V2.4

Hyperledger Fabric / Composer Developer Lab Guide

BLOCKCHAIN TRAINING ALLIANCE
KRIS BENNETT

Table of Contents

Hyperledger Fabric (v1.4) / Composer Lab Guide

Section 1 – Getting Hands-On with Hyperledger

In this section we'll get hands-on with Composer Playground. A solution template will be deployed which will be investigated and explained. At the end of this section you should have a solid understanding of the components of a Composer solution and how they all interact.

Lab 1 – Introduction to Composer Playground

In this lab we'll start off using the web-based Composer Playground IDE to generate a sample solution and explore its components.

Step 1 – Go to the web-based Composer Playground site

On your development environment, open a browser and navigate to:

```
http://composer-playground.mybluemix.net
```

Then, click on the "Let's Blockchain" button to get started.

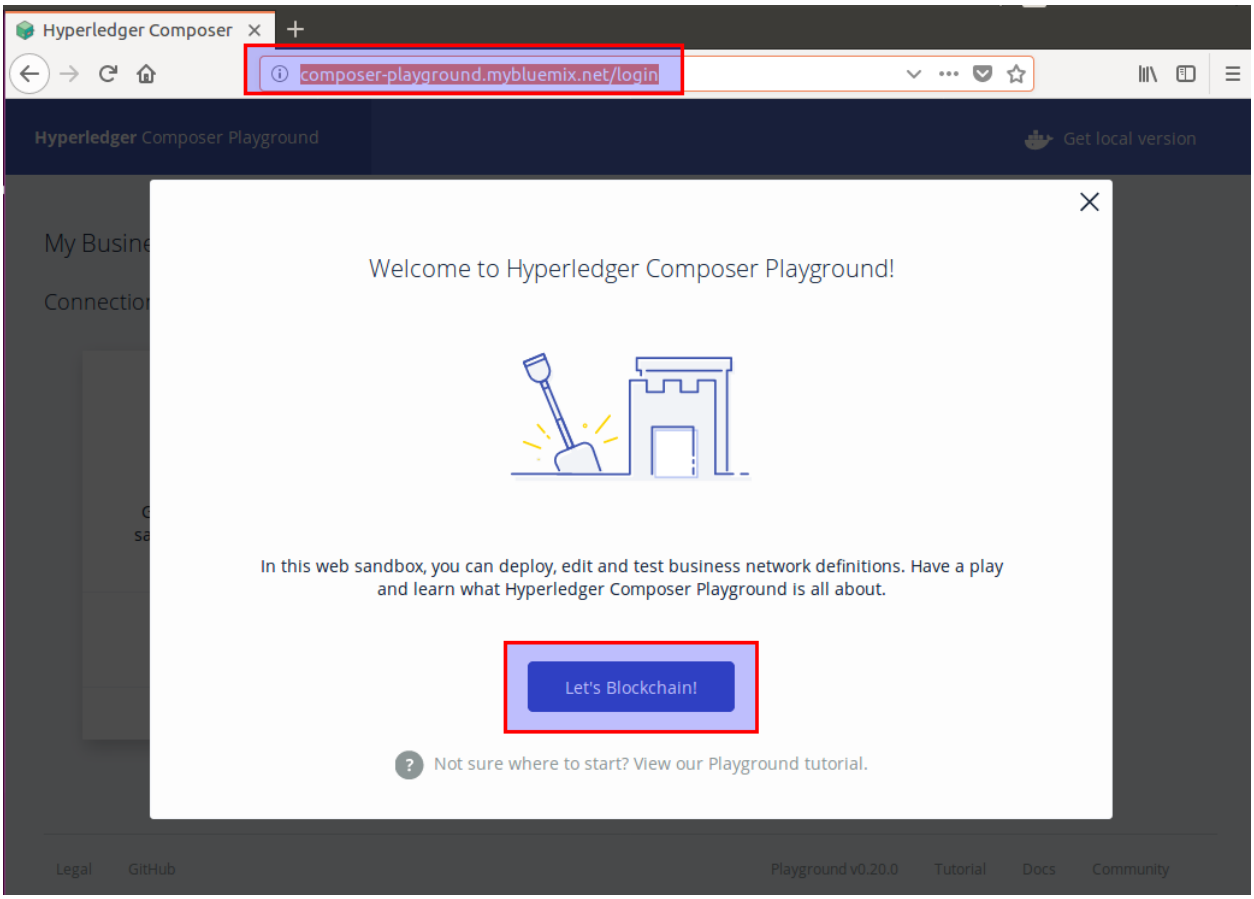


Figure 1 - Navigate to the Composer Playground site and click on the "Let's Blockchain!" button.

Step 2 – Deploy a new business network using the Perishable Network solution template

In this step you will deploy a new business network solution using the perishable network template as a starting point. To begin, click on the "Deploy a new business network" button.

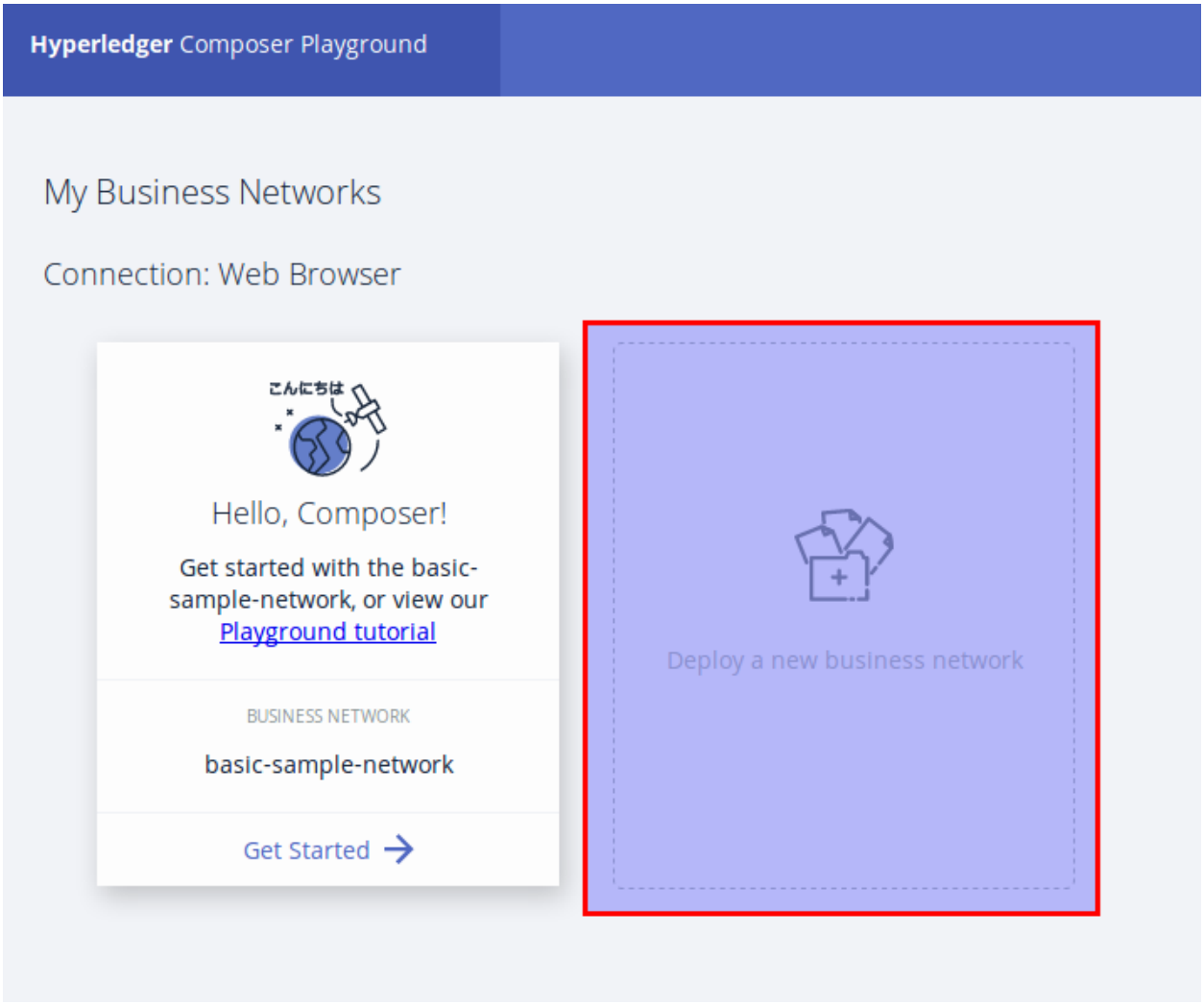


Figure 2 - Choose the option to deploy a new business network.

Be sure to leave the defaults in section "1. Basic Information".

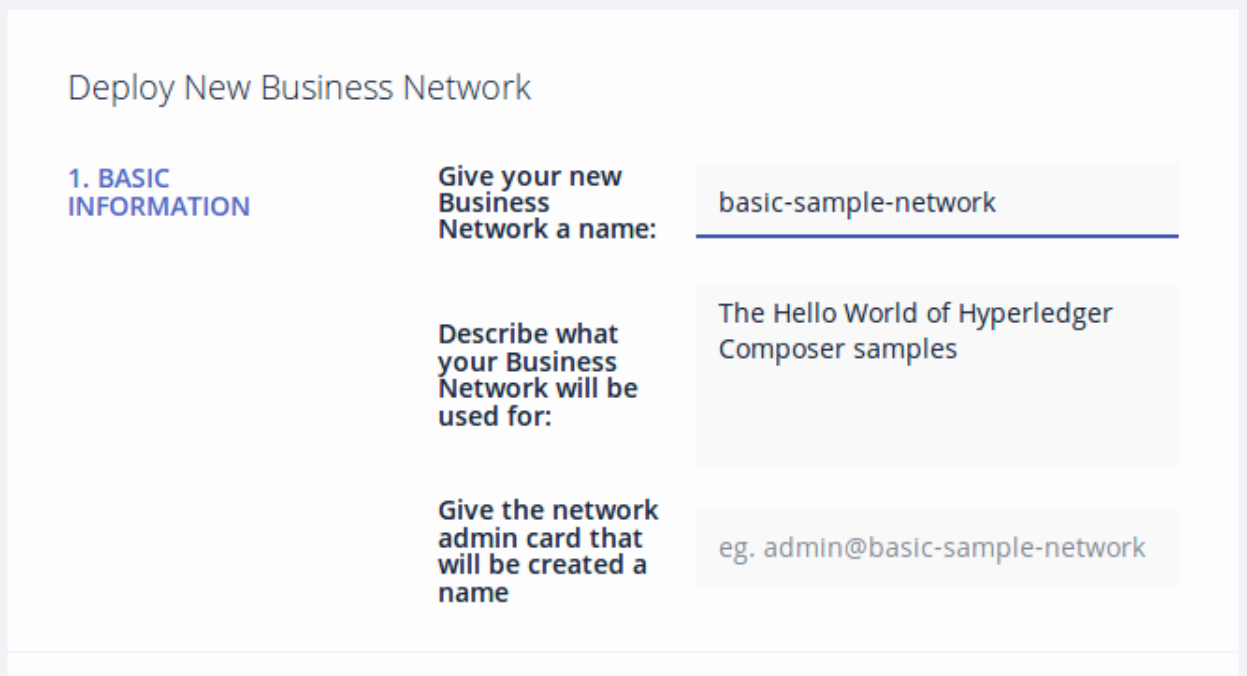


Figure 3 - Leave the default values.

In section "2. Model Network Template", select the "perishable-network" template.

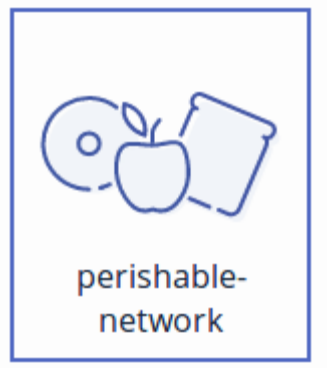


Figure 4 - Select the perishable-network template.

Next, click the "Deploy" button to deploy the solution template.

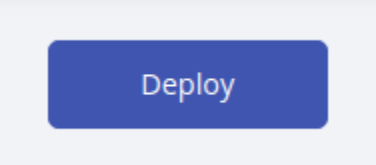


Figure 5 - Deploy the solution template.

Connect to your new Composer solution by clicking the "Connect Now" button.

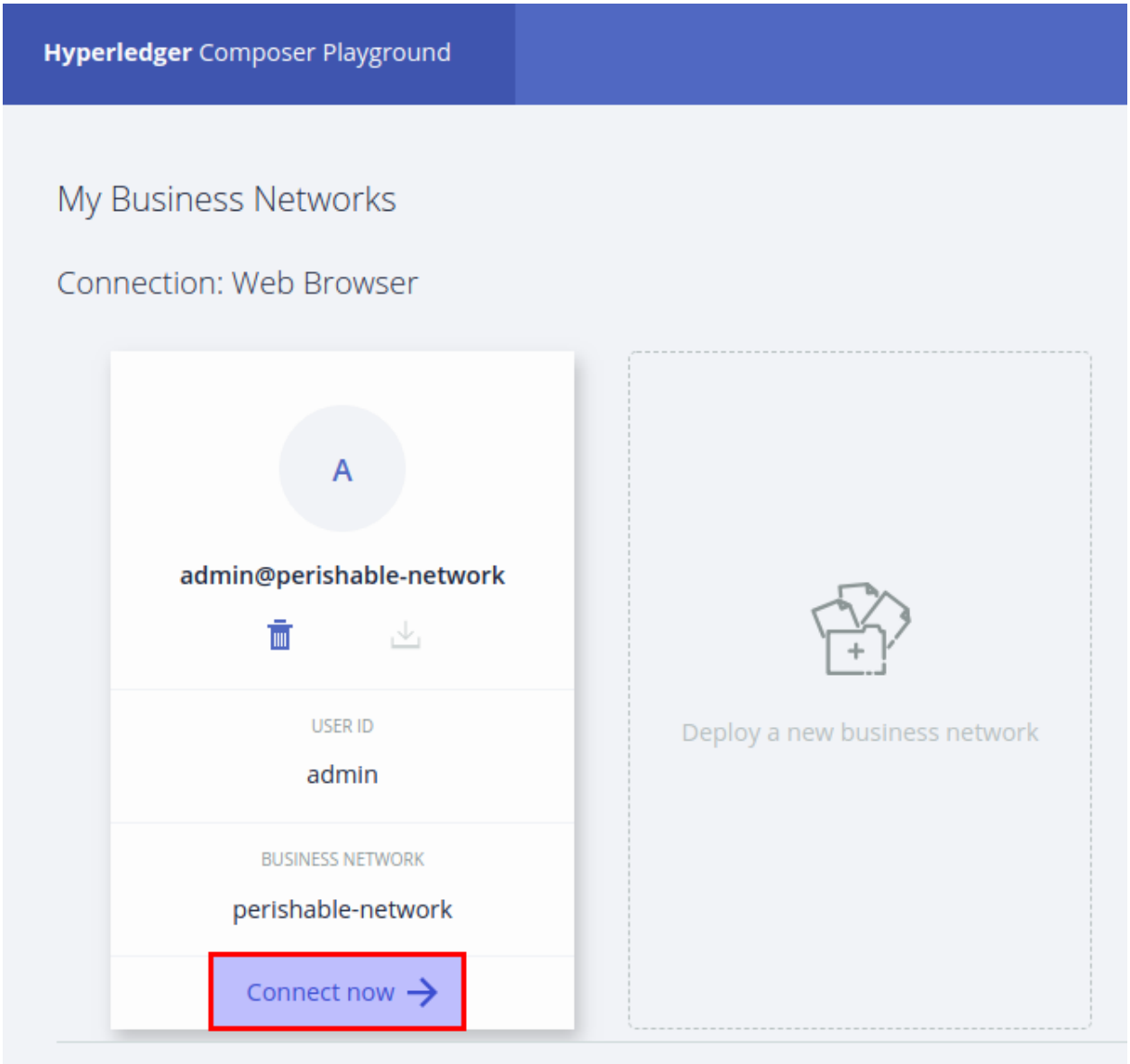


Figure 6 - Connect to your new solution.

Step 3 – Reviewing the perishable network solution template

Take a moment to read the "About File"(README.md) description file to understand the solution and how it works.

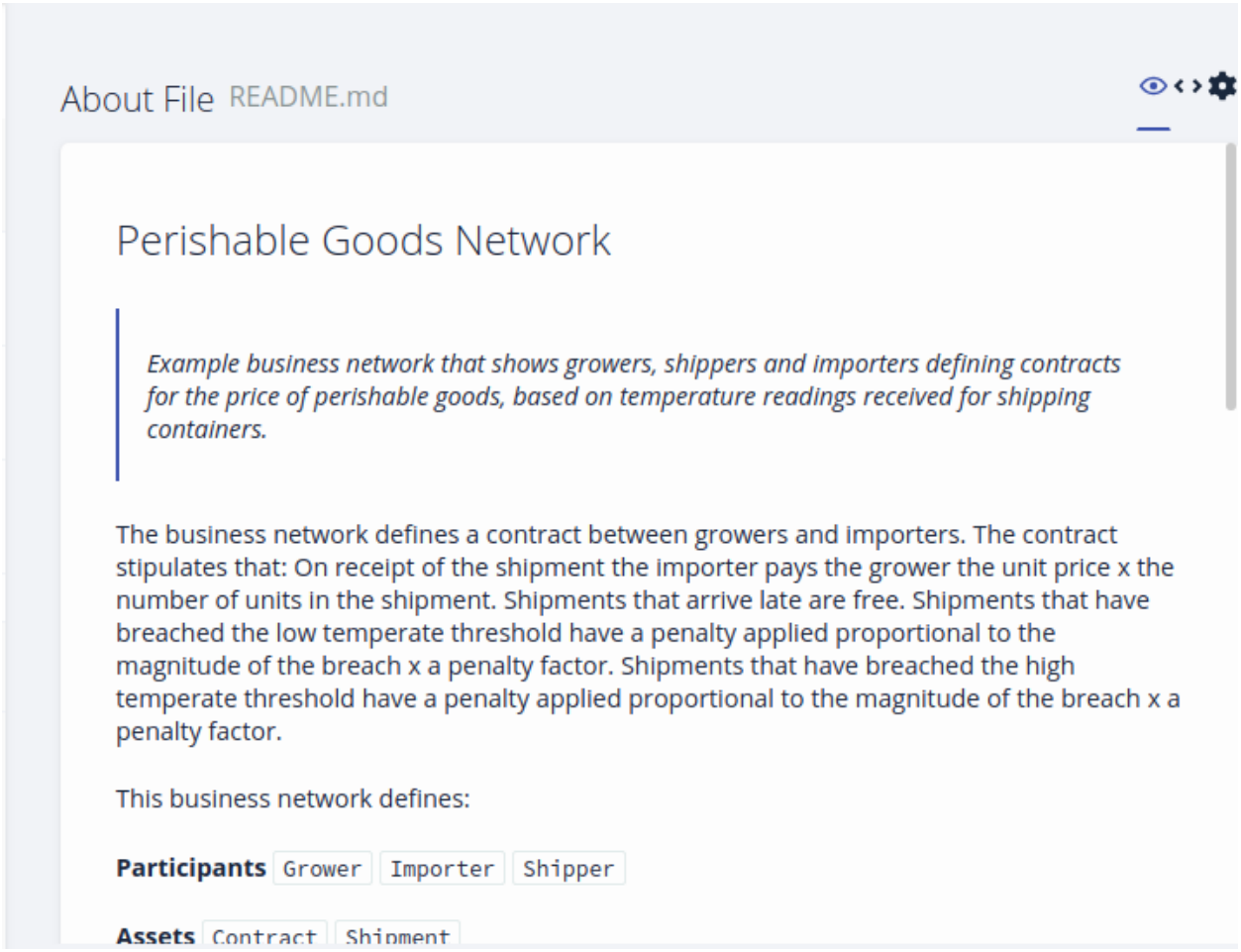


Figure 7 - The solution readme file.

Let's get hands-on and see how this sample solution works. Start by clicking on the "Test" tab at the top of the screen.

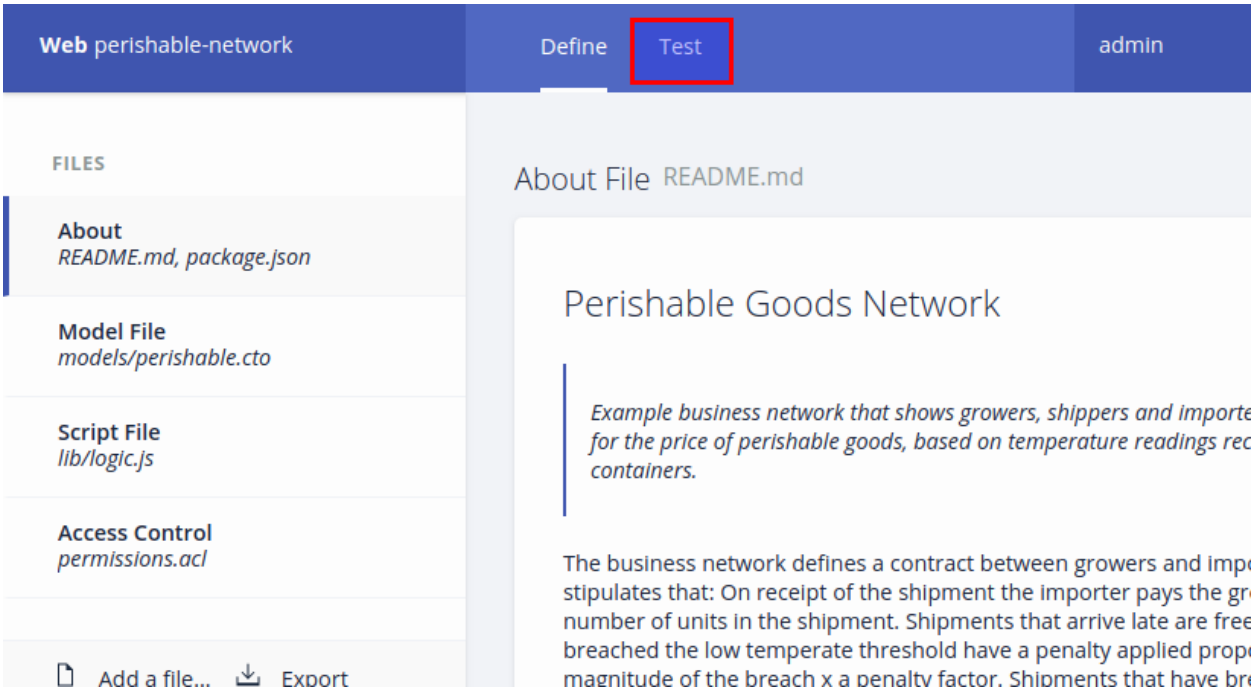


Figure 8 - Click on the Test tab to test out the solution.

We'll start by submitting a SetupDemo transaction which will populate the solution with sample participants and assets. Click on the "Submit Transaction" button to begin the process of initiating a transaction.

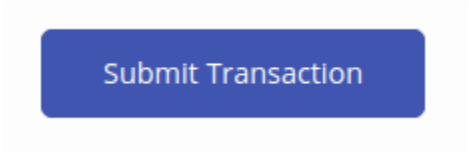


Figure 9 - The Submit Transaction button.

From the "Submit Transaction" window, select the "SetupDemo" transaction and click the "Submit" button.

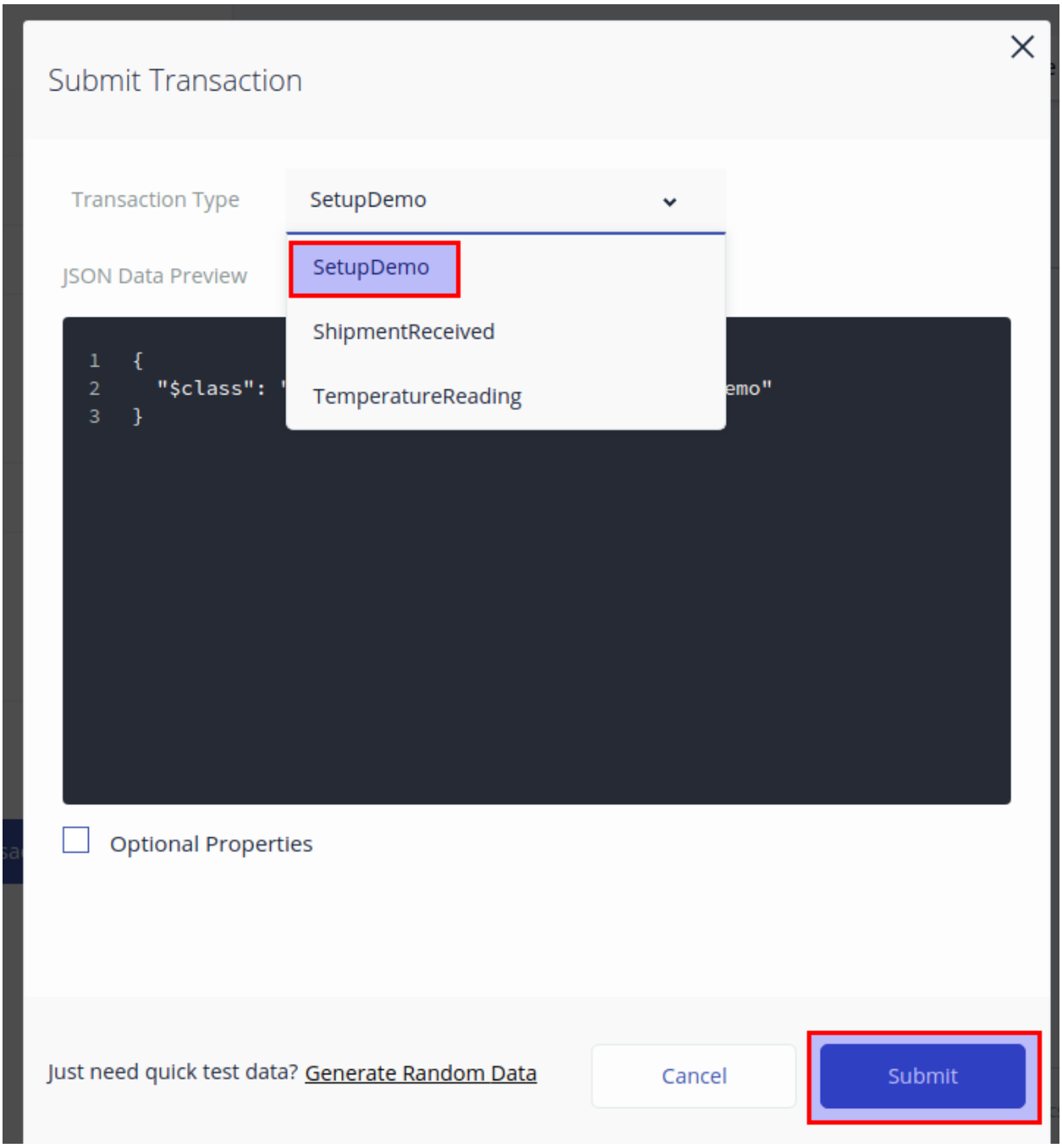


Figure 10 - Choosing the SetupDemo transaction.

The SetupDemo transaction creates three participants (one Grower, one Importer, and one Shipper). Take a moment to review each sample participant and the data which defines each one.

Figure 11 - The Grower participant.

The SetupDemo transaction also creates two sample assets - a Contract and a Shipment. Take a moment to review each sample asset and the data each one contains.

Figure 12 - A Contract asset.

Let's go ahead and submit a TemperatureReading transaction. Click on the "Submit Transaction" button, and select "TemperatureReading " transaction from the drop-down list.

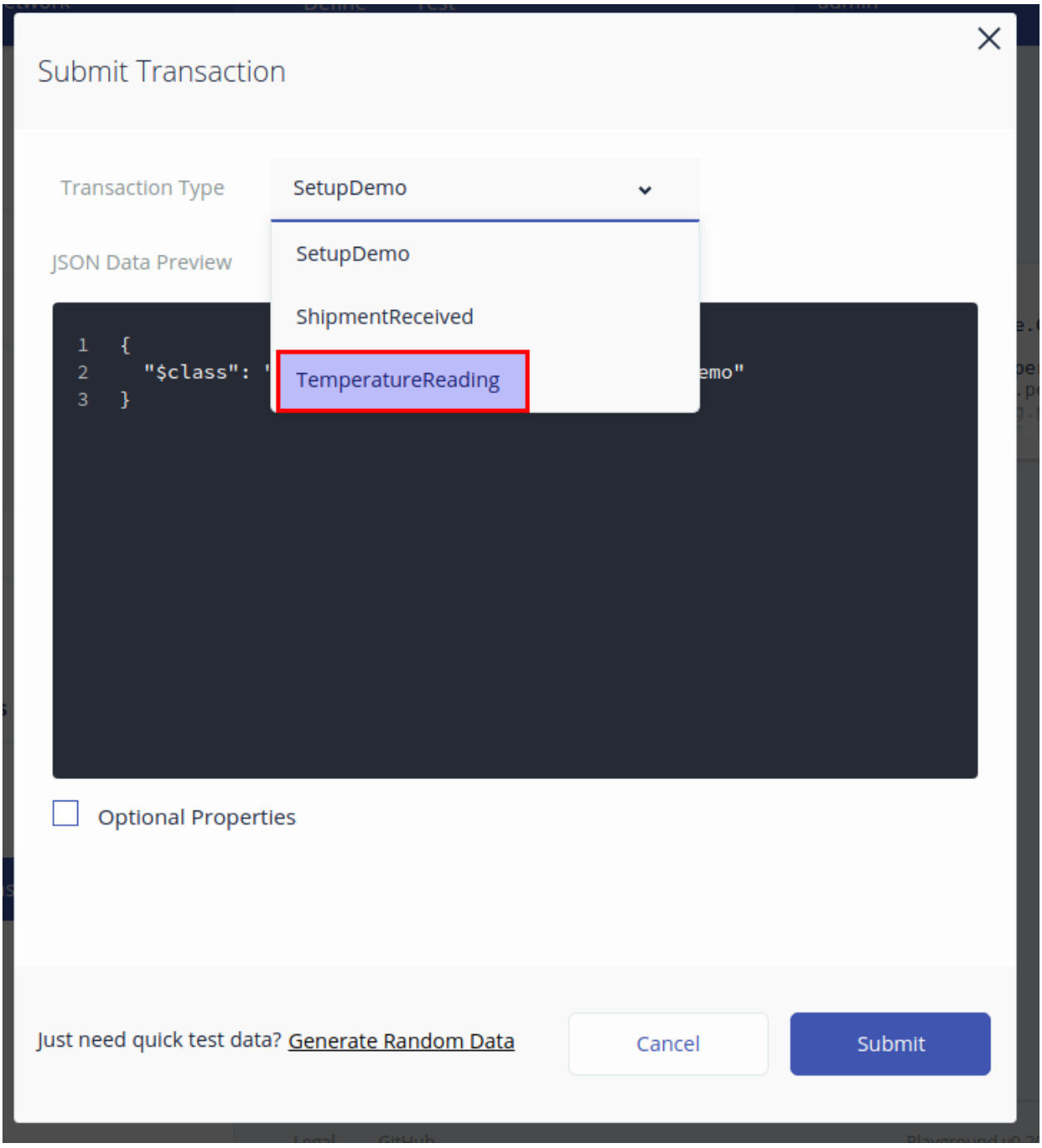


Figure 13 - Selecting the TemperatureReading transaction.

Edit the JSON TemperatureReading object so that it records a Centigrade temperature of 5 for the shipment with ID "SHIP_001". Then click the "Submit" button to submit the transaction.

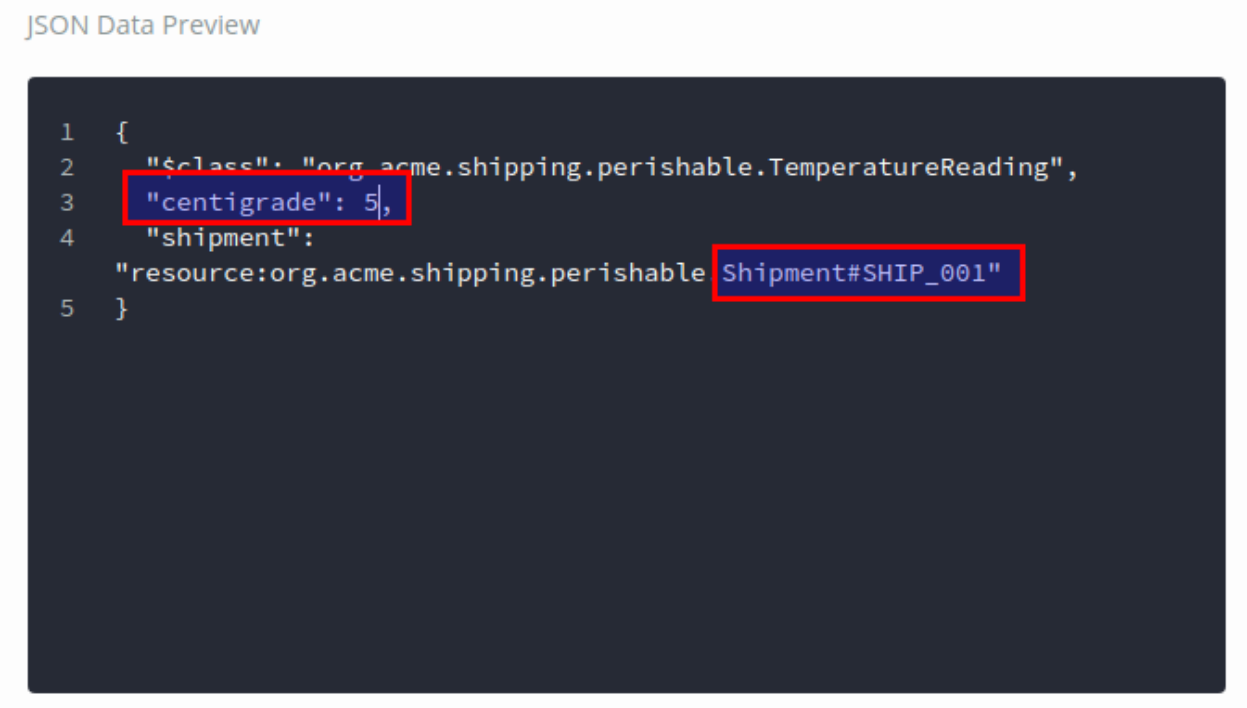


Figure 14 - Creating a new temperature reading.

You should now see a confirmation message in the top-right corner of the browser window. This message lets you know the transaction has been successfully submitted.

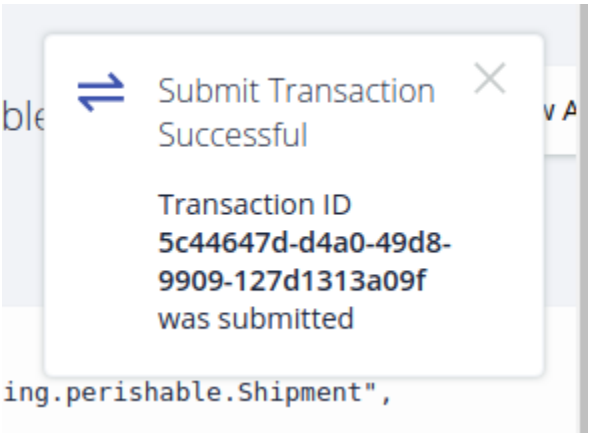


Figure 15 - The transaction was successfully submitted.

Now take a look at the shipment asset - note that the temperature reading has been added to the shipment asset.

Figure 16 - The Shipment asset now has one temperature reading.

Let's add one more temperature reading for shipment "SHIP_001". For the second reading, please supply a value of 12 degrees Centigrade.

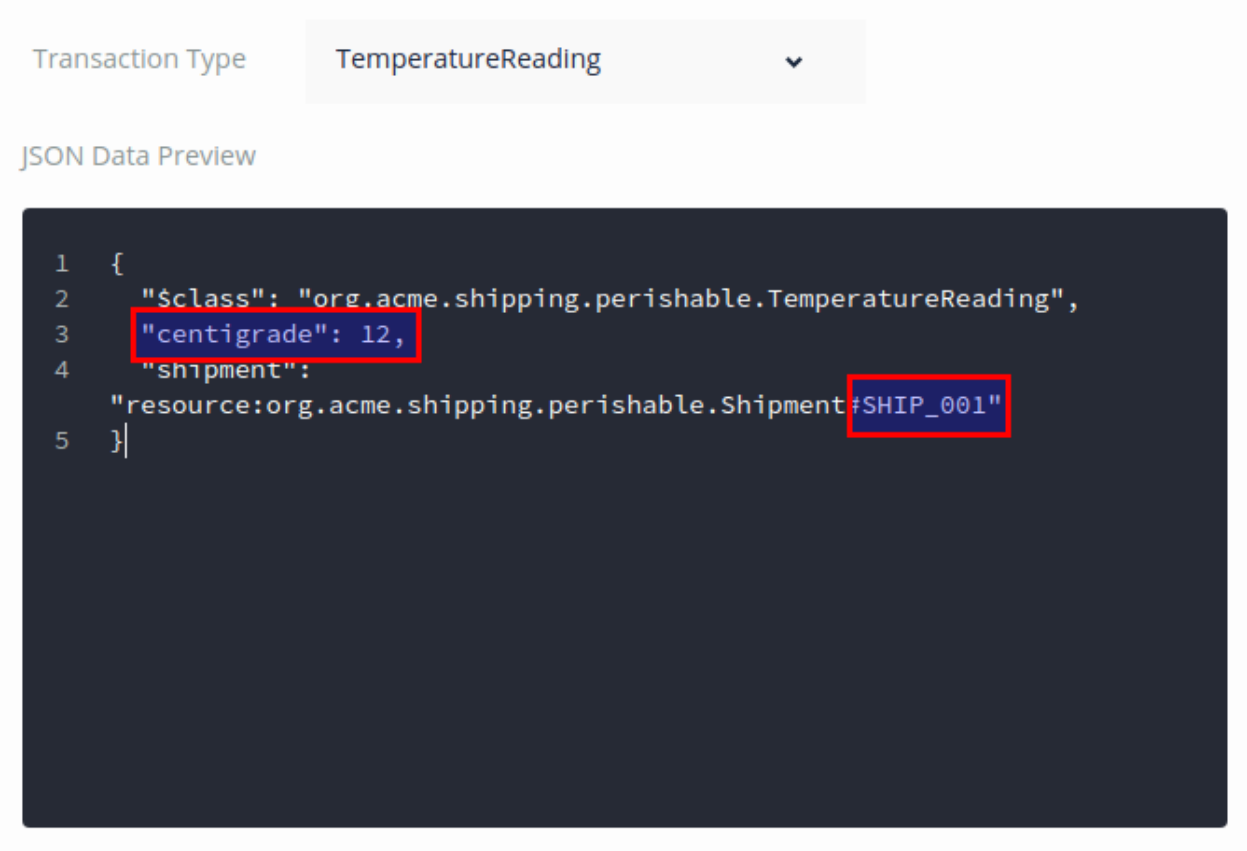


Figure 17 - Adding a new temperature reading with a temperature above the maximum allowable value.

Review the Shipment asset again. Note the two temperature readings now recorded on the shipment.

Figure 18 - The updated Shipment object.

Now let's go ahead and submit a "ShipmentReceived" transaction.

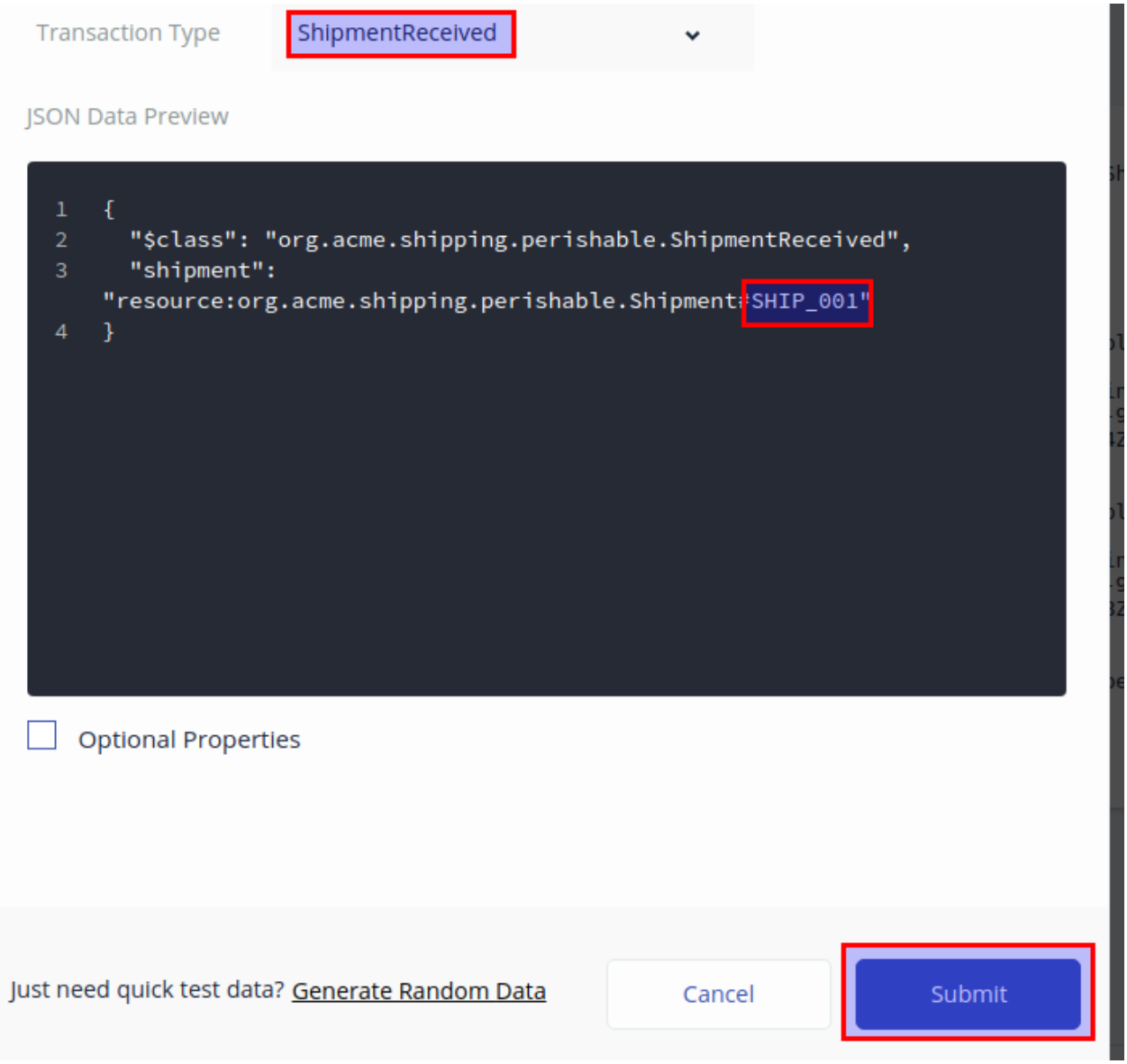


Figure 19 - Creating a ShipmentReceived transaction.

After submitting the "ShipmentReceived" transaction, take a moment to review the Grower and Importer participants. You should see the Importer now has an "accountBalance" of -1500 while the Grower has an accountBalance of +1500.

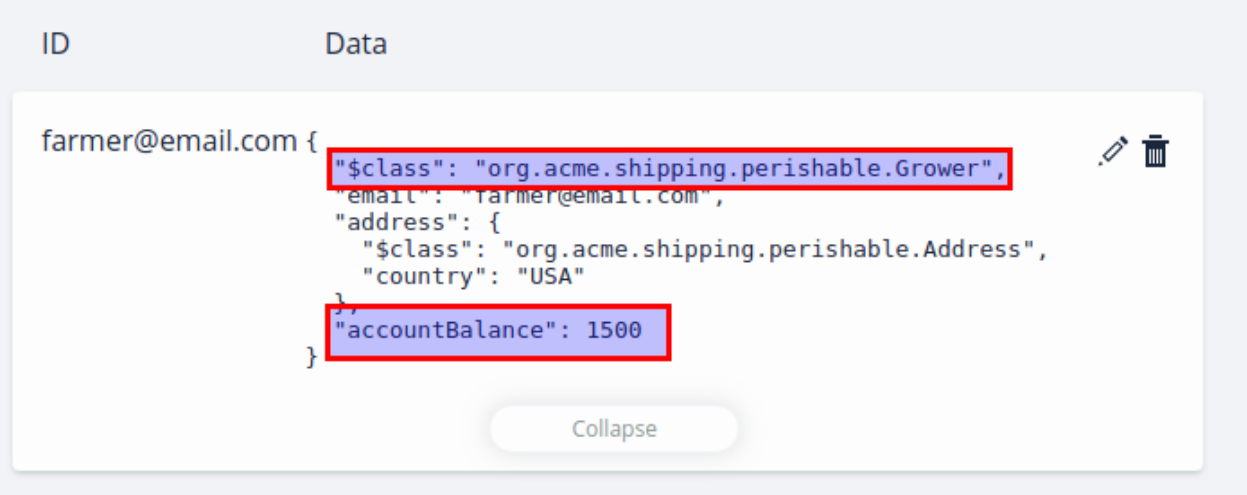


Figure 20 - The updated Grower participant.

Figure 21 - The updated Importer participant.

Section 2 – Reviewing the Perishable Network solution components

In this section we'll take a deeper look at all the components of the perishable network solution template and how they interoperate.

Lab 1 – Reviewing the Model (.cto) file

In this lab we'll take a look at the model definition file included in this solution template. This model definition file contains definitions for all the solutions participants, assets, and transactions.

Step 1 – Opening the solutions definition section

Now let's take a look at the components which make up this solution. Start by clicking on the "Define" option in Composer Playground.

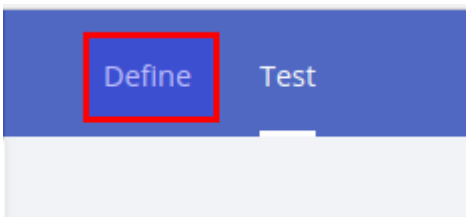


Figure 22 - Opening the solutions definition tab.

In the solution definition view you should see four different files.

- The "About" file contains information about the solution and how it works. This is the file you reviewed after deploying the Perishable Good solution template.
- The "Model" file (.cto file) defines the properties of the Assets, Participants, and Transactions that make up this solution.
- The "Script" file (.js file) contains the code that gets executed for each transaction.
- The "Access Control" file (.acl file) contains the permissions settings for this solution.

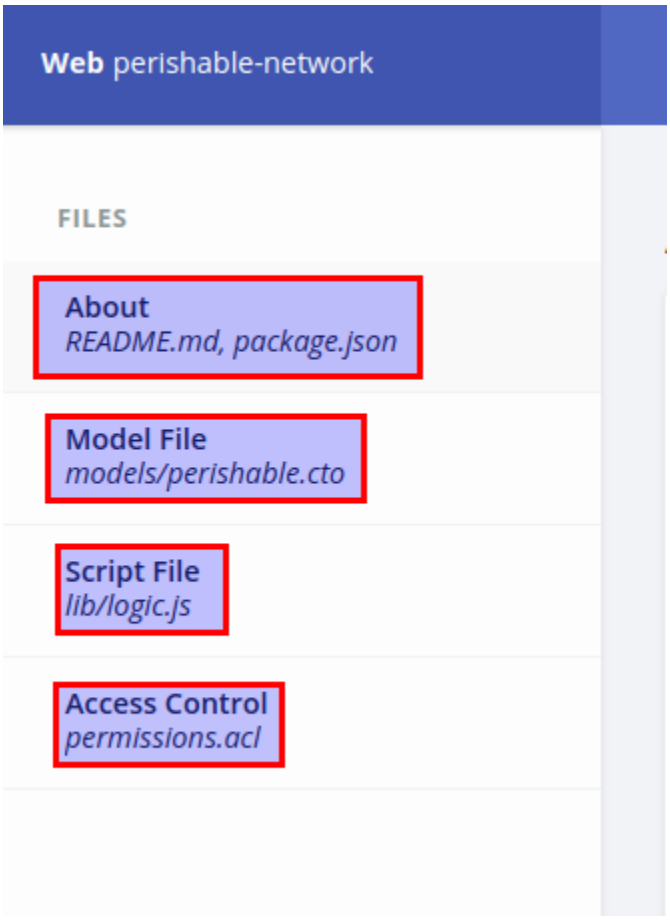


Figure 23 - The files that make up this Composer solution.

Step 2 – Open the model file for review

Let's start by reviewing the Model file. Click on the model file to view it in the browser.

