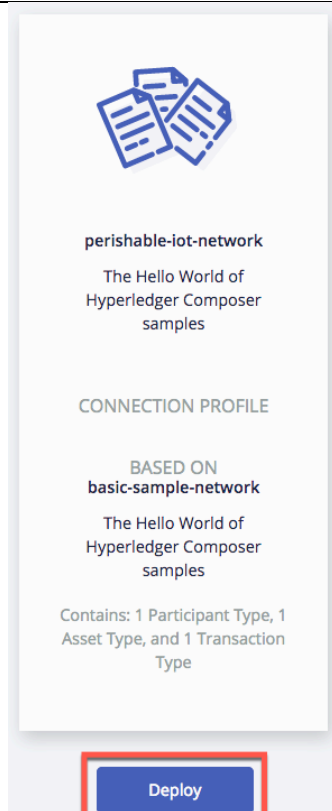
Contains: 1 Participant Type, 1 Asset Type, and 1 Transaction Type

Deploy

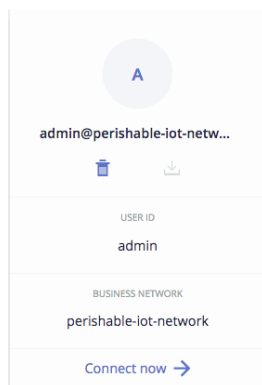
You will need to fill out the following:

Give your new Business Network a Name	perishable-iot-network
Describe what your Business Network will be used for:	<optional description>
Give the network admin card that will be created a name	<optional will default as admin@networkname>
Choose a Business Network Definition to start with:	 <div>perishable-network</div>

Click the Deploy button on the right.



You will then be returned to the main Composer page and you will see your new Business Network Card.



Click the Connect now -> button at the bottom of the card.

b. Explore the network template

As we saw earlier, there is a README.md file which describes the network. In this case, there are 3 types of Participants, Growers, Importers, and Shippers. They interact with 2 types of Assets, Contracts and Shipments.

There are 3 types of transactions, ShipmentReceived, TemperatureReading and SetupDemo.

You can see the definitions of all these by looking at the Model File.

You can see what happens during these transactions by reviewing the Script File.

For this exercise, we're going to make a few modifications to this model. The changes will include:

- Adding a GPS Reading transaction from a sensor in the shipping container to note the current location using a Lat/Lng reading.
- Add an event notification when the Temperature Reading is above the threshold
- Add an event notification when the ship arrives in port
- Modify the temperature reading transaction chaincode to create the event if it is above the contractual threshold

c. Add IoT components to the network model

In the Composer editor, click on the Model File models/perishable.cto link on the right.

In the editor for the models/perishable.cto, below the enum ShipmentStatus, we will add the following lines. This will define the valid values for a directional reading on a compass. This will help to validate the GPS readings.

```
/**
 * Directions of the compass
 */
enum CompassDirection {
    ○ N
    ○ S
    ○ E
    ○ W
}
```

Create a transaction definition to handle the GPS readings. Below the ShipmentReceived transaction definition, copy and paste the following lines.

```
/**
 * A GPS reading for a shipment. E.g. received from a
```

```
device
  * within a shipping container
  */
transaction GpsReading extends ShipmentTransaction {
  o String readingTime
  o String readingDate
  o String latitude
  o CompassDirection latitudeDir
  o String longitude
  o CompassDirection longitudeDir
}
```

Note the `CompassDirection latitudeDir` and `CompassDirection longitudeDir` attributes. This is using the Enumerated value we created above.

To store the GPS readings, we will add them to the Shipment asset as an array of values. Find the asset Shipment identified by `shipmentId` entry and make the highlighted change.

```
/**
 * 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
  o GpsReading[] gpsReadings optional
  --> Contract contract
}
```

The `gpsReadings` attribute is optional because you can have a Shipment asset that has no GPS Readings yet.