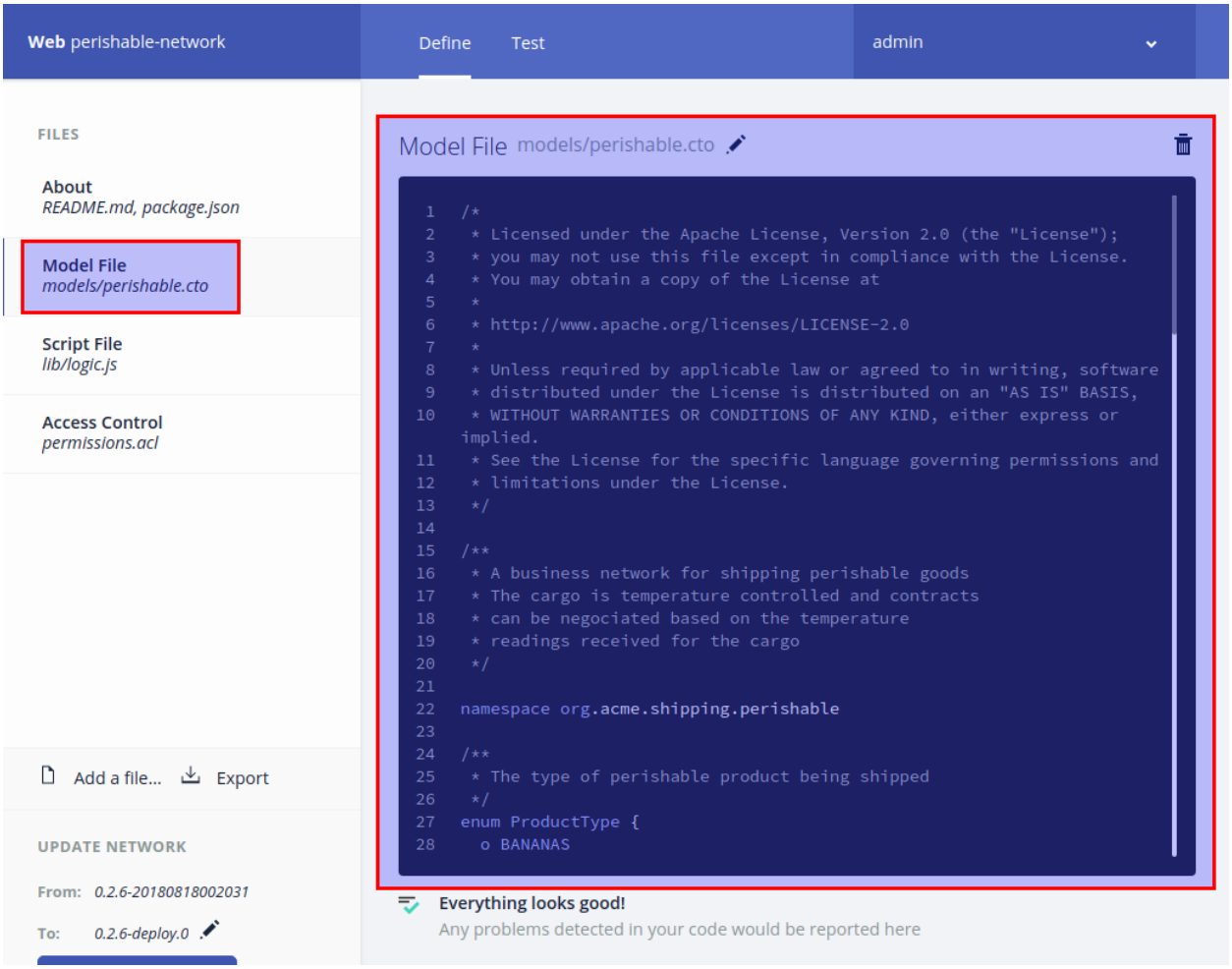


Figure 24 - Open the model (.cto) file for review.

The Model file (.cto file extension) starts by defining a namespace that all solution components will fall under. This namespace helps keep artifacts from different solutions separated.

```
21
22  namespace org.acme.shipping.perishable
23
```

Figure 25 - The namespace used in this solution.

Model files can define and contain custom data structures called enumerations (enum). This solution defines two enumerations for defining the types of products that can be sold, and the possible status values for a shipment. You will see these enumerations being used later in the Model file as properties of a transaction and an asset.

```
24  /**
25   * The type of perishable product being shipped
26   */
27  enum ProductType {
28    o BANANAS
29    o APPLES
30    o PEARS
31    o PEACHES
32    o COFFEE
33  }
34
35  /**
36   * The status of a shipment
37   */
38  enum ShipmentStatus {
39    o CREATED
40    o IN_TRANSIT
41    o ARRIVED
42  }
```

Figure 26 - Two enums are defined in this solution.

Hyperledger gives developers the ability to create "abstract" or template objects that can be inherited from and extended. This is a useful pattern when multiple objects are of the same base type or will all share properties in common. In this solution an abstract transaction object is defined called "ShipmentTransaction". This "ShipmentTransaction" abstract object contains a single property which is a relationship to the Shipment object the transaction pertains to. Two of the transactions in this solution (TemperatureReading and ShipmentReceived) contain a reference or link to a particular Shipment - this commonality can be defined using a base or abstract Transaction which both TemperatureReading and ShipmentReceived inherit from. Abstract or base objects are defined using the "abstract" keyword.

```
44  /**
45   * An abstract transaction that is related to a Shipment
46   */
47  abstract transaction ShipmentTransaction {
48    --> Shipment shipment
49  }
50
```

Figure 27 - An abstract object definition.

As discussed above, the TemperatureReading and ShipmentReceived transactions both inherit from the ShipmentTransaction abstract object by using the "extends" keyword. The TemperatureReading transaction then extends the base definition by adding an additional property called "centigrade" which is used to store the recorded temperature. The net effect of this is the both the TemperatureReading and ShipmentReceived transactions will have a property which a link back to the shipment they pertain to, and the TemperatureReading transaction will contain one additional property, "centigrade".

```
44  /**
45   * An abstract transaction that is related to a Shipment
46   */
47  abstract transaction ShipmentTransaction {
48    --> Shipment shipment
49  }
50
51  /**
52   * An temperature reading for a shipment. E.g. received from a
53   * device within a temperature controlled shipping container
54   */
55  transaction TemperatureReading extends ShipmentTransaction {
56    o Double centigrade
57  }
58
59  /**
60   * A notification that a shipment has been received by the
61   * importer and that funds should be transferred from the importer
62   * to the grower to pay for the shipment.
63   */
64  transaction ShipmentReceived extends ShipmentTransaction {
65  }
66
```

Figure 28 - Inheriting from, and extending, a base class.

Properties in any object are expressed using a standardized syntax. Each property definition begins with a "-->" or "o". Using the "-->" indicates that this property is a relationship to another object in the solution. Using the "o" indicates that the property is an "attribute" (native to the object it's defined in) and not linked to any other object. Attributes can be defined using a primitive type or an object definition from the current solution. After a property definition is started with a "-->" or an "o" the data type of the property is specified. Finally, the name of the property is specified. This allows developers to access an object's property using the ObjectName.PropertyName syntax.

```
44  /**
45   * An abstract transaction that is related to a Shipment
46   */
47  abstract transaction ShipmentTransaction {
48    --> Shipment shipment
49  }
50
51  /**
52   * An temperature reading for a shipment. E.g. received from a
53   * device within a temperature controlled shipping container
54   */
55  transaction TemperatureReading extends ShipmentTransaction {
56    o Double centigrade
57  }
58
```

Figure 29 - Defining object properties.

Transactions are defined by using the "transaction" keyword, then specifying a name for the particular transaction being defined. Transactions do not require an identifier to be defined - Hyperledger automatically assigns a unique identifier to each transaction for you.

```
51  /**
52   * An temperature reading for a shipment. E.g. received from a
53   * device within a temperature controlled shipping container
54   */
55  transaction TemperatureReading extends ShipmentTransaction {
56    o Double centigrade
57  }
58
```

Figure 30 - Defining a Transaction.

Assets are defined using the "asset" keyword. Assets must contain a property which serves as a unique identifier. This property must be specified in the asset definition.

```
67  /**
68   * A shipment being tracked as an asset on the ledger
69   */
70  asset Shipment identified by shipmentId {
71    o String shipmentId
72    o ProductType type
73    o ShipmentStatus status
74    o Long unitCount
75    o TemperatureReading[] temperatureReadings optional
76    --> Contract contract
77  }
78
```

Figure 31 - Defining an Asset.

Attributes can be defined as arrays of objects using the "[]" syntax. The "Shipment" asset contains an array of TemperatureReading objects named "temperatureReadings". This attribute can contains one or more TemperatureReadings.

```
67  /**
68   * A shipment being tracked as an asset on the ledger
69   */
70  asset Shipment identified by shipmentId {
71    o String shipmentId
72    o ProductType type
73    o ShipmentStatus status
74    o Long unitCount
75    o TemperatureReading[] temperatureReadings optional
76    --> Contract contract
77  }
78
```

Figure 32 - Using an array as an object property.

Attributes can also be marked using the "optional" keyword. This keyword indicates that this attribute is optional and not required.

```
67  /**
68   * A shipment being tracked as an asset on the ledger
69   */
70  asset Shipment identified by shipmentId {
71    o String shipmentId
72    o ProductType type
73    o ShipmentStatus status
74    o Long unitCount
75    o TemperatureReading[] temperatureReadings optional
76    --> Contract contract
77  }
```

Figure 33 - Using the optional keyword on a property.

An object can use more than one links or references to other solution objects. The "Contract" asset uses this model - each Contract references a Grower, a Shipper, and an Importer.

```
79  /**
80   * Defines a contract between a Grower and an Importer to ship using
81   * a Shipper, paying a set unit price. The unit price is multiplied by
82   * a penality factor proportional to the deviation from the min and
    max
83   * negociated temperatures for the shipment.
84   */
85  asset Contract identified by contractId {
86    o String contractId
87    --> Grower grower
88    --> Shipper shipper
89    --> Importer importer
90    o DateTime arrivalDateTime
91    o Double unitPrice
92    o Double minTemperature
93    o Double maxTemperature
94    o Double minPenaltyFactor
95    o Double maxPenaltyFactor
96  }
```

Figure 34 - Defining multiple object references as properties is allowed.

Concepts are abstract classes that are not assets, participants or transactions. They are typically contained by an asset, participant or transaction. In the perishable network example an Address concept is defined. This concept contains optional data fields for city, street, and zip, as well as one required data field for country. Concepts are useful when a data structure will be used in multiple object types.

```
 98  /**
 99   * A concept for a simple street address
100   */
101  concept Address {
102    o String city optional
103    o String country
104    o String street optional
105    o String zip optional
106  }
107
```

Figure 35 - Defining a Concept in Composer.

In the sample solution, the Address concept is used in the abstract participant definition of a Business. This abstract definition is inherited by the Grower participant. This means that the Grower participant will contain the following properties (not comprehensive):

`Grower.address.city`

`Grower.address.country`

```
 98  /**
 99   * A concept for a simple street address
100   */
101  concept Address {
102    o String city optional
103    o String country
104    o String street optional
105    o String zip optional
106  }
107
108  /**
109   * An abstract participant type in this business network
110   */
111  abstract participant Business identified by email {
112    o String email
113    o Address address
114    o Double accountBalance
115  }
116
117  /**
118   * A Grower is a type of participant in the network
119   */
120  participant Grower extends Business {
121  }
122
```

Figure 36 - Using the address concept in the participant base class.

Lab 2 – Reviewing the Logic (logic.js) file

In this lab we'll take a look at the logic / transaction processor file. This file contains the code that implements each of the transactions defined in the model file.

Step 1 – Open up the logic file

The "Script File" (logic.js) contains the code that will be executed anytime a transaction defined in the model file is submitted. In other words, the model file defines what the transactions are, while the logic file defines what each transaction actually does. To view the logic file, click on the "Script File" section on the left-hand side of the Composer Playground window.

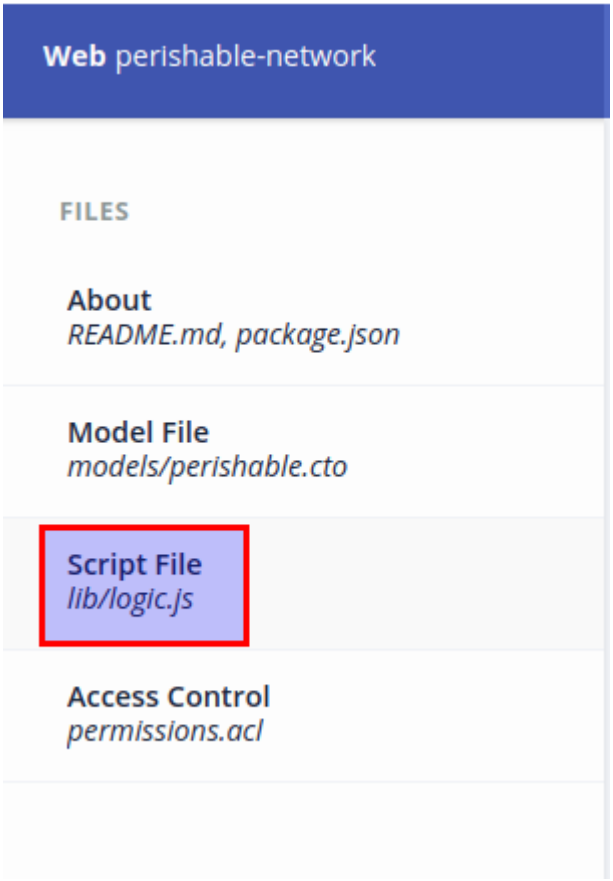


Figure 37 - Opening the logic file.

Step 2 – Review the function definition

Each transaction function (transaction processor) takes one input parameter which is passed in by Hyperledger. This parameter represents the transaction object as defined in the model file. The input parameter is defined using the @param syntax, followed by the object type, and a name for the object to reference in the function.

```
17  /**
18   * A shipment has been received by an importer
19   * @param {org.acme.shipping.perishable.ShipmentReceived} shipmentReceived
    - the ShipmentReceived transaction
20   * @transaction
21   */
22  async function payOut(shipmentReceived) {  // eslint-disable-line no-
    unused-vars
```

Step 3 – Accessing object properties

In the model file, a ShipmentTransaction transaction object simply extends the ShipmentTransaction abstract definition. ShipmentTransaction contains only one property, a link to the associated shipment. This shipment property, and all of its properties, are accessible via the "shipmentReceived" object.

```
44  /**
45   * An abstract transaction that is related to a Shipment
46   */
47  abstract transaction ShipmentTransaction {
48    --> Shipment shipment
49  }
50
```

Figure 38 - The base class definition in the model file.

```
59  /**
60   * A notification that a shipment has been received by the
61   * importer and that funds should be transferred from the importer
62   * to the grower to pay for the shipment.
63   */
64  transaction ShipmentReceived extends ShipmentTransaction {
65  }
66
```

Figure 39 - The definition of the ShipmentReceived object in the model file.

```
22  async function payOut(shipmentReceived) {  // eslint-disable-
    unused-vars
23
24      const contract = shipmentReceived.shipment.contract;
25      const shipment = shipmentReceived.shipment;
26      let payOut = contract.unitPrice * shipment.unitCount;
27
```

Figure 40 - Accessing properties of the shipmentReceived's shipment property.

Step 4 – Updating an object and saving it back to the registry

At the end of the "payOut" function, the balance for the grower and importer, as well as the state of the shipment are all updated. Growers and importers are both participants, while shipments are assets. The collection of growers and importers can be accessed via the participant registry (getParticipantRegistry in code) - the collection of assets is similarly accessed from the asset registry (getAssetRegistry). Each call to get a registry expects a type definition, indicating the type of object to be returned. To update the object affected by the transaction, simply pass it into the update function attached to each registry. This will update this object on the registry.

```
79      // update the grower's balance
80      const growerRegistry = await
    getParticipantRegistry('org.acme.shipping.perishable.Grower');
81      await growerRegistry.update(contract.grower);
82
```

Figure 41 - Saving an updated object back to the registry.

Step 5 – Creating new objects and adding them to the registry

The setupDemo transaction processor function is used to populate the solution with test data. Examining the code in this function will reveal the process for creating new objects and adding them to the proper registries. The process begins by creating a reference to a "factory" object; the factory object is used to create instances of objects (assets, participants, and transactions). This is done by calling the "getFactory" method which returns a factory object for use in creating new objects. Note that a constant has also been defined ("NS") to reference the targeted namespace - you will see this "NS" constant used throughout this function.

```
113
114  /**
115   * Initialize some test assets and participants useful for running a demo.
116   * @param {org.acme.shipping.perishable.SetupDemo} setupDemo - the
     SetupDemo transaction
117   * @transaction
118   */
119  async function setupDemo(setupDemo) {  // eslint-disable-line no-unused-
     vars
120
121      const factory = getFactory();
122      const NS = 'org.acme.shipping.perishable';
123
```

Figure 42 - Getting a Factory reference and creating a constant for the namespace being used.

Creating a new object is quite simple. A new grower object can be obtained from the factory using the "newResource" method. This method expects inputs of a namespace, the type of object to create within that namespace, and a unique identifier (if required). In this example note the use of the "newConcept" method as well. Remember that growers implement the address concept via their base class. A concept reference must be obtained from the factory to set any properties contained within that concept.

```
124      // create the grower
125      const grower = factory.newResource(NS, 'Grower', 'farmer@email.com');
126      const growerAddress = factory.newConcept(NS, 'Address');
127      growerAddress.country = 'USA';
128      grower.address = growerAddress;
129      grower.accountBalance = 0;
130
```

Figure 43 - Creating a new object using the Factory.

Once created, adding a new grower to the participant registry is easy. Simply obtain a reference to the proper registry (in this case the participant registry, but this pattern holds true for assets and transactions as well), and use the "addAll" method to add one or more objects of the specified type.

```
166       // add the growers
167       const growerRegistry = await getParticipantRegistry(NS + '.Grower');
168       await growerRegistry addAll([grower]);
```

Figure 44 - Adding the new object to the registry.

Step 6 - Deleting Items from the Registry, and other supported functions.

Although not shown in the perishable network example, it's still quite easy to remove objects from a registry as well as check for their existence. All registry objects in Composer inherit from a base "Registry" class. The supported methods of this class are outlined below. Take a moment to familiarize yourself with these methods, and remember they've been documented here should you need to reference them later in your development activities.

Method Summary

Name	*Returns*	*Description*
add	Promise	Adds a new resource to the registry
addAll	Promise	Adds a list of new resources to the registry
exists	Promise	Determines whether a specific resource exists in the registry
get	Promise	Get a specific resource in the registry
getAll	Promise	Get all of the resources in the registry
remove	Promise	Remove an asset with a given type and id from the registry
removeAll	Promise	Removes a list of resources from the registry
resolve	Promise	Get a specific resource in the registry, and resolve all of its relationships to other assets, participants, and transactions
resolveAll	Promise	Get all of the resources in the registry, and resolve all of their relationships to other assets, participants, and transactions
update	Promise	Updates a resource in the registry
updateAll	Promise	Updates a list of resources in the registry

Lab 3 – Reviewing the Permissions (.acl) file

Composer allows you to define two different types of permissions described below.

- Business Access Control
    - Access control for resources within a business network.
- Network Access Control
    - Access control for network administrative changes.

In Composer solutions, both types of permissions are defined in the Access Control file (.acl). In this lab we'll be taking a look at the access control file supplied with the perishable network sample. We'll also be reviewing some of the concepts around access control in Composer.

Step 1 - Viewing the access control file

From the "Define" tab in Composer Playground, select the "Access Control" (permissions.acl) file from the left-hand navigation menu.

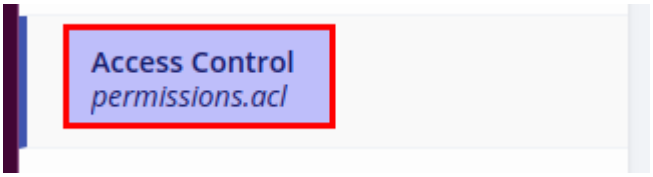


Figure 45 - Viewing the Access Control file.

Step 2 - Viewing the solution rules - Default

The default .acl file included with this solution template outlines four very simple rules which we'll cover over the next few steps. It is important to remember in Composer that all access is denied unless otherwise specified.

Start with the first rule, "Default". This rule allows any participant ("participant: "ANY"") in the solution the ability to perform all operations ("operation: ALL" and "action: ALLOW") on anything in the namespace "org.acme.shipping.perishable.*".

This rule opens up permissions on all solution components to all users. This is a useful pattern to follow when creating and testing proof-of-concepts.

```
15  /**
16   * Sample access control list.
17   */
18  rule Default {
19      description: "Allow all participants access to all resources"
20      participant: "ANY"
21      operation: ALL
22      resource: "org.acme.shipping.perishable.*"
23      action: ALLOW
24  }
25
```

Figure 46 - The Default rule.

Step 3 - Viewing the solution rules - SystemACL

The SystemACL rule grants permission to the composer system itself to access all resources within the system namespace. This is a fairly typical approach although restricted system access to system functions can be a useful way to 'disable' unneeded or unwanted Composer features.

```
26  rule SystemACL {
27      description:  "System ACL to permit all access"
28      participant: "org.hyperledger.composer.system.Participant"
29      operation: ALL
30      resource: "org.hyperledger.composer.system.**"
31      action: ALLOW
32  }
33
```

Figure 47 - The SystemACL rule.

Step 4 - Viewing the solution rules - NetworkAdminUser

The NetworkAdminUser rule allows all network administrators to have full access to all the solution components. This is a useful pattern when developing and testing proof-of-concept solutions as it allows network administrators to help debug and troubleshoot potential configuration and deployment issues.

```
34  rule NetworkAdminUser {
35      description: "Grant business network administrators full access to user resources"
36      participant: "org.hyperledger.composer.system.NetworkAdmin"
37      operation: ALL
38      resource: "**"
39      action: ALLOW
40  }
```

Figure 48 - The NetworkAdminUser rule.

Step 5 - Viewing the solution rules - NetworkAdminSystem

The final rule, NetworkAdminSystem, grants all network administrators access to all system administrative functions. This configuration is typical although restricting access with this rule could be an effective way to separate network admin roles and responsibilities.

```
41
42  rule NetworkAdminSystem {
43      description: "Grant business network administrators full access to system resources"
44      participant: "org.hyperledger.composer.system.NetworkAdmin"
45      operation: ALL
46      resource: "org.hyperledger.composer.system.**"
47      action: ALLOW
48  }
```

Step 6 - Simple vs. Conditional rules

Composer allows you to define rules in two different formats - the simple rule and the conditional rule. Simple rules are used to control access to a namespace or asset by a participant type or participant instance. Conditional rules introduce variable bindings for the participant and the resource being accessed, and a Boolean JavaScript expression, which, when true, can either ALLOW or DENY access to the resource by the participant. Here's an example of each:

```
rule SimpleRule {
    description: "Description of the ACL rule"
    participant: "org.example.SampleParticipant"
    operation: ALL
    resource: "org.example.SampleAsset"
    action: ALLOW
}
```

Figure 49 -

This rule states that any instance of the org.example.SampleParticipant type can perform all operations on all instances of org.example.SampleAsset. NOTE - This rule is provided as a sample, and does NOT exist in the perishable-network solution demo.

```
rule SampleConditionalRule {
    description: "Description of the ACL rule"
    participant(m) "org.example.SampleParticipant"
    operation: ALL
    resource(v): "org.example.SampleAsset"
    condition: (v.owner.getIdentifier() == m.getIdentifier())
    action: ALLOW
}
```

Figure 50 -

In contrast to above, this rule states that any instance of the org.example.SampleParticipant type can perform ALL operations on all instances of org.example.SampleAsset ONLY IF the participant is the owner of the asset. Notice that variable names are declared in parenthesis next to the objects to be evaluated in the "condition" clause. NOTE - This rule is provided as a sample, and does NOT exist in the perishable-network solution demo.

Step 7 - More on Conditional Rules

Any conditional rule you create can also include an optional transaction clause. If the transaction clause is used, the effect is that the rule only allows access to the resource by the participant if the participant submitted a transaction, AND that transaction is of the type specified in the transaction clause.

```
rule SampleConditionalRuleWithTransaction {
    description: "Description of the ACL rule"
    participant(m): "org.example.SampleParticipant"
    operation: READ, CREATE, UPDATE
    resource(v): "org.example.SampleAsset"
    transaction(tx): "org.example.SampleTransaction"
    condition: (v.owner.getIdentifier() == m.getIdentifier())
    action: ALLOW
}
```

Figure 51 -

In this example rule, any instance of the org.example.SampleParticipant type can perform ALL operations on all instances of org.example.SampleAsset IF the participant is the owner of the asset AND the participant submitted a transaction of the org.example.SampleTransaction type to perform the operation. If either of those two conditions are not met, access will be denied. NOTE - This rule is provided as a sample, and does NOT exist in the perishable-network solution demo.

Step 8 - The "Resource" Clause

The "Resource" clause identifies the object that access is being granted (or denied) to. Resource clauses can specify a class, all classes within a given namespace, all classes under a given namespace, or an instance of a class. For example, all of the resource clauses below are valid:

- resource: "org.example.*"
    - Used to identify a namespace
- resource: "org.example.**"
    - Used to identify all namespaces under org.example
- resource: "org.example.Car"
    - Used to identify a particular class within a namespace
- resource: "org.example.Car#ABC123"
    - Used to identify a specific instance of an object

Step 9 - The "Operation" Clause

The "Operation" clause defines all the actions the rule applies to. There are five supported actions, listed below. If the ALL option is not used, comma-separated list can be provided.

- ALL
  - Implies all functions are allowed (create, read, update, and delete).
- CREATE
  - New items can be created.
- READ
  - Existing items can be viewed.
- UPDATE
  - Existing items can be updated.
- DELETE
  - Existing items can be deleted.

Step 10 - The "Participant" Clause

The "Participant" clause defines the entity or individual who has submitted a transaction in Composer. Any participant specified must exist in the participant registry. The value of "ANY" may be specified to indicate that all participants should be treated according to the conditions of the current rule.

Step 11 - The "Condition" Clause

The "Condition" clause specifies the conditions that must be met in order for access to be granted or denied. A Condition is any boolean JavaScript expression. Any JavaScript expression that is legal within an "if" statement may be used. JavaScript expressions used for the condition of an ACL rule can refer to JavaScript utility functions in a script file. This allows a user to easily implement complex access control logic, and re-use the same access control logic functions across multiple ACL rules.

Step 12 - The "Action" clause

The Action clause defines the action the rule is taking. There are two possible values:

- ALLOW
- DENY

Lab 4 - The Transaction Log

In this lab we'll be taking a look at the transaction log. The transaction log keeps a permanent record of all transactions, even transactions against assets which no longer exist.

NOTE: If you are using a VirtualBox lab image provided by Blockchain Training Alliance please run the commands below in a command prompt before starting this lab!

```
cd ~/fabric-dev-servers/
./startFabric.sh
./createPeerAdminCard.sh
exit
```

Step 1 - Create a sample Grower participant

From the Composer Playground environment, click on the "Test" tab at the top then select the Grower participant.

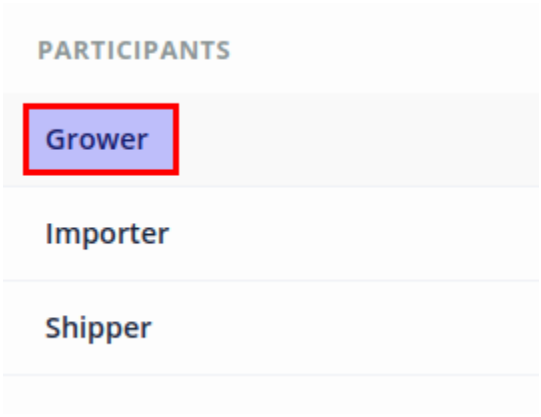


Figure 52 - Select the Grower view.

Select the option to "Create New Participant" from the top-right hand corner of the environment.

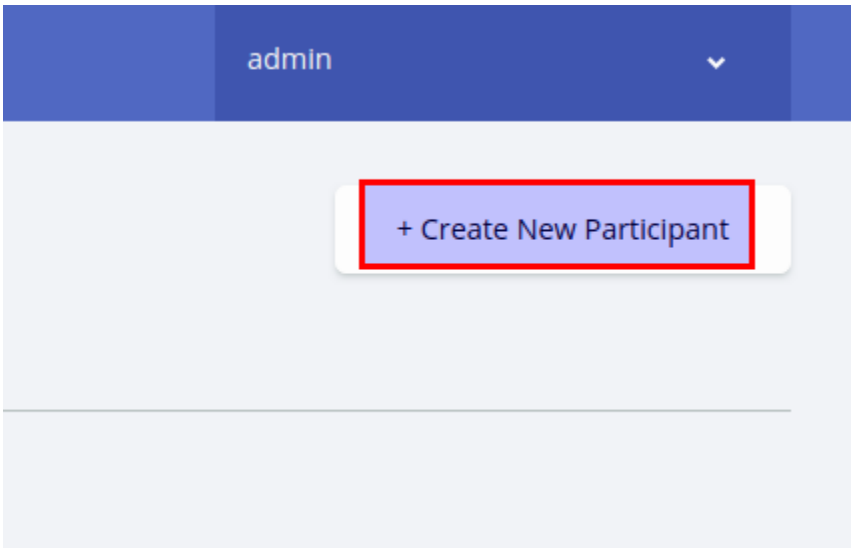


Figure 53 - Creating a new Grower.

Populate the JSON Grower object using the values displayed below, then click on the "Create New" button.

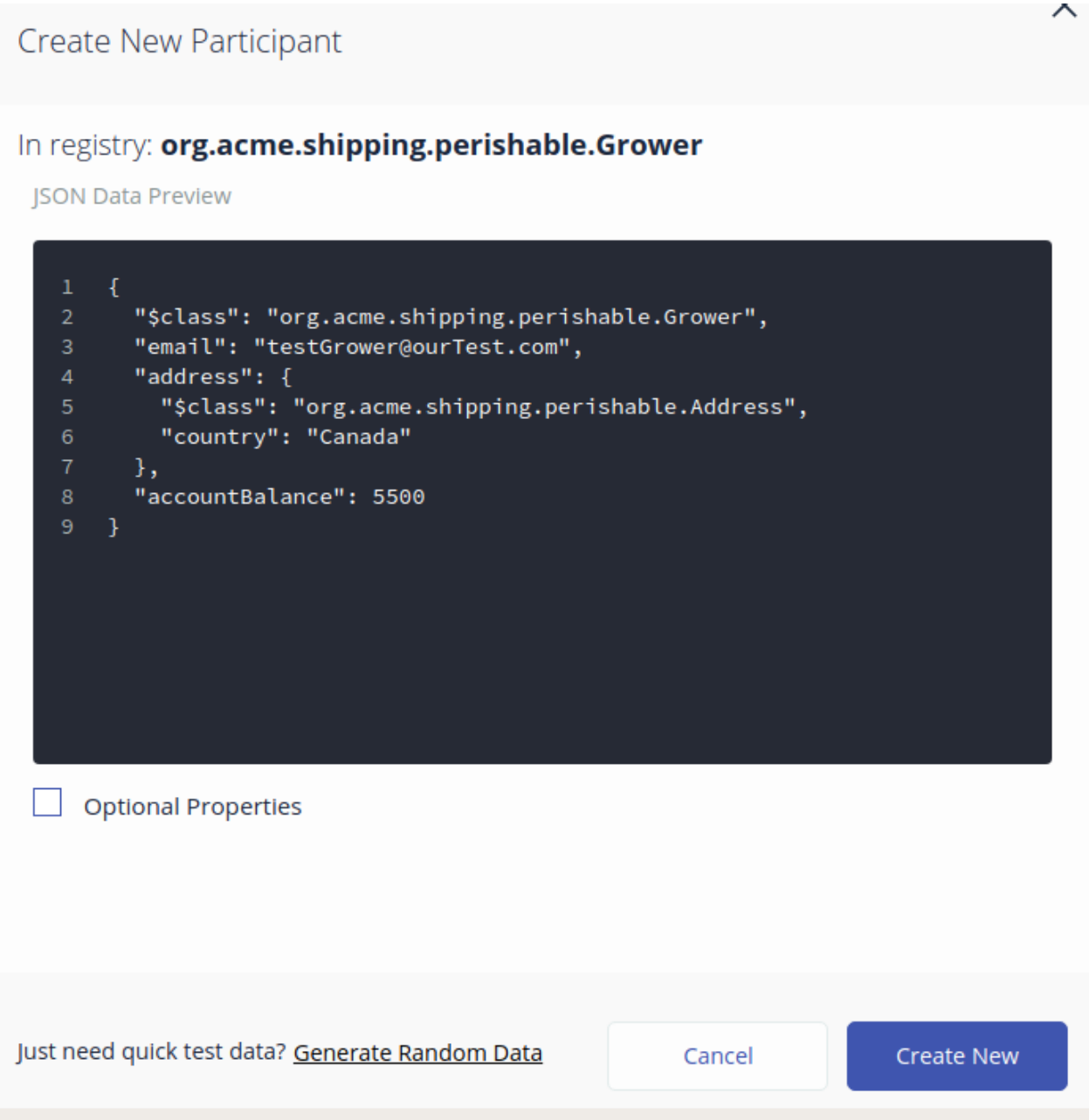


Figure 54 - Adding details about the new Grower.

Step 2 - Open up the transaction log

Select the "All Transactions" option under the "TRANSACTIONS" section.

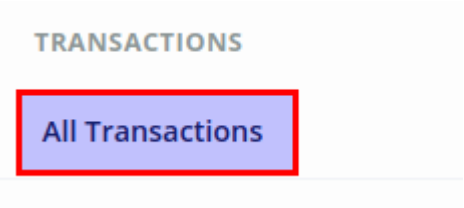


Figure 55 - Open the transaction log.

Step 3 - View a transaction

The transaction log lists all transactions, displays their type, includes a date and time stamp, and identifies any involved participants. At the top of the list you should see the AddParticipant transaction performed in Step 1 on this lab. Click on the "view record" option to view the transaction in detail.

Figure 56 - Viewing a transaction.

Step 4 - Viewing transaction details

After clicking the "view record" option you should see a JSON record of the transaction. You should see the following JSON record, although your transactionId and timestamp values will differ.


```
{
 "$class": "org.hyperledger.composer.system.AddParticipant",
 "resources": [
  {
   "$class": "org.acme.shipping.perishable.Grower",
   "email": "testGrower@ourTest.com",
   "address": {
    "$class": "org.acme.shipping.perishable.Address",
    "country": "Canada"
   },
   "accountBalance": 5500
  }
 ],
 "targetRegistry":
"resource:org.hyperledger.composer.system.ParticipantRegistry#org.acme
.shipping.perishable.Grower",
 "transactionId": "00000000-0000-0000-0000-000000000000",
 "timestamp": "20XX-XX-XXTXX:XX:XX.XXXZ"
}
```

Figure 57 - Viewing the transaction detail.

Step 5 - Delete the new Grower participant

Click on the "Grower" option under the "PARTICIPANTS" section of the Composer Playground environment. Once in the Grower section, find the record we just created for testGrower@ourTest.com and select the option to delete the record.

Figure 58 - Deleting the Grower.

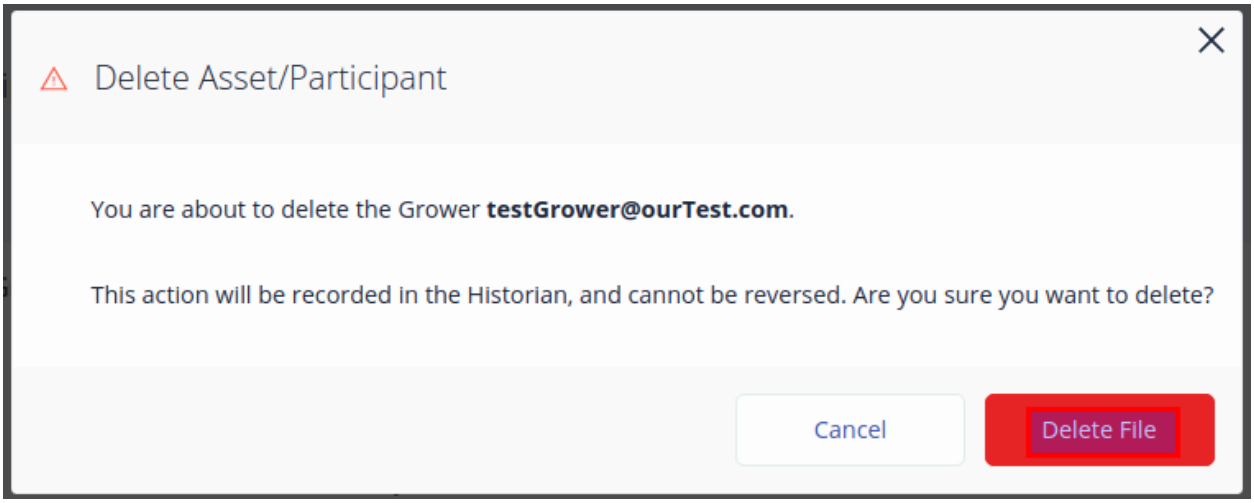


Figure 59 - Confirm the deletion.

Step 6 - View the deletion transaction in the log

Return to the "All Transaction" view under the "TRANSACTIONS" section of Composer Playground. You should see a RemoveParticipant Transaction. Click on the "view record" option to view the details. Note that the transaction log keeps permanent details of each transaction and the objects effected.

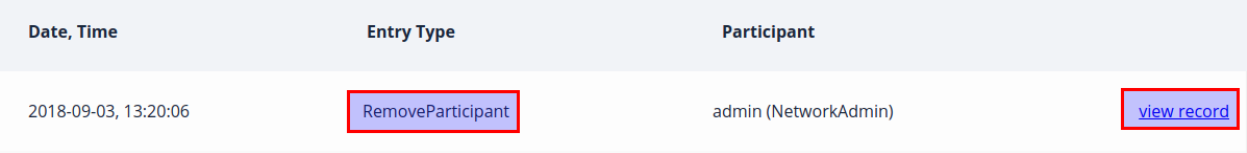


Figure 60 - Viewing the deletion transaction.

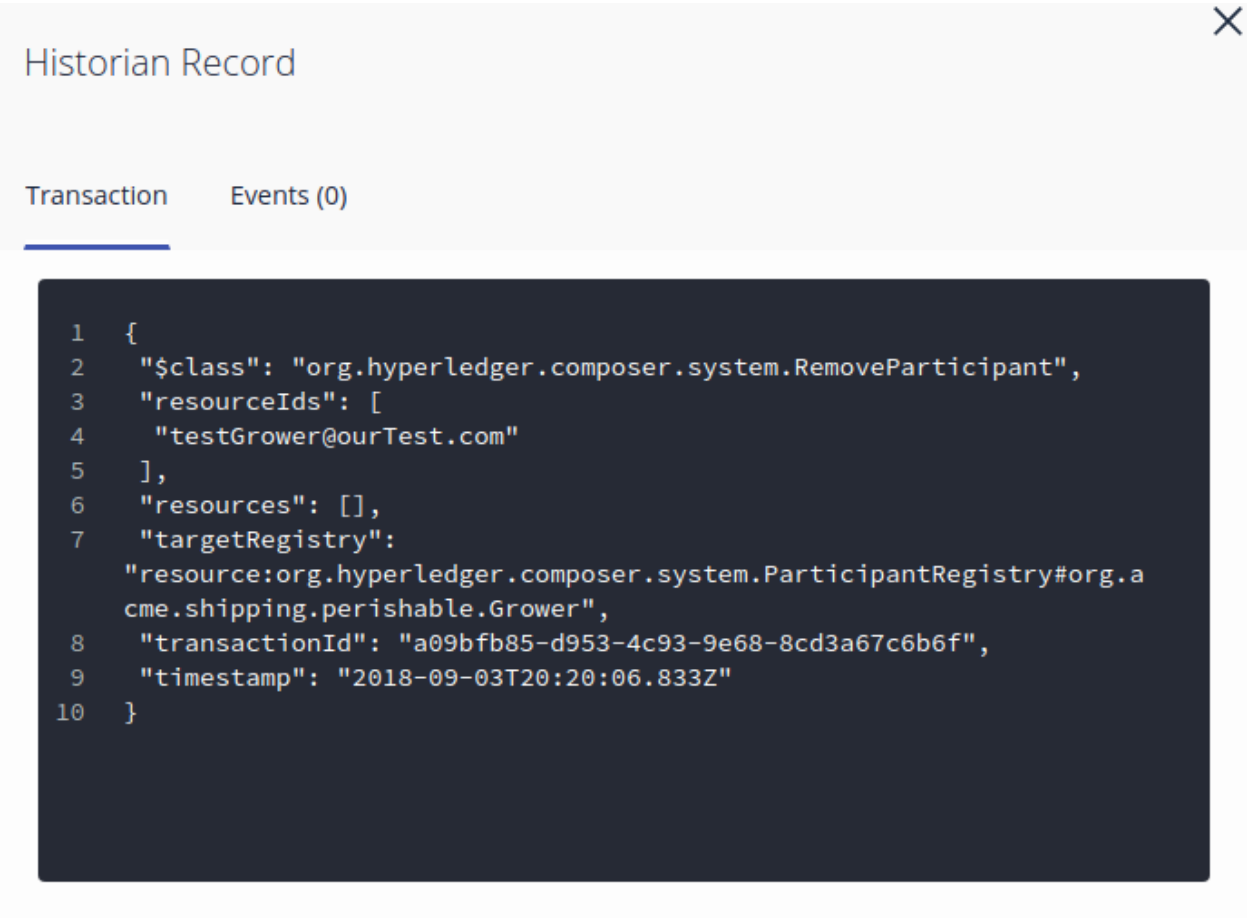


Figure 61 - Viewing the transaction detail.

Lab 5 - Updating and Re-Deploying

In this lab we're going to make changes to our model file, then re-deploy the solution and view the effects.

Step 1 - Open up the model file

From the Composer Playground environment, click on the "Define" tab at the top, then select the model file (perishable.cto) from the side navigation menu.

Step 2 - Add a new field to Grower

From the model file, locate the definition for the Grower participant (lines 117-121). Add a single attribute to the definition of a Grower. The data type should be string, and the property should be called websiteURL.

```
117  /**
118   * A Grower is a type of participant in the network
119   */
120  participant Grower extends Business {
121  }
122
```

Figure 62 - The Grower object BEFORE any changes are made.

```
117  /**
118   * A Grower is a type of participant in the network
119   */
120  participant Grower extends Business {
121    o String websiteURL
122  }
123
```

Figure 63 - The Grower object AFTER being updated.

Step 3 - Re-deploy the updated solution

Check the debugger at the bottom of the window to make sure there are no code problems before deploying.

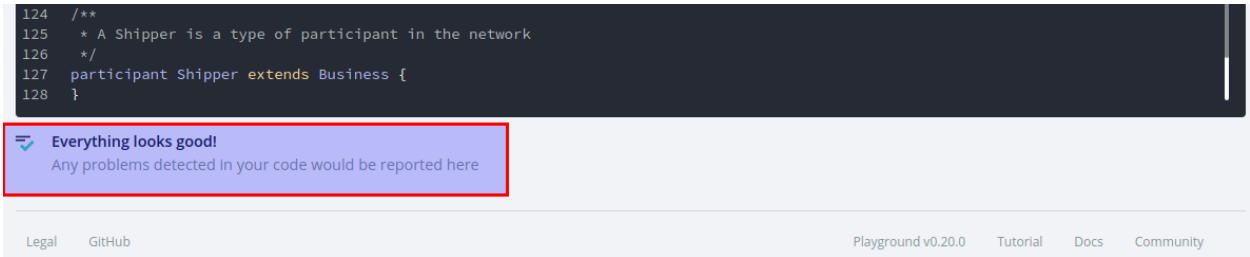


Figure 64 - Confirm the solution contains no errors.

If no problems are reported, click the "Deploy changes" button located in the bottom-left hand corner of the application environment.

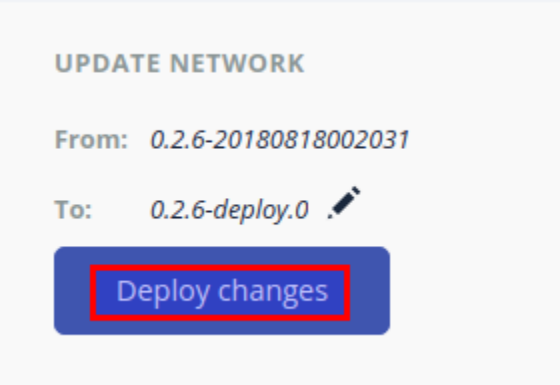


Figure 65 - Deploying the updated solution.

You should see a confirmation message displayed after deploying.

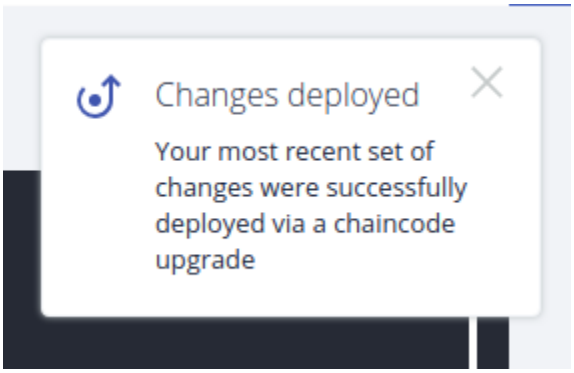


Figure 66 - Confirmation of a successful deployment.

Step 4 - Create a new Grower

After deploying your changes, click on the "Test" tab, then select the Grower section. Create a new Grower by selecting the "Create New Participant" button. You should now see the websiteURL field listed in the Grower JSON object. Create a new grower with a website URL value.

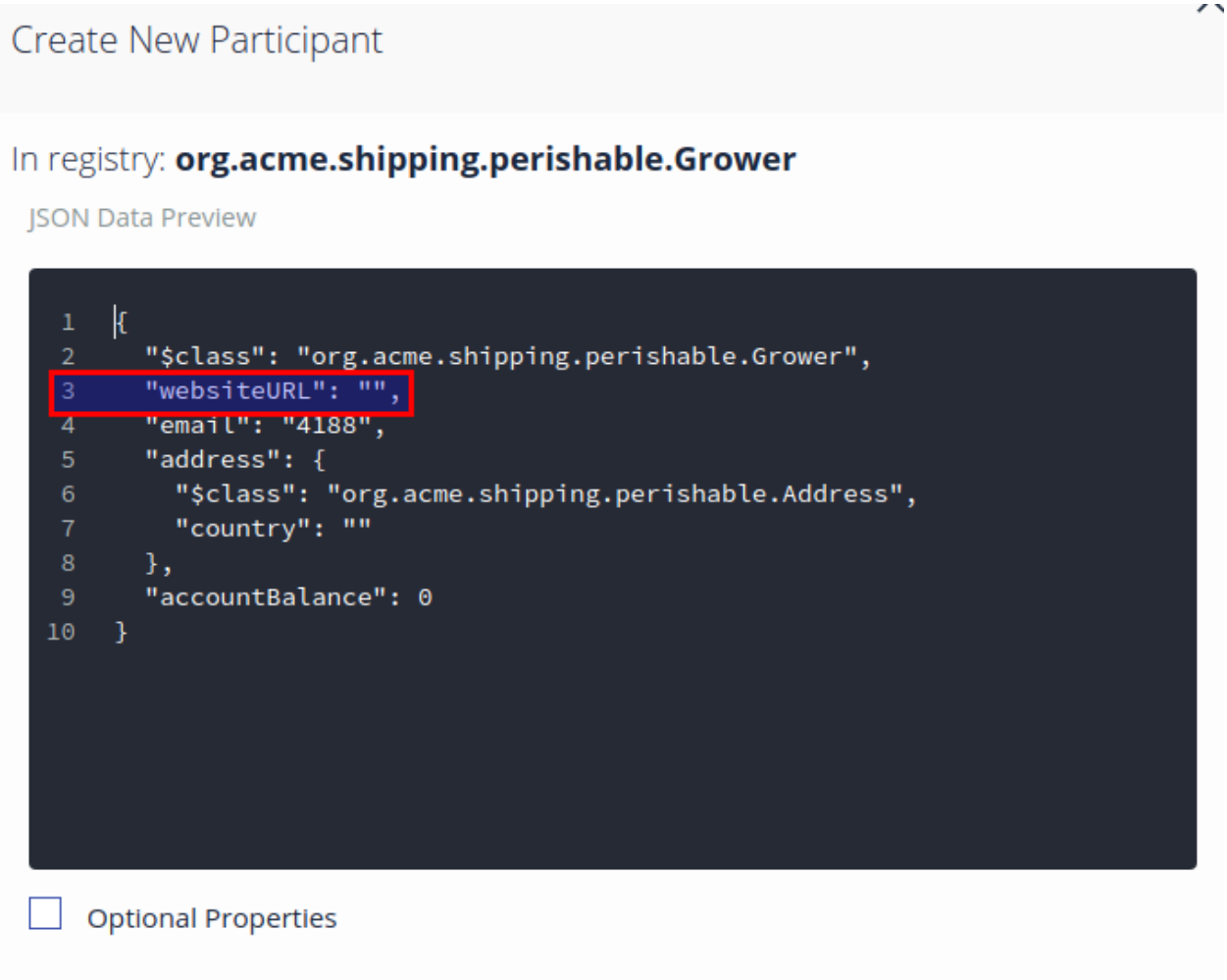


Figure 67 - The updated Grower object.

Section 3 - Solution Packaging and Deployment

In this section we'll take a look at the steps involved in packaging up a solution from Composer Playground and deploying it into a Fabric / Composer environment. We'll also take a look at the REST API generated for us by Composer and build a sample Angular application against our API using the Yeoman tool.

Lab 1 - Package and migrate a Composer Playground solution

In this lab we'll look at the steps required to package up a Composer Playground solution for deployment in a Fabric / Composer environment.

Step 1 - Creating a Business Network Archive (.bna) file

From the Composer Playground environment, select the "Define" tab at the top of the window. Look for the "Export" button near the bottom of the left-hand side navigation elements. Select this Export button and you will notice a .bna file being downloaded by your browser.

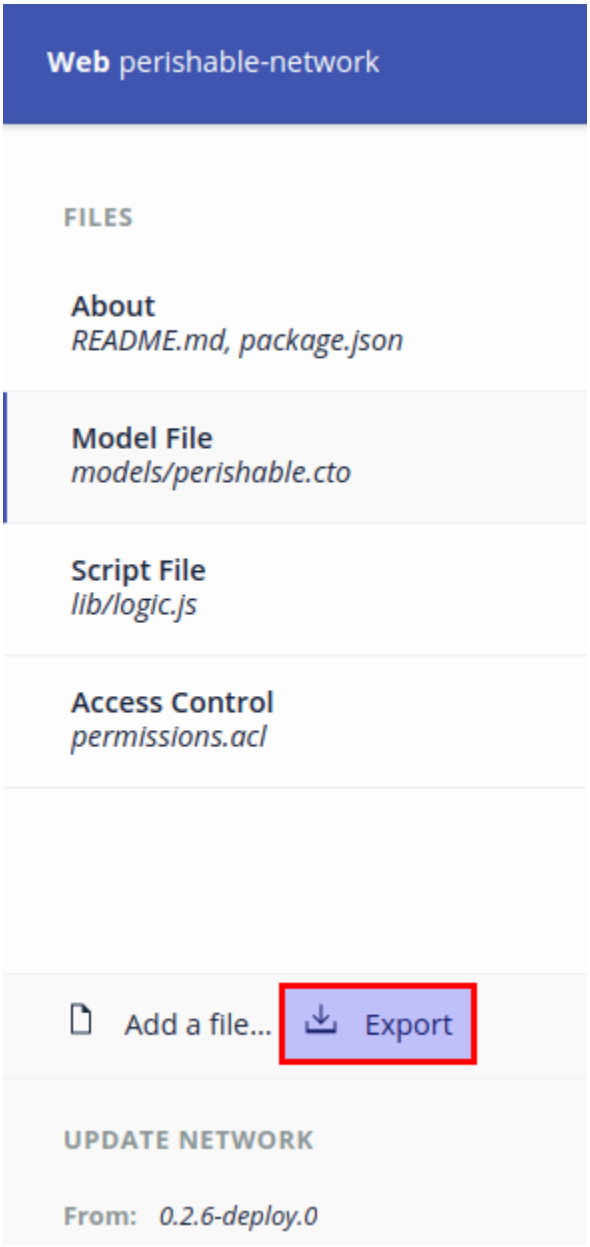


Figure 68 - The Export function in Composer Playground.

Figure 69 - The downloaded .bna file.

Step 2 - Creating a solutions folder

From the command bar on your desktop, select the "Files" tool.

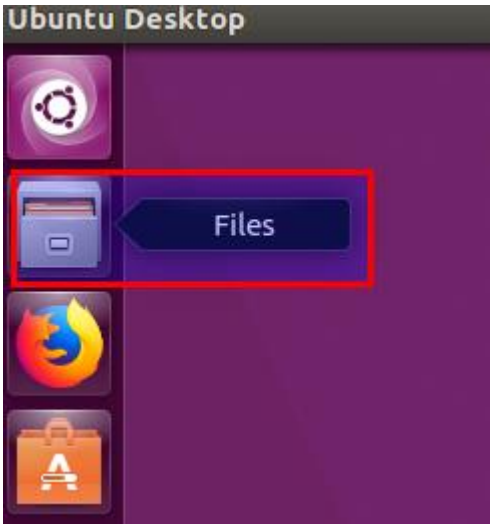


Figure 70 - Open the Files tool.

Select the "Home" folder from the left-hand side navigation.

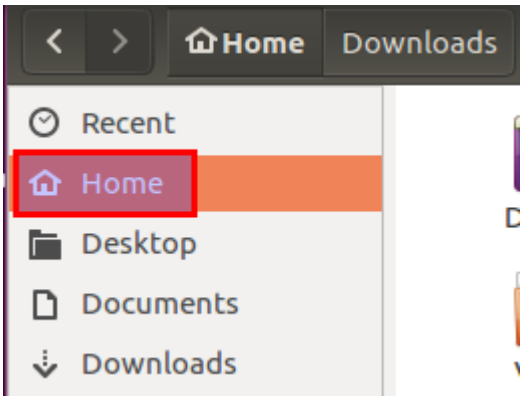


Figure 71 - Select the Home folder

Right-click on the files tool and select the "New Folder" option from the flyout menu. Name the new folder "perishable-network".

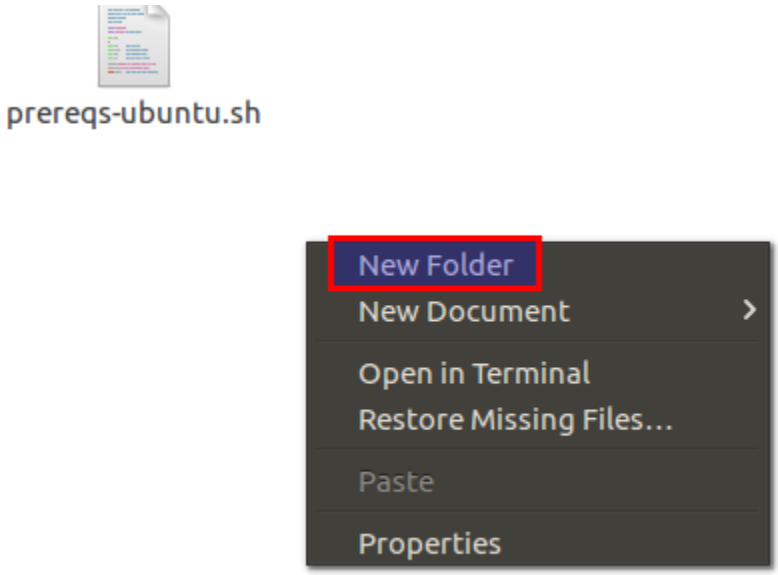


Figure 72 - Creating a new folder.

Figure 73 - The newly created folder.

Step 3 - Move the .bna file into the perishable-network folder

Select the "Downloads" folder using the left-hand navigation. From the Downloads folder, right-click on the perishable-network.bna file and select the "Cut" option.

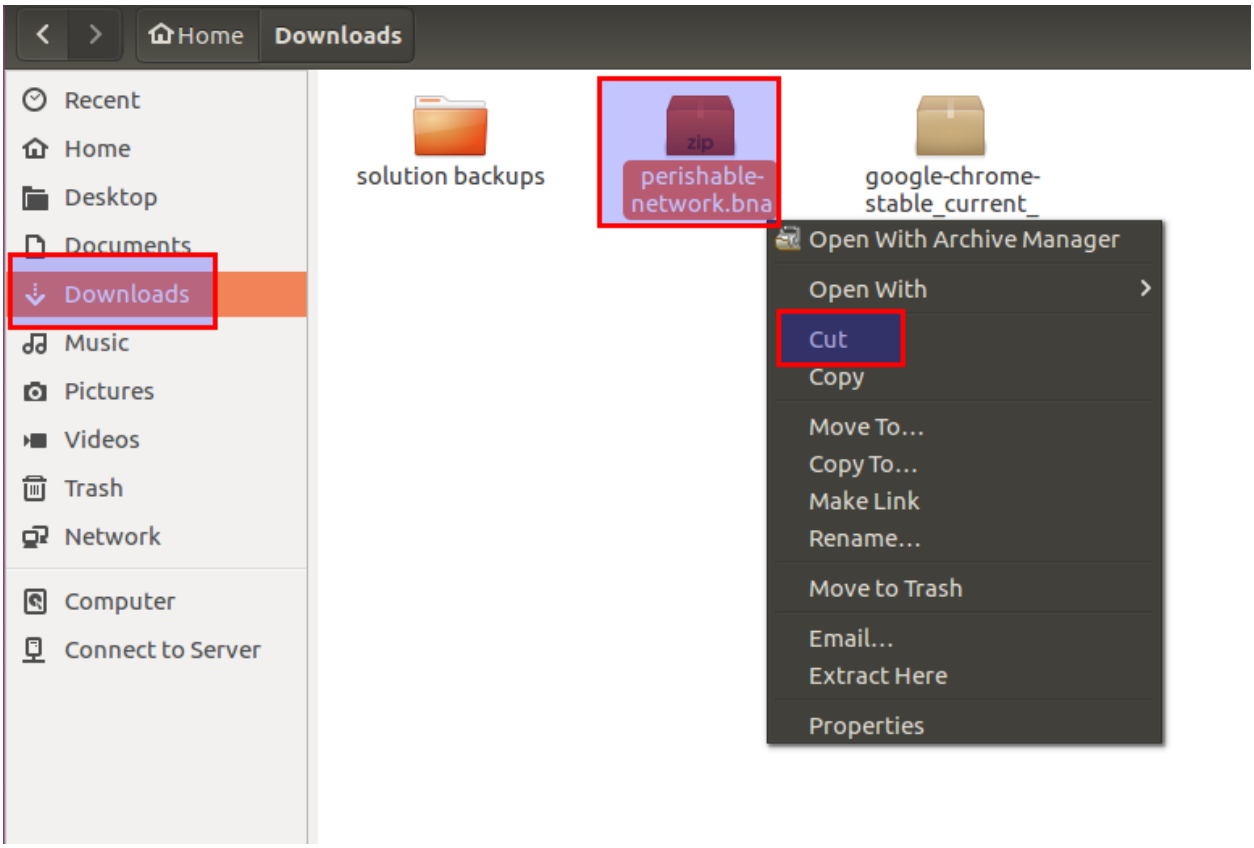


Figure 74 - Cutting the .bna file from the Downloads directory.

Return to the perishable-network folder under Home. Right-click on the files tool and select the "Paste" option to move the .bna file. The .bna file should now reside in the perishable-network folder.

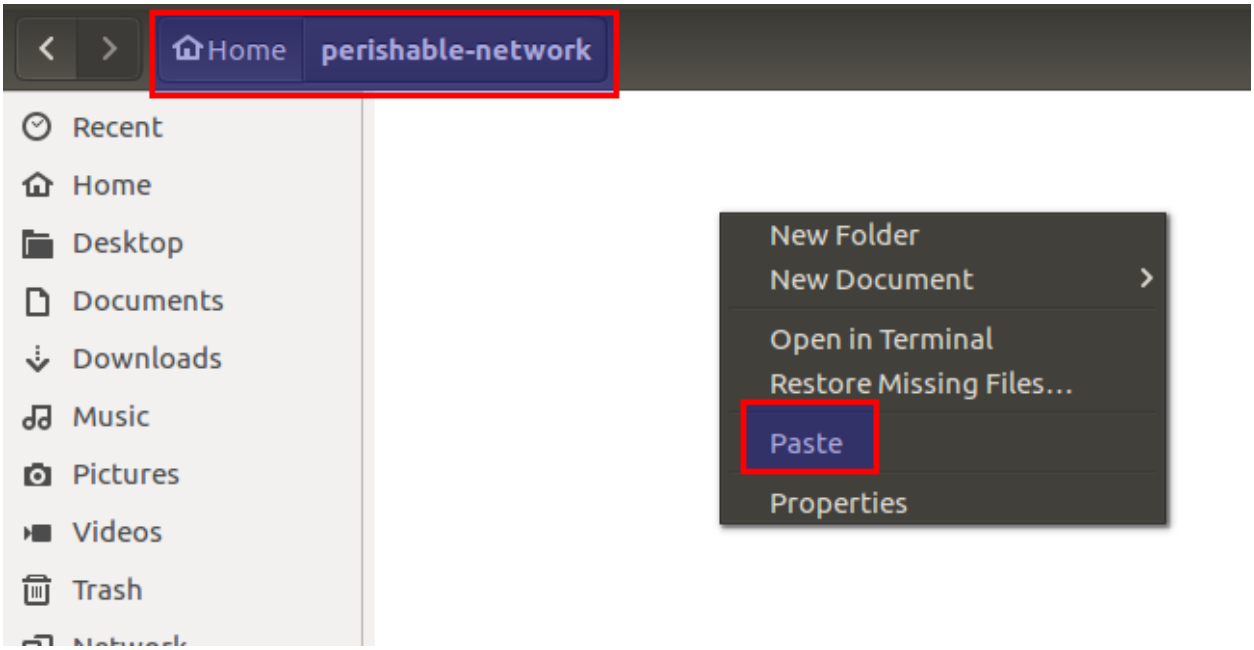


Figure 75 - Pasting the .bna file.

Step 4 - Start the Fabric environment

Open up a terminal and navigate to the fabric-dev-servers folder using the following commands:

```
export PATH=/bin:/usr/bin:/usr/local/bin:/sbin:/usr/sbin
```

```
cd ~/fabric-dev-servers/
```

From the fabric dev servers folder, run the startFabric script using the following command:

```
./startFabric.sh
```

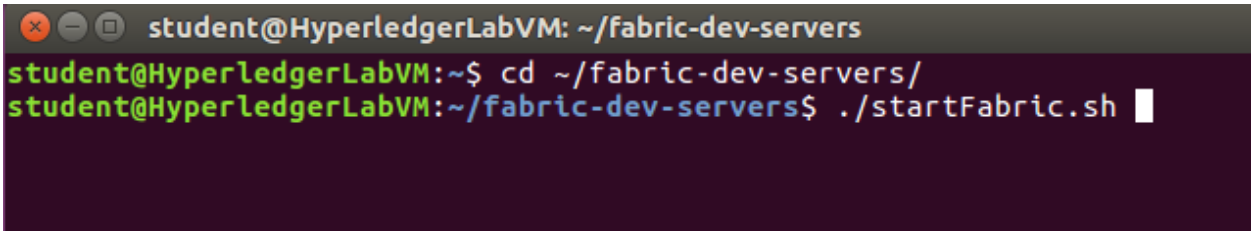


Figure 76 - Running the startFabric script.

Step 5 - Deploy the solution to your Fabric / Composer environment

Navigate to the perishable-network folder by running the following command:

```
cd ~/perishable-network/
```

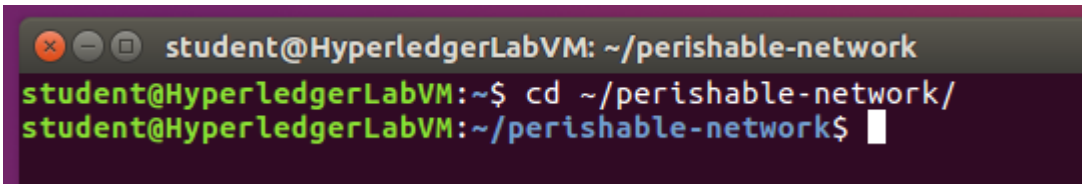


Figure 77 - Navigating to the perishable-network directory.

To install the archived solution, enter the following command:

```
composer network install --card PeerAdmin@hlfv1 --archiveFile
perishable-network.bna
```

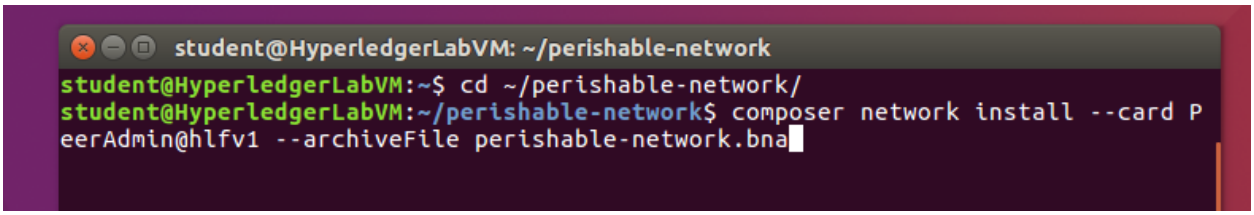


Figure 78 - Running the command.

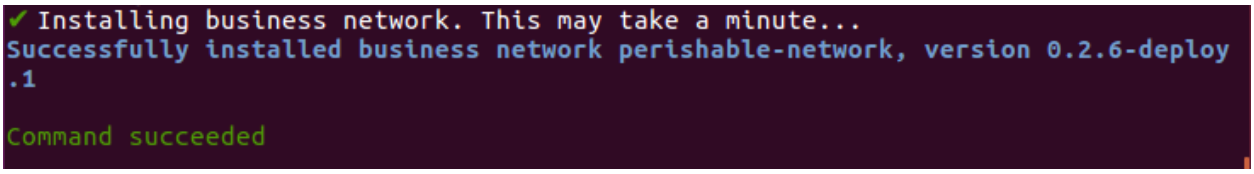


Figure 79 - Successful completion!

Step 6 - Start the solution

To start the solution, use the "composer network start" command. Enter the command below:

```
composer network start --networkName perishable-network --
networkVersion 0.2.6-deploy.1 --networkAdmin admin --
networkAdminEnrollSecret adminpw --card PeerAdmin@hlfv1 --file
networkadmin.card
```

```
student@HyperledgerLabVM:~/perishable-network$ composer network start --networkN
ame perishable-network --networkVersion 0.2.6-deploy.1 --networkAdmin admin --ne
tworkAdminEnrollSecret adminpw --card PeerAdmin@hlfv1 --file networkadmin.card
```

Figure 80 - Running the command.

```
Starting business network perishable-network at version 0.2.6-deploy.1

Processing these Network Admins:
        userName: admin

✔ Starting business network definition. This may take a minute...
Successfully created business network card:
        Filename: networkadmin.card

Command succeeded
```

Figure 81 - Successful completion!

Step 7 - Import the Network Admin identity

To import the network administrator identity as a usable business network card, run the command below. The composer card import command requires the filename specified in composer network start command to create a card.

```
composer card import --file networkadmin.card
```

```
student@HyperledgerLabVM:~/perishable-network$ composer card import --file netwo
rkadmin.card

Successfully imported business network card
        Card file: networkadmin.card
        Card name: admin@perishable-network

Command succeeded
```

Figure 82 - Running the command.

Step 8 - Verify the Business Network is running

To check that the business network has been deployed successfully, run the command below to ping the network. The composer network ping command requires a business network card to identify the network to ping.

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

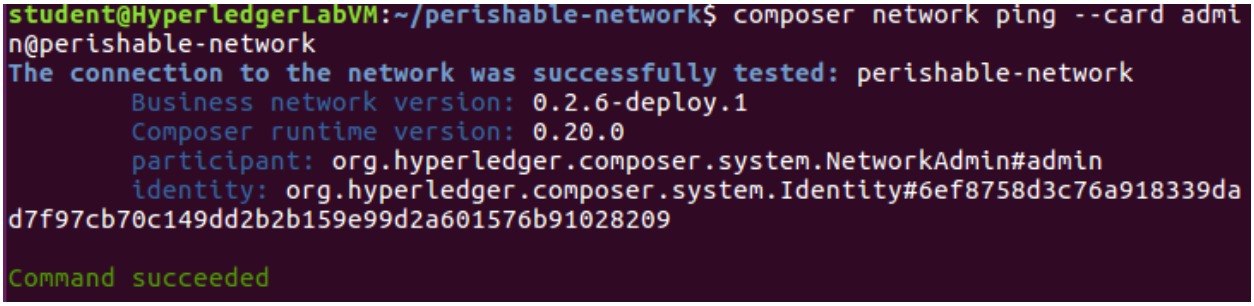


Figure 83 - Running the command.

Lab 2 - Generate RESTful APIs from the deployed solution

In this lab you'll be using Composer to generate a set of RESTful APIs against the solution you deployed in the previous lab.

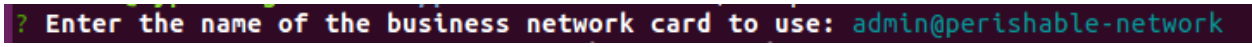
Step 1 - Create the REST API

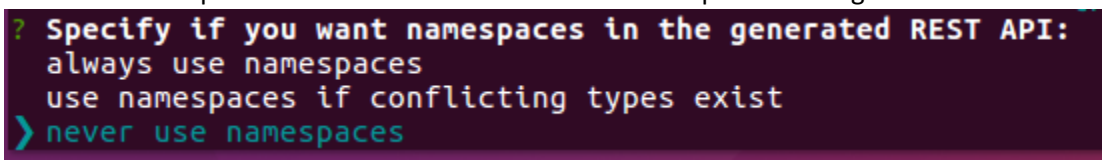
To create the REST API, navigate to the perishable-network directory and run the following command:

```
composer-rest-server
```

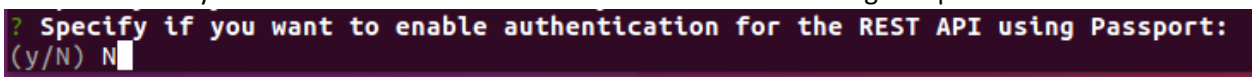
Figure 84 - Running the command.

Use the same answers shown below when prompted:

- Enter admin@perishable-network as the card name.
  - 

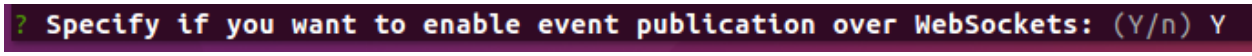
- Select "never use namespaces" when asked whether to use namespaces in the generated API.
  - 

- Select No when asked whether to use an API key to secure the REST API.
  - 

- Select No when asked if you want to enable authentication for the REST API using Passport.
  - 

- Select Yes when asked if you want to enable the explorer test interface.
  - 

- Press Enter (don't enter anything) when asked if you want to enable dynamic logging.
  - 

- Select Yes when asked whether to enable event publication over WebSockets.
  - 

- Select No when asked whether to enable TLS security for the REST API.
  - 

Once you see the following message, your REST API is up and running! Be sure to leave the command terminal window up and running!

Figure 85 - The REST API server is up and running.

Step 2 - View the REST API in the browser

Open a browser and navigate to:

```
http://localhost:3000/explorer
```

You should see the following interface:

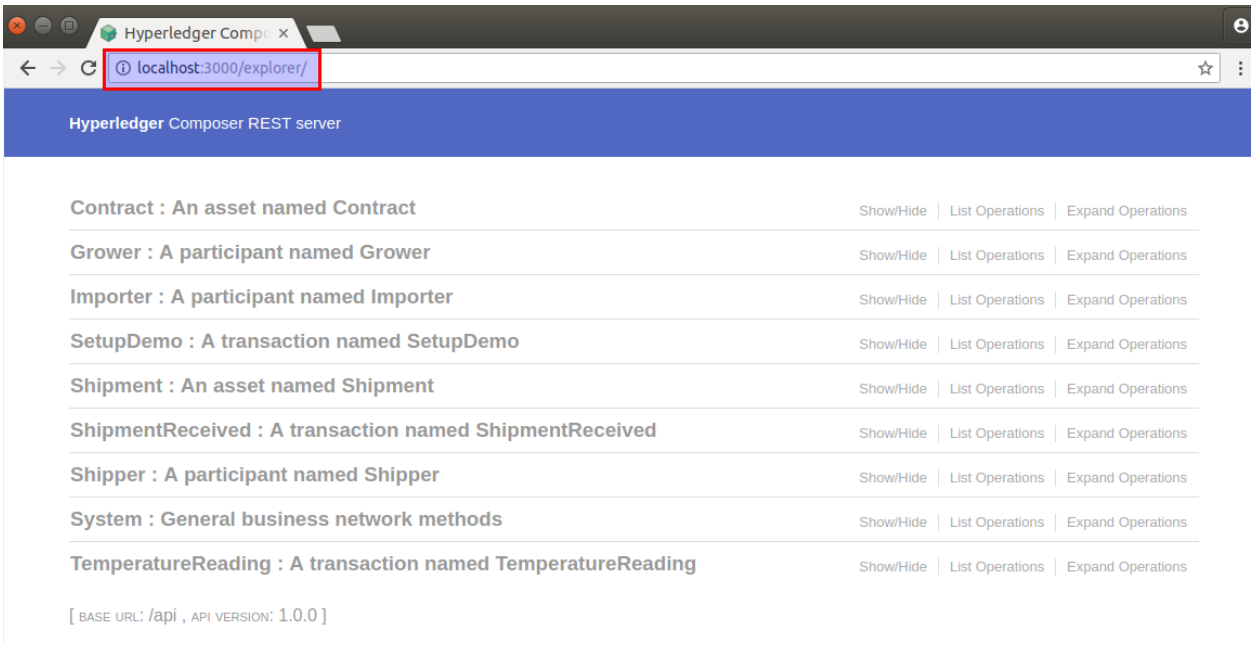


Figure 86 - The REST API web-based explorer interface.

Step 3 - Use the REST API explorer to create a new Grower

From the REST API explorer interface, click on the Grower participant.

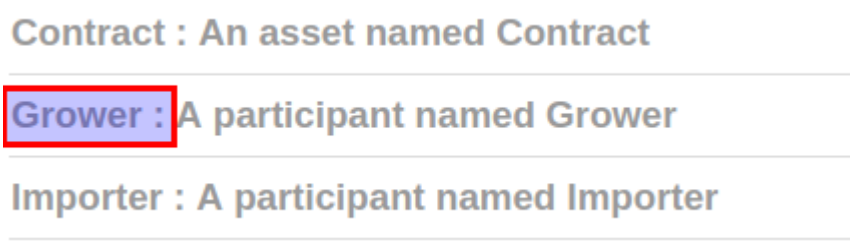


Figure 87 - Open the Grower section.

Step 4 - Review the available operations

After clicking on the Grower participant you should see the full list of available REST functions.

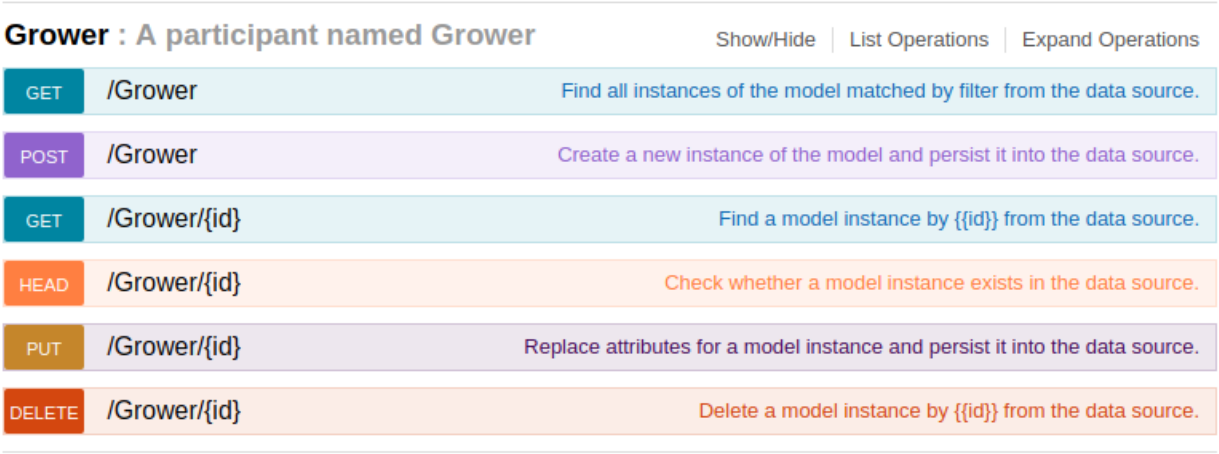


Figure 88 - The exposed REST functions.

Step 5 - Create a new Grower

Click on the "POST" operation to expand the interface.

Figure 89 - Detailing the POST operation.

In the "data" section of the window, paste the JSON code below and click on the "Try it out!" button.

```
{
  "$class": "org.acme.shipping.perishable.Grower",
  "websiteURL": "www.myGrowerWebsite.com",
  "email": "testGrower@testData.com",
  "address": {
    "$class": "org.acme.shipping.perishable.Address",
    "city": "Springfield",
    "country": "USA",
    "street": "123 Main St",
    "zip": "12345"
  },
  "accountBalance": 5500
}
```

Figure 90 - Insert the JSON object into the 'data' field and click on the 'Try it out!' button.

Validate the transaction was successful by scrolling down on the page and ensuring you received a "Response Code" of 200.

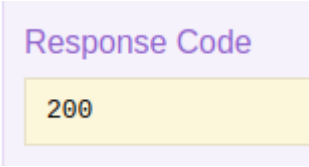


Figure 91 - Look for a 200 Response Code.

Step 6 - View the new Grower participant

To view the new Grower participant created in the last step, first click on the "POST" header under the Grower category to collapse the section you were working with on the last step.

Figure 92 - Collapse the POST section under Grower.

Next, click on the first "GET" header under the Grower section.

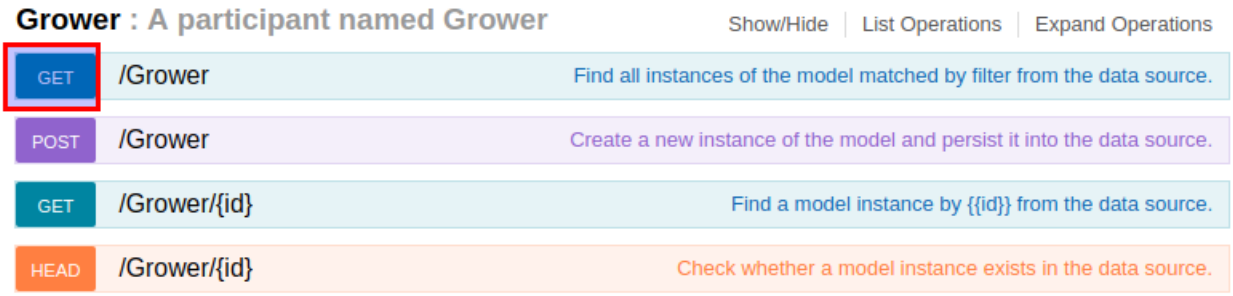


Figure 93 - Click on the first 'GET' section under the Grower Participant.

Scroll down and click on the "Try it out!" button.

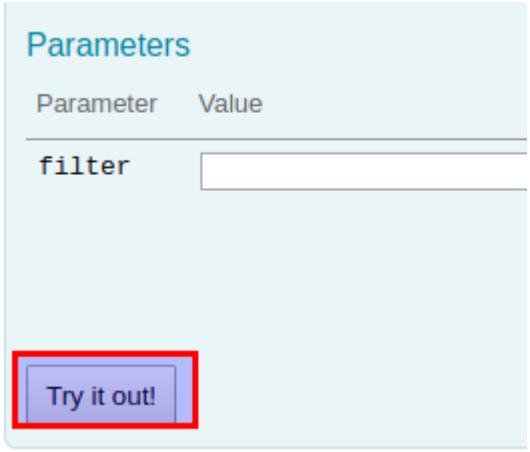


Figure 94 - Click on the 'Try it out!' button.

Scroll down to the Response Body and Response Code fields. The Response Body should contain the same information you used to create a new Grower in the previous step. The Response Code field should contain 200, indicating a successful REST call.

Figure 95 - The results.

Step 7 - Continue to experiment

Feel free to continue to experiment with the functions made available via the REST API.

Lab 3 - Use Yeoman to build an Angular app against your REST API

In this lab you'll use the Yeoman tool to construct an Angular application against the REST API you created in the last lab.

Step 1 – Prereqs

Ensure the terminal window from the previous lab is still open and that the REST API explorer is still available at:

```
http://localhost:3000/explorer
```

If the terminal has been closed, please repeat the steps in the last lab.

Step 2 – Navigate to the perishable-network directory

Open a new terminal window and navigate to the perishable-network directory using the following command:

```
cd ~/perishable-network/
```

Step 3 - Run the Yo command

From the perishable-network directory, run the following command:

```
yo hyperledger-composer:angular
```

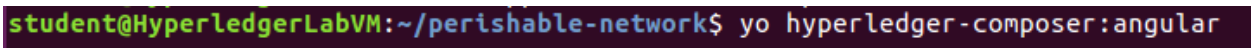


Figure 96 - The yo command to generate the Angular solution.

If this is the first time the Yeoman tool has been run, you may be asked if you'd like to send usage statistics to Yeoman. Choose whatever answer you prefer.

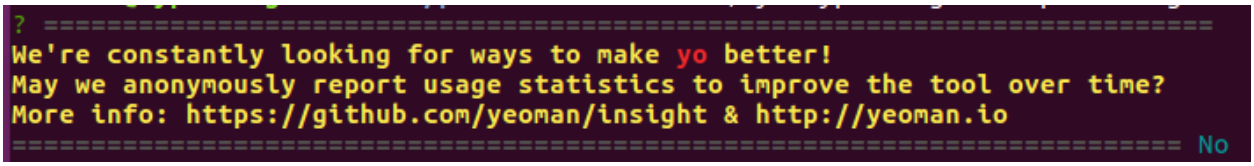


Figure 97 - If this is the first time yo has been run, you may see this dialog.

When asked if you want to connect to a running Business Network, select Yes.

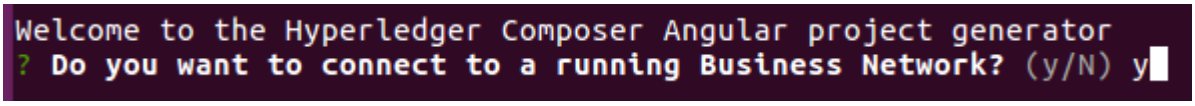


Figure 98 - Select Yes.

When asked for a project name, you can use:

`ThePerishableNetwork`

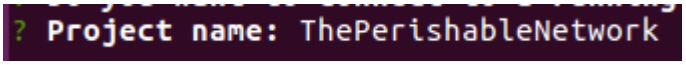


Figure 99 - Use this project name.

Provide any project description you like.

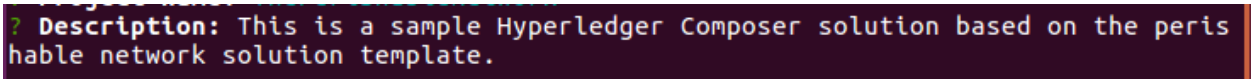


Figure 100 - Project Description prompt.

Enter your name in the "Author name" field.

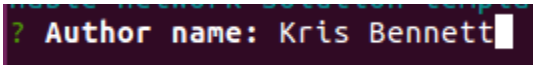


Figure 101 - The project author.

Enter an email address (does not have to be real for this lab)

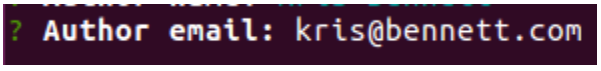


Figure 102 - Author email address. Does not have to be a real email address for this lab.

Accept the default value (`Apache-2.0`) for the License field by pressing Enter.

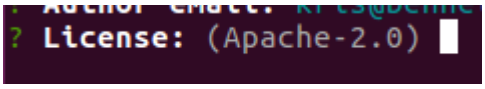


Figure 103 - Accept the default.

When asked for the name of the Business Network card, use the following value:

`admin@perishable-network`

Figure 104 - Use the admin@perishable-network card.

When asked whether to generate a new REST API or connect to an existing one, select the "`Connect to an existing REST API`" option.

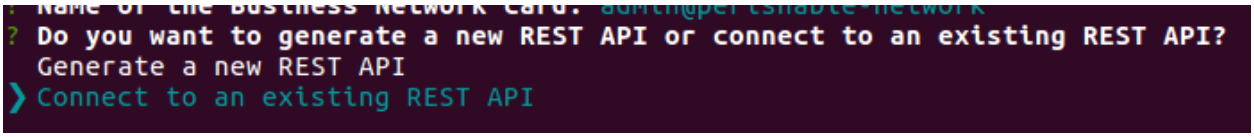


Figure 105 - Choose the option to 'Connect to an existing REST API'.

Accept the default value for the REST server address (`http://localhost`).

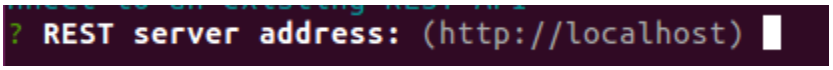


Figure 106 - Accept the default value.

Accept the default REST server port value (`3000`).

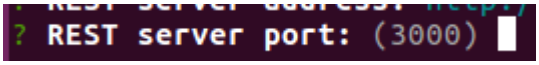


Figure 107 - Accept the default value.

When asked if namespaces should be used in the generated REST API, select the "`Namespaces are not used`" option.

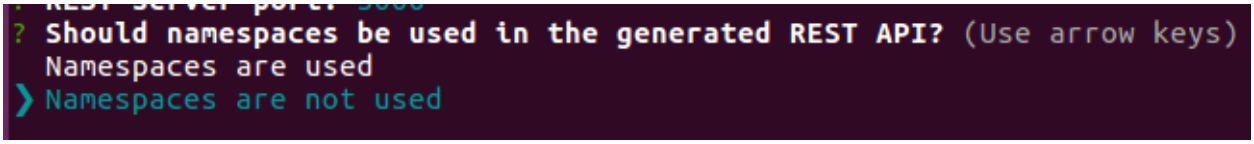


Figure 108 - Select the 'Namespaces are not used' option.

At this point Yeoman will start to build the Angular app for you. This make take several minutes.

Step 4 - Start the NPM web server

Now the Yeoman has generated an Angular app from your REST API, you can start the NPM web server and interact with the generated solution at localhost:4200. Start by navigating to the ThePerishableNetwork directory using the following command

```
cd ~/perishable-network/ThePerishableNetwork/
```

Now run the command to start NPM:

```
npm start
```

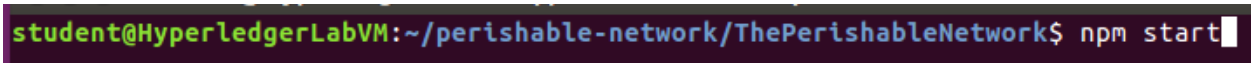


Figure 109 - Starting the npm web server.

It may take several minutes for NPM to start. Once it's started successfully, you should see the following message:

```
webpack: Compiled successfully.
```

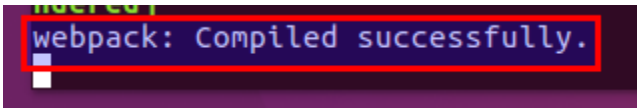


Figure 110 - Everything worked!

Step 5 - View the Angular app in a browser

Make sure you leave the terminal window from the last step open, as well as the terminal window from the previous lab. Open a browser and navigate to:

```
http://localhost:4200/
```

You should see the following interface:

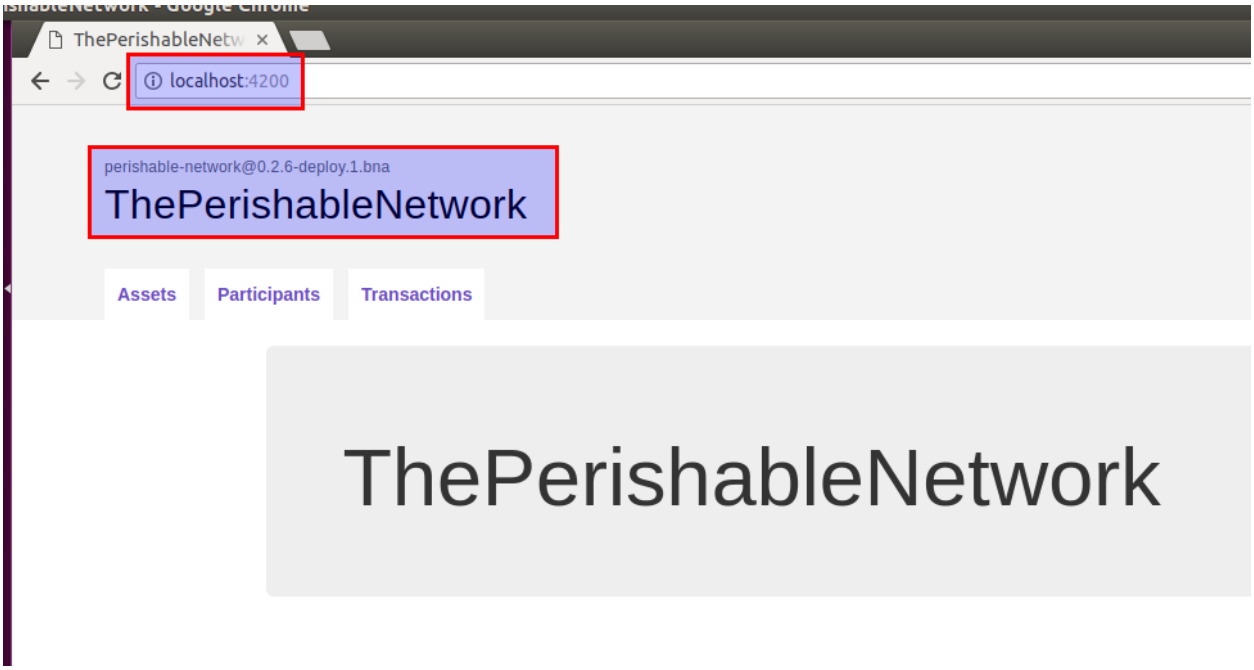


Figure 111 - The generated Angular app.

Bonus Step – Can you deploy another solution template and create an Angular app for it?

Based on what you've done in Labs 1, 2, and 3 of Section 4 can you deploy a solution template (other than the Perishable Network) in Composer Playground, export it, stand up a REST API and then create an Angular app around it? Try using the Car Auction demo…

Section 4 - Building a Solution directly from your development environment

In the previous sections we've covered building a Composer solution using the Composer Playground environment, the migrating the solution to your development environment. In this section we'll look at building a solution directly in your development environment using the Yeoman tool to create a project template.

Lab 1 - Create a Business Network solution using Yeoman

In this lab we'll use the Yeoman code generation tool to setup a project template structure. We'll then edit the solution resource files, and deploy our solution to the Fabric runtime.

NOTE - Any terminal windows from previous labs can be closed at this point.

Step 1 - Create the solution structure

From your development environment, open up a terminal window and run the following command:

```
yo hyperledger-composer:businessnetwork
```

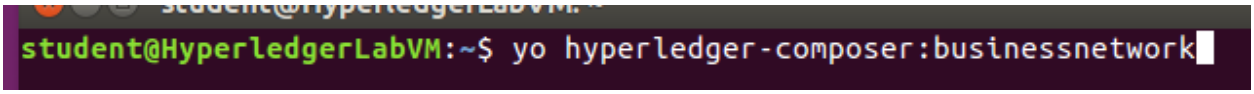


Figure 112 - Entering the command.

When prompted for the "Business network name", use the following value:

```
tutorial-network
```

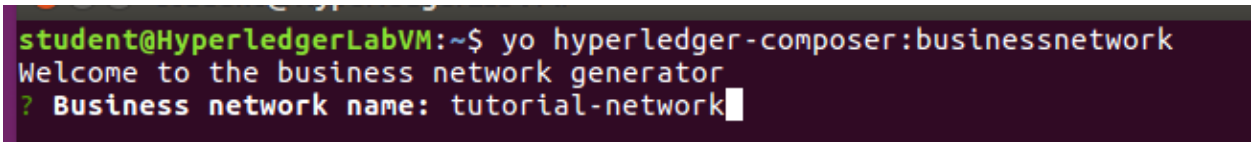


Figure 113 - Entering the Business network name.

Provide any Description, Author name, and Author email values you like.

```
student@HyperledgerLabVM:~$ yo hyperledger-composer:businessnetwork
Welcome to the business network generator
? Business network name: tutorial-network
? Description: This is a sample project.
? Author name:  Kris Bennett
? Author email: kris@abc.com
```

Figure 114 - Project metadata.

When asked for a License, accept the default (Apache-2.0). When asked for a namespace, use the following value:

```
org.example.mynetwork
```

```
student@HyperledgerLabVM:~$ yo hyperledger-composer:businessnetwork
Welcome to the business network generator
? Business network name: tutorial-network
? Description: This is a sample project.
? Author name:  Kris Bennett
? Author email: kris@abc.com
? License: Apache-2.0
? Namespace: org.example.mynetwork
```

Figure 115 - Entering a namespace.

When asked if "you want to generate an empty template network", select the "No: generate a populated sample network" option.

```
student@HyperledgerLabVM:~$ yo hyperledger-composer:businessnetwork
Welcome to the business network generator
? Business network name: tutorial-network
? Description: This is a sample project.
? Author name:  Kris Bennett
? Author email: kris@abc.com
? License: Apache-2.0
? Namespace: org.example.mynetwork
? Do you want to generate an empty template network?
  Yes: generate an empty template network
❯ No: generate a populated sample network
```

Figure 116 - Select the 'No' option.

Once the yo command has completed, you should see the following:

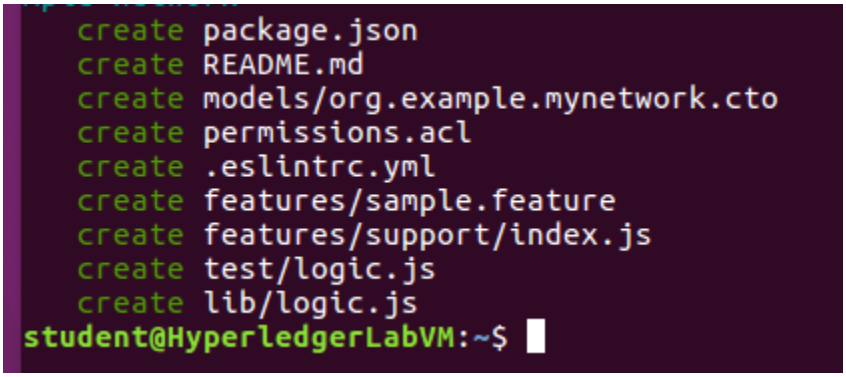


Figure 117 - The completed yo command.

Step 2 - Update the model file

From your desktop, open up the Files utility. Find the tutorial-network folder and double-click to open it. You should see the following files and folders:

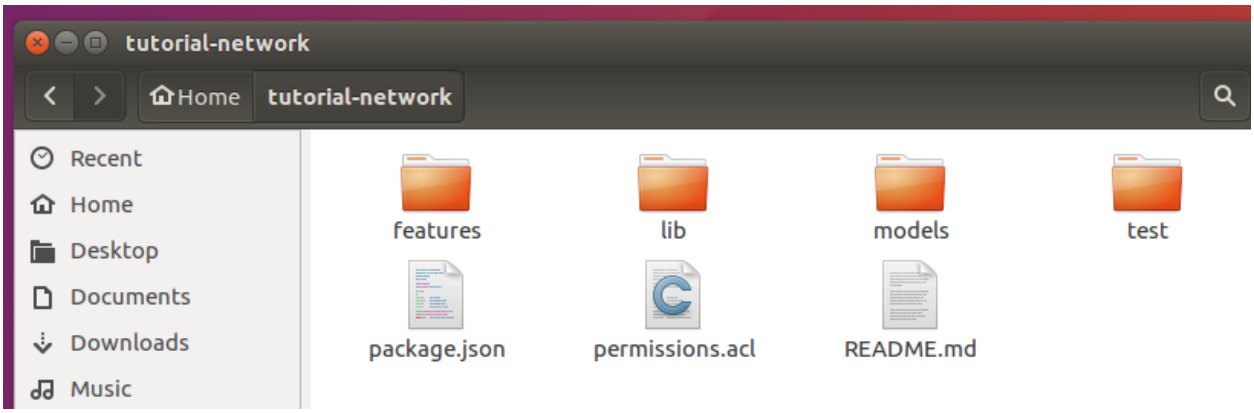


Figure 118 - The tutorial-network solution folder.

Double-click the model folder to open it. This folder will contain all the model (.cto) files for the current solution. Right-click on the file "org.example.mynetwork.cto" and select "Open With" and "Visual Studio Code" from the flyout menu to open the .cto file in Visual Studio Code.

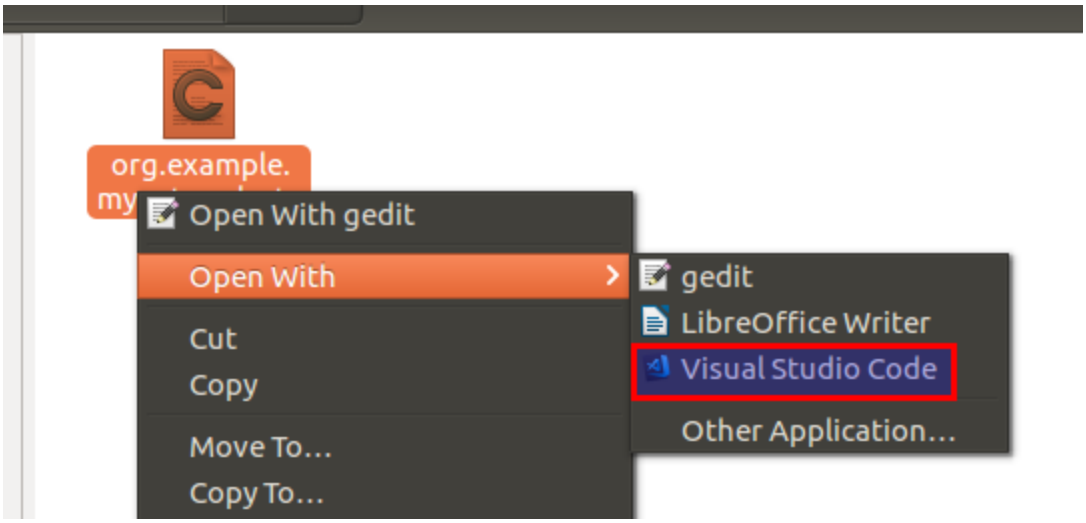


Figure 119 - Opening a file in Visual Studio Code.