Now, let's create the definition of our two new events. At the bottom of the model file, paste the following lines.

```
/**
 * An event - when the temperature goes outside the
 * agreed-upon boundaries
 */
event TemperatureThresholdEvent {
  o String message
}
```

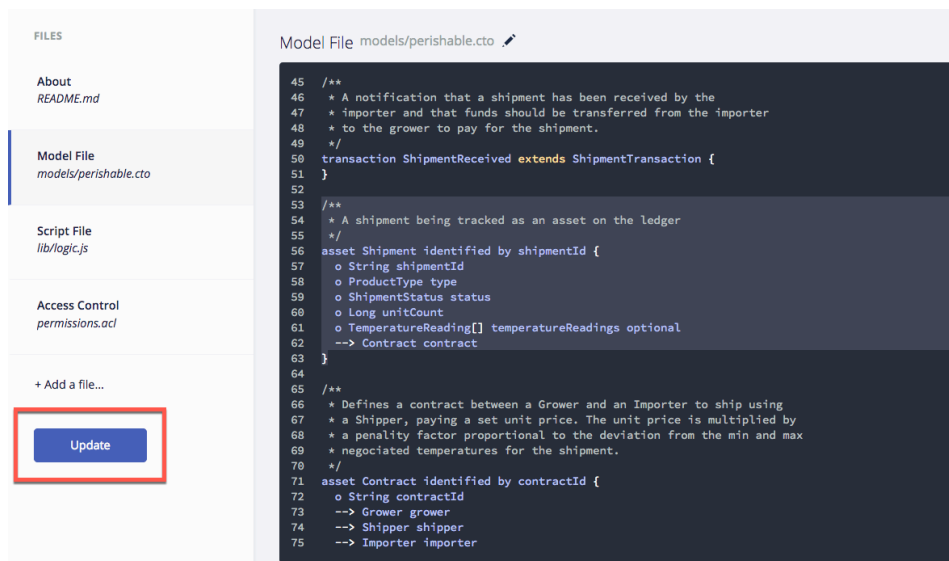
```

    o Double temperature
    --> Shipment shipment
  }

/**
 * An event - when the ship arrives at the port
 */
event ShipmentInPortEvent {
  o String message
  --> Shipment shipment
}

```

Click the Update button off to the lower left.



- d. Add chaincode to monitor the temperature readings from the sensor

Click on the Script File lib/logic.js link on the left to edit the chaincode.

First, we want to modify the temperatureReading function. You can copy and paste this whole function. The changes are highlighted. This will check the contract associated with this shipment and compare the temperature reading to the contract min/max temperatures. If the current temperature reading is below the min or above the max, it will trigger a TemperatureThresholdEvent.

```

/**
 * 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;
    var NS = "org.acme.shipping.perishable";
    var contract = shipment.contract;
    var factory = getFactory();

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

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

    if (temperatureReading.centigrade < contract.minTemperature ||
temperatureReading.centigrade > contract.maxTemperature) {
        var temperatureEvent = factory.newEvent(NS,
'TemperatureThresholdEvent');
        temperatureEvent.shipment = shipment;
        temperatureEvent.temperature =
temperatureReading.centigrade;
        temperatureEvent.message = 'Temperature threshold
violated! Emitting TemperatureEvent for shipment: ' +
shipment.$identifier;
        emit(temperatureEvent);
    }

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

```

e. Add chaincode to register the GPS location

Now, we will add in the new GPS Reading transaction. At the bottom of the file, paste in the following lines. Something important to note. The comments above the function are critical. The @param () specifies the Transaction Type that is defined in your model file. This has to match exactly. The name directly after it needs to match the function name exactly so that it can be invoked when that Transaction is submitted. The

@transaction is required to specify that this is a Transaction function.