Visual Studio Code should now be displaying the contents of "org.example.mynetwork.cto" file.

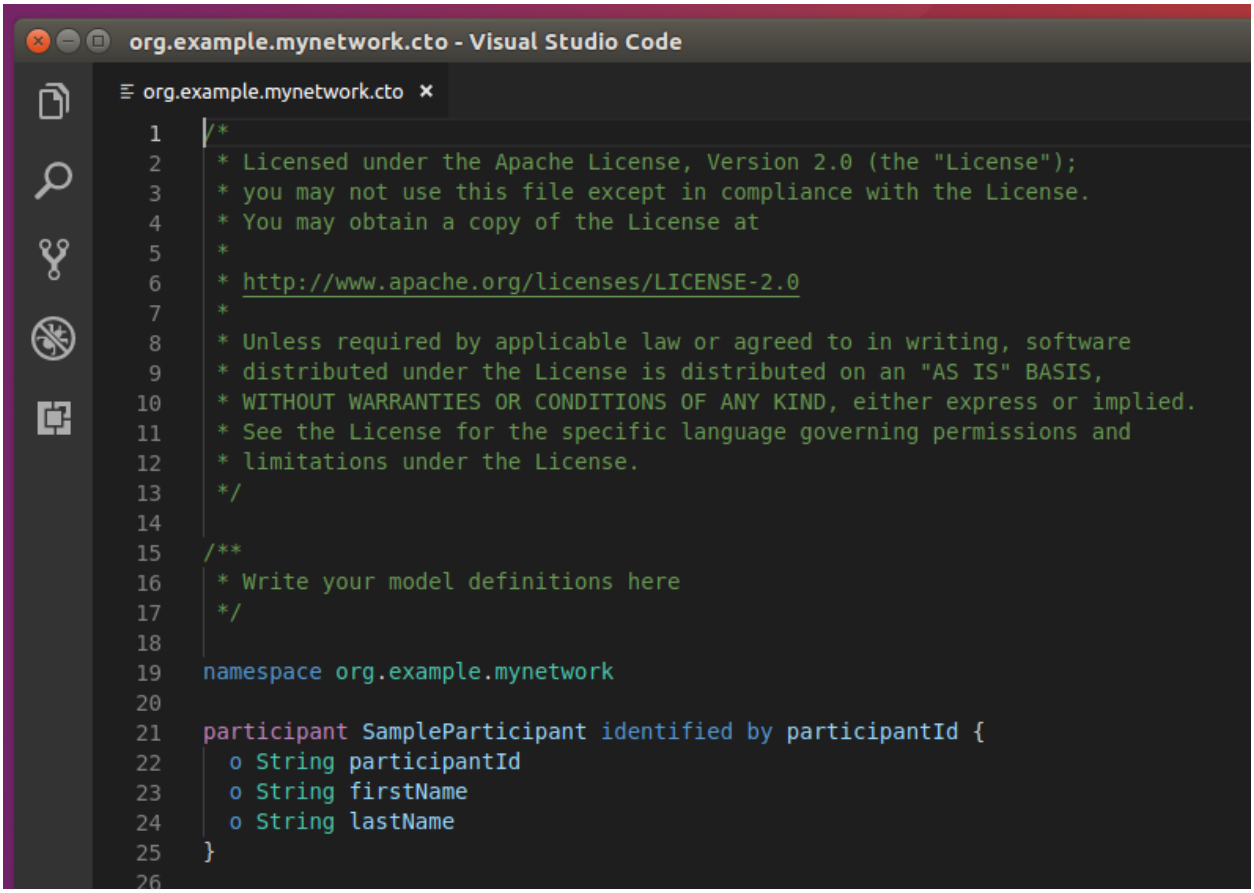


Figure 120 - The default .cto file.

DELETE all existing code from the model file. The model file should now be blank. Copy & Paste the following code into the model file:

```
/**
 * My commodity trading network
 */
namespace org.example.mynetwork
asset Commodity identified by tradingSymbol {
    o String tradingSymbol
    o String description
    o String mainExchange
    o Double quantity
    --> Trader owner
}
participant Trader identified by tradeId {
    o String tradeId
    o String firstName
    o String lastName
}
transaction Trade {
    --> Commodity commodity
    --> Trader newOwner
}
```

Select the "Save" option from the "File" menu to save your changes.

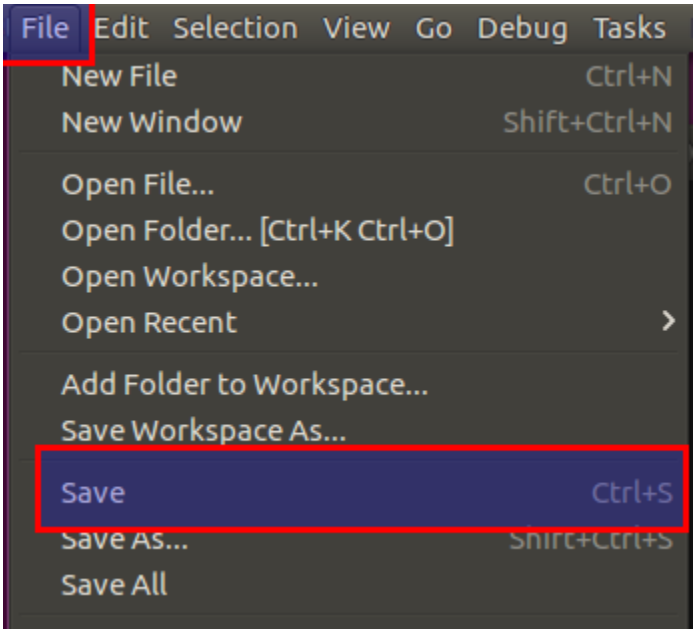


Figure 121 - Saving your changes in Visual Studio Code.

Step 3 - Update the logic.js file

Using the "Files" utility, open up the "lib" folder of the "tutorial-network" project folder. Right-click on the "logic.js" file and open the file in Visual Studio Code.

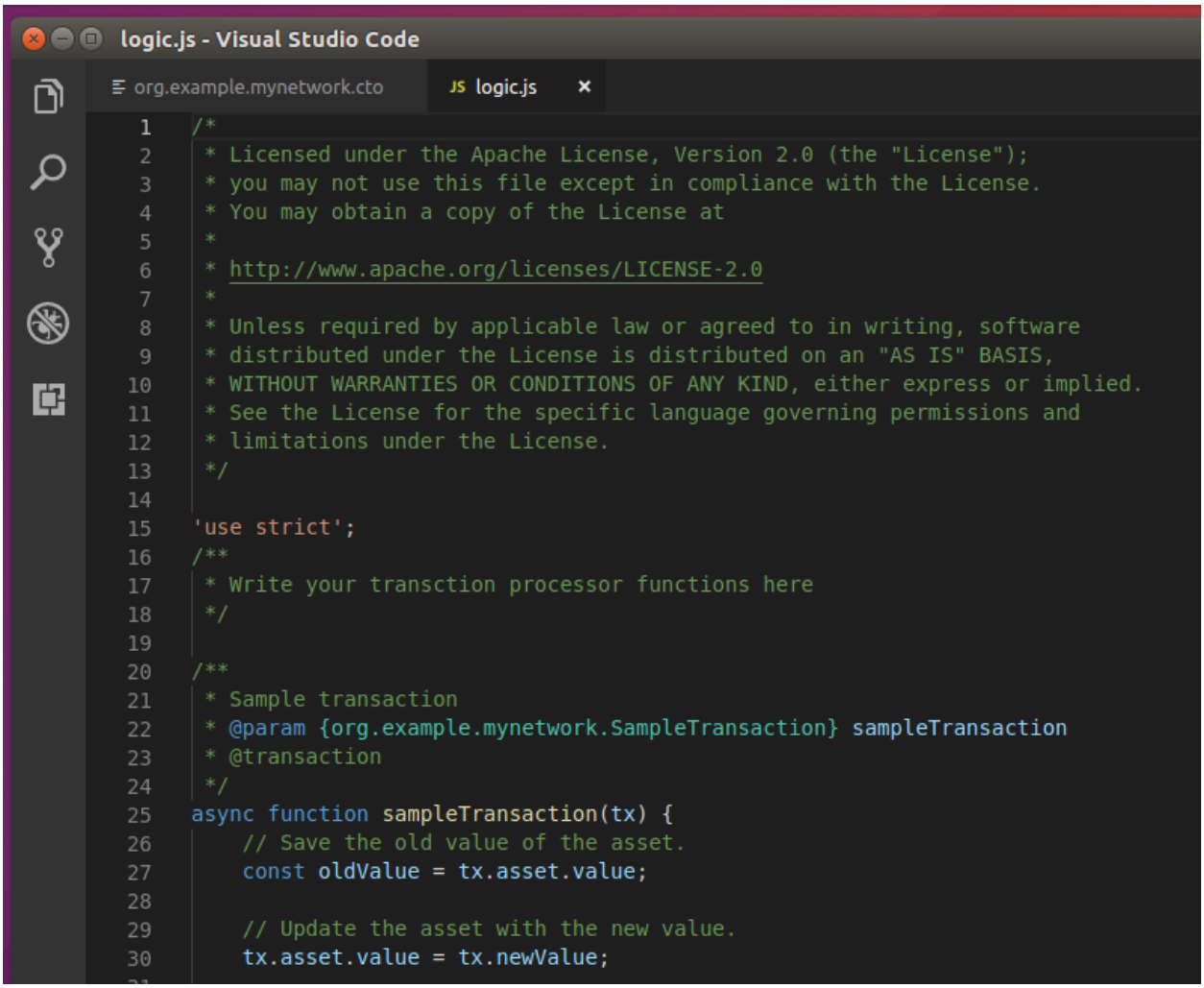


Figure 122 - The default logic.js file.

DELETE the contents of the logic.js file. The file should now be blank. Copy and paste the code below into the logic.js file:

```
/**
 * Track the trade of a commodity from one trader to another
 * @param {org.example.mynetwork.Trade} trade - the trade to be processed
 * @transaction
 */
async function tradeCommodity(trade) {
    trade.commodity.owner = trade.newOwner;
    let assetRegistry = await getAssetRegistry('org.example.mynetwork.Commodity');
    await assetRegistry.update(trade.commodity);
}
```

Use the Save option from the File menu to save your changes.

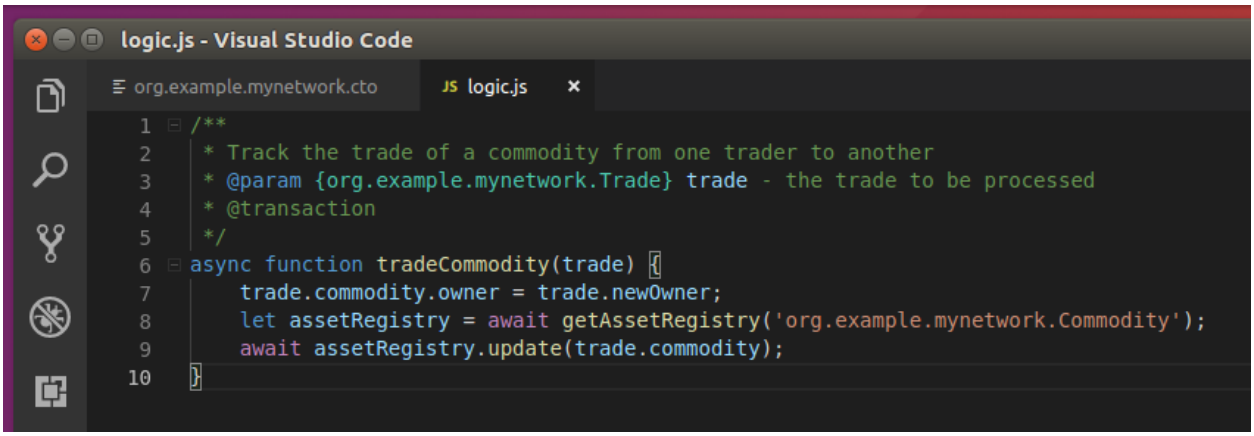


Figure 123 - The updated logic.js file.

Step 4 - Update the access control file

From the tutorial-network folder, right click on the "permissions.acl" file and open it in Visual Studio Code. Delete the contents and replace them with the code below:

```
/**
 * Access control rules for tutorial-network
 */
rule Default {
    description: "Allow all participants access to all resources"
    participant: "ANY"
    operation: ALL
    resource: "org.example.mynetwork.*"
    action: ALLOW
}

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

Save your changes.

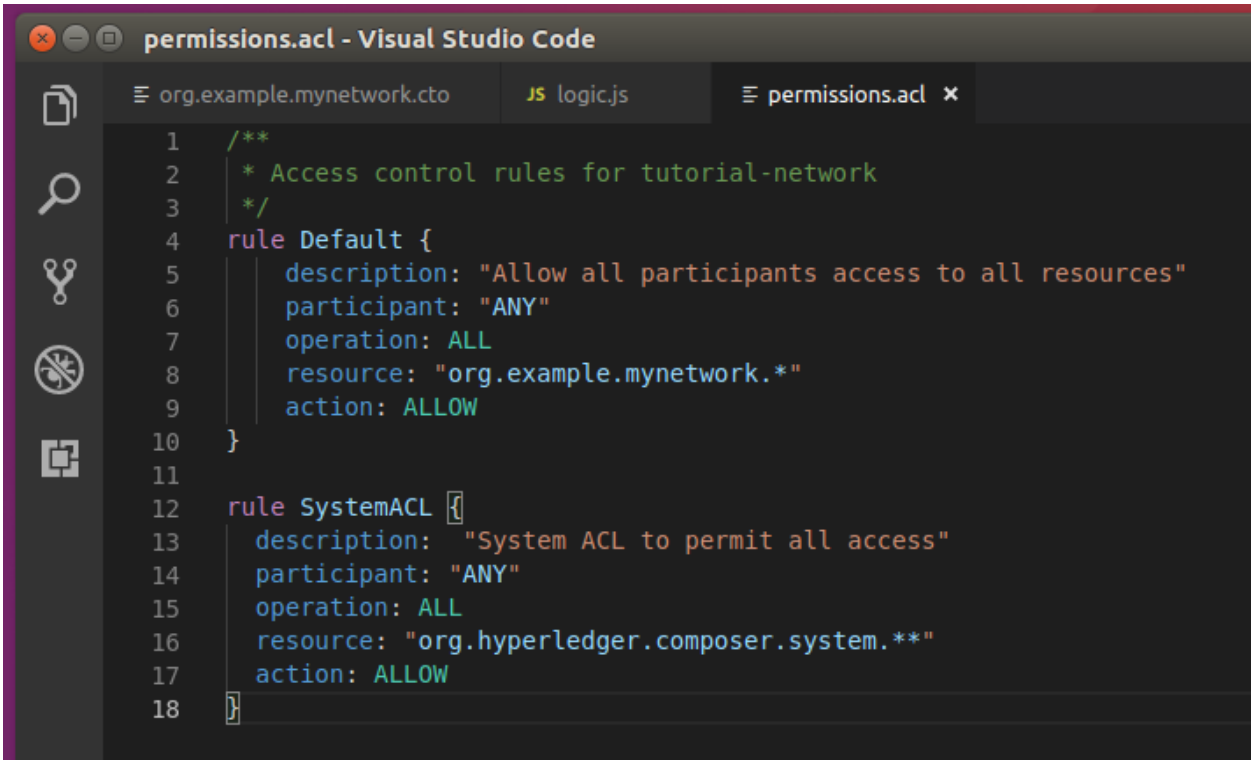


Figure 124 - The access control file.

Step 5 - Generate a .bna file from your solution

If all your changes have been saved from the previous steps, feel free to close Visual Studio Code (if still open).

From your terminal window, navigate to the tutorial-network folder using the following command:

```
cd ~/tutorial-network/
```

In this step we'll be creating a .bna archive of the solution we just edited in Visual Studio Code. Remember in Section 4, Lab 1 we exported a .bna file from Composer Playground. In this step we'll be creating the .bna from the artifacts in the tutorial-network solution folder. To create the .bna file, run the following command:

```
composer archive create -t dir -n .
```

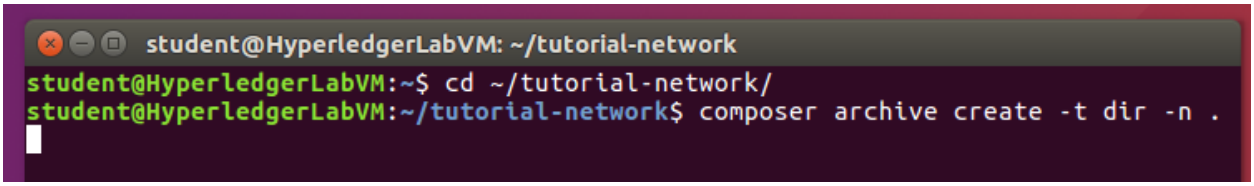


Figure 125 - Building the .bna file.

Once the command has completed, you should see the screen below. Take note of the "Output file" name (tutorial-network@0.0.1.bna).

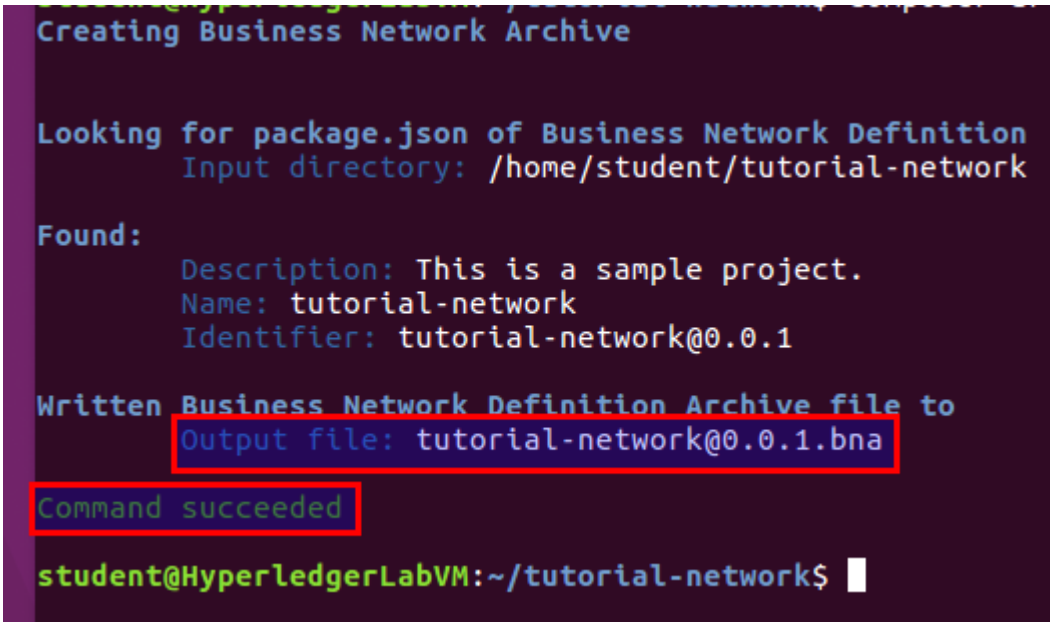


Figure 126 - Command successful, note the output file value.

Step 6 - Deploy the .bna file to the Fabric environment

To install the solution into the Fabric environment, run the following command from a terminal window in your development environment. Before running the command ensure you are the tutorial-network directory. For reference, this is the same "composer network install" command you used in Section 4, Lab 1 to deploy the .bna file exported from Composer Playground. The only difference is the "archiveFile" value.

```
composer network install --card PeerAdmin@hlfv1 --archiveFile
tutorial-network@0.0.1.bna
```

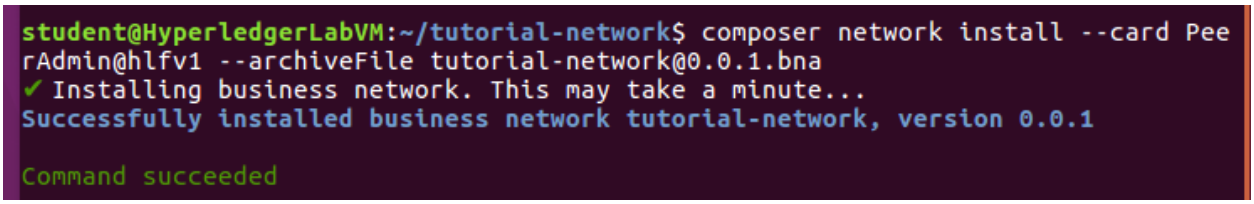


Figure 127 - Command ran successfully.

Step 7 - Start the business network

To start the solution, use the "composer network start" command. This is the same command you entered in Section 4, Lab 1. Enter the command below:

```
composer network start --networkName tutorial-network --networkVersion
0.0.1 --networkAdmin admin --networkAdminEnrollSecret adminpw --card
PeerAdmin@hlfv1 --file networkadmin.card
```

Step 8 - Import the network administrator identity card

To import the network administrator identity as a usable business network card, run the command below. The composer card import command requires the filename specified in composer network start command to create a card. This is the same command you entered in Section 4, Lab 1.

```
composer card import --file networkadmin.card
```

Step 9 - Verify the Business Network is running

To check that the business network has been deployed successfully, run the command below to ping the network. The composer network ping command requires a business network card to identify the network to ping.

```
composer network ping --card admin@tutorial-network
```

If the solution is running correctly, you should see the following message:

```
student@HyperledgerLabVM:~/tutorial-network$ composer network ping --card admin@
tutorial-network
The connection to the network was successfully tested: tutorial-network
        Business network version: 0.0.1
        Composer runtime version: 0.20.0
        participant: org.hyperledger.composer.system.NetworkAdmin#admin
        identity: org.hyperledger.composer.system.Identity#a5bc3f78fa824921556c2
459f16f3e35b6e28f52ea9321e5b94a8cf7226fdeca

Command succeeded
```

Figure 128 - Command ran successfully.

Step 10 - Continue

At this point your tutorial-network solution is in the same state your perishable-network solution was in at the end of Section 4, Lab 1. If desired, proceed to follow the instructions in the remaining Section 4 Labs (Lab 2 and beyond) to build the REST API and Angular app around your template-network solution.

Section 5 - Queries

In this section, you'll get hands-on with building queries in Hyperledger Composer. The native query language in Composer can filter results using criteria and can be invoked in transactions to perform operations, such as updating or removing assets on result sets. Queries are defined in a query file (.qry) in the parent directory of the business network definition. Queries contain a WHERE clause, which defines the criteria by which assets or participants are selected.

Lab 1 - Creating Queries

In this lab, you'll be updating the tutorial-network solution created during the previous lab to include several queries.

Step 1 - Update the model file

Start by opening the model (`org.example.mynetwork.cto`) file from the `tutorial-network/models` solution folder in Visual Studio Code. Add the following code to the end of the model file and save your changes once complete:

```
event TradeNotification {
    --> Commodity commodity
}
transaction RemoveHighQuantityCommodities {
}
event RemoveNotification {
    --> Commodity commodity
}
```

Step 2 - Update the logic.js file

Start by opening the logic (logic.js) file from the tutorial-network/lib solution folder in Visual Studio Code. Replace the contents of the logic.js file with the following code and save your changes once complete:

```
/**
 * Track the trade of a commodity from one trader to another
 * @param {org.example.mynetwork.Trade} trade - the trade to be
processed
 * @transaction
 */
async function tradeCommodity(trade) {

    // set the new owner of the commodity
    trade.commodity.owner = trade.newOwner;
    let assetRegistry = await
getAssetRegistry('org.example.mynetwork.Commodity');

    // emit a notification that a trade has occurred
    let tradeNotification =
getFactory().newEvent('org.example.mynetwork', 'TradeNotification');
    tradeNotification.commodity = trade.commodity;
    emit(tradeNotification);

    // persist the state of the commodity
    await assetRegistry.update(trade.commodity);
}

/**
 * Remove all high volume commodities
 * @param {org.example.mynetwork.RemoveHighQuantityCommodities} remove
- the remove to be processed
 * @transaction
```

```
 */
async function removeHighQuantityCommodities(remove) {


    let assetRegistry = await
getAssetRegistry('org.example.mynetwork.Commodity');
    let results = await query('selectCommoditiesWithHighQuantity');


    for (let n = 0; n < results.length; n++) {
        let trade = results[n];


        // emit a notification that a trade was removed
        let removeNotification =
getFactory().newEvent('org.example.mynetwork','RemoveNotification');
        removeNotification.commodity = trade;
        emit(removeNotification);
        await assetRegistry.remove(trade);
    }
}
```

NOTE: *The first function tradeCommodity will change the owner property on a commodity (with a new owner Participant) on an incoming Trade transaction and then emit a Notification event. The modified Commodity is then saved back into the asset registry. The second function calls a query 'selectCommoditiesWithHighQuantity' (defined in queries.qry) which will return all Commodity assets that have a quantity greater than 60. The function will then emit an event, and finally remove the Commodity from the AssetRegistry.*

Step 3 - Create the Query Definition (.qry) file

The queries used by logic.js are defined in a file which must be called "queries.qry". Each query entry defines the resources and criteria against which the query is executed. Use Visual Studio Code to create a new file by using the "New File" option under the "File" menu:

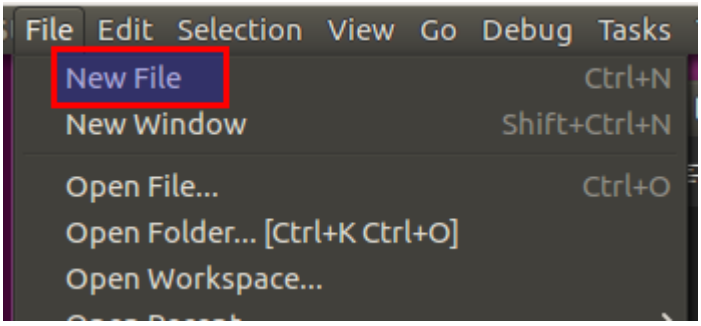


Figure 129 - Creating a new file in Visual Studio Code.

Paste the following code into the newly created file:

```
/** Sample queries for Commodity Trading business network
*/

query selectCommodities {
  description: "Select all commodities"
  statement:
      SELECT org.example.mynetwork.Commodity
}

query selectCommoditiesByExchange {
  description: "Select all commodities based on their main exchange"
  statement:
      SELECT org.example.mynetwork.Commodity
          WHERE (mainExchange==_$exchange)
}

query selectCommoditiesByOwner {
  description: "Select all commodities based on their owner"
```

```
  statement:
      SELECT org.example.mynetwork.Commodity
          WHERE (owner == _$owner)
}

query selectCommoditiesWithHighQuantity {
  description: "Select commodities based on quantity"
  statement:
      SELECT org.example.mynetwork.Commodity
          WHERE (quantity > 60)
}
```

Use the "Save" option under the "File" menu in Visual Studio Code to save the file. Make sure you save the file in the tutorial-network directory and name the file:

`queries.qry`

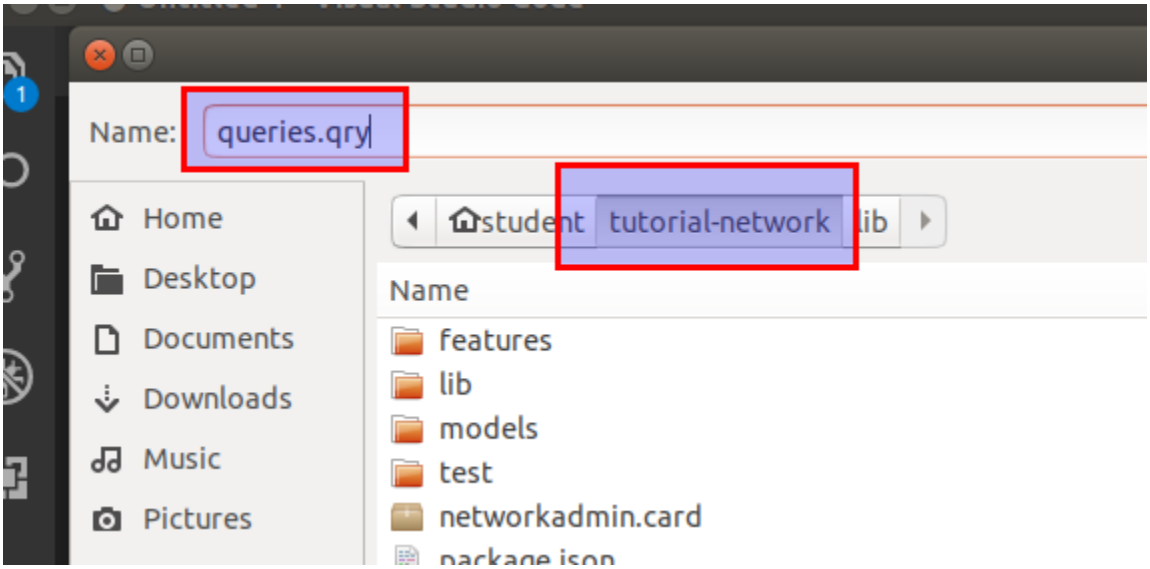


Figure 130 - Name the file 'queries.qry' and save it in the 'tutorial-network' folder.

Step 4 - Re-Generate your .bna file

After changing the files in a business network solution, the business network must be repackaged as a .bna and re-deployed to the Fabric environment. Upgrading a deployed solution requires that the new version being deployed have a newer (higher) version number than the solution version being replaced. Using Visual Studio Code, open the package.json file from the tutorial-network directory. Update the version property from 0.0.1 to 0.0.2 as per the image below and save your changes.

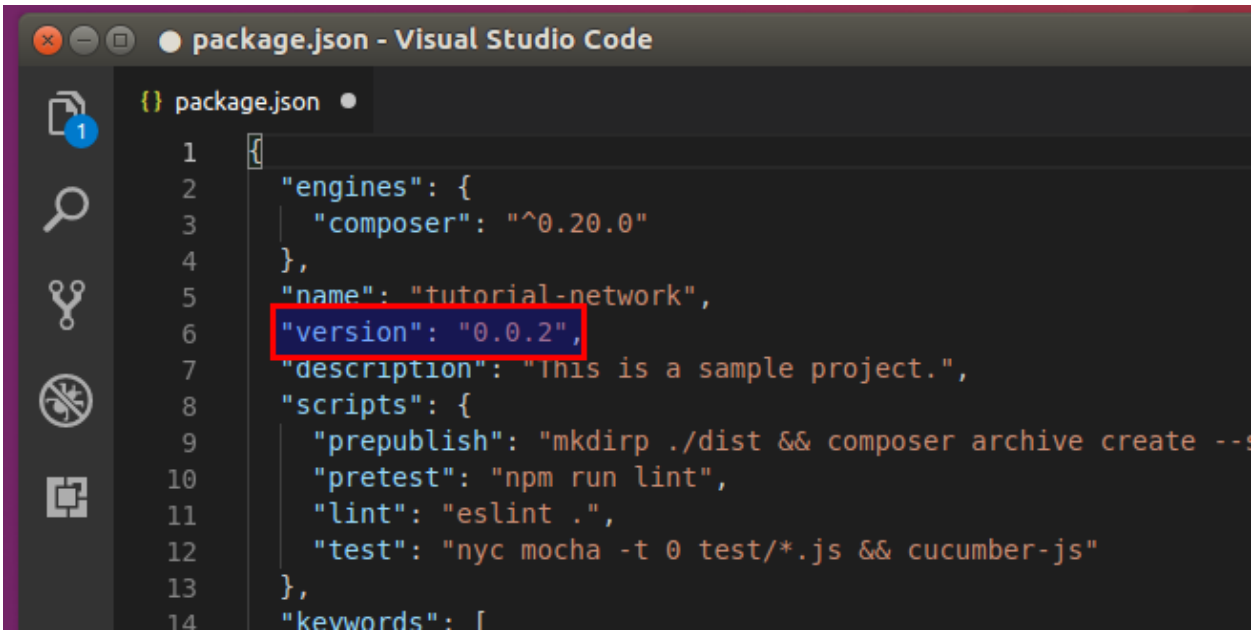


Figure 131 - Updating the version in the package.json file.

Using a terminal window, navigate to the tutorial-network directory (cd ~/tutorial-network/). From the tutorial-network directory, run the following command:

```
composer archive create --sourceType dir --sourceName . -a tutorial-network@0.0.2.bna
```

Ensure your command was successful.

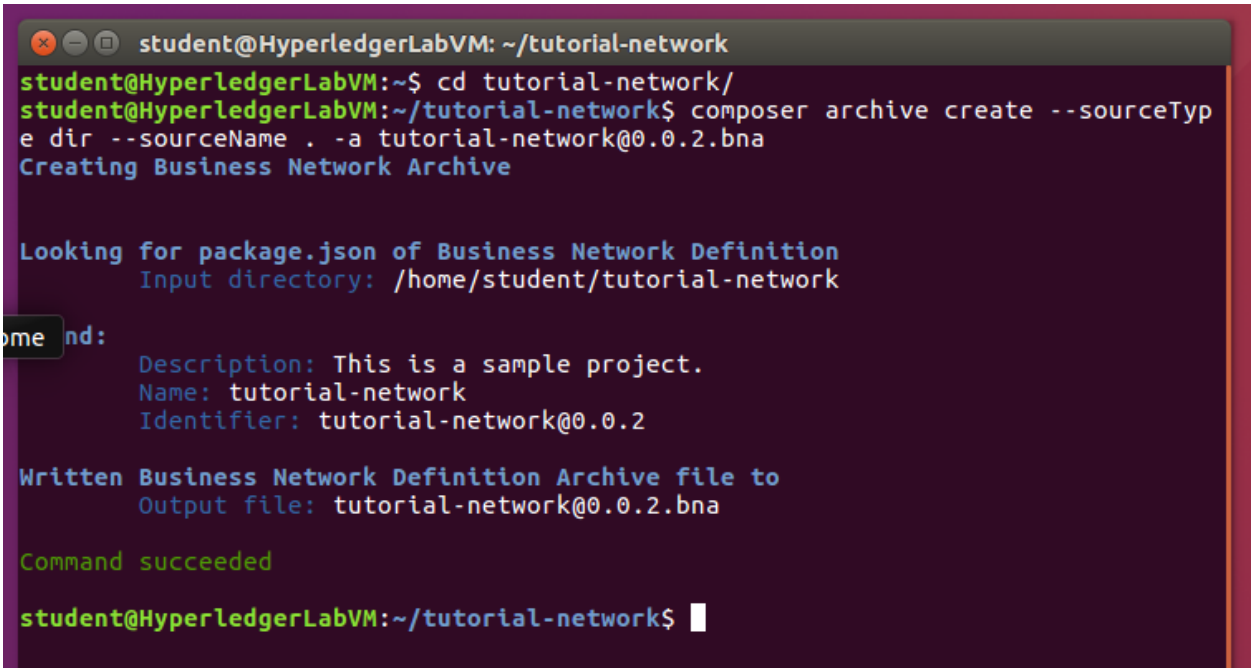


Figure 132 - Creating a new .bna file.

Step 5 - Deploy the updated solution

From a terminal window in the tutorial-network directory, run the following command to install the updated business network:

```
composer network install --card PeerAdmin@hlfv1 --archiveFile
tutorial-network@0.0.2.bna
```

```
student@HyperledgerLabVM:~/tutorial-network$ composer network install --card Pee
rAdmin@hlfv1 --archiveFile tutorial-network@0.0.2.bna
✔ Installing business network. This may take a minute...
Successfully installed business network tutorial-network, version 0.0.2

Command succeeded
```

Figure 133 - Deploying the updated solution.

Run the following command to upgrade the solution to the new version:

```
composer network upgrade -c PeerAdmin@hlfv1 -n tutorial-network -V
0.0.2
```

```
student@HyperledgerLabVM:~/tutorial-network$ composer network upgrade -c PeerAdm
in@hlfv1 -n tutorial-network -V 0.0.2
Upgrading business network tutorial-network to version 0.0.2

✔ Upgrading business network definition. This may take a minute...

Command succeeded
```

Figure 134 - Upgrading the tutorial-network solution.

Finally, ping the business network to ensure the proper solution version is deployed and running by issuing the command below. Note the "Business network version" value.

```
composer network ping -c admin@tutorial-network
```

```
student@HyperledgerLabVM:~/tutorial-network$ composer network ping -c admin@tuto
rial-network
The connection to the network was successfully tested: tutorial-network
        Business network version: 0.0.2
        Composer runtime version: 0.20.0
        participant: org.hyperledger.composer.system.NetworkAdmin#admin
        identity: org.hyperledger.composer.system.Identity#a5bc3f78fa824921556c2
459f16f3e35b6e28f52ea9321e5b94a8cf7226fdeca

Command succeeded
```

Figure 135 - Testing for successful upgrade. Note the 'Business network version' value.

Step 6 - Regenerate the REST API for the updated solution

From a terminal window in the tutorial-network directory, run the following command to build the REST API:

```
composer-rest-server
```

Use the same answers below when prompted:

- Enter admin@tutorial-network as the card name.
- Select "never use namespaces" when asked whether to use namespaces in the generated API.
- Select No when asked whether to use an API key to secure the REST API.
- Select No when asked if you want to enable authentication for the REST API using Passport.
- Select Yes when asked if you want to enable the explorer test interface.
- Press Enter (don't enter anything) when asked if you want to enable dynamic logging.
- Select Yes when asked whether to enable event publication over WebSockets.
- Select No when asked whether to enable TLS security for the REST API.

Step 7 - Create several new Trader Participants

Open up a browser and navigate to:

```
http://localhost:3000/explorer
```

You should see the REST API explorer interface, allowing you to inspect and test the generated REST API. You should be able to see that the REST Endpoint called 'Query' has been added and, upon expanding, reveals the list of REST Query operations defined in the tutorial-network solution.

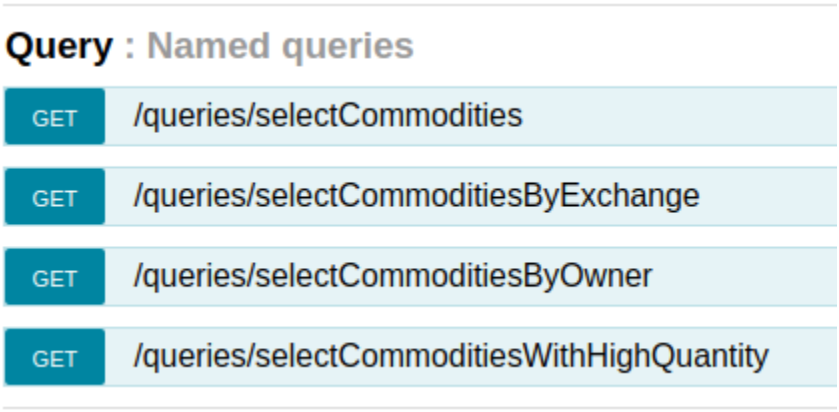


Figure 136 - The available queries in the REST API explorer.

In order to test whether the new queries are working as designed, you'll need some sample test data. Use the REST API explorer interface to create several new Trader participants as well as some more Commodity assets. First, click on 'Trader' in the REST Explorer, then click on the 'POST' method. From the Trader view, scroll down to the Parameters section and create a new Trader by pasting the JSON below into the "data" field and clicking the "Try it out!" button.

```
{
  "$class": "org.example.mynetwork.Trader",
  "tradeId": "TRADER1",
  "firstName": "Jenny",
  "lastName": "Jones"
}
```

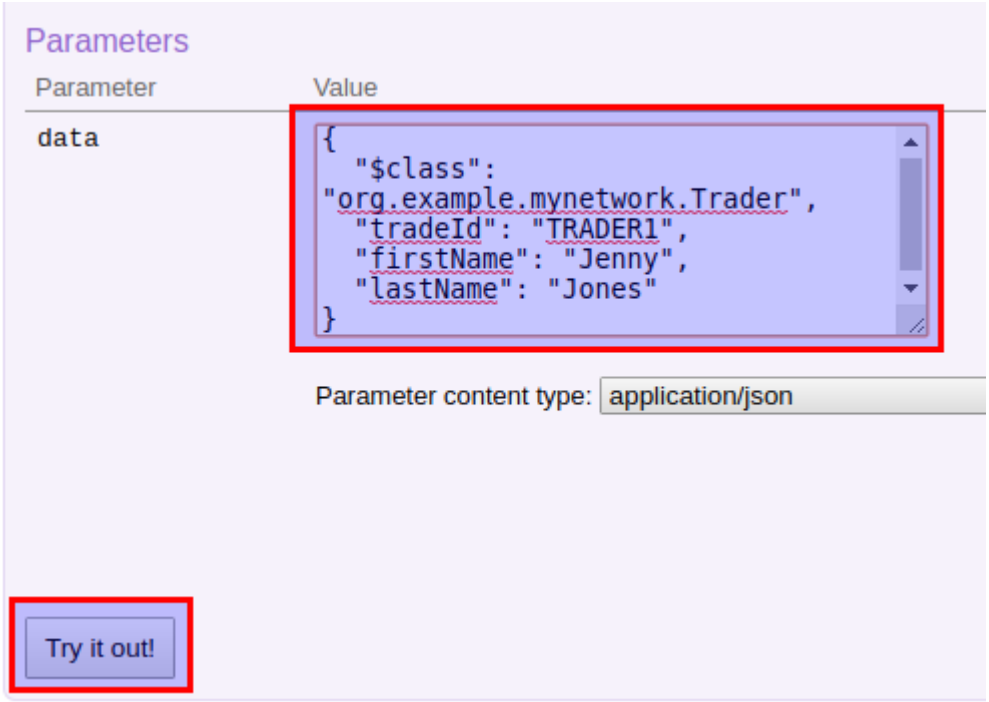


Figure 137 - Adding the new Trader via the REST API explorer.

After clicking 'Try it out' to create the Participant, the 'Response Code' should be 200, indicating a successful transaction.

Figure 138 - A Response Code of 200 indicates success.

Repeat the process to create another trader using the JSON below:

```
{
  "$class": "org.example.mynetwork.Trader",
  "tradeId": "TRADER2",
  "firstName": "Jack",
  "lastName": "Sock"
}
```

Repeat the process to create a third trader using the JSON below:

```
{
  "$class": "org.example.mynetwork.Trader",
  "tradeId": "TRADER3",
  "firstName": "Rainer",
  "lastName": "Valens"
}
```

Step 8 - Create new Commodity Assets

From the REST API explorer interface at http://localhost:3000/explorer click on 'Commodity' view. From the Commodity view click on the POST operation and scroll down to the Parameters section. Repeat the same process outlined in the last step to add two new Commodity Assets.

First Commodity to add:

```
{
  "$class": "org.example.mynetwork.Commodity",
  "tradingSymbol": "EMA",
  "description": "Corn",
  "mainExchange": "EURONEXT",
  "quantity": 10,
  "owner": "resource:org.example.mynetwork.Trader#TRADER1"
}
```

Second Commodity to add:

```
{
  "$class": "org.example.mynetwork.Commodity",
  "tradingSymbol": "CC",
  "description": "Cocoa",
  "mainExchange": "ICE",
  "quantity": 80,
  "owner": "resource:org.example.mynetwork.Trader#TRADER2"
}
```

Step 9 - Test out the selectCommodities query

Now that you have some sample assets and participants you can get hands-on with the new queries you added. The simplest query you can try is named selectCommodities. Expand the 'Query' section to view all the queries defined in the current solution. Note the description of the query (defined in .qry query definition file) is shown on the right hand side.

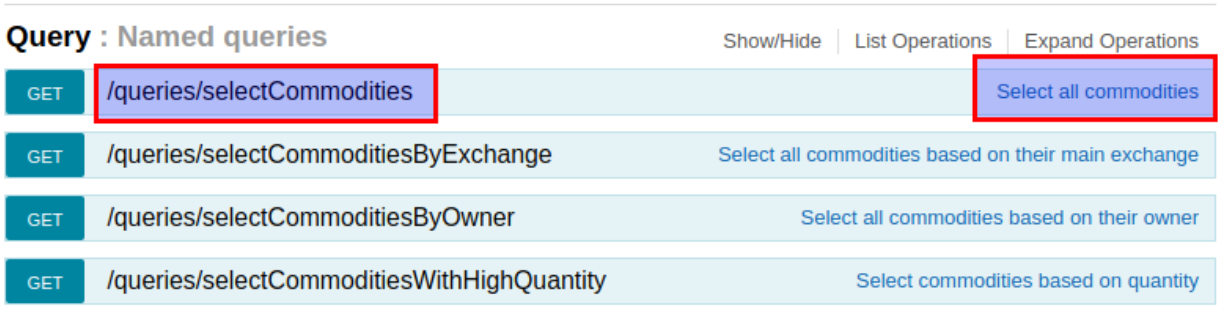


Figure 139 - The available queries, including selectCommodities.

Expand the "selectCommodities" query and click on the "Try it out!" button. You should see the two commodities you created in the last step returned from this query.

Response Body


```
[
  {
    "$class": "org.example.mynetwork.Commodity",
    "tradingSymbol": "CC",
    "description": "Cocoa",
    "mainExchange": "ICE",
    "quantity": 80,
    "owner": "resource:org.example.mynetwork.Trader#TRADER2"
  },
  {
    "$class": "org.example.mynetwork.Commodity",
    "tradingSymbol": "EMA",
    "description": "Corn",
    "mainExchange": "EURONEXT",
    "quantity": 10,
    "owner": "resource:org.example.mynetwork.Trader#TRADER1"
  }
]
```


Figure 140 - The results of selectCommodities.

Step 10 - Perform a filtered query

The query "selectCommoditiesByExchange" takes an input parameter - the exchange to filter returned results on. In this step, we'll use this query to view only commodities on the "EURONEXT" exchange. Start by expanding the query "selectCommoditiesByExchange" and scroll to the Parameters section. Enter "EURONEXT" in the "exchange" parameter and click "Try it out!".

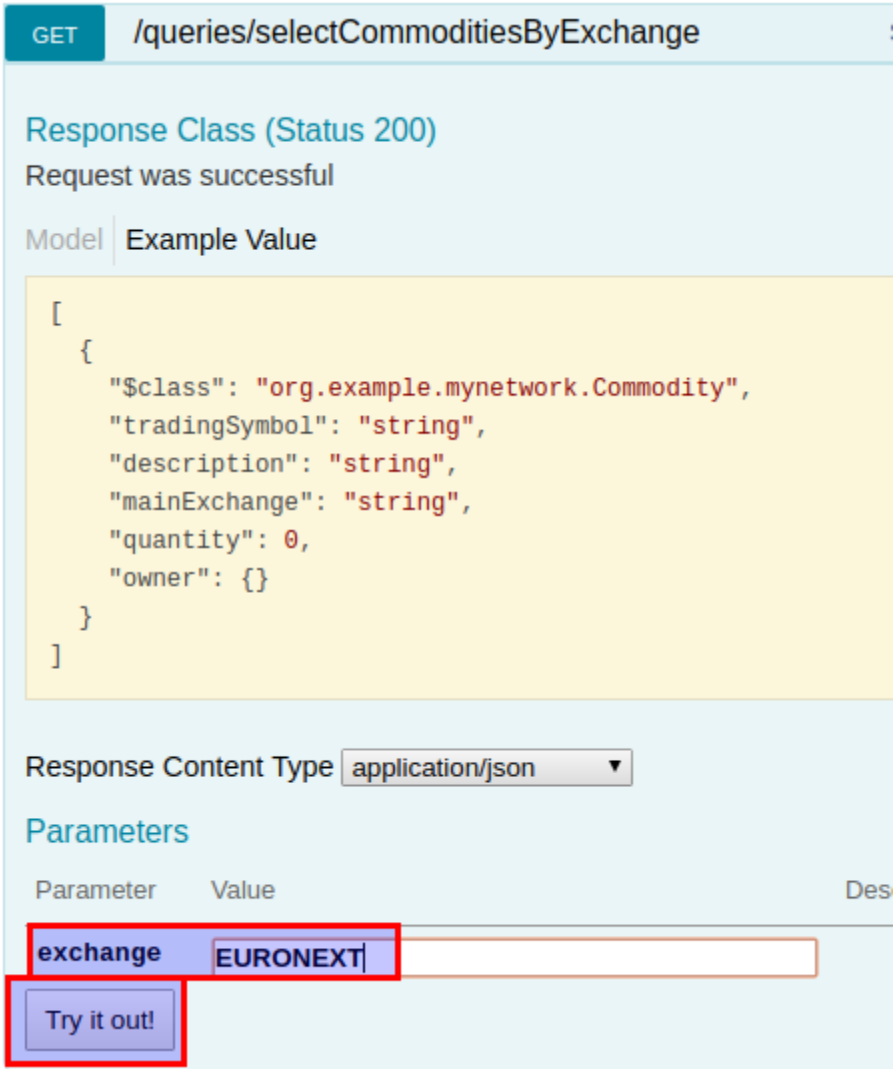


Figure 141 - Querying for EURONEXT commodities.

You should only see the "Corn" commodity returned.

Figure 142 - The results of the query.

Step 11 - Using queries in a transaction

In the queries.qry file you added to the tutorial-network solution, there was one query "selectCommoditiesWithHighQuantity" which returned only those Commodity objects with a quantity property greater than 60.

```
23
24  query selectCommoditiesWithHighQuantity {
25    description: "Select commodities based on quantity"
26    statement:
27        SELECT org.example.mynetwork.Commodity
28            WHERE (quantity > 60)
29  }
```

Figure 143 - The selectCommoditiesWithHighQuantity query definition from the queries.qry file.

Queries are very powerful when used in combination with transaction functions. Queries can allow transaction logic to define a set of assets or participants to perform updates or removals on. The "selectCommoditiesWithHighQuantities" query is used within the logic.js function "removeHighQuantityCommodities". Calling this function (or invoking this transaction) will remove all Commodity objects from the Asset Registry which have a quantity property greater than 60.

```
20
21  /**
22   * Remove all high volume commodities
23   * @param {org.example.mynetwork.RemoveHighQuantityCommodities} remove - the remove to be processed
24   * @transaction
25   */
26  async function removeHighQuantityCommodities(remove) {
27
28      let assetRegistry = await getAssetRegistry('org.example.mynetwork.Commodity');
29      let results = await query('selectCommoditiesWithHighQuantity');
30
31      for (let n = 0; n < results.length; n++) {
32          let trade = results[n];
33
34          // emit a notification that a trade was removed
35          let removeNotification = getFactory().newEvent('org.example.mynetwork','RemoveNotification');
36          removeNotification.commodity = trade;
37          emit(removeNotification);
38          await assetRegistry.remove(trade);
39      }
40  }
```

Figure 144 - The removeHighQuantityCommodities function from the logic.js file. Note the use of the 'query' Composer object to run the query and return results.

Use the "GET" operation on the Commodity object to display a list of all the Commodity objects in the registry. Note the value of the "quantity" property of all results.

Response Body


```
[
  {
    "$class": "org.example.mynetwork.Commodity",
    "tradingSymbol": "CC",
    "description": "Cocoa",
    "mainExchange": "ICE",
    "quantity": 80,
    "owner": "resource:org.example.mynetwork.Trader#TRADER2"
  },
  {
    "$class": "org.example.mynetwork.Commodity",
    "tradingSymbol": "EMA",
    "description": "Corn",
    "mainExchange": "EURONEXT",
    "quantity": 10,
    "owner": "resource:org.example.mynetwork.Trader#TRADER1"
  }
]
```


Figure 145 - The returned results.

Before using the "removeHighQuantityCommodities" transaction to delete those Commodity objects with a quantity greater than 60, let's first inspect the "selectCommoditiesWithHighQuantity" query which drive the transaction. Expand the "Query" section and select the "GET" operation for the "selectCommoditiesWithHighQuantity" query. Use the "Try it out!" button to test the query, and inspect the results.

Response Body


```
[
  {
    "$class": "org.example.mynetwork.Commodity",
    "tradingSymbol": "CC",
    "description": "Cocoa",
    "mainExchange": "ICE",
    "quantity": 80,
    "owner": "resource:org.example.mynetwork.Trader#TRADER2"
  }
]
```


Figure 146 - Only those Commodity objects with a quantity value over 60 are returned.

The objects returned by this query will be the ones deleted once the transaction is invoked. Go ahead and invoke the "removeHighQuantityCommodities" transaction by expanding its section and going to the POST operation. You can test the transaction using the "Try it out!" button, no input data is required. Verify the transaction was successful.

Response Body


```
{
  "$class": "org.example.mynetwork.RemoveHighQuantityCommodities",
  "transactionId": "4f44fa03d21f04374017c506f0d87ffcff1775916e741ccaa2215d4b8aa570ed"
}
```


Response Code

200

Figure 147 - A successful transaction.

Finally, use the GET operation on the Commodity object to view all the remaining commodities. You will only see those with a quantity of less than 60.

Figure 148 - The remaining Commodity objects.

Section 6 - Identity Management and Access Control in Hyperledger Composer

Up until this point, all the solutions you've been working with have only had one user and fully open permissions. In this section you will create multiple user identities and then add ACL rules to your permissions file to test access.

Lab 1 - Working with Multiple Identities

This lab uses the online Composer Playground environment to create mutiple user identites and test access rules. This solution will work with two different Participant types, Traders and Regulators. You will create and test the following rules for the solution:

- Traders have the ability to:
    - View and update their own profile
    - View and update their own Assets, but not the Assets of others
    - View history of their Transactions, but not the Transactions of others
    - The Trade Transaction will only be available to Trader Participants
- A regulator Participant can see the full transaction history for all traders

Step 1 - Create the Solution

You will start this lab by deploying the trade-network solution template.

Right-click the Chrome icon and select "New Incognito Window"

Figure 149 - Launching the Chrome browser in Incognito Window. Doing so prevents any cached objects from creating conflicts in Composer Playground.

Navigate to Composer Playground at https://composer-playground.mybluemix.net

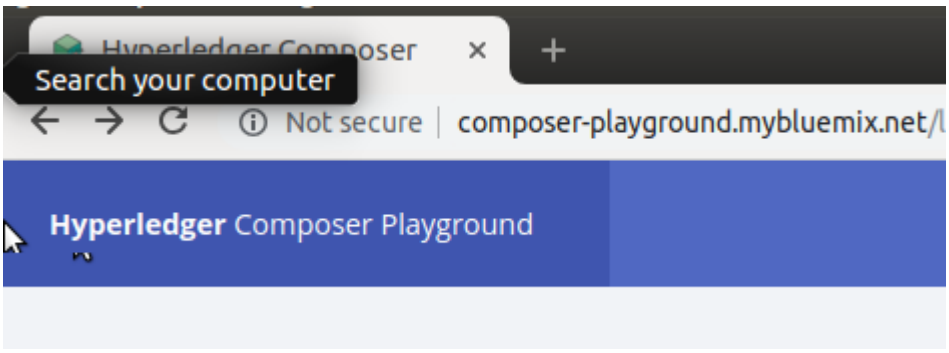


Figure 150 - Navigating to the Composer Playground site.

Select the option to deploy a new business network

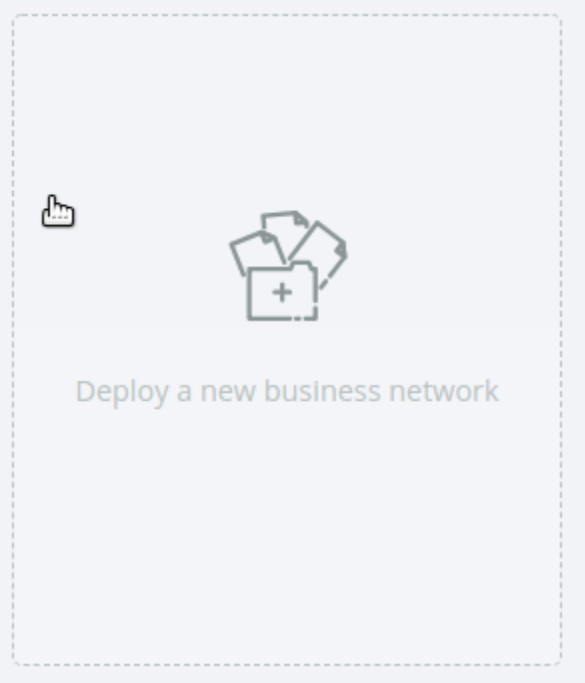


Figure 151 - Deploying a new Business Network.

Select the "trade-network" template and click "Deploy"

Figure 152 - Select the trade-network template.

Connect to the "trade-network" solution

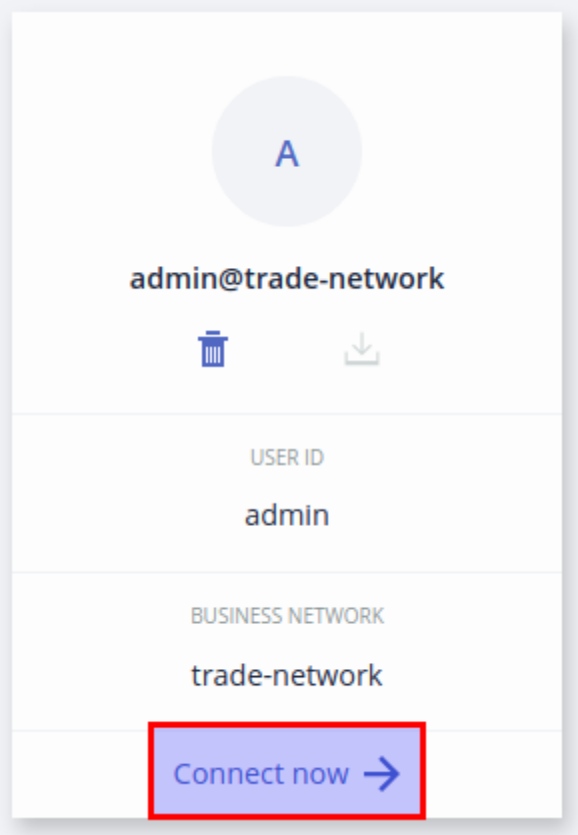


Figure 153 - Connecting to the trade-network solution.

Step 2 - Create Trader Participants

In this step you will create six new Trader Participants using the data provided in this step.

Click on the Test tab

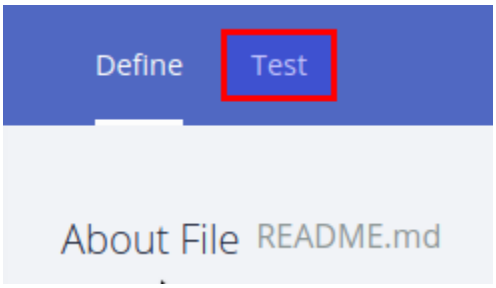


Figure 154 - Switch to the Test view.

Select the Trader Participant and click the "Create New Participant" button to add a new Trader. Use the information below:

```
{
      "$class": "org.example.trading.Trader",
      "tradeId": "TRADER1",
      "firstName": "Jenny",
      "lastName": "Jones"
}
```

Repeat this process for Traders 2 through 6. Use the information below:

```
{
    "$class": "org.example.trading.Trader",
    "tradeId": "TRADER2",
    "firstName": "Jack",
    "lastName": "Sock"
}
```

```
{
  "$class": "org.example.trading.Trader",
```

```
  "tradeId": "TRADER3",
  "firstName": "Rainer",
  "lastName": "Valens"
}

{
  "$class": "org.example.trading.Trader",
  "tradeId": "TRADER4",
  "firstName": "Davor",
  "lastName": "Dolittle"
}

{
  "$class": "org.example.trading.Trader",
  "tradeId": "TRADER5",
  "firstName": "Steve",
  "lastName": "Alonso"
}

{
  "$class": "org.example.trading.Trader",
  "tradeId": "TRADER6",
  "firstName": "Lars",
  "lastName": "Graf"
}
```

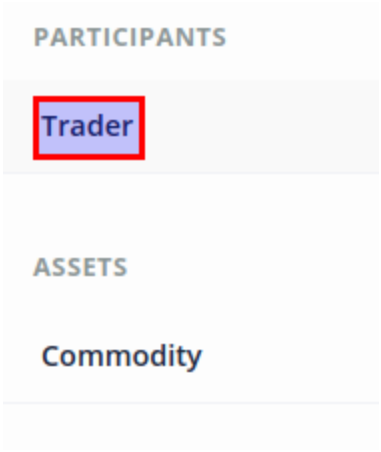


Figure 155 - Select the Trader Participant from the Test view.

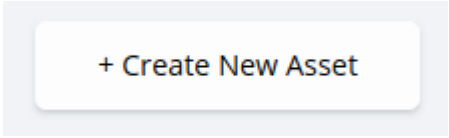


Figure 156 - Select the Create New Participant option.

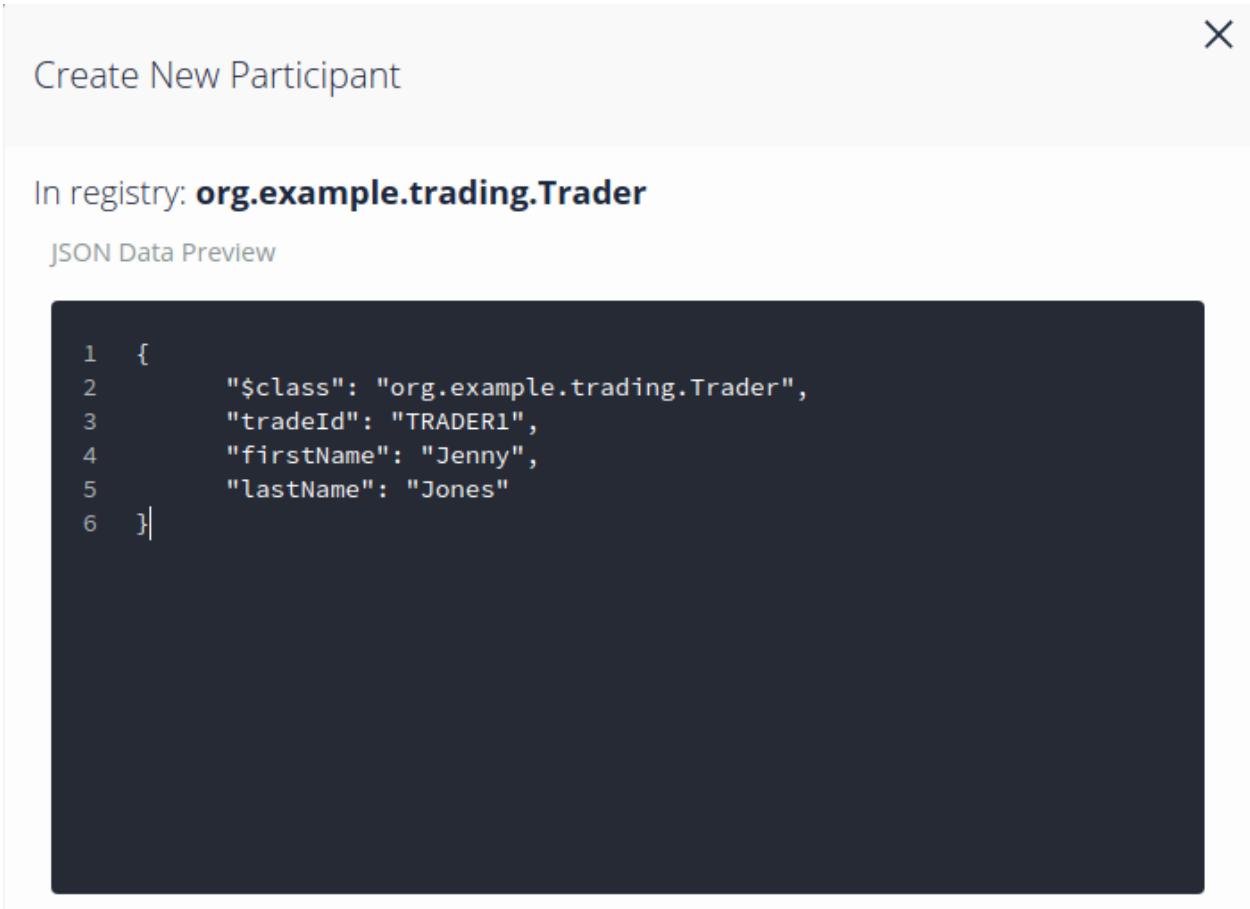


Figure 157 - Creating the new Participant record.

Step 3 - Create Commodity Assets

In this step you will create six Commodity Assets.

Repeat the process from the last step to create six Commodity records using the data below. NOTE: Make sure you create new Commodity objects, **NOT** new Trader objects!

```
{
  "$class": "org.example.trading.Commodity",
  "tradingSymbol": "EMA",
  "description": "Corn",
  "mainExchange": "EURONEXT",
  "quantity": 10,
  "owner": "resource:org.example.trading.Trader#TRADER1"
}

{
  "$class": "org.example.trading.Commodity",
  "tradingSymbol": "CC",
  "description": "Cocoa",
  "mainExchange": "ICE",
  "quantity": 80,
  "owner": "resource:org.example.trading.Trader#TRADER2"
}

{
  "$class": "org.example.trading.Commodity",
  "tradingSymbol": "HO",
  "description": "Heating Oil",
  "mainExchange": "NYMEX",
  "quantity": 40,
  "owner": "resource:org.example.trading.Trader#TRADER3"
}
```

```
{
  "$class": "org.example.trading.Commodity",
  "tradingSymbol": "HG",
  "description": "Copper",
  "mainExchange": "COMEX",
  "quantity": 100,
  "owner": "resource:org.example.trading.Trader#TRADER4"
}

{
  "$class": "org.example.trading.Commodity",
  "tradingSymbol": "SM",
  "description": "Soybean Meal",
  "mainExchange": "CBOT",
  "quantity": 70,
  "owner": "resource:org.example.trading.Trader#TRADER5"
}

{
  "$class": "org.example.trading.Commodity",
  "tradingSymbol": "AG",
  "description": "Silver",
  "mainExchange": "CBOT",
  "quantity": 60,
  "owner": "resource:org.example.trading.Trader#TRADER6"
}
```

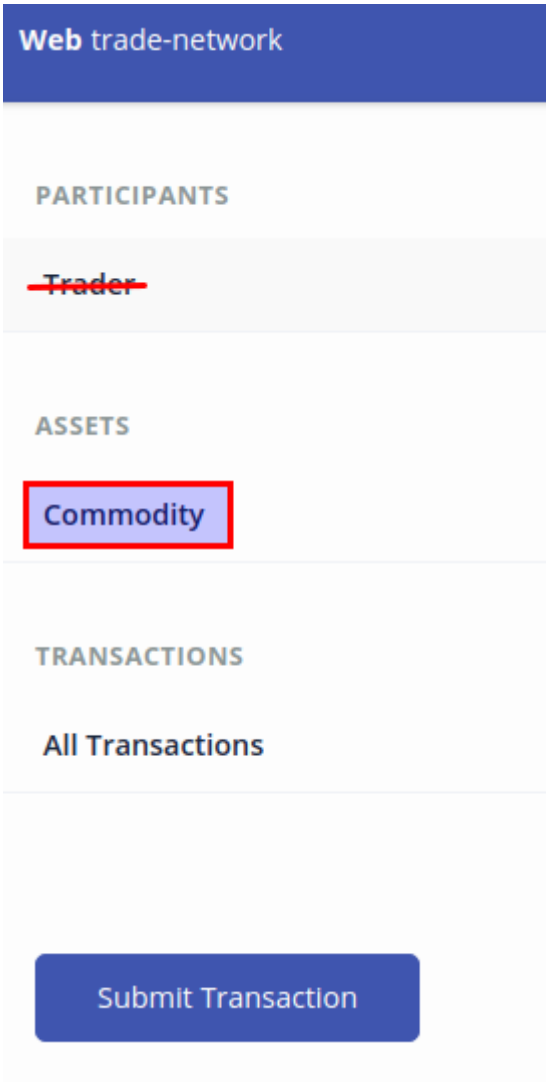


Figure 158 - Select the Commodity Asset from the Test view.

Step 4 - Create new Identity records and map each to a Participant

In this step you will create new Identities for your solution and map each to a Participant record.

In the top-right corner, click the "admin" button and select "ID Registry" from the drop-down menu

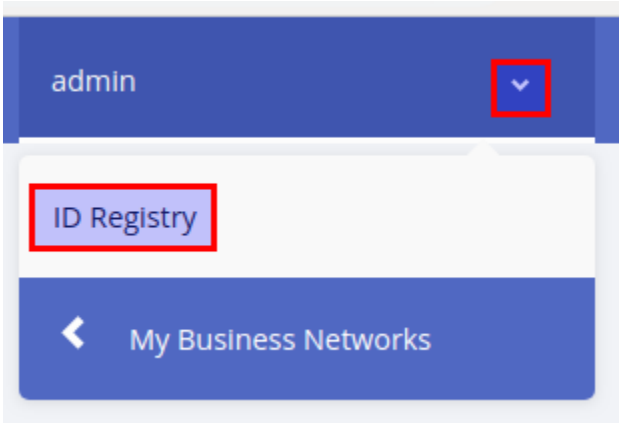


Figure 159 - Accessing the ID Registry feature in Composer.

In the top-right corner, click the "Issue New ID" button

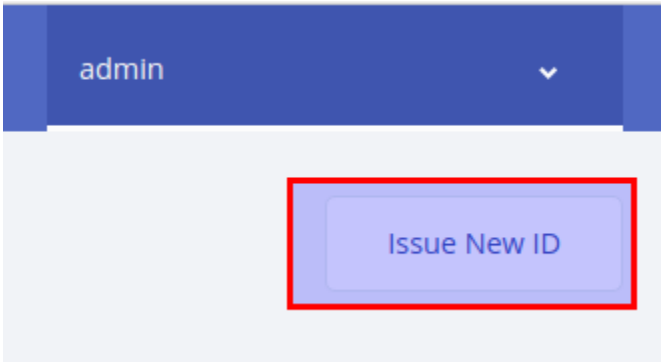


Figure 160 - To create a new Identity use the Issue New ID option.

In the "Issue New Identity" window enter "tid1" as the "ID_Name" value. In the "Participant" window enter "TRADER1" to search for the Participant.

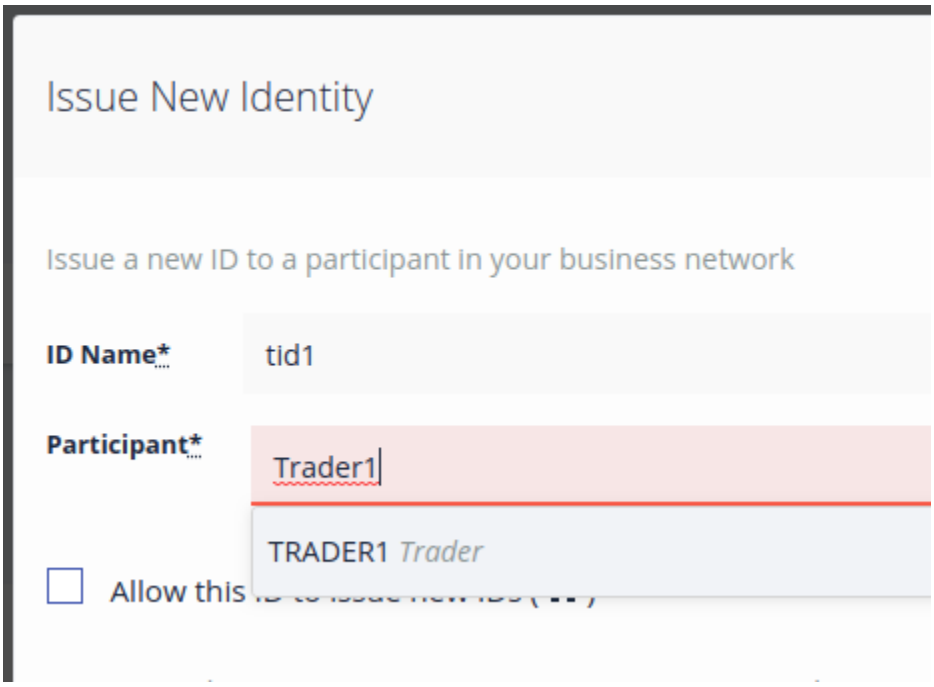


Figure 161 - Creating the tid1 Identity.

Participant Identity should resolve to a fully-qualified name

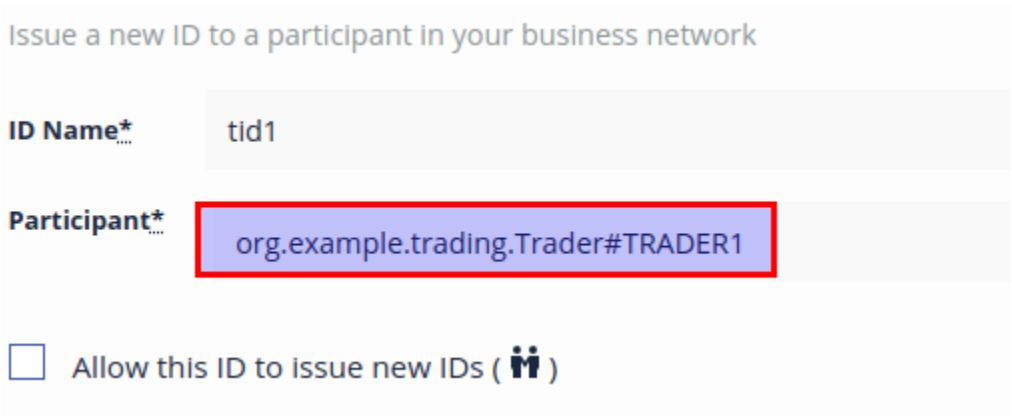


Figure 162 - The resolution to a fully-qualified Participant record.

Click the "Create New" button to create a new Identity

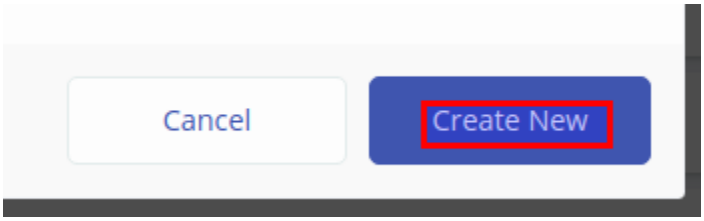


Figure 163 - Clicking the Create New option will create the new Identity.

Repeat the above process to create an Identity for tid2, tid3, tid4, tid5, and tid6

Figure 164 - You should have a total of six Trader Identity records, tid1 - tid6.

Step 5 - Remove the Default Access Rule

By default, all users of this solution will have full access due to the "Default" rule in the permissions.acl file. This rule must first be removed, then additional rules will be created.

Click on the Define tab at the top of the window

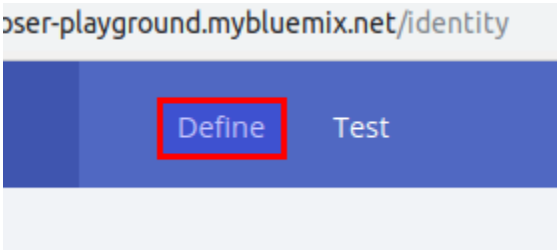


Figure 165 - Selecting the Define view.

Select the "permissions.acl" file

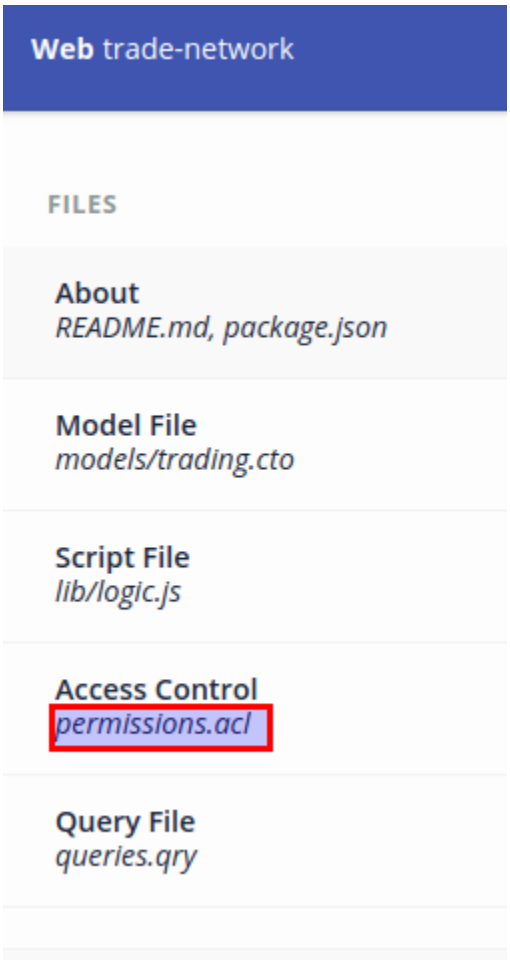


Figure 166 - Accessing the permissions.acl file.

Delete the entire "Default" rule, leave all other rules as they are.

ACL File permissions.acl

```
 * limitations under the License.
 */
/**
 * Access control rules for mynetwork
 */
rule Default {
    description: "Allow all participants access to all resources"
    participant: "ANY"
    operation: ALL
    resource: "org.example.trading.*"
    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 {
```

Figure 167 - Removing the default rule.

In the bottom-left corner of the window, click on the "Deploy changes" button

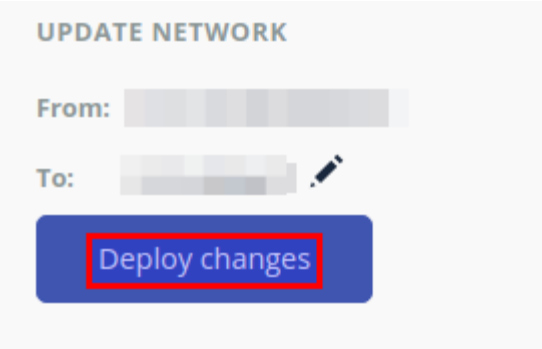


Figure 168 - To update your solution, re-deploy it using the Deploy changes button.

Step 6 - Create a rule enforcing that Traders can only see and update their own profile record.

In this step, you will create the rule that allows for a Trader to view and edit their own profile, but not the profiles of anyone else.

Click on the "Test" tab at the top of the window. Select the "admin" button on the top-right, then select the "ID Registry" option.

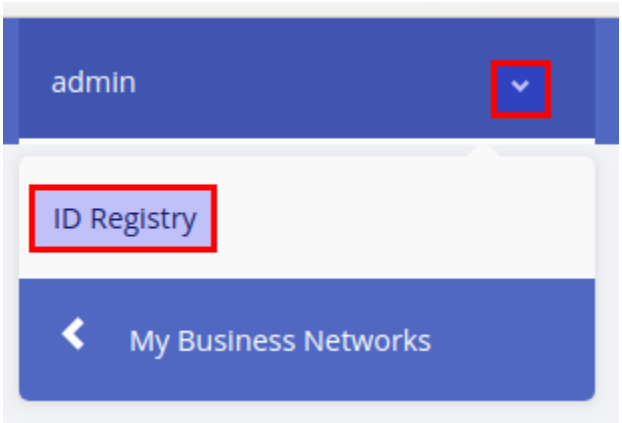


Figure 169 - Accessing the ID Registry feature.

Select the tid1 Identity, then select the "Use now" option.

Figure 170 - Select the tid1 Identity

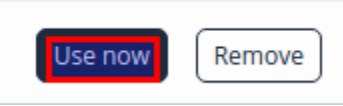


Figure 171 - Once the tid1 Identity has been selected, click the Use now option to change your Identity.

You should see the "tid1" identity appear in the drop-down box in the top-right corner.

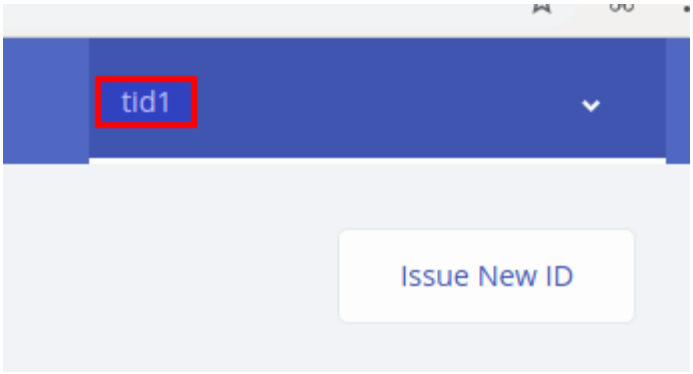


Figure 172 - Look for the tid1 Identity to ensure you've successfully changed your Identity.

Select the "Test" tab at the top of the window, then select Trader under the "PARICIPANTS" section. Verify that no Trader Participants are visible. Remember that by default all access is denied in Composer unless a rule specifically grants access. The "Default" rule from the Access Control file which granted access has been removed and no new rules have been added.

Figure 173 - As tid1 you will not be able to view any Participant records.

Verify no Commodity objects are visible using the "tid1" Identity.

Figure 174 - As tid1 you will not be able to view any Commodity records.

Step 7 - Create a rule enforcing that Traders can only see and update their own Commodity records.

In this step you will create the rule that allows Traders to see and edit their own Commodity Assets. You will then test this rule.

Go back to the "ID Registry" and select the "admin" user.

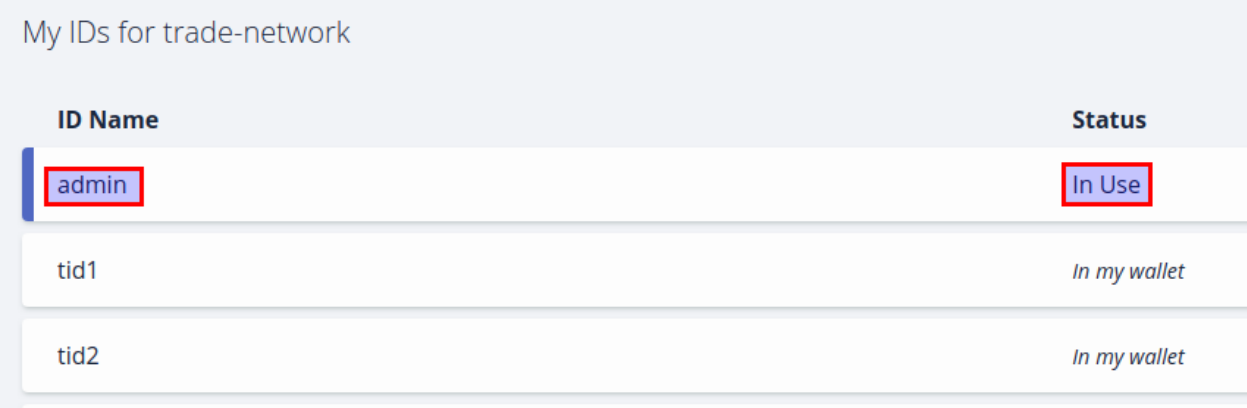


Figure 175 - Switch back to the admin user using the ID Registry function.

Click on the "Define" tab and select the "permissions.acl" file. Place the following rule at the top of the Access Control file (just below the comments, on line 17)

```
rule R1a_TraderSeeUpdateThemselvesOnly {
  description: "Trader can see and update their own record only"
  participant(t): "org.example.trading.Trader"
  operation: READ, UPDATE
  resource(v): "org.example.trading.Trader"
  condition: (v.getIdentifier() == t.getIdentifier())
  action: ALLOW
}
```

```
13   */
14  /**
15   * Access control rules for mynetwork
16   */
17  rule R1a_TraderSeeUpdateThemselvesOnly {
18    description: "Trader can see and update their own record only"
19    participant(t): "org.example.trading.Trader"
20    operation: READ, UPDATE
21    resource(v): "org.example.trading.Trader"
22    condition: (v.getIdentifier() == t.getIdentifier())
23    action: ALLOW
24  }
25
26  rule SystemACL {
27      description:  "System ACL to permit all access"
```

Figure 176 - Adding the new rule to the permissions.acl file.

In the bottom-left corner of the window, click on the "Deploy changes" button

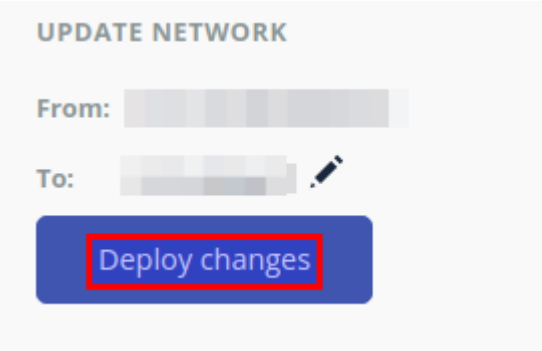


Figure 177 - Re-deploy the solution for the changes to take effect.

Select the tid1 Identity, then select the "Use now" option.

Figure 178 - Select the tid1 Identity in the ID Registry.

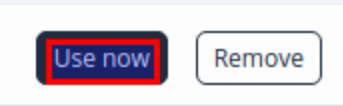


Figure 179 - Switch your Identity using the Use now option.

You should see the "tid1" identity appear in the drop-down box in the top-right corner.

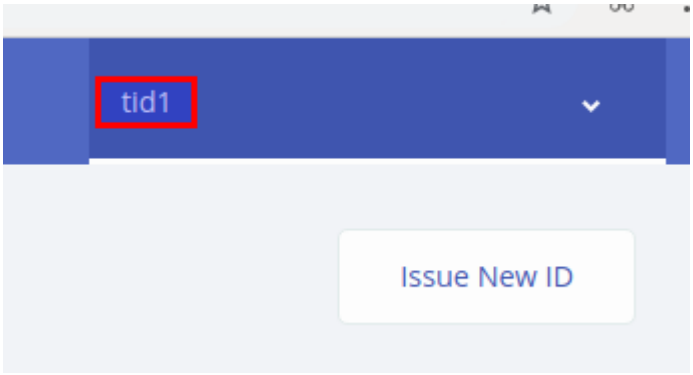


Figure 180 - Verify you are now using the tid1 Identity.

Select the "Test" tab at the top of the window, then select Trader under the "PARTICIPANTS" section. Verify that the identity "tid1" can only see the Trader object for "TRADER1" (Jenny Jones).

Figure 181 - You will now be able to see your own Participant record, but not the record of anyone else.

Use the "ID Registry" to switch back to the "admin" Identity. Click on the "Define" tab at the top and navigate to your "permissions.acl" file. Paste the following rule at the top of the Access Control file (below the comments, on line 17)

```
rule R1b_TraderSeeTheirCommodities {
  description: "Trader can see/work with their own Commodities"
  participant(t): "org.example.trading.Trader"
  operation: ALL
  resource(c): "org.example.trading.Commodity"
  condition: (c.owner.getIdentifier() == t.getIdentifier())
  action: ALLOW
}
```

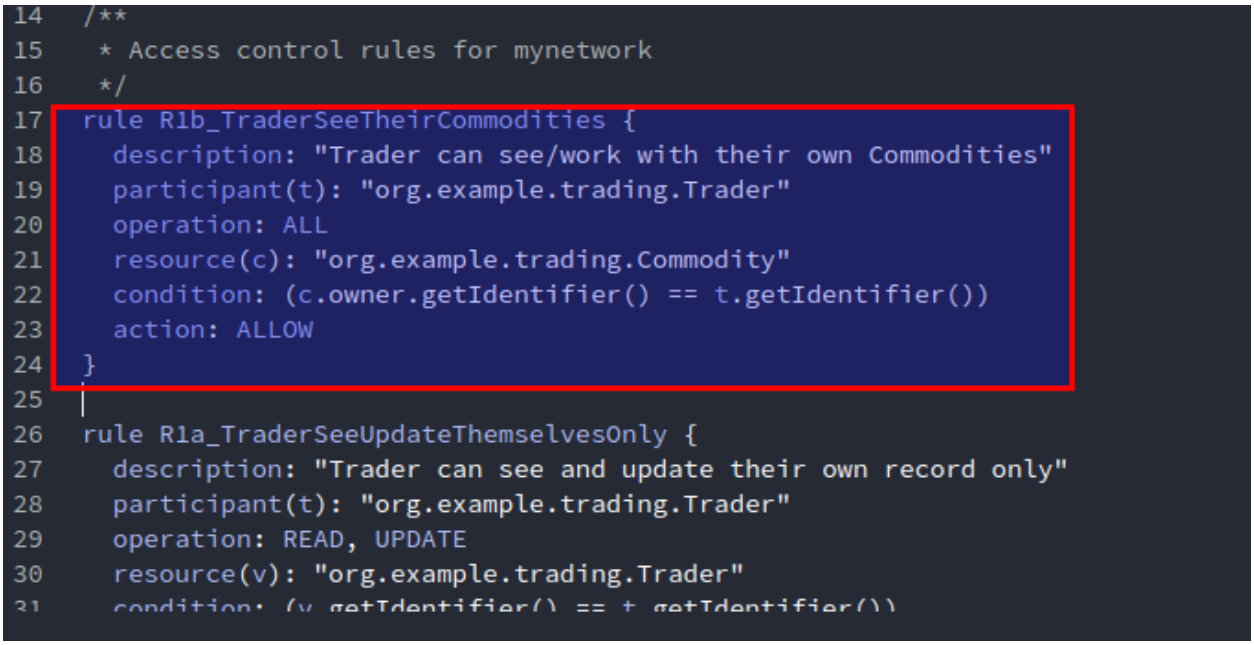


Figure 182 - Adding the new rule to permissions.acl

In the bottom-left corner of the window, click on the "Deploy changes" button.

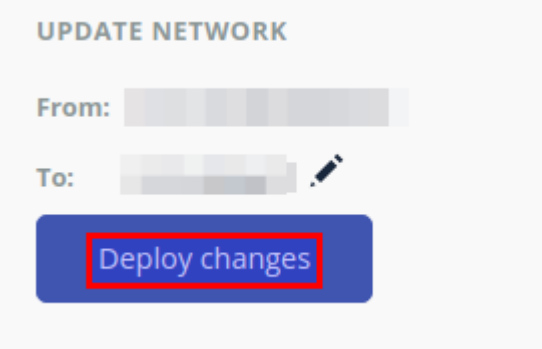


Figure 183 - Re-deploy the solution for changes to take effect.

Using the "tid1" Identity, verify that the only Commodity Asset visible is "EMA" (Corn).

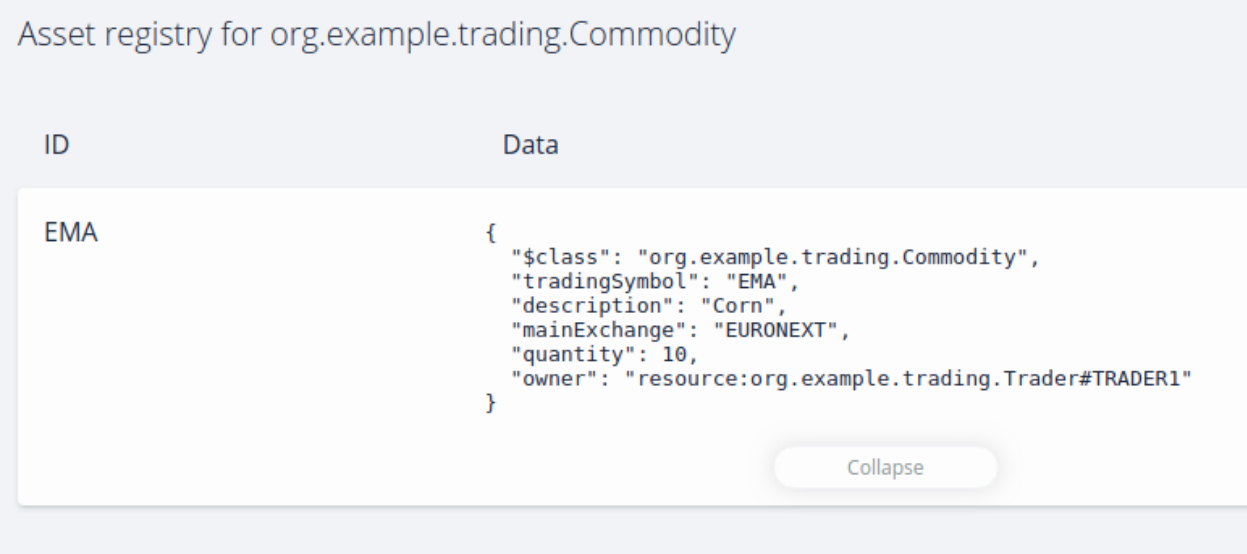


Figure 184 - You will now be able to see the records of Commodities you own, but not the Commodity records of anyone else.

Step 8 - Create a rule enforcing that only Traders can submit Trade transactions.

In this step, you will create the rule that allows Traders to submit Trade transactions.

Using the "tid1" Identity, click on the "Test" tab and select "Submit Transaction"

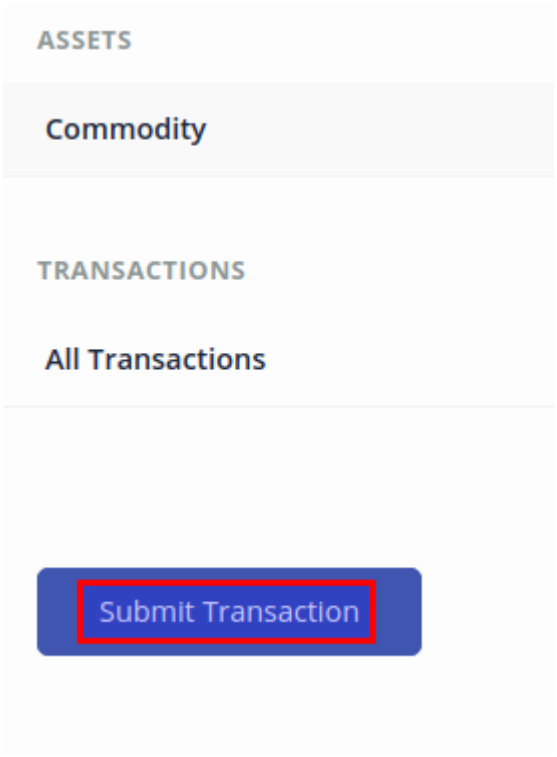


Figure 185 - Use the Submit Transaction option to submit a new Transaction.

Select the "Trade" Transaction Type.

Figure 186 - Select the Trade transaction from the list of available transactions.

Enter the following Transaction data:

```
{
  "$class": "org.example.trading.Trade",
  "commodity": "resource:org.example.trading.Commodity#EMA",
  "newOwner": "resource:org.example.trading.Trader#TRADER2"
}
```

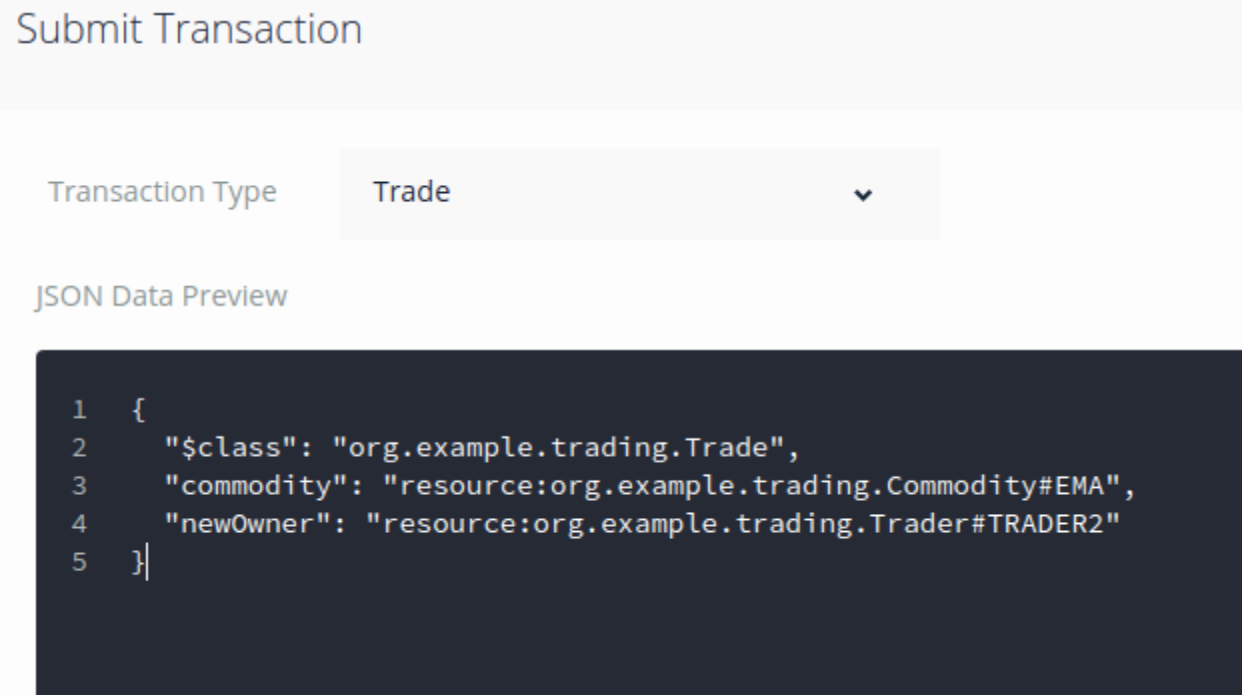


Figure 187 - Transaction data to submit.

Click the "Submit" button.

Figure 188 - Use the Submit option to submit the Transaction data you entered above.

Verify that you receive the error depicted below. NOTE: The specific transaction ID will not match the one you receive, this is okay.

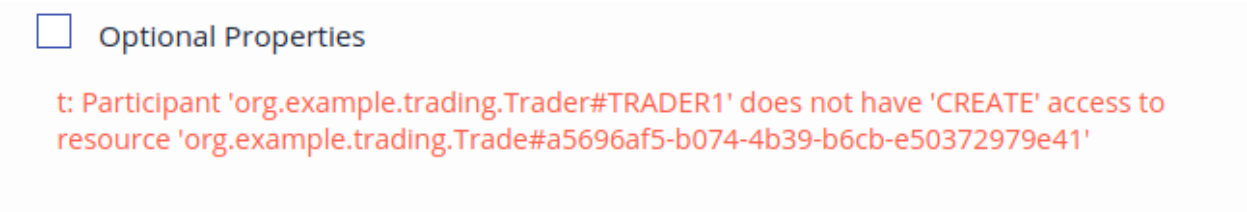


Figure 189 - You should receive an error when trying to submit the transaction.

Switch back to the "admin" identity and add the following rule to the top of the "permissions.acl" file.