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

    var factory = getFactory();
    var NS = "org.acme.shipping.perishable";
    var shipment = gpsReading.shipment;
    var PORT_OF_NEW_YORK = '/LAT:40.6840N/LONG:74.0062W';

    var latLong = '/LAT:' + gpsReading.latitude +
gpsReading.latitudeDir + '/LONG:' +
        gpsReading.longitude + gpsReading.longitudeDir;

    if (shipment.gpsReadings) {
        shipment.gpsReadings.push(gpsReading);
    } else {
        shipment.gpsReadings = [gpsReading];
    }

    if (latLong == PORT_OF_NEW_YORK) {
        var shipmentInPortEvent = factory.newEvent(NS,
'ShipmentInPortEvent');
        shipmentInPortEvent.shipment = shipment;
        var message = 'Shipment has reached the destination port
of ' + PORT_OF_NEW_YORK;
        shipmentInPortEvent.message = message;
        emit(shipmentInPortEvent);
    }

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

When a GPS Reading comes in, the Lat/Lng is pushed into the array attribute on the Shipment Asset. If the Lat/Lng matches that of the Port of New York, a ShipmentInPortEvent stating that the shipment has arrived in port is created and the Shipment is updated in the Registry.

In the setupDemo function, change the following highlighted statements to seed each participant with 5000 dollars.

```
// create the grower
```

	<pre> var grower = factory.newResource(NS, 'Grower', 'farmer@email.com'); var growerAddress = factory.newConcept(NS, 'Address'); growerAddress.country = 'USA'; grower.address = growerAddress; grower.accountBalance = 5000; // create the importer var importer = factory.newResource(NS, 'Importer', 'supermarket@email.com'); var importerAddress = factory.newConcept(NS, 'Address'); importerAddress.country = 'UK'; importer.address = importerAddress; importer.accountBalance = 5000; // 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 = 5000; </pre> <p>Click the Update button to save the changes.</p>
5	<p><u>Test the Network</u></p> <p>a. Run the Setup Demo transaction</p> <p>In the Navigation section, click on Test.</p> <p>You will now notice that you have different types of Participants than we saw in the Basic Network example earlier. You now have Grower, Importer, Shipper as Participant Types. You have Contract and Shipment Asset Types. If you click on each, you'll see that as we saw earlier, there are no entries in the registries for each type. If you click Submit Transaction, you can choose Setup Demo and click Submit.</p>

Submit Transaction

Transaction Type

SetupDemo

JSON Data Preview

```

1  {
2    "$class": "org.acme.shipping.perishable.SetupDemo"
3  }

```

☐ Optional Properties

Just need quick test data? [Generate Random Data](#)

Cancel

Submit

Once that is complete, you will see a new Grower, Importer, Shipper, Contract and Shipment in the registries. The ID's are as follows:

Grower	farmer@email.com
Importer	supermarket@email.com
Shipper	shipper@email.com
Contract	CON_001
Shipment	SHIP_001

Take a look at the Contract. You will notice that it has some terms & conditions to the contract. There is a unitPrice which is the agreed upon price that will paid to the grower from the importer upon delivery. However, if the shipment is late, the importer pays nothing according to the chaincode. Also, there is a minTemperature and maxTemperature which are the threshold values to ensure that the shipments were kept at acceptable temperatures. There is also a min and max penalty factor that can be applied if the temperature thresholds are violated.

This is a very simple example of a Smart Contract.

Next, take a look at the Shipment. You will see the type of product being shipped, the current status, the number of units. You will also notice that there are no temperatureReadings or gpsReadings yet. Let's add some.

b. Test adding temperature readings

Click on the Submit Transaction button and choose TemperatureReading. Enter the following in the editor and click Submit.

```
{
  "$class": "org.acme.shipping.perishable.TemperatureReading",
  "centigrade": 5,
  "shipment": "SHIP_001"
}
```

This will successfully add a temperature reading of 5 degrees centigrade. This falls into the min/max thresholds.