```
rule R2_EnableTradeTxn {
    description: "Enable Traders to submit transactions"
    participant: "org.example.trading.Trader"
    operation: ALL
    resource: "org.example.trading.Trade"
    action: ALLOW
}
```

ACL File permissions.acl

```
 9    * distributed under the License is distributed on an "AS IS" BASIS,
10    * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
11    * See the License for the specific language governing permissions and
12    * limitations under the License.
13    */
14   /**
15    * Access control rules for mynetwork
16    */
17   rule R2_EnableTradeTxn {
18       description: "Enable Traders to submit transactions"
19       participant: "org.example.trading.Trader"
20       operation: ALL
21       resource: "org.example.trading.Trade"
22       action: ALLOW
23   }
24
25   rule R1b_TraderSeeTheirCommodities {
26     description: "Trader can see/work with their own Commodities"
27     participant(t): "org.example.trading.Trader"
28     operation: ALL
29     resource(c): "org.example.trading.Commodity"
30     condition: (c.owner.getIdentifier() == t.getIdentifier())
```

Figure 190 - Adding a new rule to the permissions.acl file.

In the bottom-left corner of the window, click on the "Deploy changes" button.

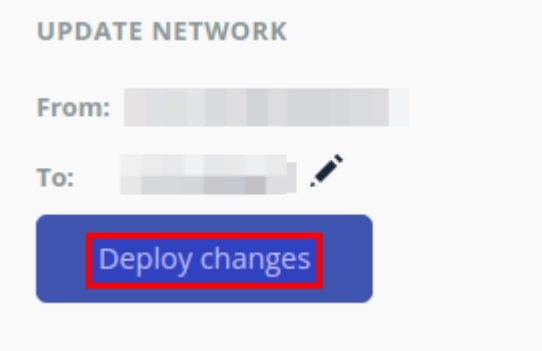


Figure 191 - Re-deploy for changes to take effect.

Switch to the "tid1" Identity. Attempt to submit another "Trade" transaction using the data below.

```
{
  "$class": "org.example.trading.Trade",
  "commodity": "resource:org.example.trading.Commodity#EMA",
  "newOwner": "resource:org.example.trading.Trader#TRADER2"
}
```

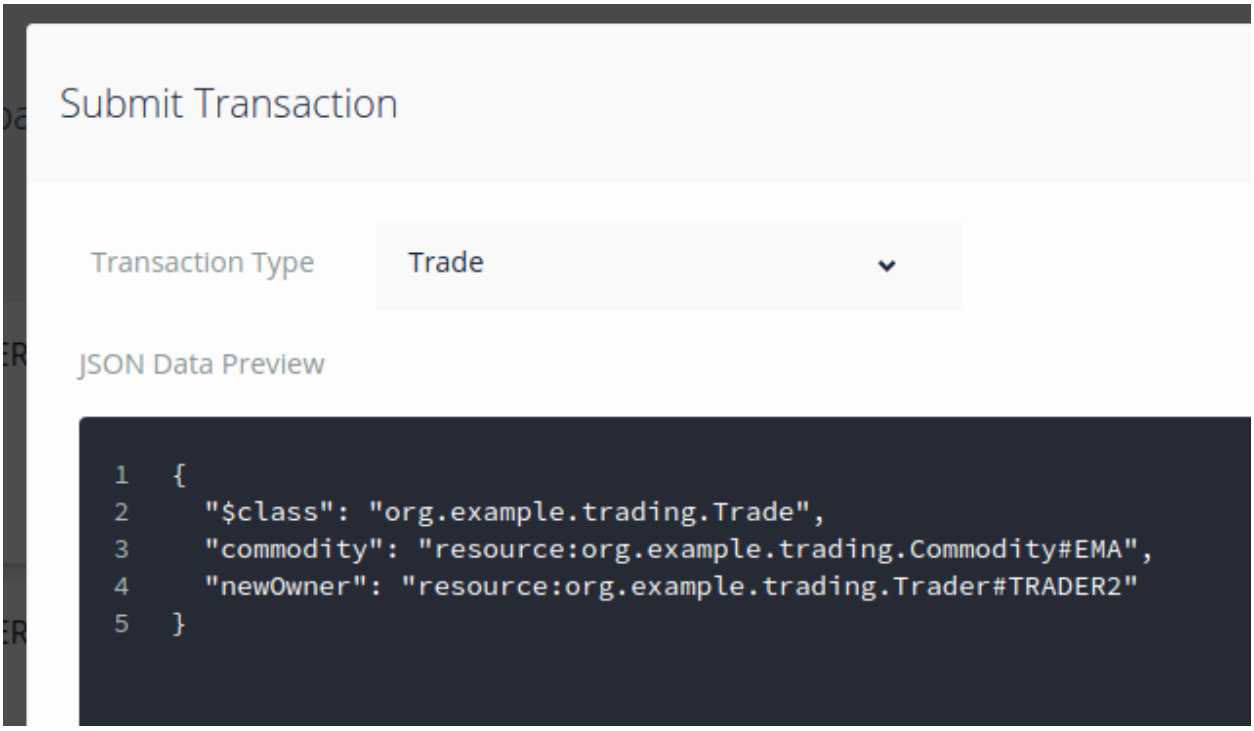


Figure 192 - Data for a new Trade Transaction.

To confirm the transaction, click on the "All Transactions" option under the "TRANSACTIONS" section. Verify the "Trade" transaction was recorded on the Transaction Log. Click the "view record" option to view Transaction details.

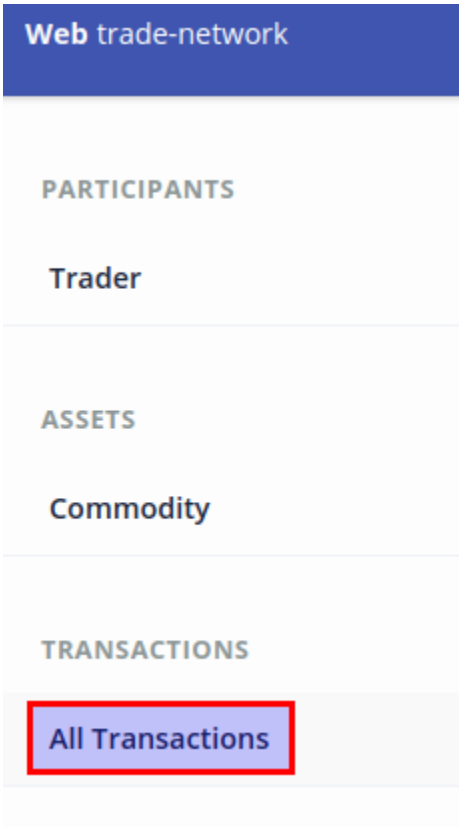


Figure 193 - Select the All Transactions option.

Date, Time	Entry Type	Participant	
2019-02-08, 15:50:25	Trade	admin (NetworkAdmin)	view record
2019-02-08, 15:01:44	ActivateCurrentIdentity	none	view record
2019-02-08, 14:46:15	IssueIdentity	admin (NetworkAdmin)	view record

Figure 194 - Viewing the Transaction log.

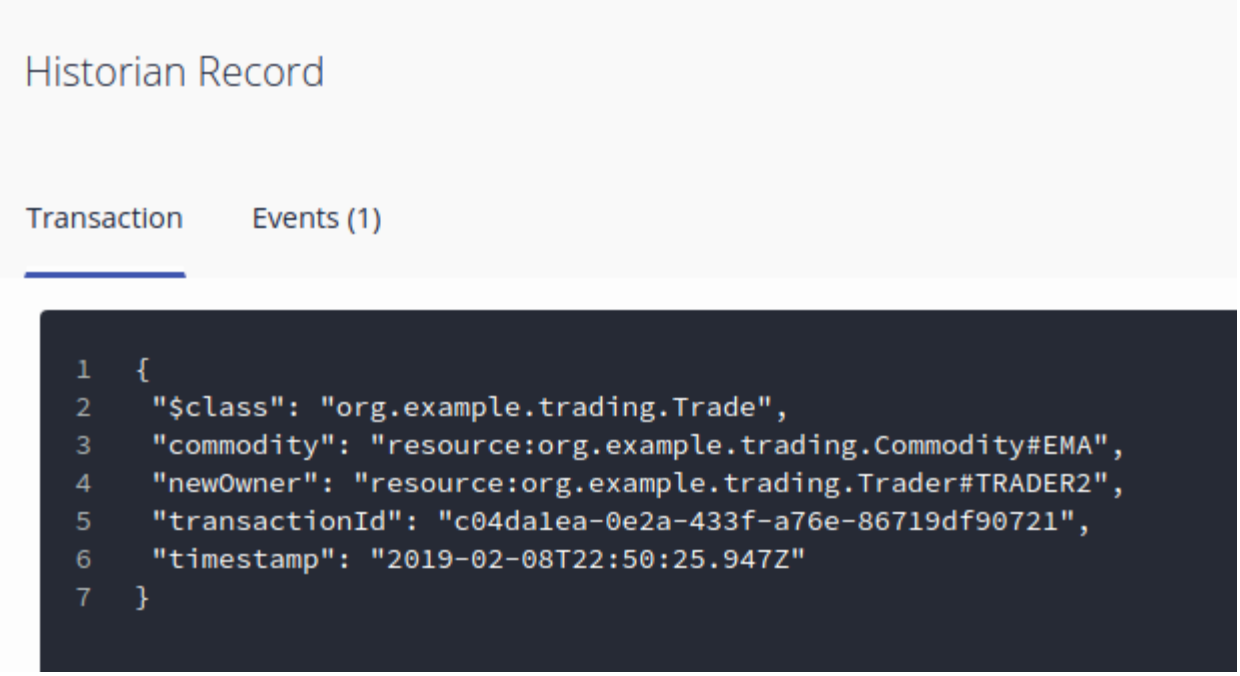


Figure 195 - Viewing the Trade Transaction detail.

You can also verify the success of the transaction by switching between "tid1" and "tid2". Use the "ID Registry" to switch between Identities. Verify that Identity "tid1" is unable to view any Commodity Assets. Remember that this Identity traded away their Commodity in the "Trade" transaction. Using Identity "tid2", verify that you can see two Commodity Assets in the Registry (CC/Cocoa and EMA/Corn). NOTE: When switching between Identities you may have to first switch to the "admin" Identity, then to the desired second Identity.

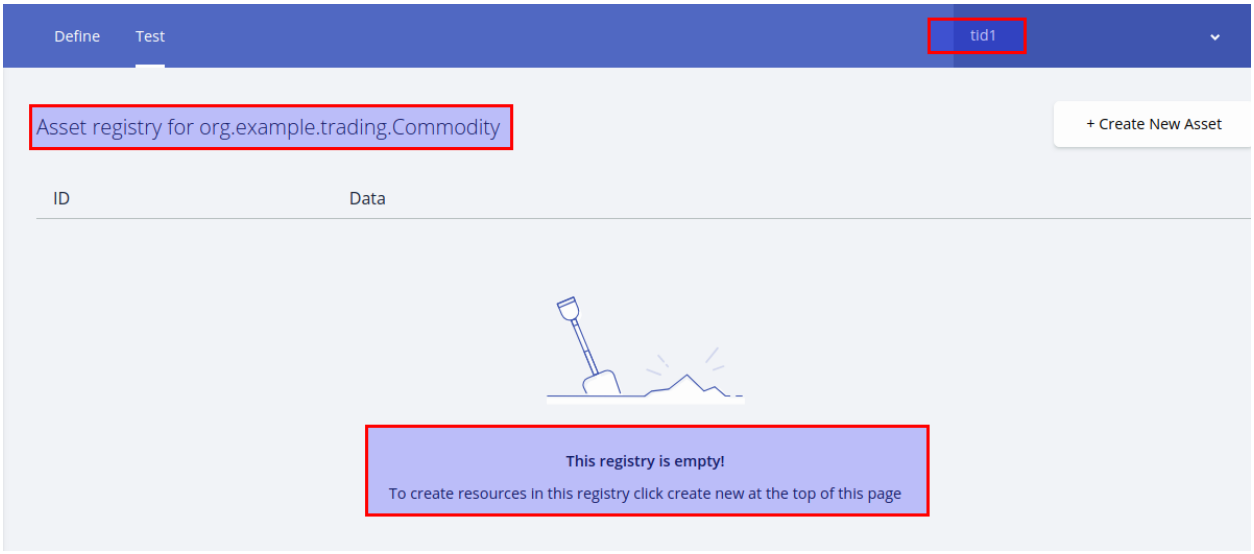


Figure 196 - When using the tid1 Identity, no Assets are visible in the Asset Registry.

Figure 197 - Using the tid2 Identity, two Commodity Assets will be visible in the Asset registry.

Step 9 - Create a rule allowing Traders to see only their own records in the Transaction Log.

In this step, you will create the rule that allows a Trader to view their own Transaction records only.

First verify that all transactions are visible to all users. Switch to Identity "tid3" using the "ID Registry". Remember to first switch to the "admin" Identity when switching between users. Using the "tid3" Identity, view the Transaction Log ("All Transactions" under "TRANSACTIONS") and note that this Identity is able to view the "TRADE" transaction between "tid1" and "tid2".

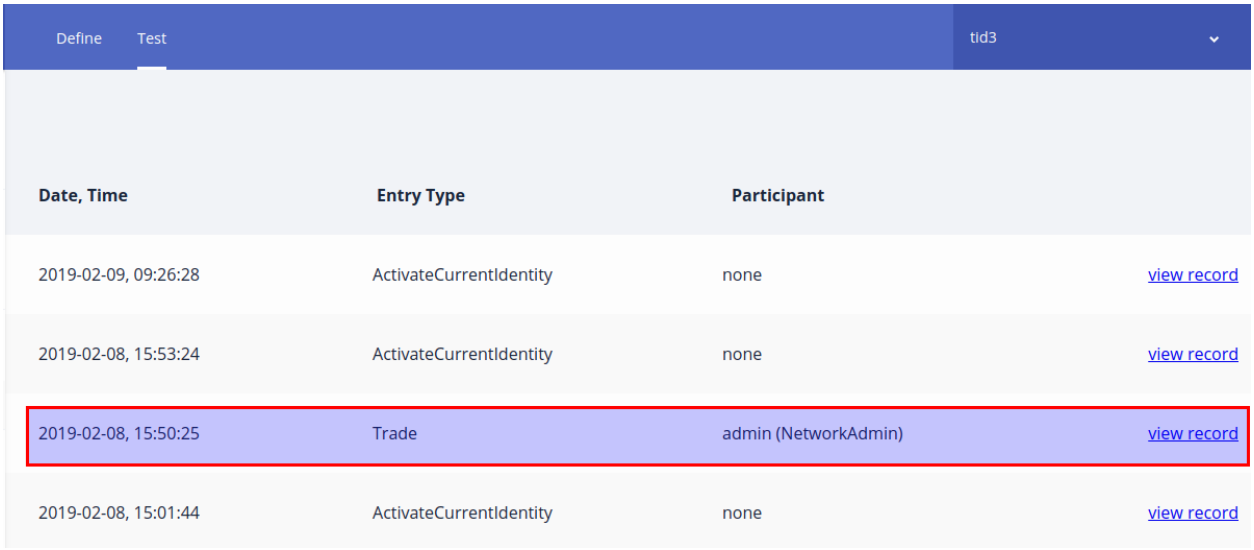


Figure 198 - Using the tid3 Identity all Transaction data is viewable, even those Transactions that did not involve tid3.

Figure 199 - tid3 is able to view a Transaction between tid1 and tid2.

Switch to the "admin" Identity, then add the rule below at the top of the "permissions.acl" file.

```
rule R3_TradersSeeOwnHistoryOnly {
  description: "Traders should be able to see the history of their own transactions only"
  participant(t): "org.example.trading.Trader"
  operation: READ
  resource(v): "org.hyperledger.composer.system.HistorianRecord"
  condition: (v.participantInvoking.getIdentifier() != t.getIdentifier())
  action: DENY
}
```

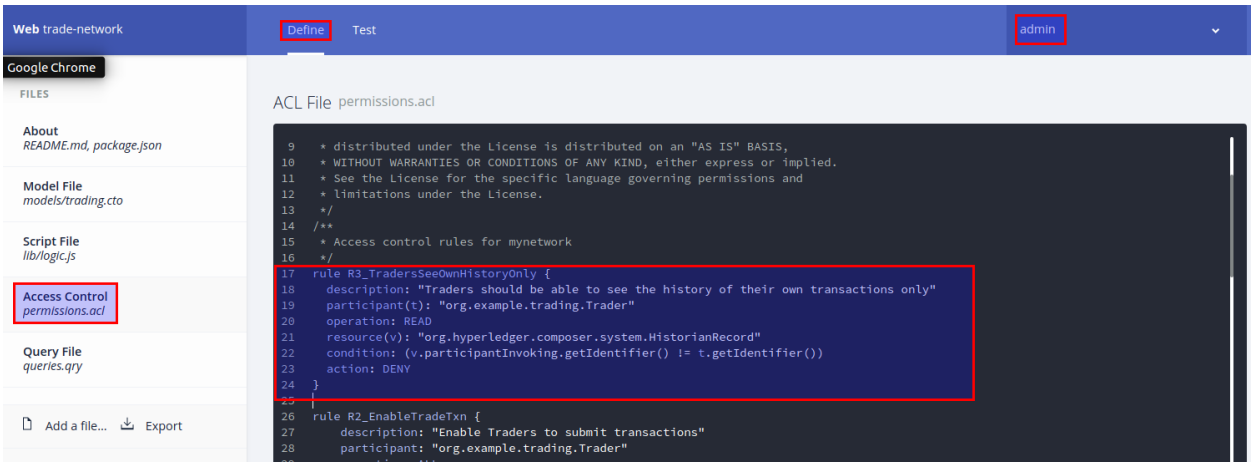


Figure 200 - Adding a new rule to the permissions.acl file.

In the bottom-left corner of the window, click on the "Deploy changes" button.

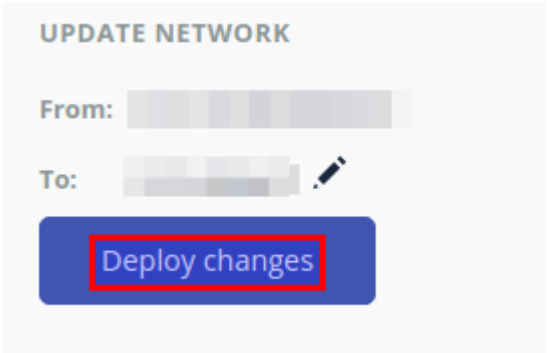


Figure 201 - Re-deploy the solution for changes to take effect.

Switch to the "tid3" Identity and view the Transaction Log again. The previous "TRADE" transaction should no longer be visible.

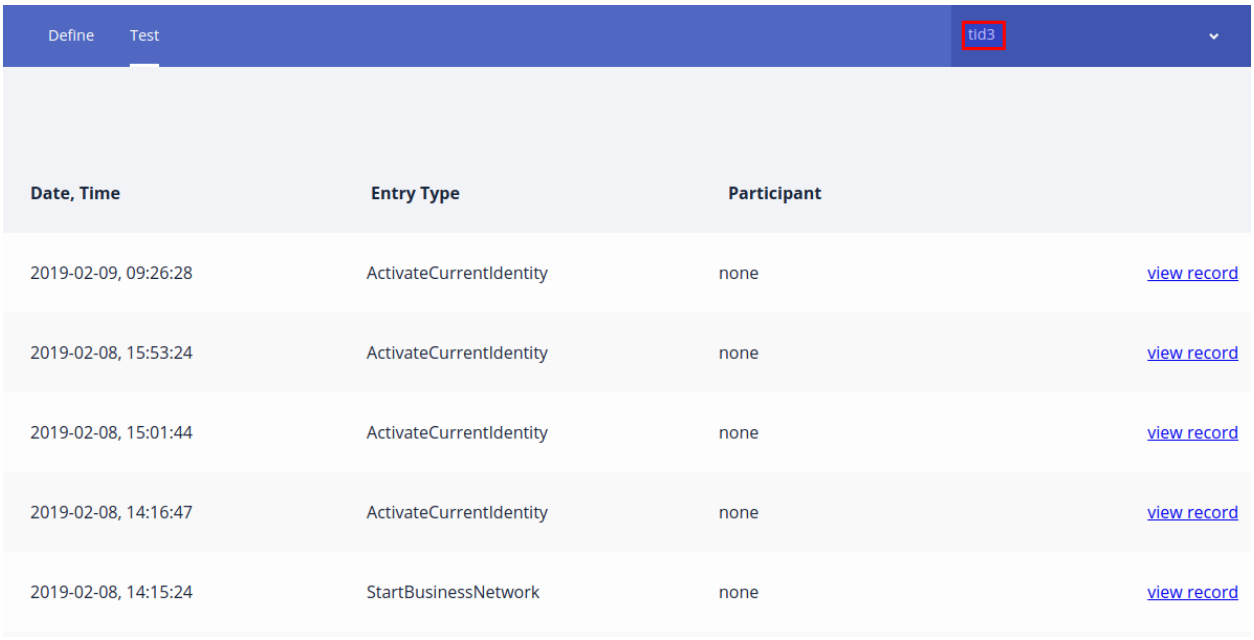


Figure 202 - tid3 is no longer able to view the Transaction between tid1 and tid2.

Step 10 - Create the rule allowing Regulators to see all Transactions

In this step you will create the rule which allows Regulator Participants to see all Transaction data as well as their own profile data.

Switch to the "admin" Identity, the update the "Model File" ("trading.cto") with the markup below. Add the markup below the "Trader" Participant.

```
participant Regulator identified by regId {
    o String regId
    o String firstName
    o String lastName
}
```

Model File models/trading.cto

```
23    o Double quantity
24    --> Trader owner
25  }
26
27  participant Trader identified by tradeId {
28    o String tradeId
29    o String firstName
30    o String lastName
31  }
32
33  participant Regulator identified by regId {
34      o String regId
35      o String firstName
36      o String lastName
37  }
38
39  transaction Trade {
40    --> Commodity commodity
41    --> Trader newOwner
42  }
43
44  event TradeNotification {
```

Figure 203 - Adding the Regulator participant to the Model file.

In the bottom-left corner of the window, click on the "Deploy changes" button.

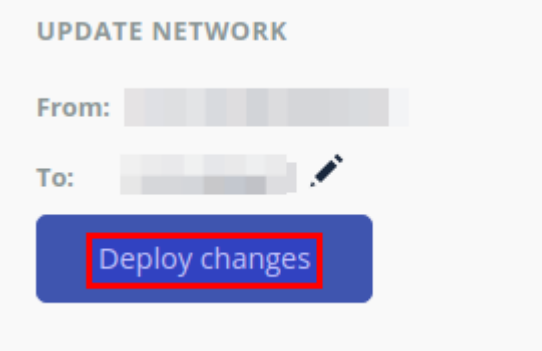


Figure 204 - Re-deploy the solution for changes to take effect.

Create a new "Regulator" Participant using the information below.

```
{
  "$class": "org.example.trading.Regulator",
  "regId": "Reg101",
  "firstName": "Jon",
  "lastName": "Doe"
}
```

Figure 205 - Creating a new Regulator Participant.

Figure 206 - The new Regulator Participant record.

Use the "Issue New ID" option in the "ID Registry" to create a new Identity with an ID Name of "101". Map this Identity to the Regulator you just created (Reg101). NOTE - At this point, the Regulator can now see the history of system transactions in Transaction Log, but cannot their own profile data.

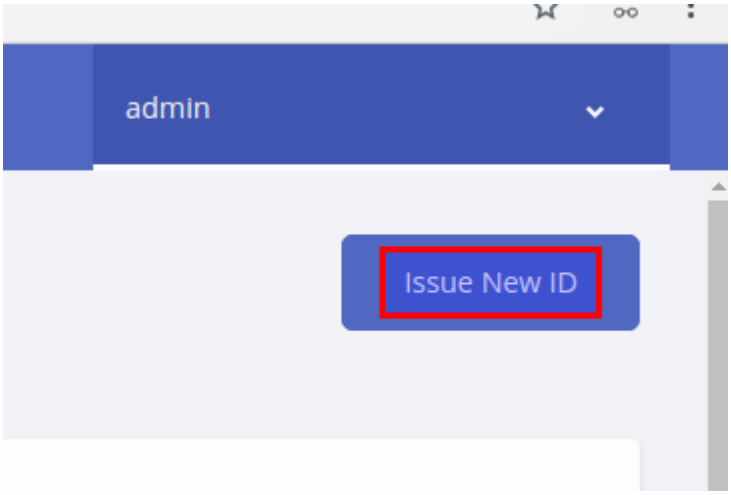


Figure 207 - Issue a new ID using the ID Registry feature.

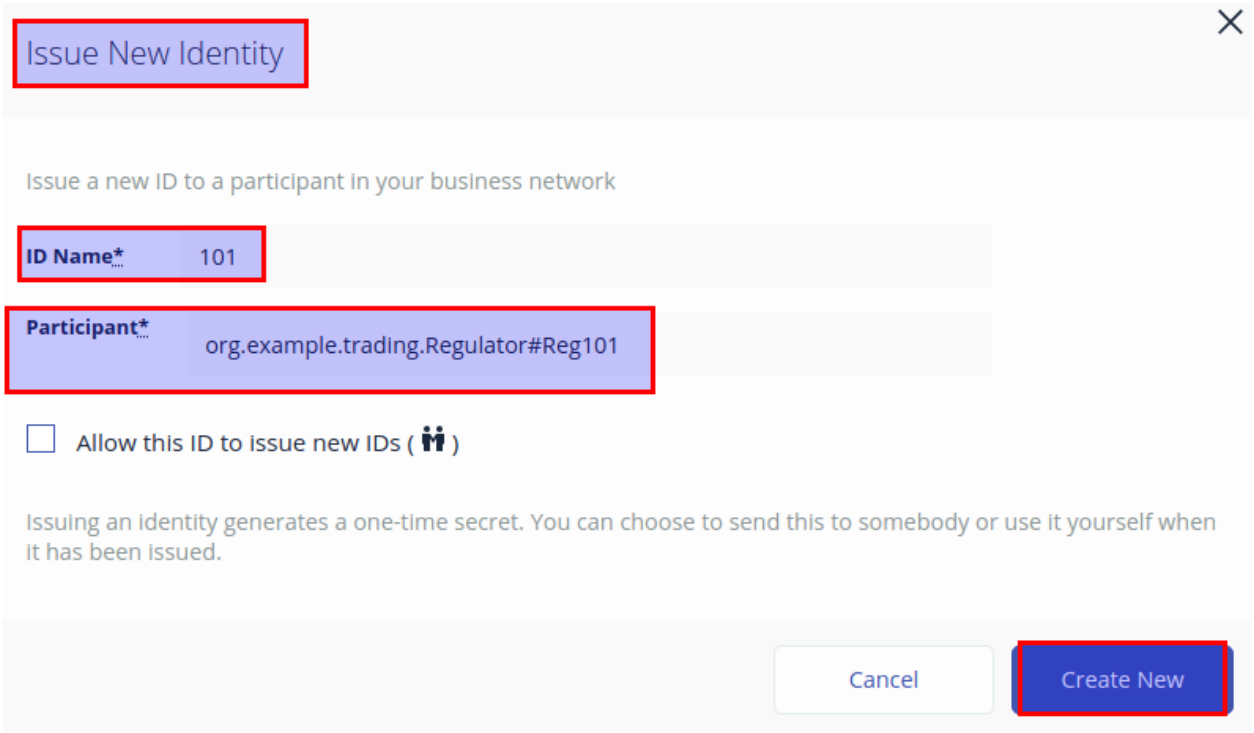


Figure 208 - Map the new Identity to the Regulator Participant record.

Add the rule below to the "permissions.acl" file.

```
rule R4a_RegulatorSeeThemselves {
  description: "Regulators can see and update their own record"
  participant: "org.example.trading.Regulator"
  operation: READ, UPDATE
  resource: "org.example.trading.Regulator"
  action: ALLOW
}
```

ACL File permissions.acl

```
11   * See the License for the specific language governing permissio
12   * limitations under the License.
13   */
14  /**
15   * Access control rules for mynetwork
16   */
17  rule R4a_RegulatorSeeThemselves {
18    description: "Regulators can see and update their own record"
19    participant: "org.example.trading.Regulator"
20    operation: READ, UPDATE
21    resource: "org.example.trading.Regulator"
22    action: ALLOW
23  }
24
25  rule R3_TradersSeeOwnHistoryOnly {
26    description: "Traders should be able to see the history of the
27    participant(t): "org.example.trading.Trader"
28    operation: READ
```

Figure 209 - Updating the permissions.acl file.

In the bottom-left corner of the window, click on the "Deploy changes" button.

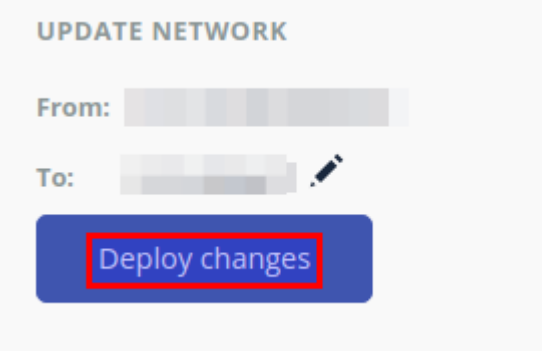


Figure 210 - Re-deploy your solution for changes to take effect.

Using the "ID Registry", switch to the "101" Identity for the Regulator. Verify that you can now view your own record in the Participant registry.

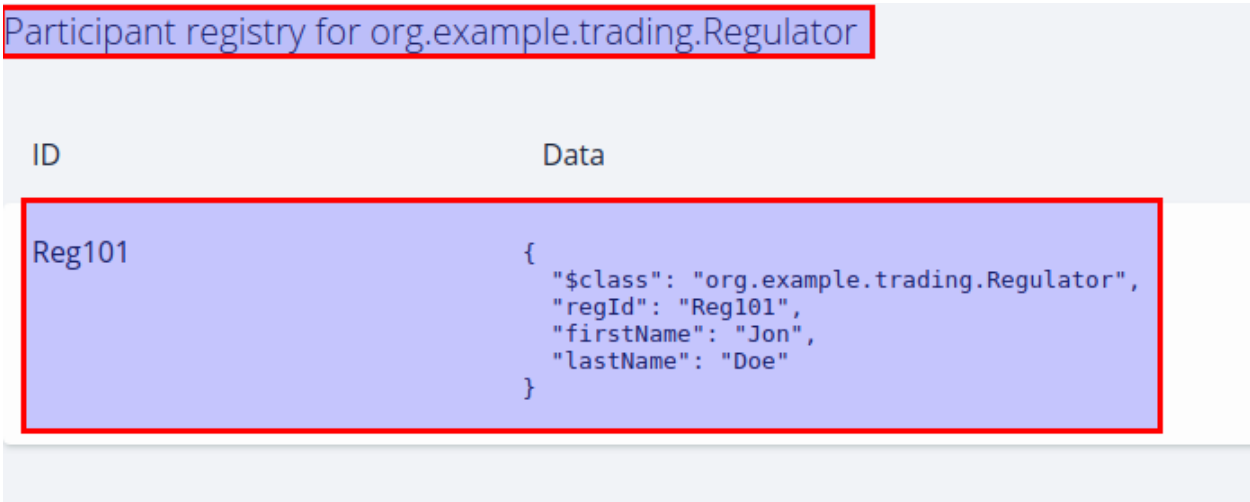
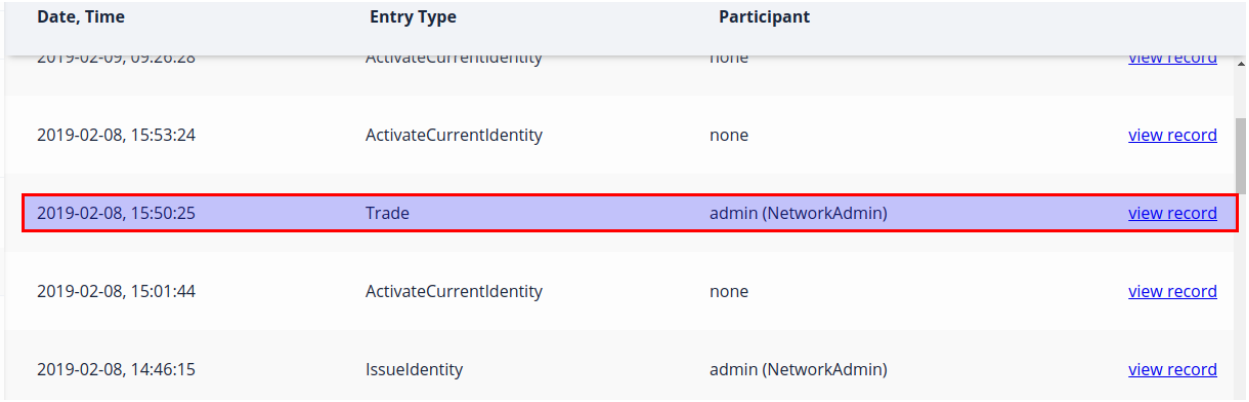


Figure 211 - The Regulator should be able to view their own record.

Now view the Transaction Log. Verify that you can view all Transactions, including the "TRADE" transaction made earlier. Try clicking on the "view record" option for the "TRADE" transaction - verify nothing happens. This is because the Regulator Identity does not have authority to view the transaction record.



Date, Time	Entry Type	Participant	
2019-02-08, 15:53:24	ActivateCurrentIdentity	none	view record
2019-02-08, 15:50:25	Trade	admin (NetworkAdmin)	view record
2019-02-08, 15:01:44	ActivateCurrentIdentity	none	view record
2019-02-08, 14:46:15	IssueIdentity	admin (NetworkAdmin)	view record

Figure 212 - You can view all Transactions in the Transaction Log, but are not able to view any Transaction details.

Switch back to the "admin" Identity. Add the rule below to the top of the "permissions.acl" file.

```
rule R4b_RegTransView {
    description: "Grant Regulator full access to Trade Transactions"
    participant: "org.example.trading.Regulator"
    operation: ALL
    resource: "org.example.trading.Trade"
    action: ALLOW
}
```

ACL File permissions.acl

```
 9    * distributed under the License is distributed on an "AS IS" BASIS,
10    * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
11    * See the License for the specific language governing permissions and
12    * limitations under the License.
13    */
14   /**
15    * Access control rules for mynetwork
16    */
17   rule R4b_RegTransView {
18       description: "Grant Regulator full access to Trade Transactions"
19       participant: "org.example.trading.Regulator"
20       operation: ALL
21       resource: "org.example.trading.Trade"
22       action: ALLOW
23   }
24
25   rule R4a_RegulatorSeeThemselves {
26     description: "Regulators can see and update their own record"
27     participant: "org.example.trading.Regulator"
28     operation: READ, UPDATE
29     resource: "org.example.trading.Regulator"
30     action: ALLOW
```

Figure 213 - Adding a new rule to permissions.acl

In the bottom-left corner of the window, click on the "Deploy changes" button.

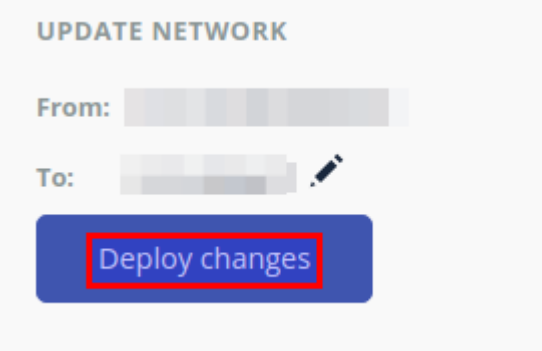


Figure 214 - Re-deploy the solution for changes to take effect.

This rule enables a Regulator to access the Trade transaction resources, such that it can view the Trade transactions from Historian's 'view record'). This rule also applies to any subsequent identity mapped to the regulator role and in the Regulator participant registry. Verify the rule works by switching to the "101" Regulator Identity and clicking on the "view record" option in the Transaction Log for the "TRADE" Transaction. You should now be able to see the Transaction detail.

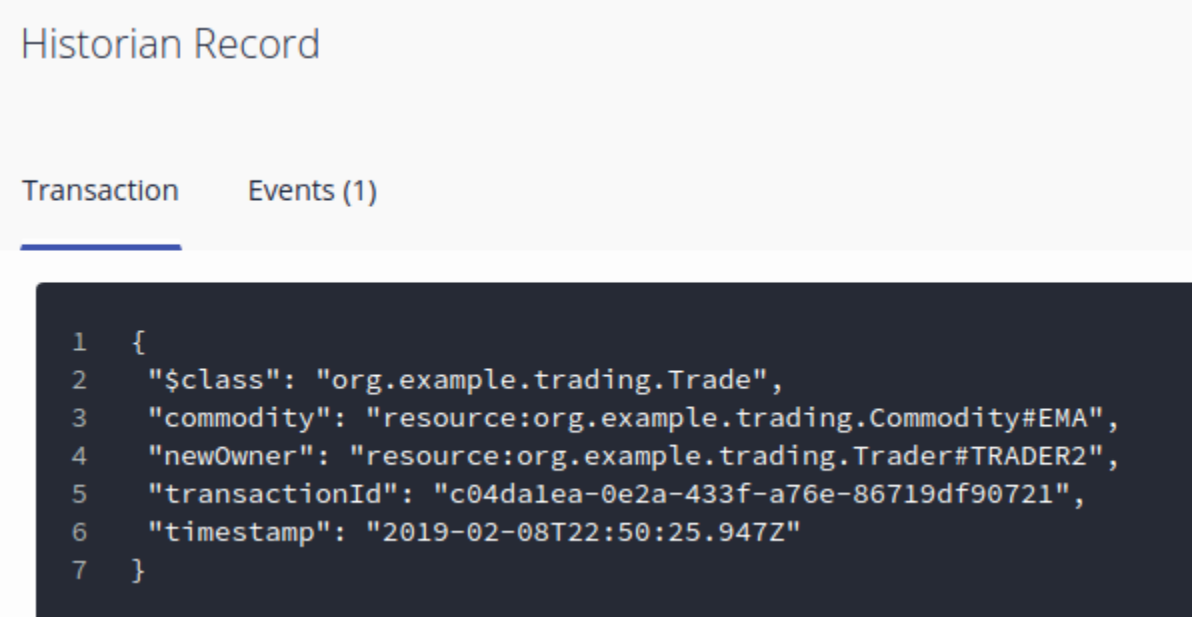


Figure 215 - Transaction detail data will now be visible to the Regulator.

Bonus Step – Create a SetupDemo Transaction

In this lab you had to manually create 6 Traders, 6 Commodities, and 1 Regulator. Can you create a SetupDemo Transaction (as seen in the perishable-network solution template) to handle this? NOTE: The code for the SetupDemo Transaction from the perishable-network solution is listed below for reference.

```
/**
 * Initialize some test assets and participants useful for running a
demo.
 * @param {org.acme.shipping.perishable.SetupDemo} setupDemo - the
SetupDemo transaction
 * @transaction
 */
async function setupDemo(setupDemo) {  // eslint-disable-line no-
unused-vars

    const factory = getFactory();
    const NS = 'org.acme.shipping.perishable';

    // create the grower
    const grower = factory.newResource(NS, 'Grower',
'farmer@email.com');
    const growerAddress = factory.newConcept(NS, 'Address');
    growerAddress.country = 'USA';
    grower.address = growerAddress;
    grower.accountBalance = 0;

    // create the importer
    const importer = factory.newResource(NS, 'Importer',
'supermarket@email.com');
    const importerAddress = factory.newConcept(NS, 'Address');
    importerAddress.country = 'UK';
    importer.address = importerAddress;
    importer.accountBalance = 0;
```

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


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

```
    shipment.unitCount = 5000;

    shipment.contract = factory.newRelationship(NS, 'Contract',
'CON_001');


    // add the growers

    const growerRegistry = await getParticipantRegistry(NS +
'.Grower');

    await growerRegistry.addAll([grower]);


    // add the importers

    const importerRegistry = await getParticipantRegistry(NS +
'.Importer');

    await importerRegistry.addAll([importer]);


    // add the shippers

    const shipperRegistry = await getParticipantRegistry(NS +
'.Shipper');

    await shipperRegistry.addAll([shipper]);


    // add the contracts

    const contractRegistry = await getAssetRegistry(NS + '.Contract');

    await contractRegistry.addAll([contract]);


    // add the shipments

    const shipmentRegistry = await getAssetRegistry(NS + '.Shipment');

    await shipmentRegistry.addAll([shipment]);

}
```

Section 7 - Interacting with other Composer Solutions

Up to this point you've been working with single solutions in Hyperledger Composer. In this lab you'll explore how one solution can invoke a transaction in a second solution. For this example both solutions will be operating on a single channel, but the same concepts apply when interacting with solutions across channels as well.

Lab 1 - Connecting two Composer Solutions

In this lab you will create and deploy two Composer solutions. You will then invoke a transaction in network B from a transaction in network A.

Step 1 - Creating and Deploying the two solutions

Create and deploy two business network solutions using the steps outlined below.

Start a clean instance of Hyperledger Fabric by opening a terminal window, and entering the commands below.

```
cd ~/fabric-dev-servers
```

```
./fabricRestart.sh
```

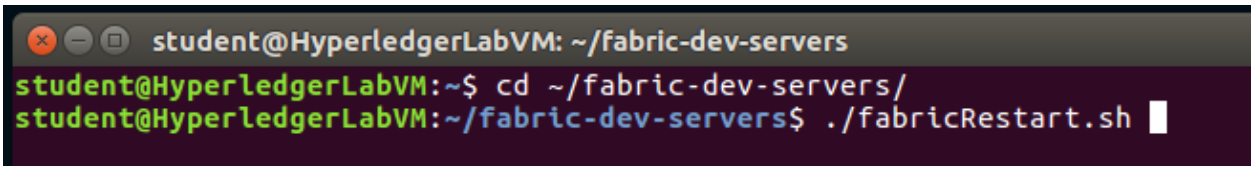


Figure 216 - Starting a clean Fabric instance

Use the commands below to delete any business network cards from your wallet. You can safely ignore any errors stating that cards cannot be found.

```
composer card delete -c PeerAdmin@hlfv1
```

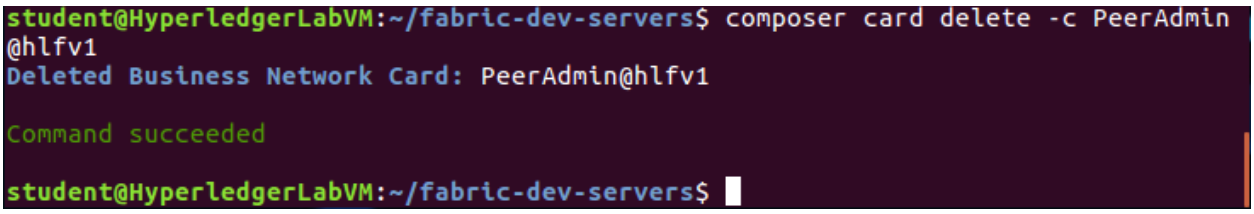


Figure 217 - Deleting the peer admin card

Then create the Peer Admin Card using the commands below.

```
./createPeerAdminCard.sh
```

```
cd ..
```

> ***NOTE:*** *If these commands fail, then you have business network cards from a previous version and you will have to delete the file system card store using the command below.*
>
> ```
> rm -fr ~/.composer
> ```

In this lab guide, refer back to "Section 5 - Building a Solution directly from your development environment". Performs steps 1 through 4 of the lab "Lab 1 - Create a Business Network solution using Yeoman". Perform the same steps again (see above). Use "other-tutorial-network" for the Business Network Name. Once complete you should see project folders for both "tutorial-network" and "other-tutorial-network" under your "Home" folder.

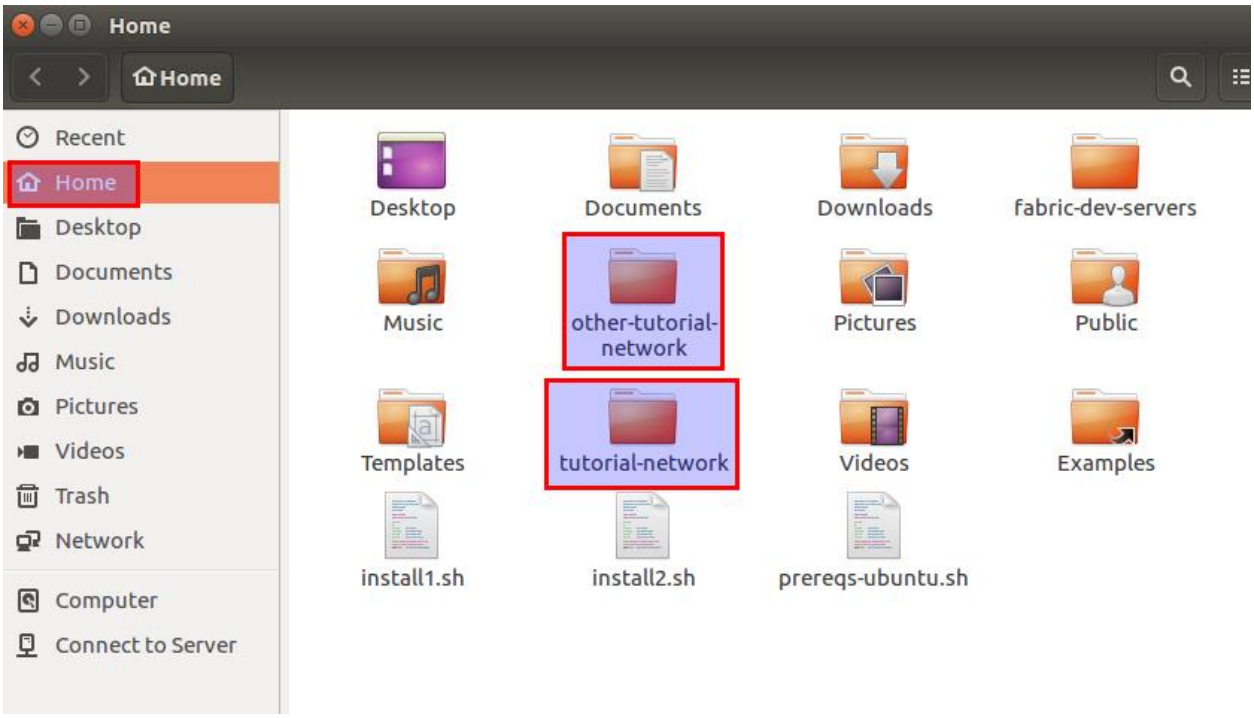


Figure 218 - The two deployed solutions

The transaction logic needs to be updated in the tutorial-network and to query an asset in the other-tutorial-network solution and then update the quantity property of an asset in business network A. Replace the contents of the logic.js script file to update the transaction processor function to be the following.

```
/**
* Track the trade of a commodity from one trader to another
* @param {org.example.mynetwork.Trade} trade - the trade to be processed
* @transaction
*/
async function tradeCommodity(trade) {
   trade.commodity.owner = trade.newOwner;

   const otherNetworkData = await getNativeAPI().invokeChaincode('other-tutorial-network', ['getResourceInRegistry', 'Asset', 'org.example.mynetwork.Commodity', trade.commodity.tradingSymbol], 'composerchannel');
   const stringAsset = new Buffer(otherNetworkData.payload.toArrayBuffer()).toString('utf8');
   const asset = getSerializer().fromJSON(JSON.parse(stringAsset));

   trade.commodity.quantity = asset.quantity;

   const assetRegistry = await getAssetRegistry('org.example.mynetwork.Commodity');
   await assetRegistry.update(trade.commodity);
}
```

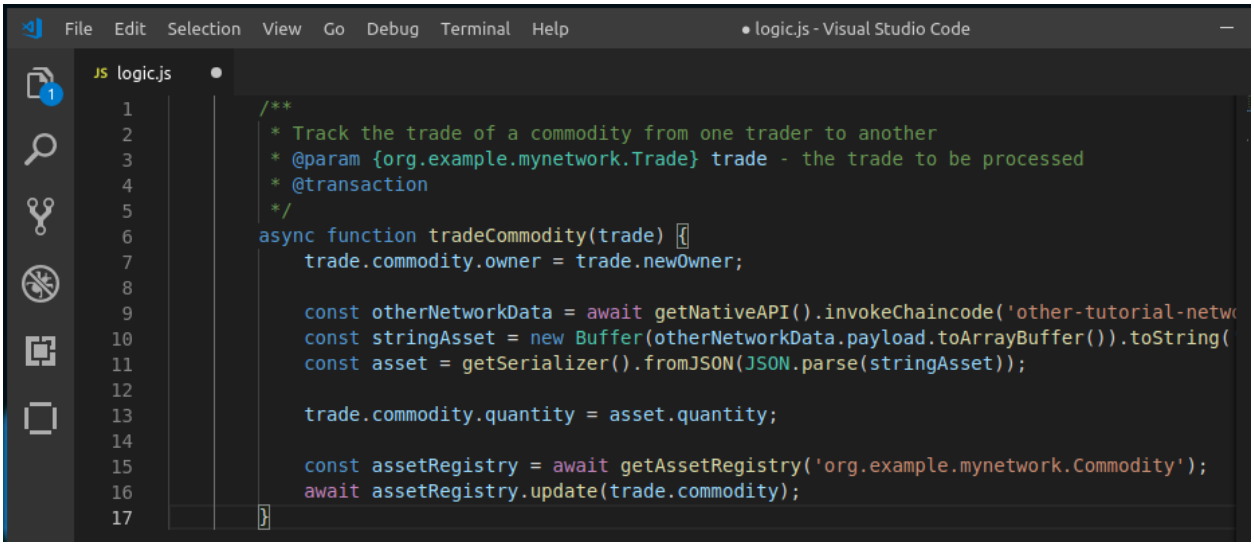


Figure 219 - Editing the code outlined above

Step 2 - Interacting across business network solutions

This step references Section 5, Lab 1. In this lab the tutorial-network solution will be network A and the other-tutorial-network solution will be network B.

Follow "Step 5 - Generate a .bna file from your solution" in Section 5, Lab 1 of the Lab Guide for both the "tutorial-network" and "other-tutorial-network" solutions.