Click Submit Transaction again and choose TemperatureReading again. This time enter the following and click Submit.

```
{
  "$class": "org.acme.shipping.perishable.TemperatureReading",
  "centigrade": 11,
  "shipment": "SHIP_001"
}
```

This will set the reading to be 11 degrees centigrade which will violate the temperature threshold and should emit a TemperatureThresholdEvent.

You can go back to see All Transactions and look at the records for the transactions you just submitted and verify that Events were created when the conditions were met.

c. Test adding a GPS reading

Click on Submit Transaction and choose GpsReading this time. Enter the following into the editor and click submit.

```
{
  "$class": "org.acme.shipping.perishable.GpsReading",
  "readingTime": "05:30",
}
```

```
"readingDate": "2018-02-12",  
"latitude": "40",  
"latitudeDir": "N",  
"longitude": "74",  
"longitudeDir": "Z",  
"shipment": "SHIP_001"  
}
```

Notice that you can't commit the transaction because there is an invalid enum value for CompassDirection. Remember, we put the valid list of values as E,N,S,W. Change the Z to W and you can click Submit.

Click Submit Transaction and choose GpsReading again. This time, paste this in and click submit.

```
{  
  "$class": "org.acme.shipping.perishable.GpsReading",  
  "readingTime": "05:30",  
  "readingDate": "2018-02-12",  
  "latitude": "40.6840",  
  "latitudeDir": "N",  
  "longitude": "74.0062",  
  "longitudeDir": "W",  
  "shipment": "SHIP_001"  
}
```

This will create the ShipmentInPortEvent notifying listeners that the shipment has arrived in port.

Also, now, if you go back to look at the Shipment SHIP_001 asset, you'll now see the GPS Readings and Temperature Readings associated with the shipment.

d. Test receiving a shipment

Lastly, we will click Submit Transaction and choose ShipmentReceived. Paste the following into the editor and click Submit.

```
{  
  "$class": "org.acme.shipping.perishable.ShipmentReceived",  
  "shipment": "SHIP_001"  
}
```

This will mark this shipment as received and will determine the payout. Remember, on the terms of the contract, the payout was based on unit price of .5. There were 5000 units, so that would be 2500 dollars if all went as planned.

	<p>In this case, the shipment arrived on time, but violated the max temperature threshold by 1 degree. The smart contract states that the penalty for a temperature threshold violation is based on the difference between the high/low reading and the threshold. In this case, that's 1 degree. The difference in centigrade is multiplied by the maxPenaltyFactor, which in this case is .1. So, $1 * .1 = .1$. Multiplying that by the number of units (5000) is 500. So, instead of exchanging 2500 dollars, the contract terms dictate an exchange of 2000 dollars.</p> <p>You should be able to go to the Grower and see that their balance is now increased by 2000 dollars to 7000. The Importer's balance is decreased by 2000 dollars and is now 3000 dollars.</p> <p>We are finished testing, so in the Navigation section, click on Define to return to the definition of the business network.</p>
6	<p><u>Export the Business Network Archive</u></p> <p>a. Export the Business Network Archive (.bna) file and save off</p> <p>On the Define view, in the lower left-hand corner, you will see some Network Management functions.</p> <p>Import/Replace, allows you to import a definition and replace the one you are currently working on.</p> <p>Export allows you to create a Business Network Archive file which can be deployed on another Hyperledger Fabric installation.</p> <p>Click on the Export button. You will be prompted to save the file. Save it to your local drive. We will use this file in the next section when we deploy this to another instance of Hyperledger Fabric and Composer.</p>



Module 1: Lab Summary

You got hands-on with the Hyperledger Composer development environment for Hyperledger Fabric, the platform for IBM Blockchain.

You viewed an existing business network and learned the basics of the modeling language and how to build smart contracts.

You were worked within the Hyperledger Composer development environment to modify an existing Business Network, test those changes and verify that your changes were working as designed.