```
student@HyperledgerLabVM:~/tutorial-network$ composer archive create -t dir -n .
Creating Business Network Archive


Looking for package.json of Business Network Definition
        Input directory: /home/student/tutorial-network

Found:
        Description: Description
        Name: tutorial-network
        Identifier: tutorial-network@0.0.1

Written Business Network Definition Archive file to
        Output file: tutorial-network@0.0.1.bna

Command succeeded

student@HyperledgerLabVM:~/tutorial-network$
```

Figure 220 - Creating the tutorial-network BNA file

```
student@HyperledgerLabVM:~/other-tutorial-network$ composer archive create -t di
r -n .
Creating Business Network Archive


Looking for package.json of Business Network Definition
        Input directory: /home/student/other-tutorial-network

Found:
        Description: Description
        Name: other-tutorial-network
        Identifier: other-tutorial-network@0.0.1

Written Business Network Definition Archive file to
        Output file: other-tutorial-network@0.0.1.bna

Command succeeded

student@HyperledgerLabVM:~/other-tutorial-network$
```

Figure 221 - Creating the other-tutorial-network BNA file

Install and start the "tutorial-network" using the commands below.

```
cd ~/tutorial-network/
```

```
composer network install --card PeerAdmin@hlfv1 --archiveFile
tutorial-network@0.0.1.bna
```

```
composer network start --networkName tutorial-network --networkVersion
0.0.1 --networkAdmin admin --networkAdminEnrollSecret adminpw --card
PeerAdmin@hlfv1 --file networkA.card
```

```
composer card import --file networkA.card --card networkA
```

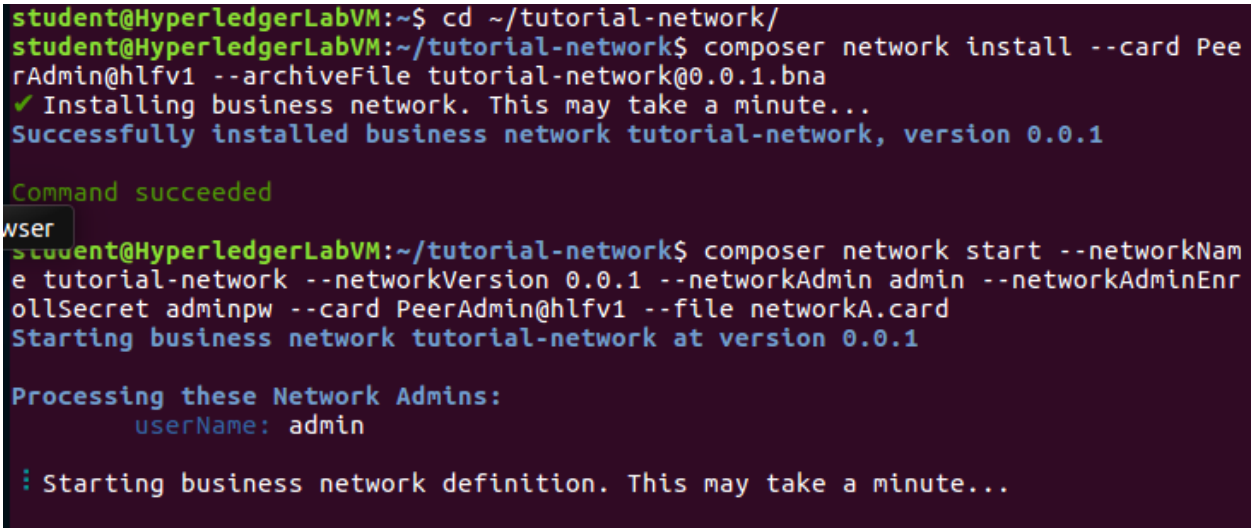


Figure 222 - Installing and starting the tutorial-network solution

Install and start the "other-tutorial-network" using the commands below.

```
cd ~/other-tutorial-network/
```

```
composer network install --card PeerAdmin@hlfv1 --archiveFile other-tutorial-network@0.0.1.bna
```

```
composer network start --networkName other-tutorial-network --networkVersion 0.0.1 --networkAdmin admin --networkAdminEnrollSecret adminpw --card PeerAdmin@hlfv1 --file networkB.card
```

```
composer card import --file networkB.card --card networkB
```

```
student@HyperledgerLabVM:~/tutorial-network$ composer network install --card PeerAdmin@hlfv1 --archiveFile tutorial-network@0.0.1.bna
✔ Installing business network. This may take a minute...
Successfully installed business network tutorial-network, version 0.0.1

Command succeeded

student@HyperledgerLabVM:~/tutorial-network$ composer network start --networkName tutorial-network --networkVersion 0.0.1 --networkAdmin admin --networkAdminEnrollSecret adminpw --card PeerAdmin@hlfv1 --file networkA.card
Starting business network tutorial-network at version 0.0.1

Processing these Network Admins:
        userName: admin

✔ Starting business network definition. This may take a minute...
Successfully created business network card:
        Filename: networkA.card

Command succeeded

student@HyperledgerLabVM:~/tutorial-network$
```

Figure 223 - Installing and starting the other-tutorial-network solution

Verify both networks are up and running by pinging them both using the commands below.

```
composer network ping --card networkA
```

```
composer network ping --card networkB
```

```
student@HyperledgerLabVM:~/other-tutorial-network$ composer network ping --card
networkA
The connection to the network was successfully tested: tutorial-network
        Business network version: 0.0.1
        Composer runtime version: 0.20.7
        participant: org.hyperledger.composer.system.NetworkAdmin#admin
        identity: org.hyperledger.composer.system.Identity#5192bfc73ccd81f381bc5
1a43755e2abd2ddb9105a8a287670c550585e60c5f1

Command succeeded

student@HyperledgerLabVM:~/other-tutorial-network$ composer network ping --card
networkB
The connection to the network was successfully tested: other-tutorial-network
        Business network version: 0.0.1
        Composer runtime version: 0.20.7
        participant: org.hyperledger.composer.system.NetworkAdmin#admin
        identity: org.hyperledger.composer.system.Identity#38996388035cf8d61fe85
752abda4811f66f04a64d5694b920aa408abc3270ff

Command succeeded
```

Figure 224 - Pinging both solutions

Create a participant in the "tutorial-network" solution by running the command below.

```
composer participant add --card networkA -d '{"$class":
"org.example.mynetwork.Trader", "tradeId": "bob@example.com",
"firstName": "Bob", "lastName": "Jones"}'
```

```
student@HyperledgerLabVM:~$ composer participant add --card networkA -d '{"$clas
s": "org.example.mynetwork.Trader", "tradeId": "bob@example.com", "firstName": "
Bob", "lastName": "Jones"}'
Participant was added to participant registry.

Command succeeded
```

Figure 225 - Creating a Participant

Create an Asset in the "tutorial-network" solution by running the command below.

```
composer transaction submit --card networkA -d '{"$class":
"org.hyperledger.composer.system.AddAsset", "targetRegistry" :
"resource:org.hyperledger.composer.system.AssetRegistry#org.example.my
network.Commodity", "resources": [{"$class":
"org.example.mynetwork.Commodity","tradingSymbol": "Ag","owner":
"resource:org.example.mynetwork.Trader#bob@example.com","description":
"a lot of gold", "mainExchange": "exchange", "quantity" : 250}]}'
```

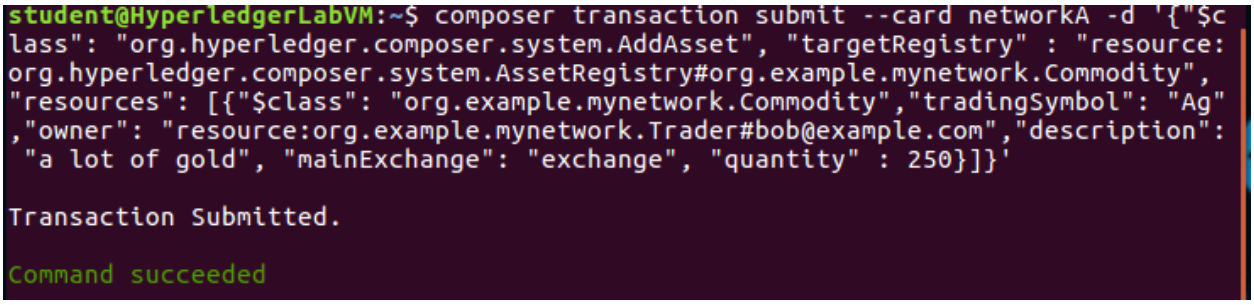


Figure 226 - Creating an Asset

Create a Participant in the "other-tutorial-network" solution using the command below.

```
composer participant add --card networkB -d '{"$class":
"org.example.mynetwork.Trader", "tradeId": "fred@example.com",
"firstName": "Fred", "lastName": "Bloggs"}'
```

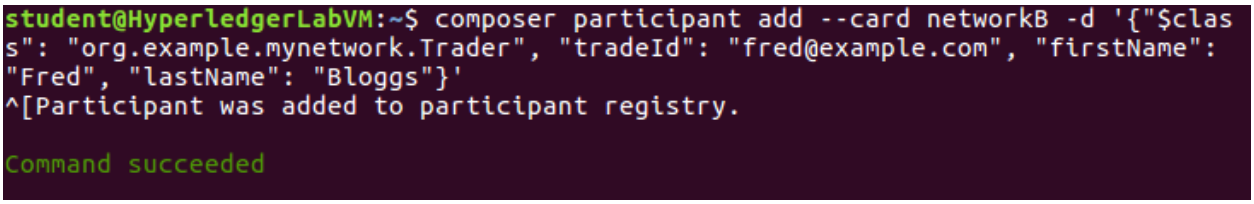


Figure 227 - Creating a Participant

Create an Asset in the "other-tutorial-network" solution by running the command below.

```
composer transaction submit --card networkB -d '{"$class":
"org.hyperledger.composer.system.AddAsset", "targetRegistry" :
"resource:org.hyperledger.composer.system.AssetRegistry#org.example.my
network.Commodity", "resources": [{"$class":
"org.example.mynetwork.Commodity","tradingSymbol": "Ag","owner":
"resource:org.example.mynetwork.Trader#fred@example.com","description"
: "a lot of gold", "mainExchange": "exchange", "quantity" : 500}]}'
```

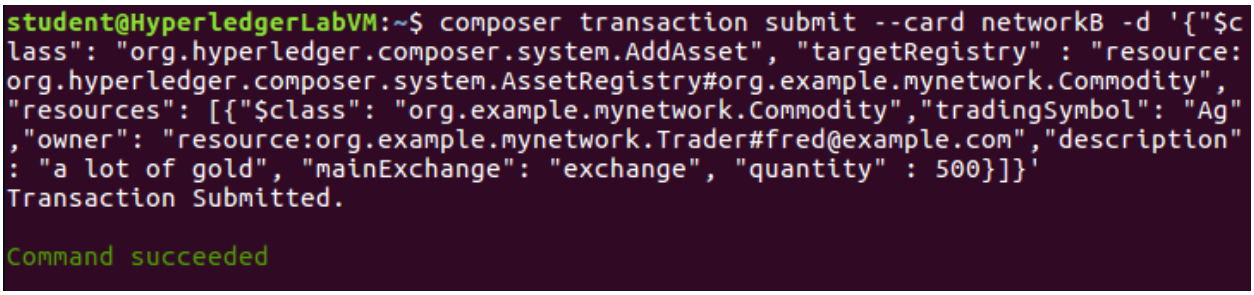


Figure 228 - Creating an Asset

Step 3 - Bind the identity on tutorial-network to the participant on other-tutorial-network

Run the commands below to export the card for the "tutorial-network" solution.

```
cd ..
```

```
composer card export -c networkA
```

```
student@HyperledgerLabVM:~$ composer card export -c networkA

Successfully exported business network card
        Card file: networkA.card
        Card name: networkA

Command succeeded
```

Figure 229 - Exporting the network card

Unzip the card you just exported. Browse to your "Home" directory using the "Files" tool. Right-click on the file "networkA.card" and select the "Open With Archive Manager" option.

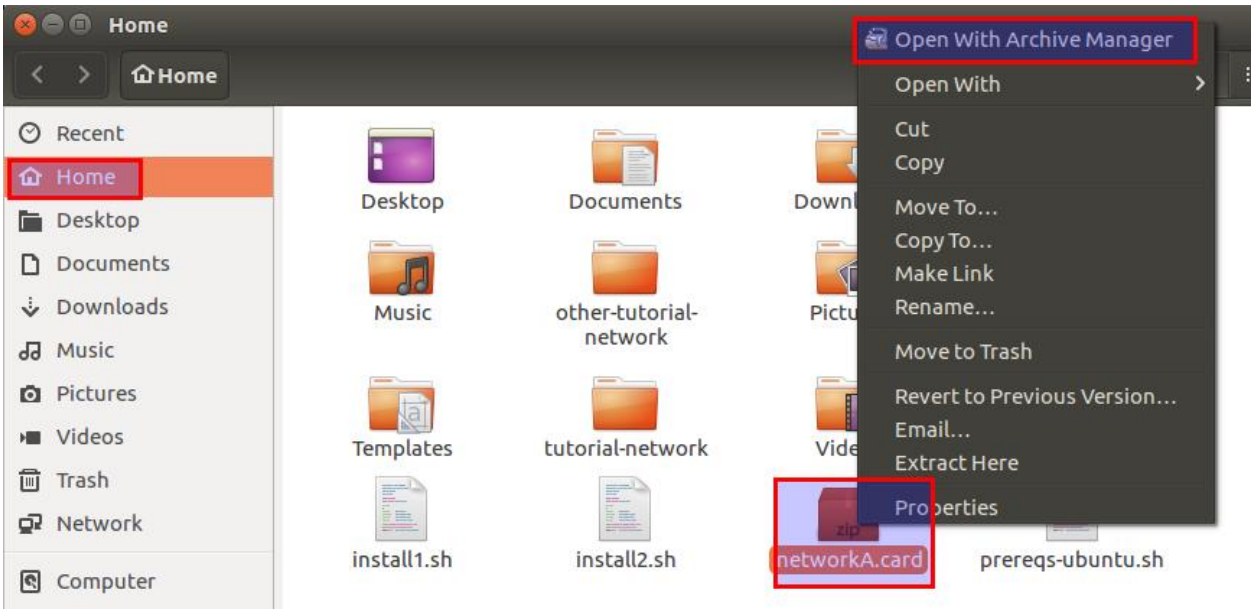


Figure 230 - Opening the network card archive

From the "Archive Manager" tool, click the "Extract" button. Make sure the "Home" directory is selected, then create a new folder called "networkA" to extract the contents to. Finally, click the "Extract" open from the Extract dialogue window.

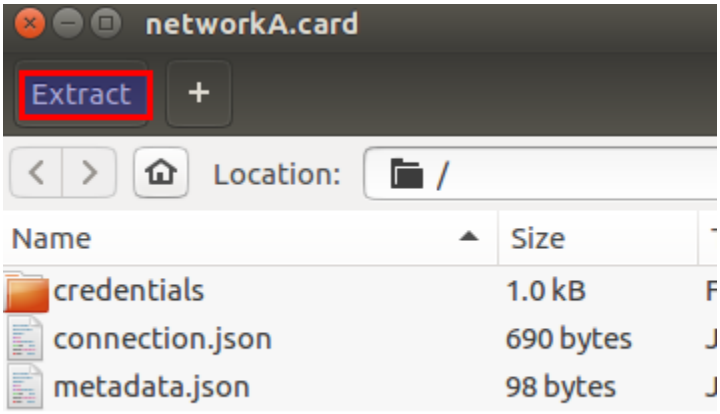


Figure 231 - Extracting the network card archive

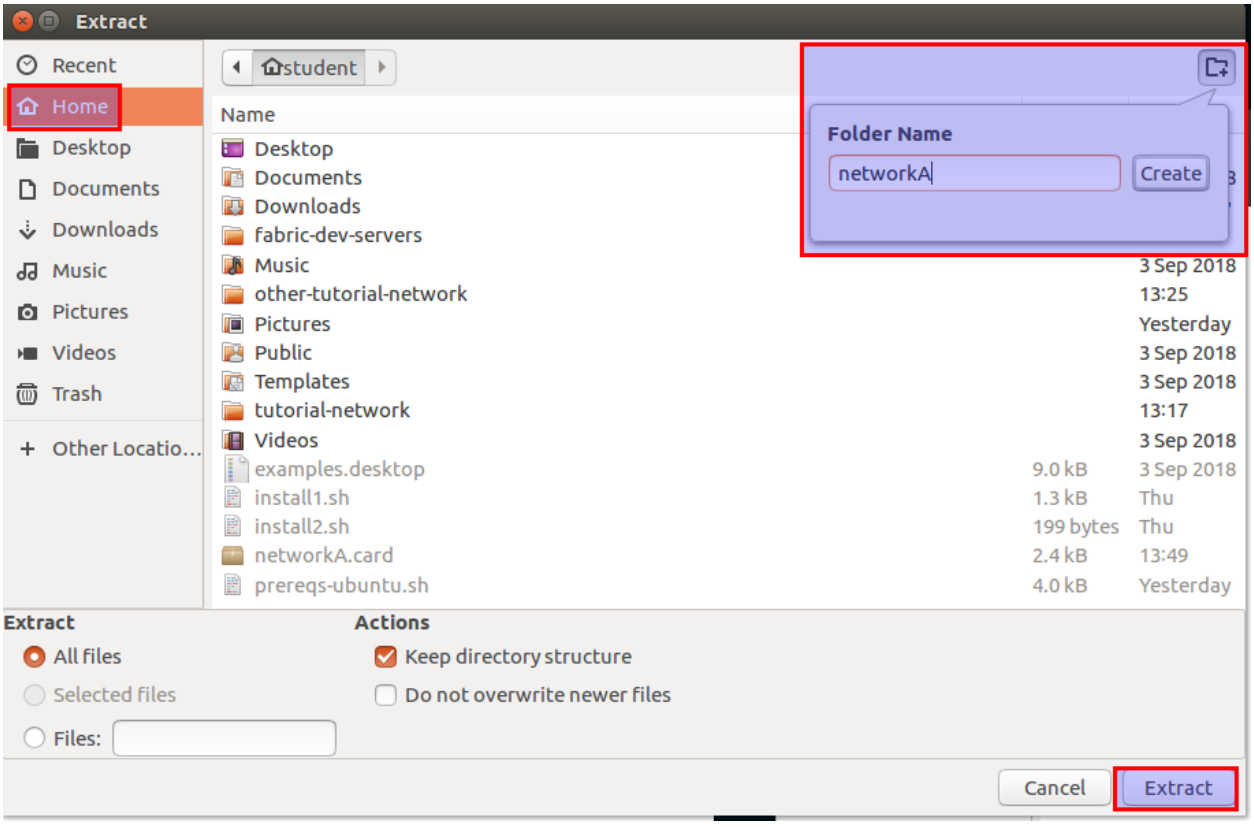


Figure 232 - Extract the archive to a folder named networkA under the Home directory

Bind the identity to the participant. Run the following command.

```
composer identity bind --card networkB --participantId
resource:org.hyperledger.composer.system.NetworkAdmin#admin --
certificateFile ./networkA/credentials/certificate
```

```
student@HyperledgerLabVM:~$ composer identity bind --card networkB --participant
Id resource:org.hyperledger.composer.system.NetworkAdmin#admin --certificateFile
 ./networkA/credentials/certificate
An identity was bound to the participant 'resource:org.hyperledger.composer.syst
em.NetworkAdmin#admin'
The participant can now connect to the business network using the identity

Command succeeded
```

Figure 233 - Binding an Identity

Create a card with the bound identity by entering the command below.

```
composer card create -p ~/.composer/cards/networkB/connection.json --
businessNetworkName other-tutorial-network -u admin -c
./networkA/credentials/certificate  -k
./networkA/credentials/privateKey -f newNetworkB.card
```

```
student@HyperledgerLabVM:~$ composer card create -p ~/.composer/cards/networkB/c
onnection.json --businessNetworkName other-tutorial-network -u admin -c ./networ
kA/credentials/certificate  -k ./networkA/credentials/privateKey -f newNetworkB.
card

Successfully created business network card file to
        Output file: newNetworkB.card

Command succeeded
```

Figure 234 - Creating a Network Card with the bound Identity

Import the card using the command below.

```
composer card import --file newNetworkB.card --card newNetworkB
```

```
student@HyperledgerLabVM:~$ composer card import --file newNetworkB.card --card
newNetworkB

Successfully imported business network card
        Card file: newNetworkB.card
        Card name: newNetworkB

Command succeeded
```

Figure 235 - Importing the newly created card

Ping the network to activate the identity using the command below.

```
composer network ping --card newNetworkB
```

```
student@HyperledgerLabVM:~$ composer network ping --card newNetworkB
The connection to the network was successfully tested: other-tutorial-network
        Business network version: 0.0.1
        Composer runtime version: 0.20.7
        participant: org.hyperledger.composer.system.NetworkAdmin#admin
        identity: org.hyperledger.composer.system.Identity#5192bfc73ccd81f381bc5
1a43755e2abd2ddb9105a8a287670c550585e60c5f1

Command succeeded
```

Figure 236 - Pinging the solution

Step 4 - Reviewing the asset data

View the asset to see that the quantity is 250 using the command below.

```
composer network list --card networkA -r
org.example.mynetwork.Commodity -a Ag
```

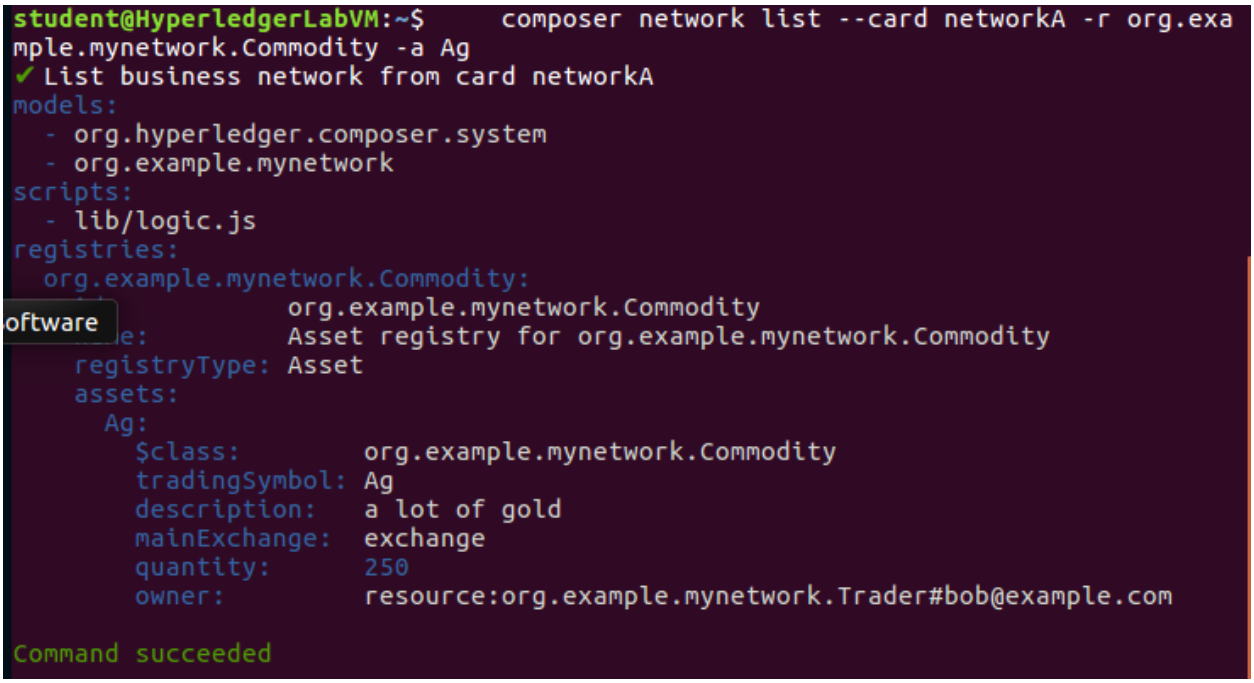


Figure 237 - Verifying Asset quantity value

Step 5 - Submiting a transaction

Submit a transaction to see the effect of querying an asset on a different business network. NOTE: NetworkB is only queried and the quantity is not changed.

```
composer transaction submit --card networkA -d '{"$class":
"org.example.mynetwork.Trade", "commodity":
"resource:org.example.mynetwork.Commodity#Ag", "newOwner":
"resource:org.example.mynetwork.Trader#bobId"}'
```

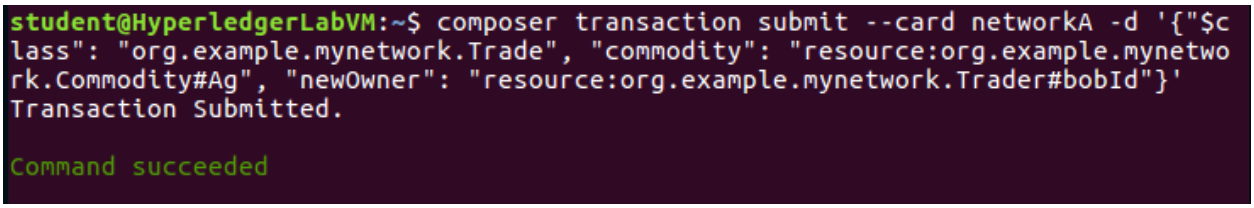


Figure 238 - Submitting a Transaction

Step 6 - Check the updated asset

View the updated asset to check that the quantity was correctly updated to 500 by using the command below.

```
composer network list --card networkA -r
org.example.mynetwork.Commodity -a Ag
```

```
student@HyperledgerLabVM:~$ composer network list --card networkA -r org.example
.mynetwork.Commodity -a Ag
✔ List business network from card networkA
models:
  - org.hyperledger.composer.system
  - org.example.mynetwork
scripts:
  - lib/logic.js
registries:
  org.example.mynetwork.Commodity:
    id:           org.example.mynetwork.Commodity
    name:         Asset registry for org.example.mynetwork.Commodity
    registryType: Asset
    assets:
      Ag:
        $class:        org.example.mynetwork.Commodity
        tradingSymbol: Ag
        description:   a lot of gold
        mainExchange:  exchange
        quantity:      500
        owner:         resource:org.example.mynetwork.Trader#bobId

Command succeeded
```

Figure 239 - Checking the updated Asset

Section 8 – Putting it All Together

In this section you will apply everything you've learned about designing and developing a Composer Fabric solution to address a real-world use case.

Lab 1 – The Capstone Project

In this lab you will identify a use case from the world around you, define some basic solution requirements, and then apply the lessons you've learned in the hands-on labs to create a working Proof-of-Concept.

Step 1 – Identify a good use case

Based on what you're learned in this class, can you identify a good use case in the world around you for a blockchain proof-of-concept? You don't need to model anything complex, in fact at this early stage the simpler your idea, the better! As you are considering use cases it may be helpful to keep in mind these common enterprise blockchain use case "symptoms".

- Large business networks / consortium
- Each participant wants to retain ownership/control of their data
- Many different audience types (each of whom will need to see or edit different parts of the data)
- Desire to standardize (and automate) business processes
- Process requires audit, need to prove you did what you said you did
- Need a permanent record of how business assets evolve over time
- Need a system of record between multiple organizations / groups

Step 2 – Identify Assets and Participants

Who are the Participants on your Business Network, and what Assets do they care about? To keep things simple, try to limit the number of Participants (for now) to around 6 and the number of Assets should be limited to 2-3.

Step 3 – Identify Transactions

Now that you've identified Assets and Participants, it's time to define your Transactions. Try listing out every unique combination of an Asset and a Participant and see if you can define a Transaction at that intersection.

Step 4 – Start building your data model

Now that you've got some high-level requirements, try building out your model (.CTO) file to match. You can start your project in Composer Playground, or work right from the file system in your IDE of choice.

Step 5 – Build your transactions

Once you've built, tested, and validated your business data model it's time to code your transactions. Create an implementation (even a basic one is okay) for each transaction defined in your model.

Step 6 – Package up your solution

Create a .BNA archive file of your solution for re-deployment.

Step 7 – Deploy your solution to your Fabric test environment

Install and start your solution from your .BNA archive file.

Step 8 – Build a REST API around your solution

Use the `composer-rest-server` command to build a REST API around your installed and running solution.

Step 9 – Build an Angular front-end for your solution

Use the Yeoman code generation tool to build an Angular front-end app for your solution. Use this front-end to review the solution with non-technical stakeholders and collect feedback.

Other Steps – Considerations

- Do you need to create any queries for your solution?
- Will your solution need to interact with other Composer / Fabric solutions?
- Will your solution need to interact with other systems?
- Do you need to define and raise any events?
- Do you need to create and test permissions with multiple identities?
- What would a development plan for this solution look like?
    - Do you have a list of features based on user feedback?
    - Have those features been prioritized by stakeholders?
    - Have you created a sprint plan and product backlog?

Appendix A – Building a Development Environment

NOTE: If you are using a VirtualBox lab image provided by Blockchain Training Alliance you DO NOT need to perform any of the labs in Appendix A. Please skip directly to Section 2!

Building out a development environment for Hyperledger Fabric / Composer solutions is relatively simple. In this section you'll build out a development environment step-by-step. These steps can be performed on any Ubuntu 16.04.5 LTS environment. The ONLY software that has been installed for you is the Ubuntu operating system and any patches or updates since the 16.04.5 LTS release. At least 4GB is memory or more is recommended for your development environment.

Lab 1 – Installing Prerequisites

In this lab you'll install all the required prerequisites for Hyperledger Fabric (v1.4) and Composer.

Step 1 – Install curl

Start by installing the curl utility. To being, open a command terminal window. This can done by right-clicking anywhere on the desktop and selecting the "Open Terminal" option from the fly-out menu. From the command line, type in the following command (you will be prompted for a password with admin privileges):

```
sudo apt install curl
```

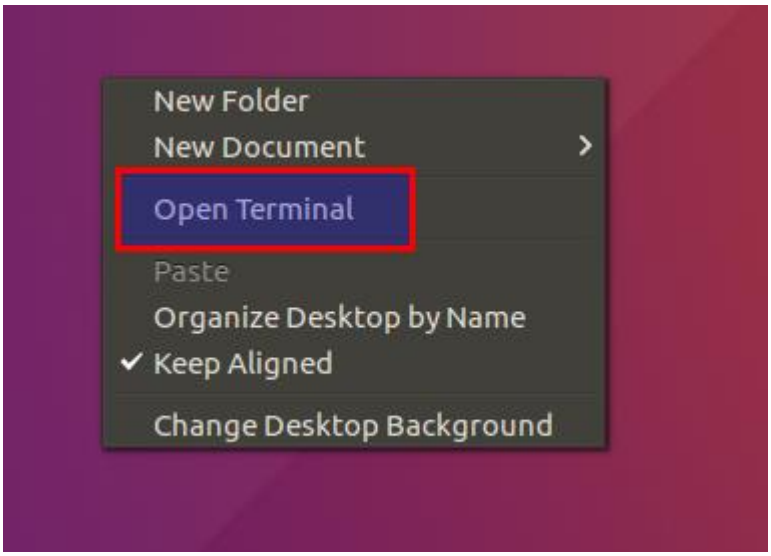


Figure 240 - Right-click on the desktop to open a terminal window.

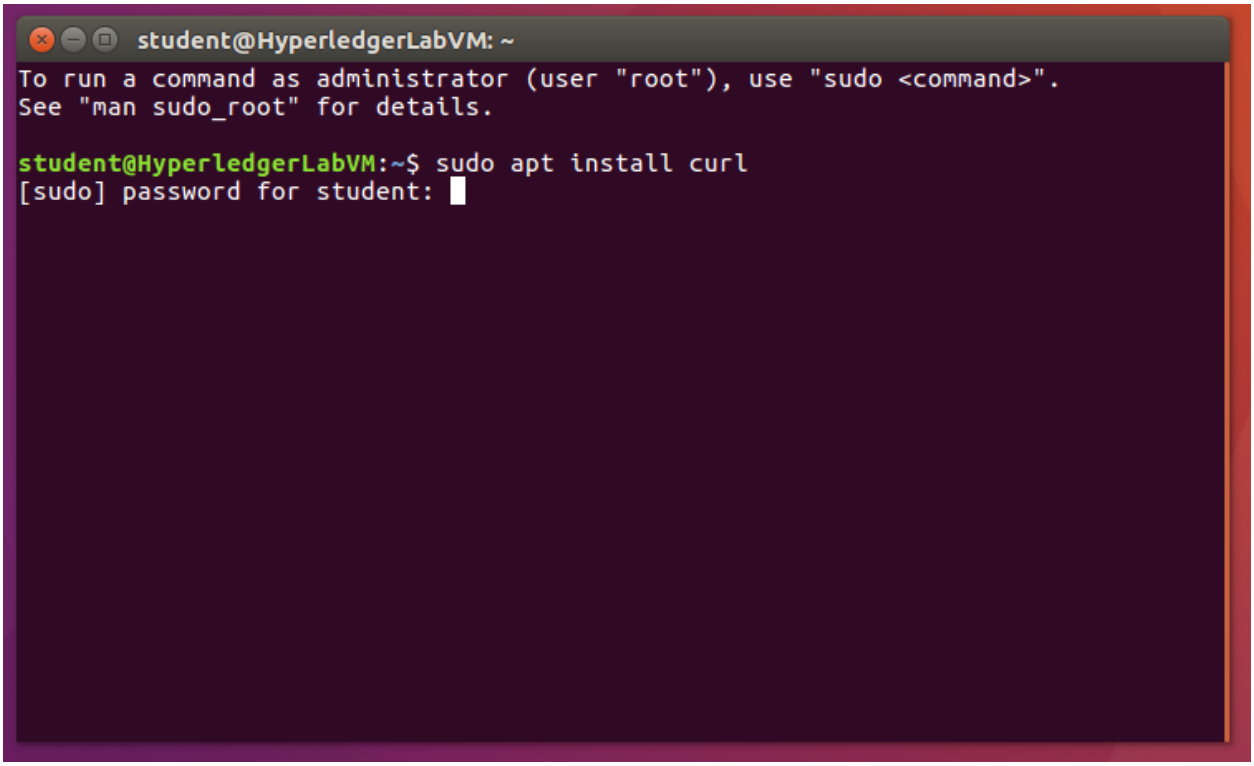


Figure 241 - Use this command to install curl

Step 2 – Download the prerequisites script.

This will download a script provided by Hyperledger which downloads and installs prerequisite components. In this step you will download the script and set execution permissions on it.

Download the prerequisites and add execute permission using the following commands:

```
curl -O https://hyperledger.github.io/composer/latest/prereqs-ubuntu.sh
```

```
chmod u+x prereqs-ubuntu.sh
```

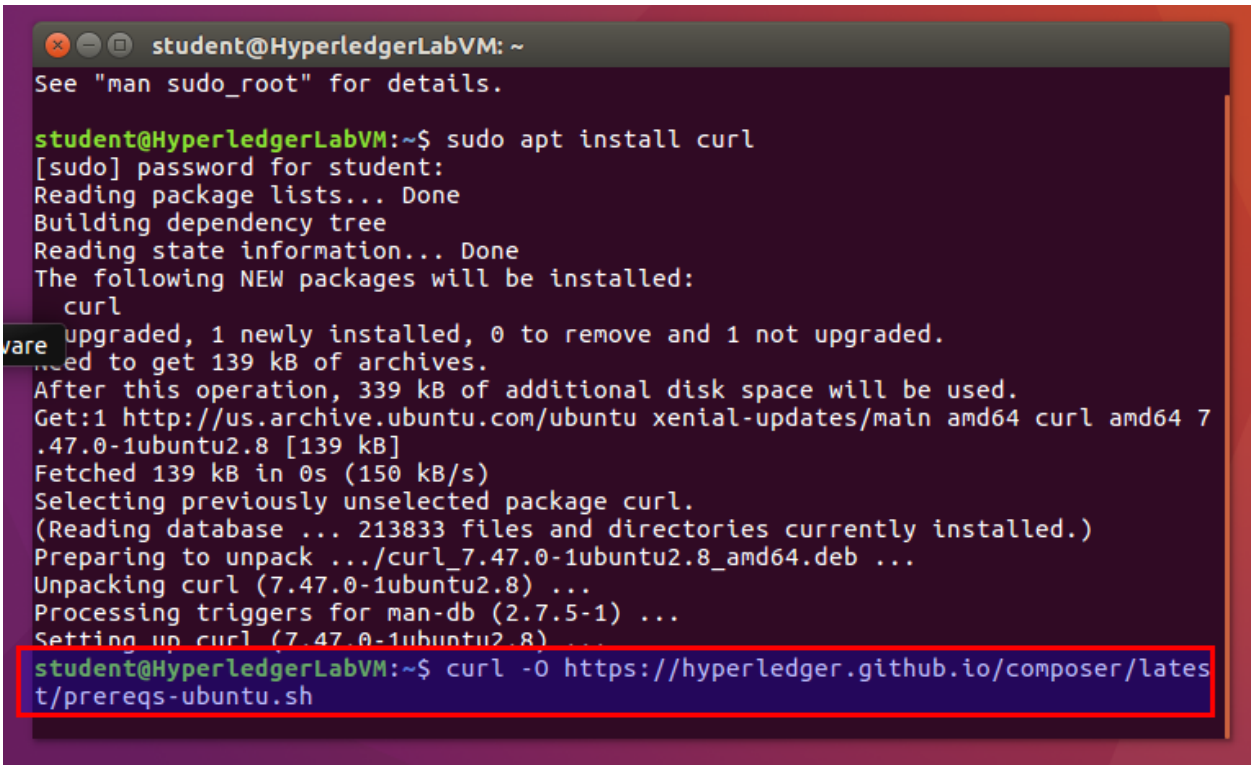


Figure 242 - Downloading the prereqs script

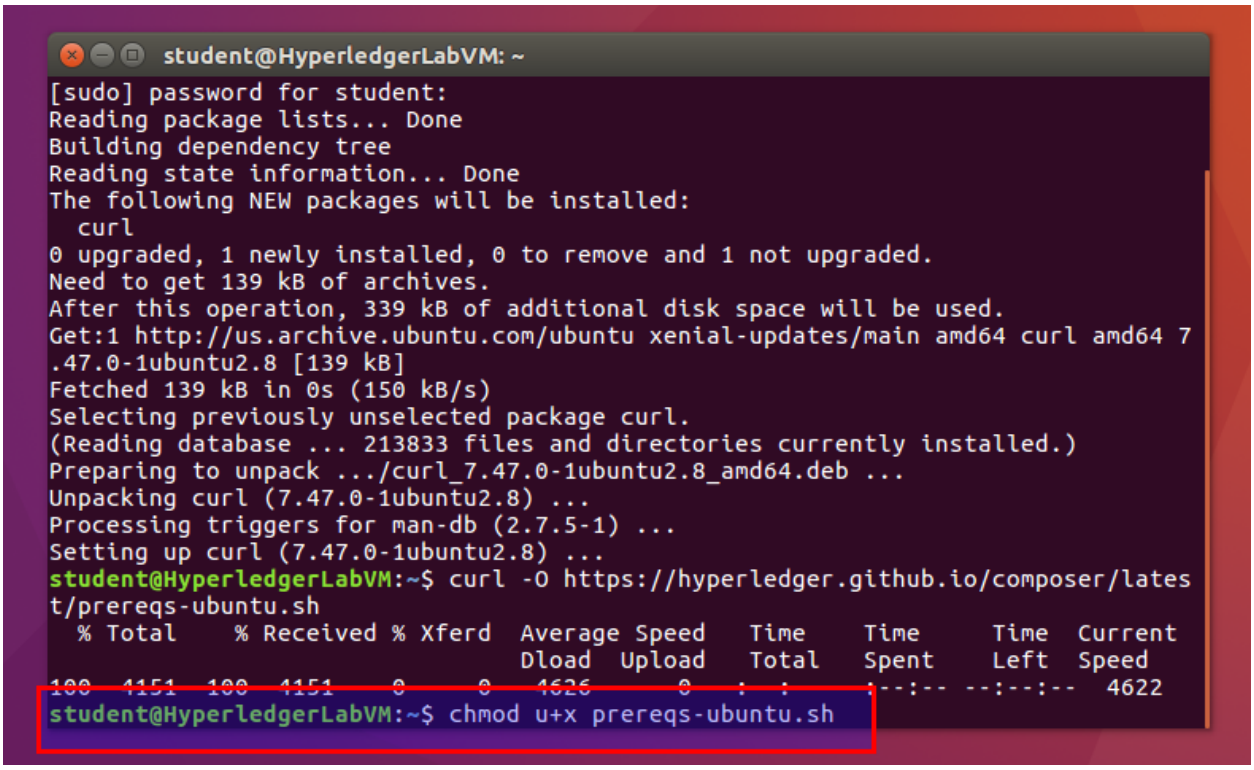


Figure 243 - Adding execute permissions to the newly downloaded script.

Step 3 – Run the prereqs installation script

Next run the script downloaded in the previous step. This script uses admin privileges during its execution – you will be asked for an admin password. To execute the script, enter the following commands:

```
export PATH=/bin:/usr/bin:/usr/local/bin:/sbin:/usr/sbin
./prereqs-ubuntu.sh
```

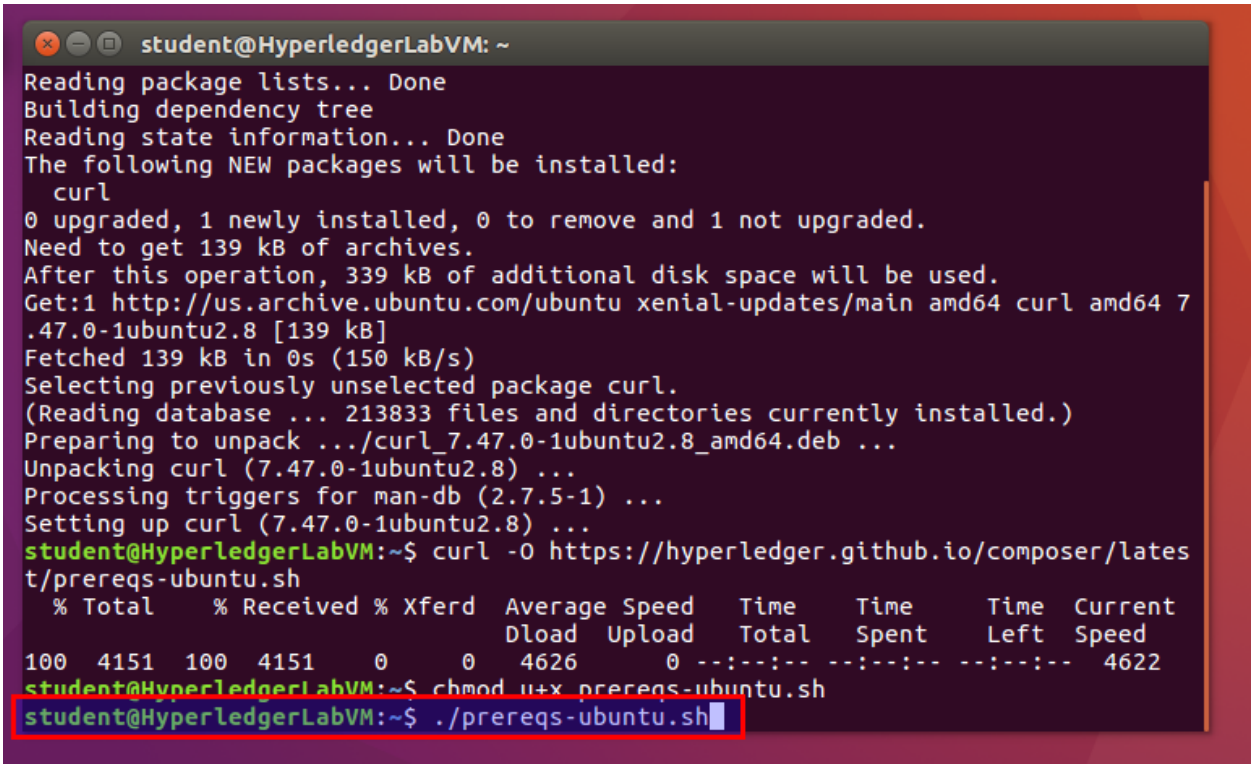


Figure 244 - Executing the prereqs script.

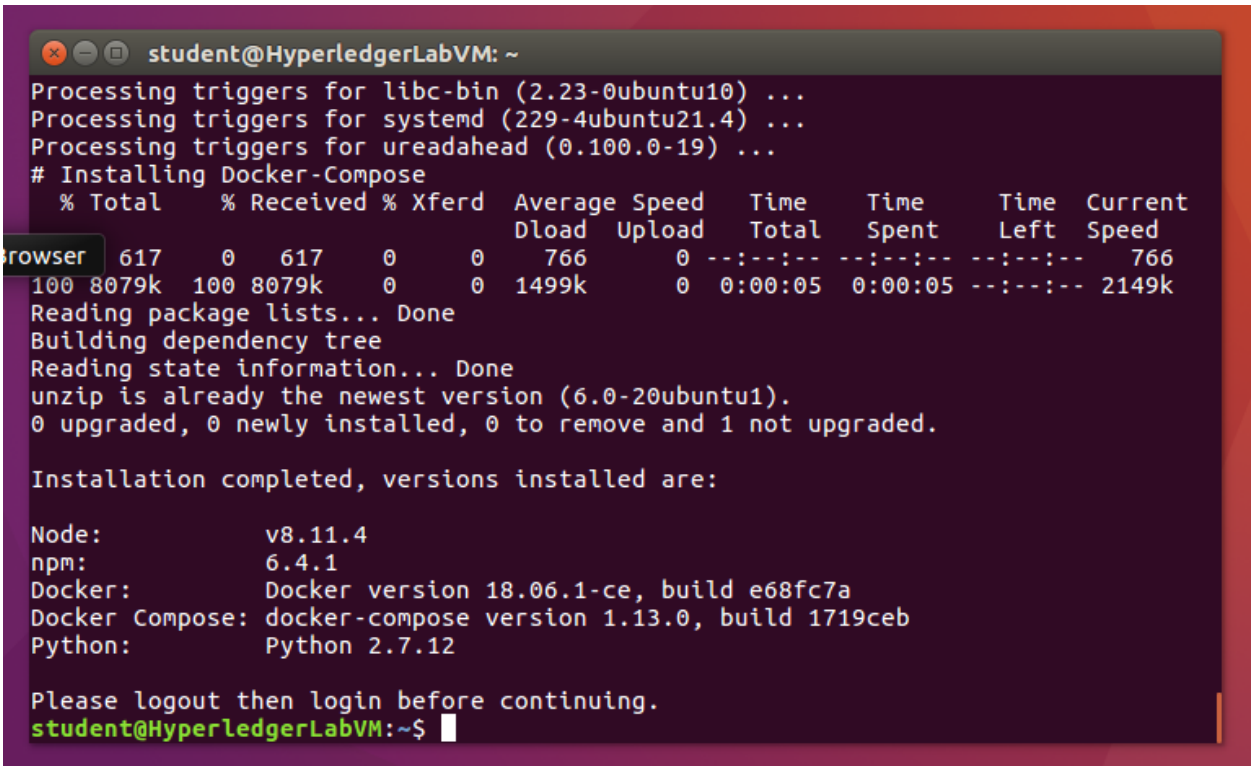


Figure 245 - After executing the script.

Step 4 – Installing the Node Package Manager (NPM)

To install NPM, run the following commands:

```
sudo apt install npm
```

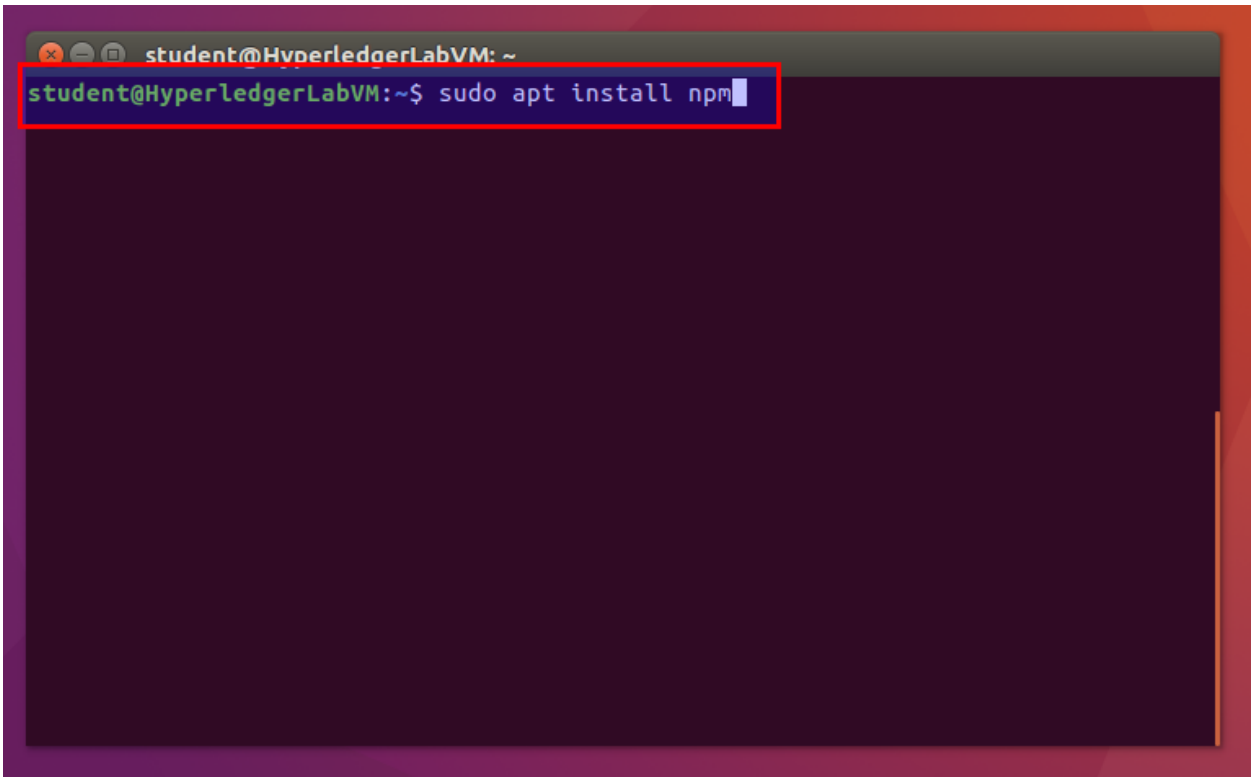


Figure 246 - Installing NPM

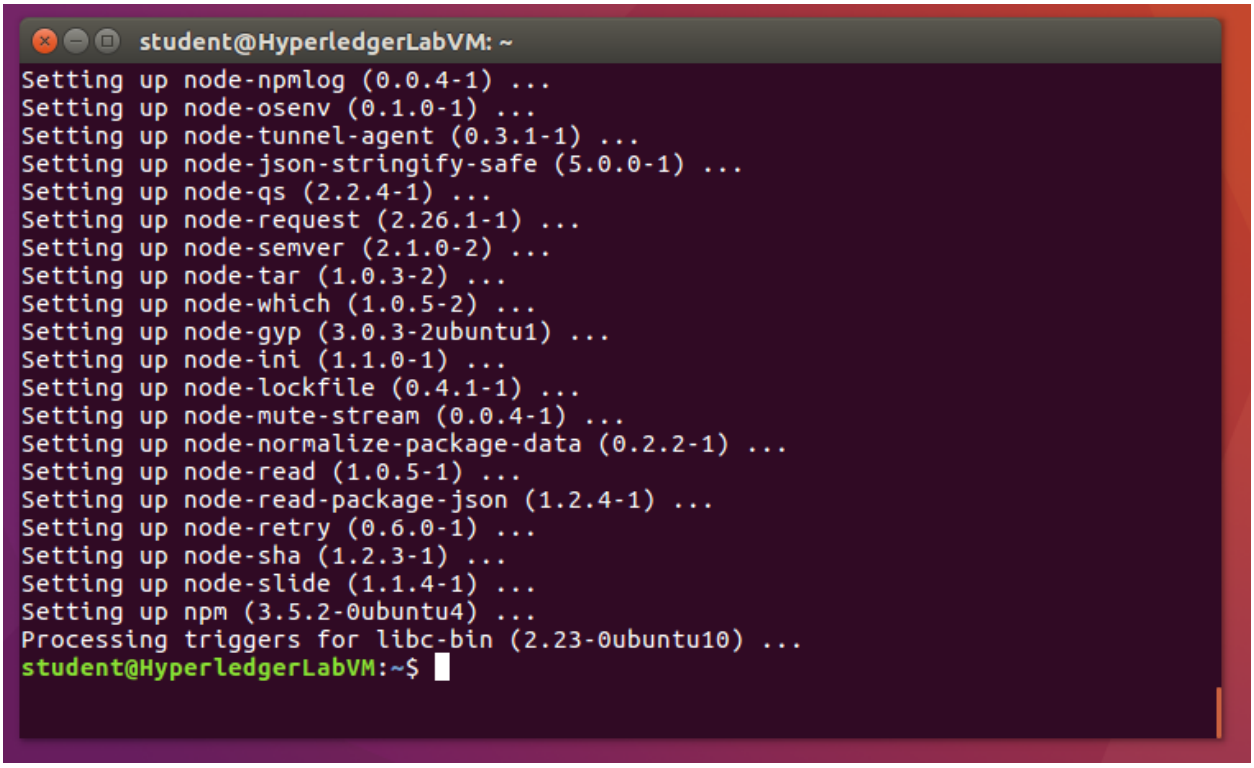


Figure 247 - After installing NPM

Step 5 - Open up permissions on /usr/local/lib folder

The permissions for the /usr/local/lib need to be opened up to allow installation without root access (installing as root can cause problems later in the labs).

To adjust folder permissions, run the following command:

```
sudo chmod +777 /usr/local/lib
```

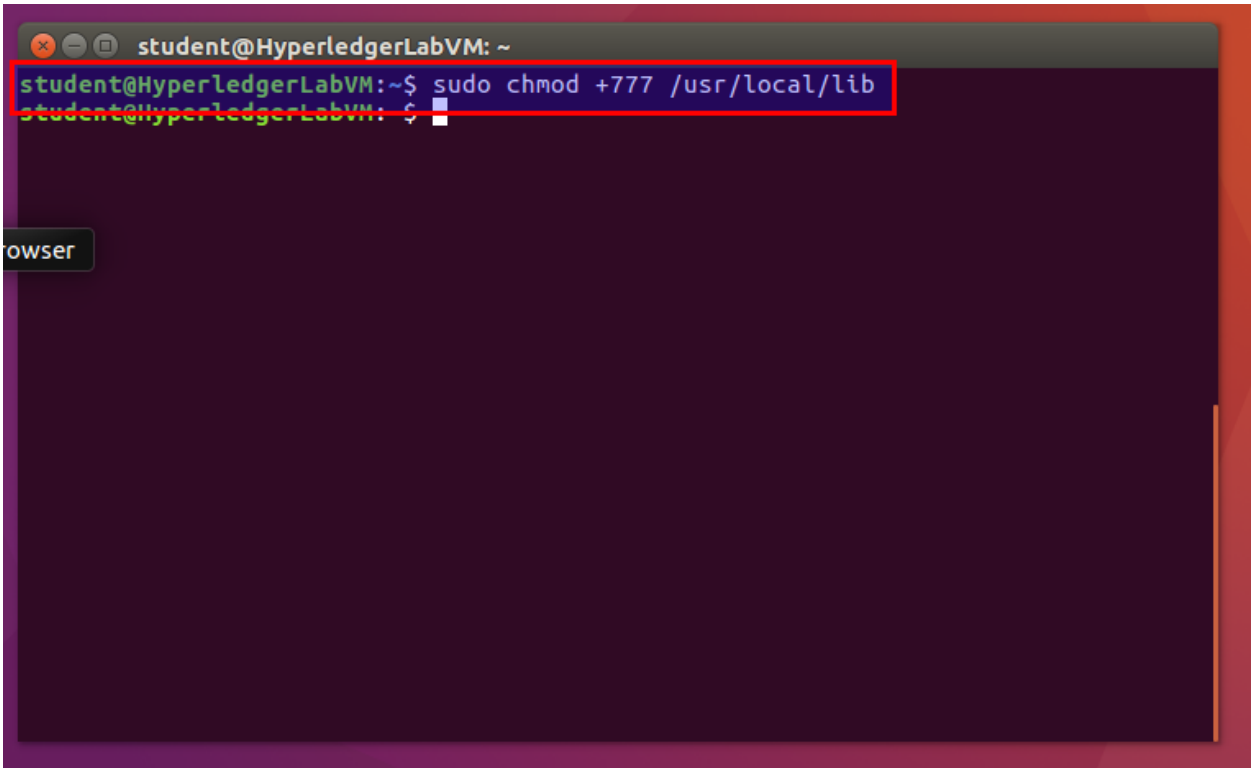


Figure 248 - Changing folder permissions

Step 6 - Open up permissions on /usr/local folder

The permissions for the /usr/local need to be opened up to allow installation without root access (installing as root can cause problems later in the labs).

To adjust folder permissions, run the following command:

```
sudo chmod +777 /usr/local
```

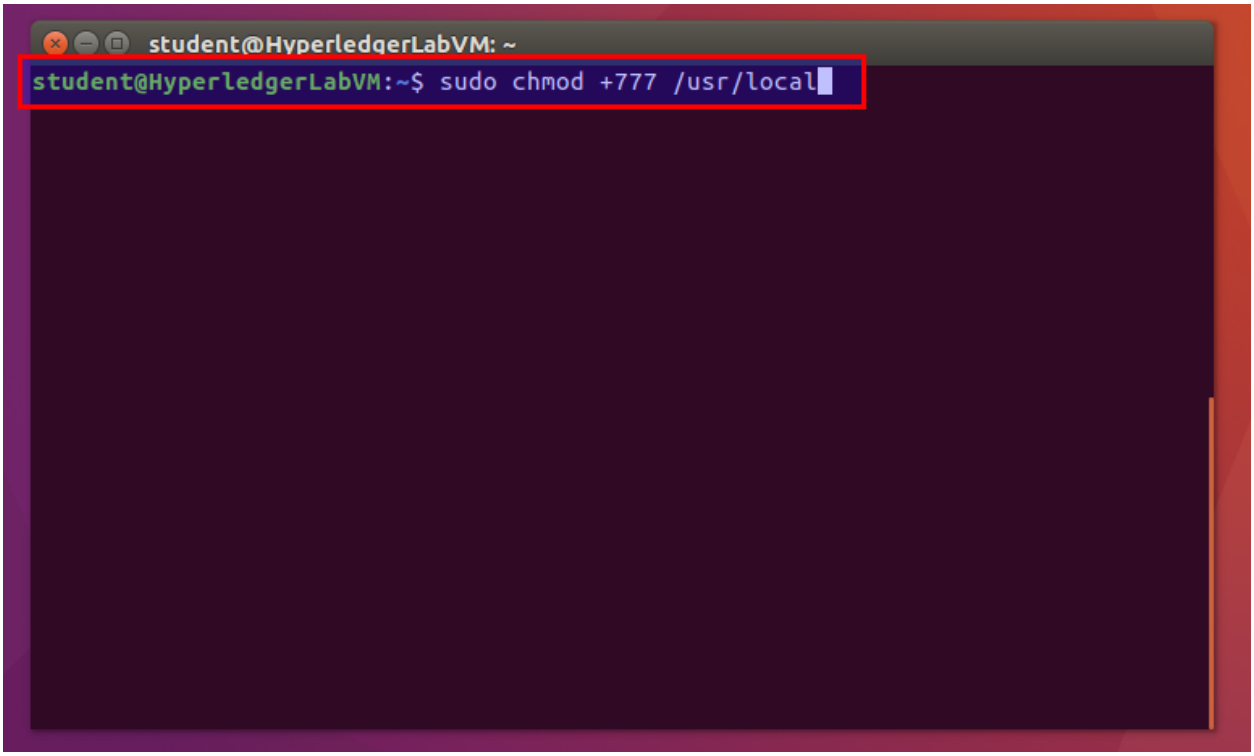


Figure 249 - Changing folder permissions

Lab 2 – Installing Hyperledger Components

In this lab you'll install the Hyperledger Fabric and Composer components into your development environment.

Step 1 – Install Node.js

To install Node.js, use the script located on the Desktop. You can run it with the following commands:

```
cd Desktop

chmod +777 nodeInstall.sh

./nodeInstall.sh

cd ..
```

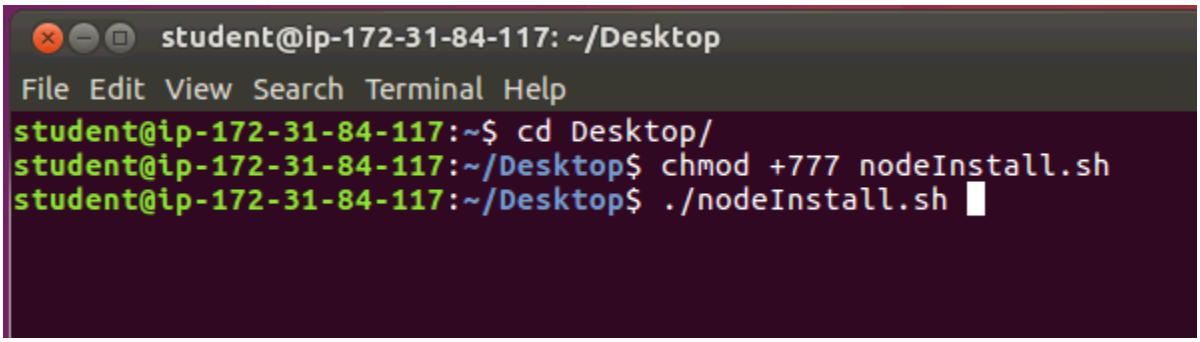


Figure 250 - Running the nodeInstall script.

Step 2 – Install the Composer command line tools

To install the Composer command line tools, use the following commands:

```
cd Desktop

./fixIt.sh

cd ..

npm install -g composer-cli@0.20
```

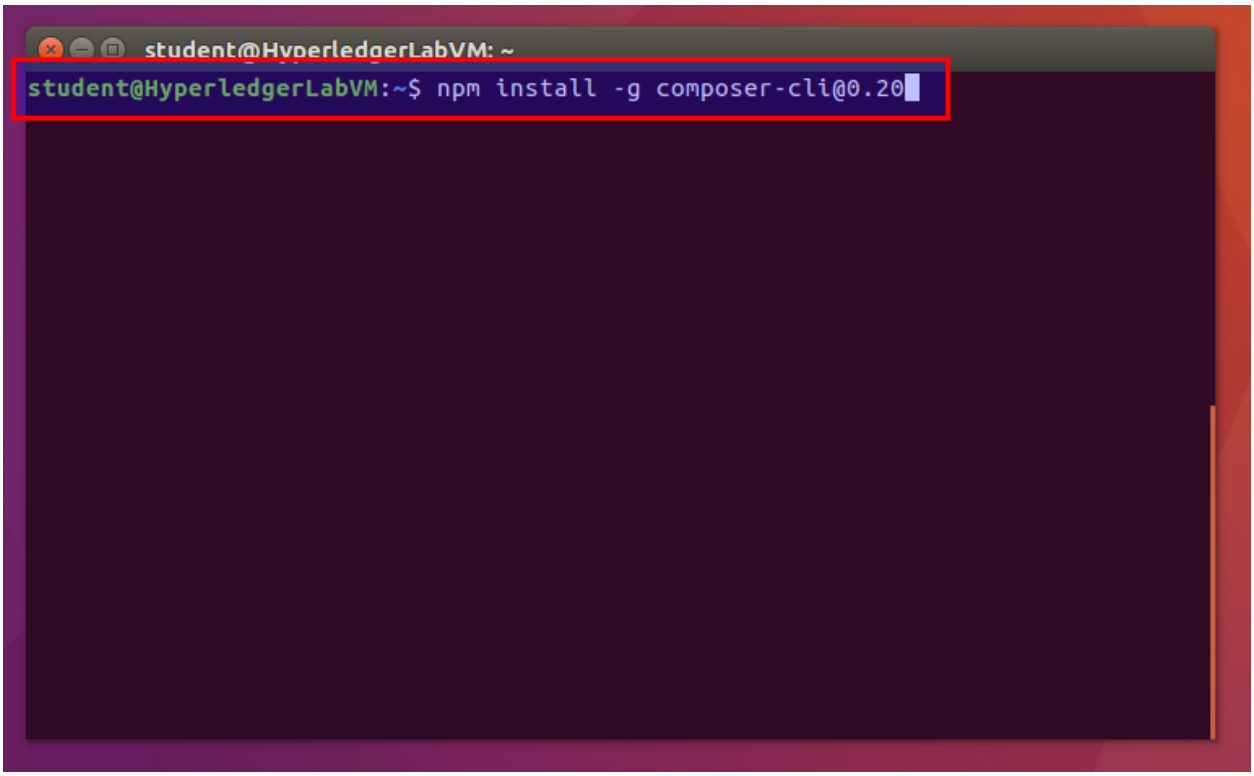


Figure 251 - Installing the CLI tools for Composer

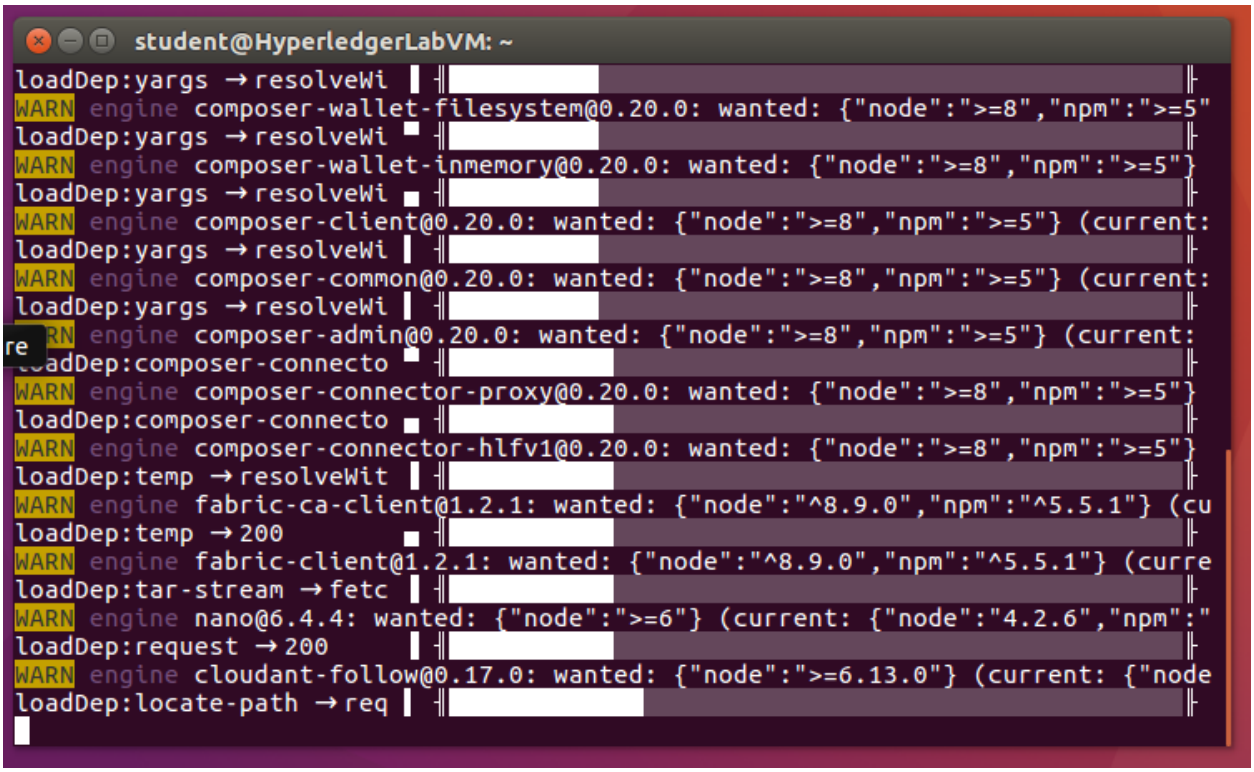


Figure 252 - Tools installing, this step may take a bit of time to complete.

Step 3 – Install the Composer Rest Server

The Composer Rest Server is used by Composer to expose your business network solutions as RESTful APIs. To install the Composer Rest Server, run the following command:

```
npm install -g composer-rest-server@0.20
```

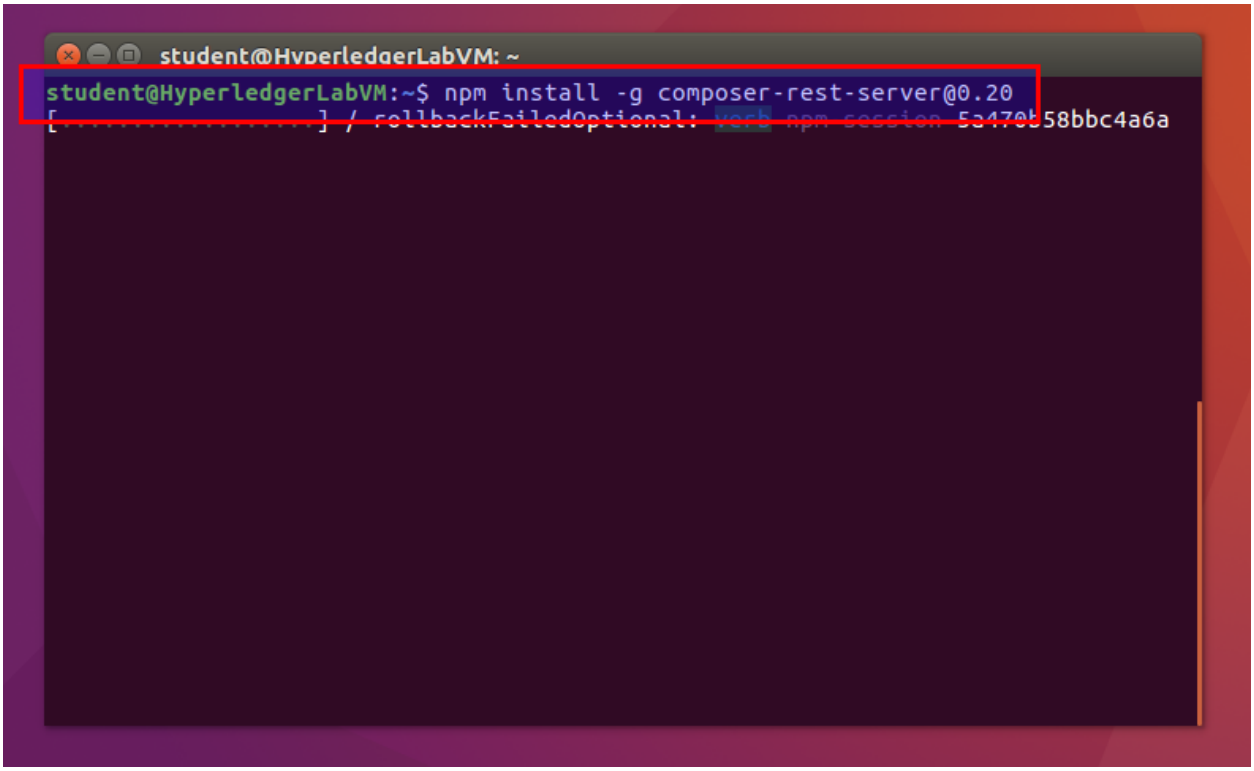


Figure 253 - Installing the Composer Rest Server

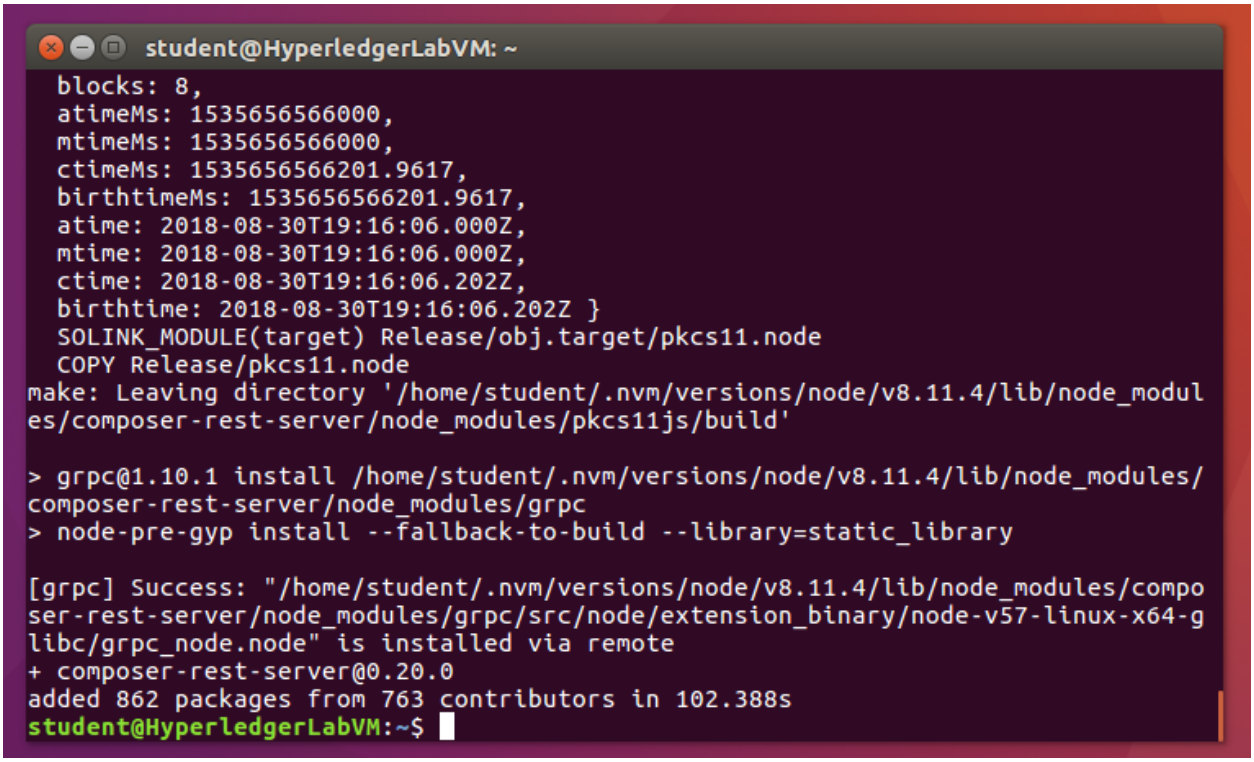


Figure 254 - Installation of the REST server is complete.

Step 4 – Install the Hyperledger Composer Generator utility

The Hyperledger Composer Generator utility is a utility which helps to generate Composer code and project scaffoldings. It is used by the Yeoman code generation tool which will be installed in the next step. To install the Composer Generator utility, run the following command:

```
npm install -g generator-hyperledger-composer@0.20
```

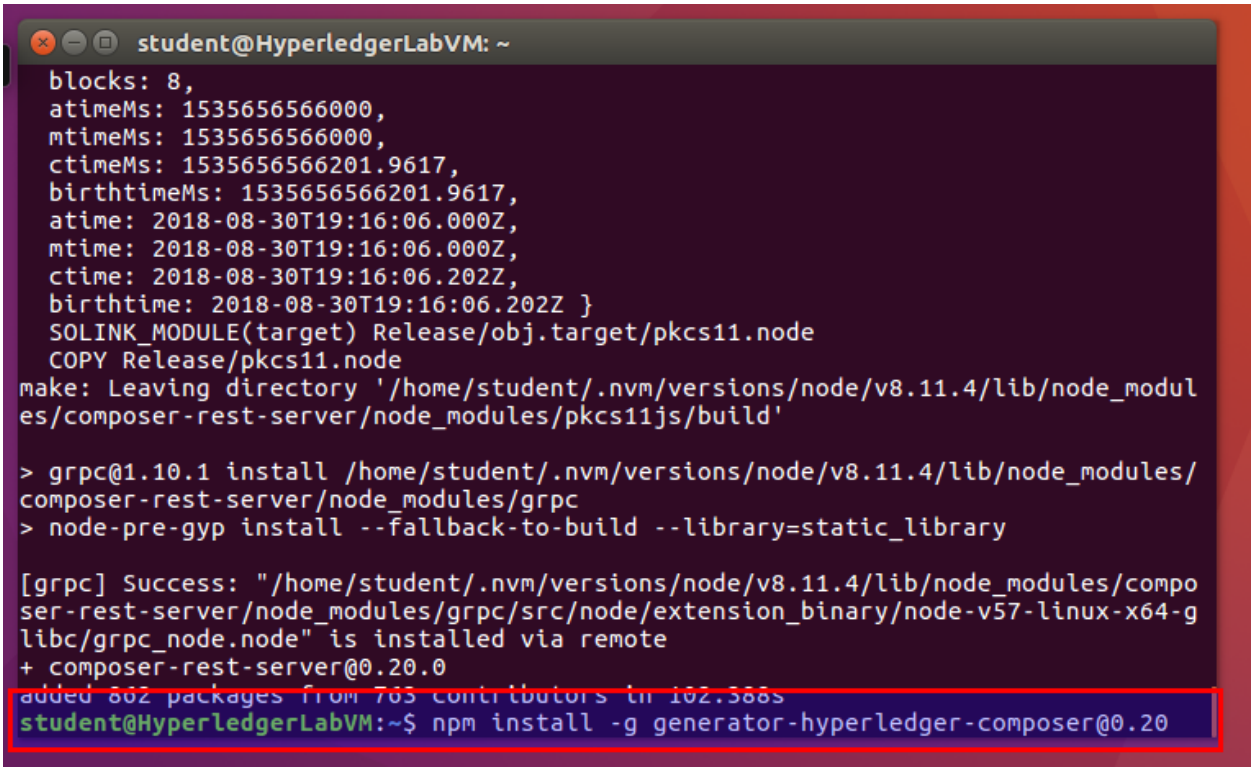


Figure 255 - Install the Composer Generator utility.

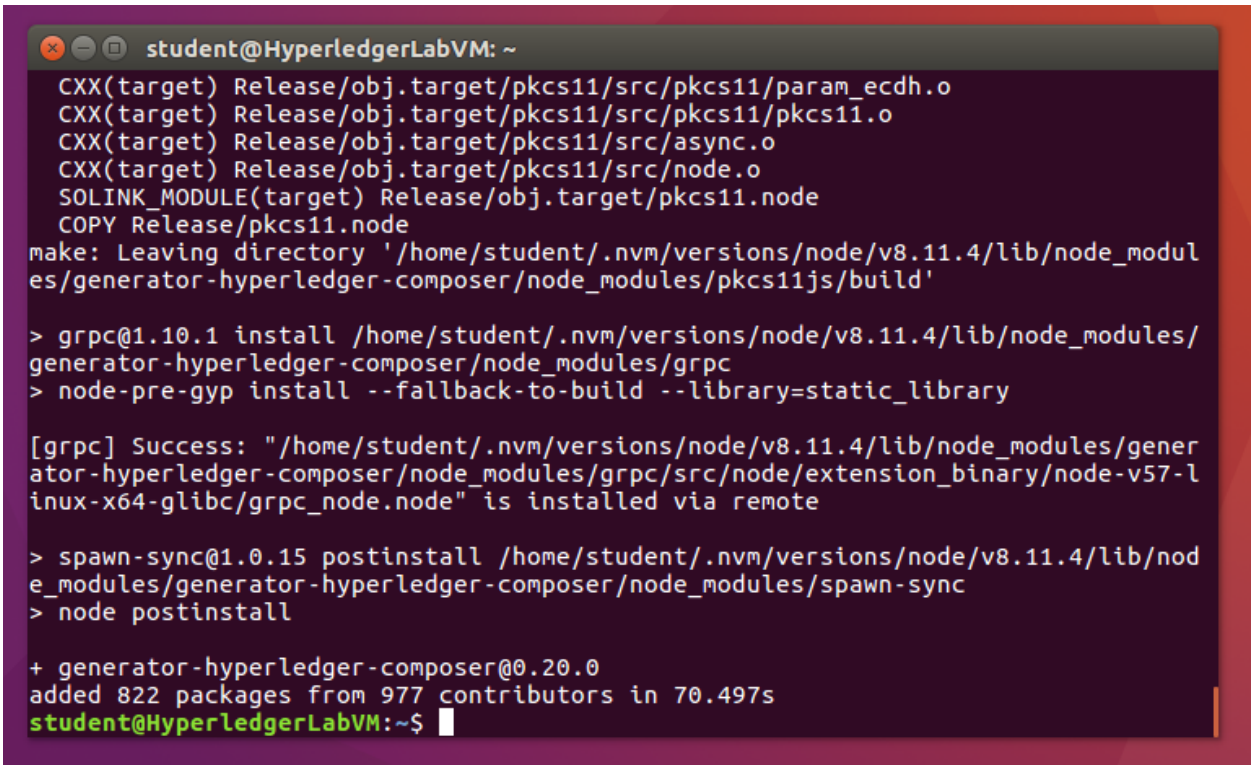


Figure 256 - Installation of the Composer Generator utility is complete.

Step 5 – Install the Yeoman code generation tool

Yeoman is a tool for generating code based on templates. Yeoman will be used throughout the labs to help generate code and project scaffoldings. It will use the Composer Generator utility installed on the previous step. To install Yeoman, run the command:

```
npm install -g yo
```

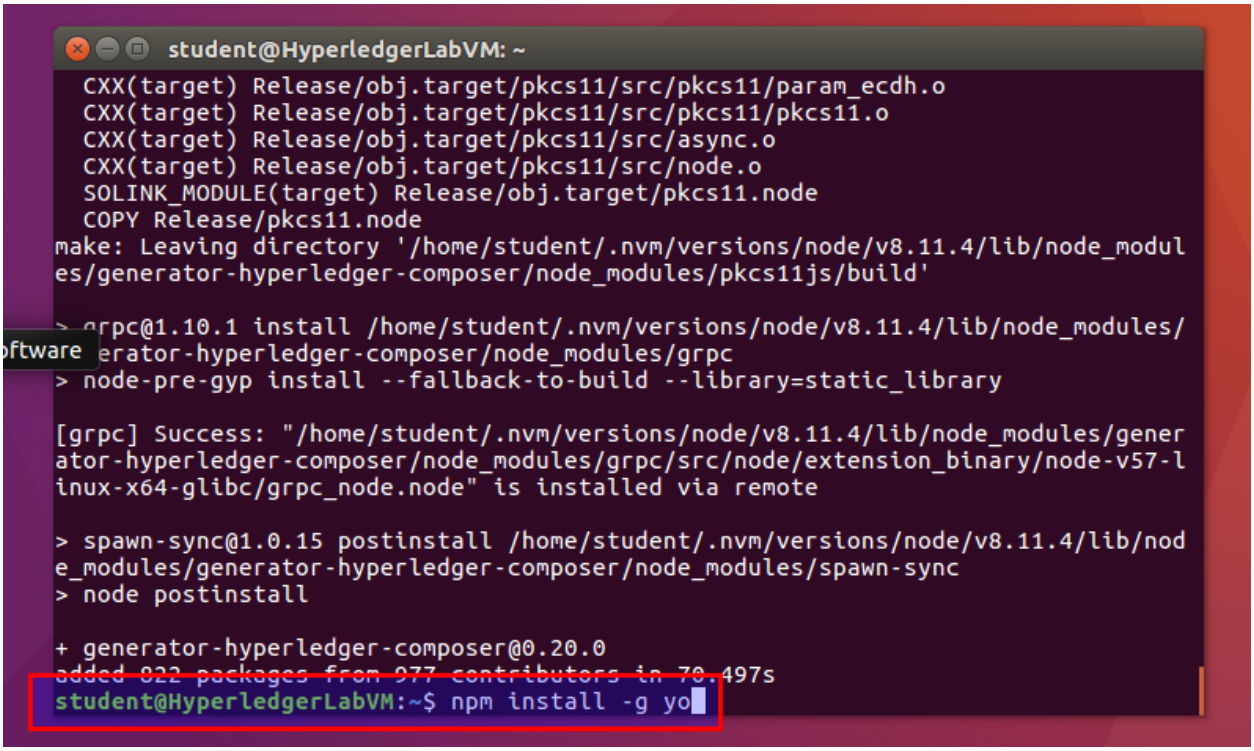


Figure 257 - Install the Yeoman tool.

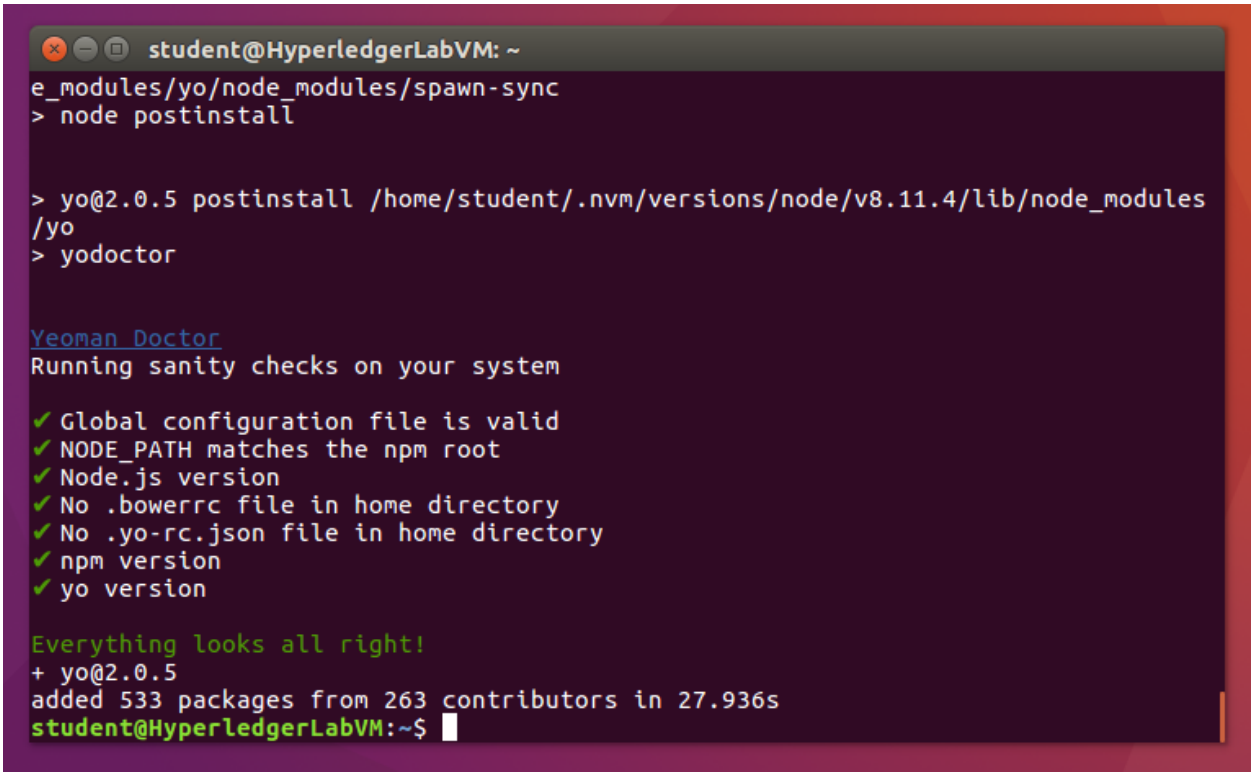


Figure 258 - Installation of Yeoman is complete.

Step 6 – Install Composer Playground

Composer Playground is a web-based toolset for designing and testing a Hyperledger Composer solution. Composer Playground can also be installed locally so that network access is not required to start creating and testing solutions. To install Composer Playground locally, run the following command:

```
npm install -g composer-playground@0.20
```

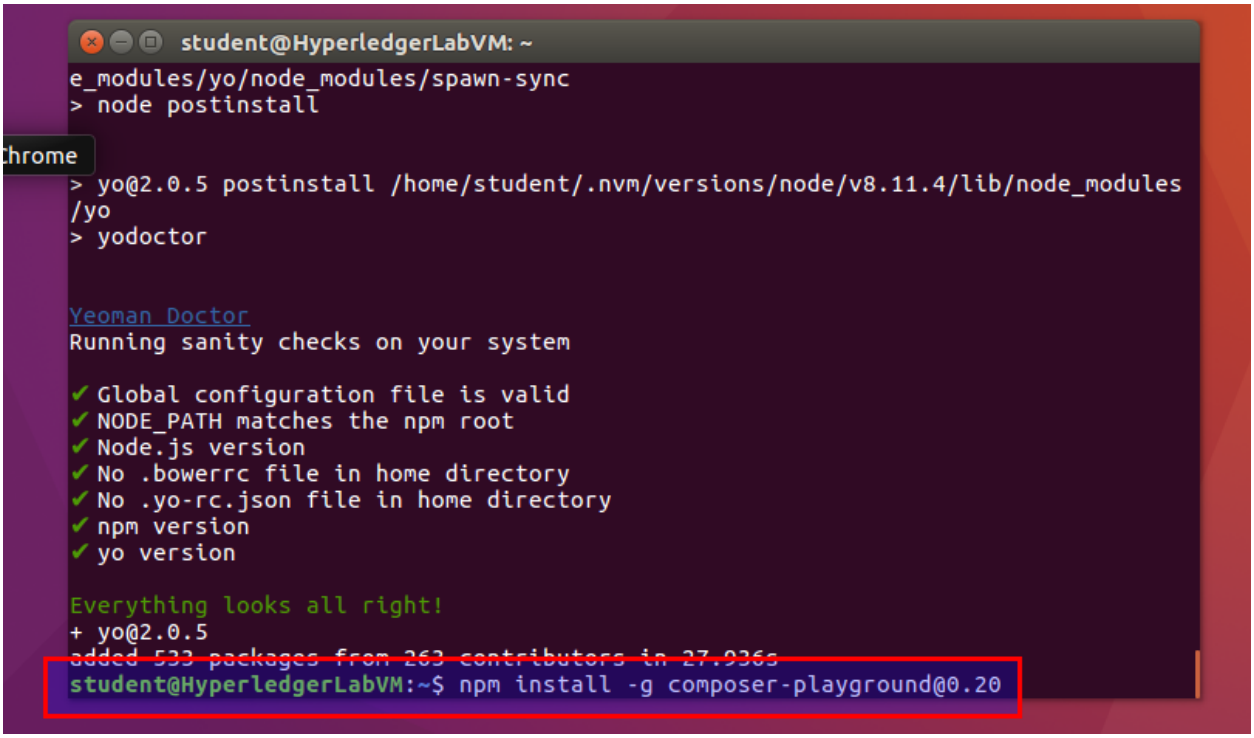


Figure 259 - Installing Composer Playground.

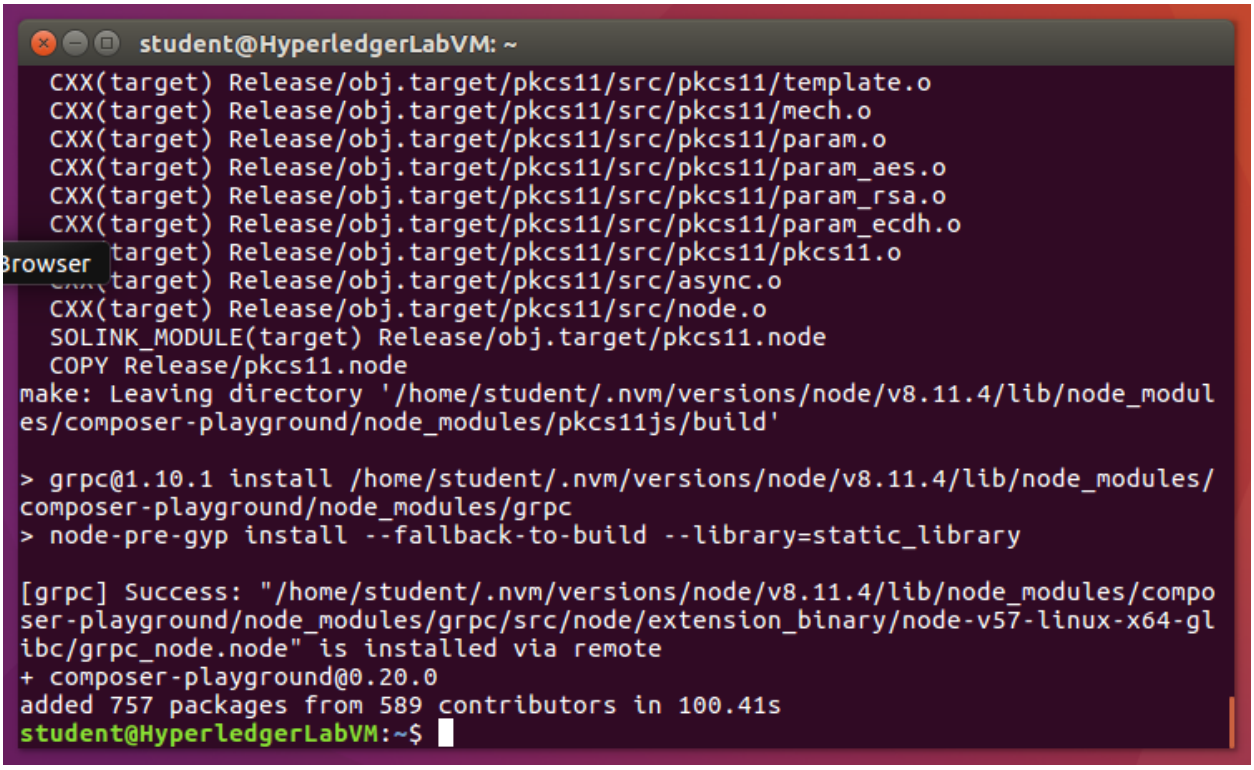


Figure 260 - Composer Playground local install complete.

Step 7 – Install the Visual Studio Code IDE

***** This step can be skipped in BTA provided lab environments!***

Many different IDEs can be used to work with Hyperledger Composer solutions. One popular choice is Visual Studio Code. Code is an open-sourced IDE and has available plug-ins for Composer solutions. In this step you'll install Visual Studio Code into your development environment, but note that practically any IDE can be used in place of Code.

Begin by opening a browser:

Figure 261 - Launch a web browser

Then navigate to:

`https://code.visualstudio.com/download`

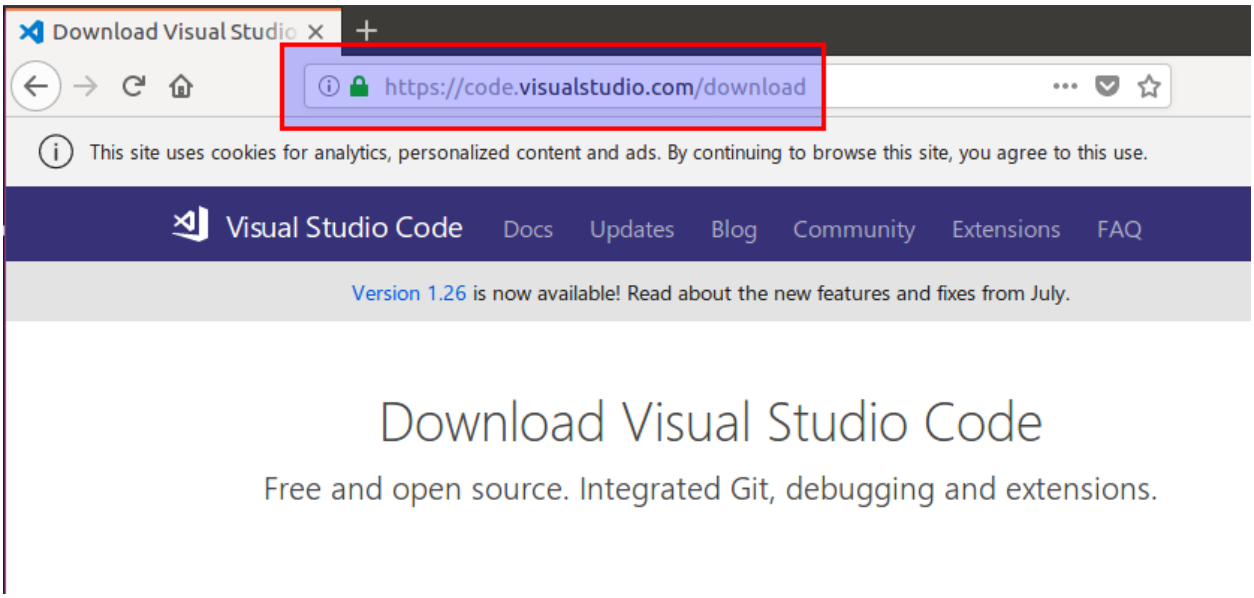


Figure 262 - Navigate to the download page.

Select the option to download the "Debian, Ubuntu" installation

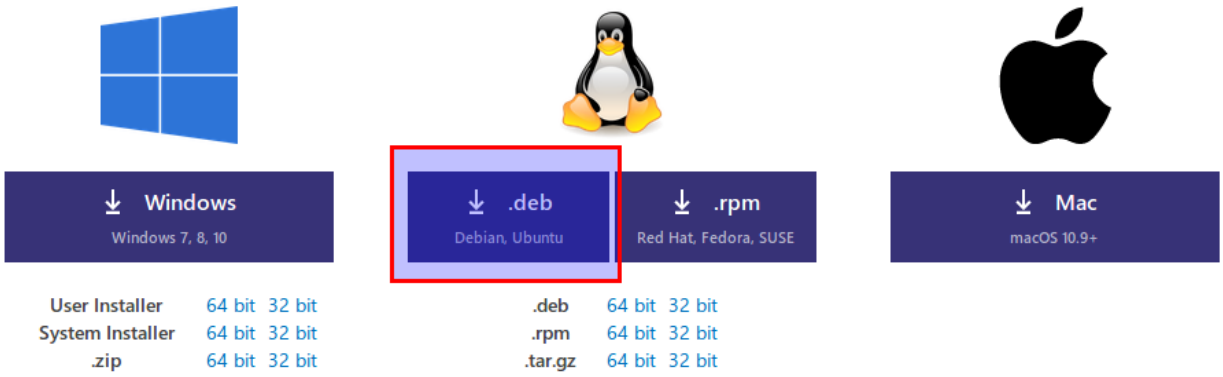


Figure 263 - Select the correct installation package.

Once the proper package has been selected, select the “Open with” option on the download window. Make sure "Software Install (default)" is selected as the application to "Open with".

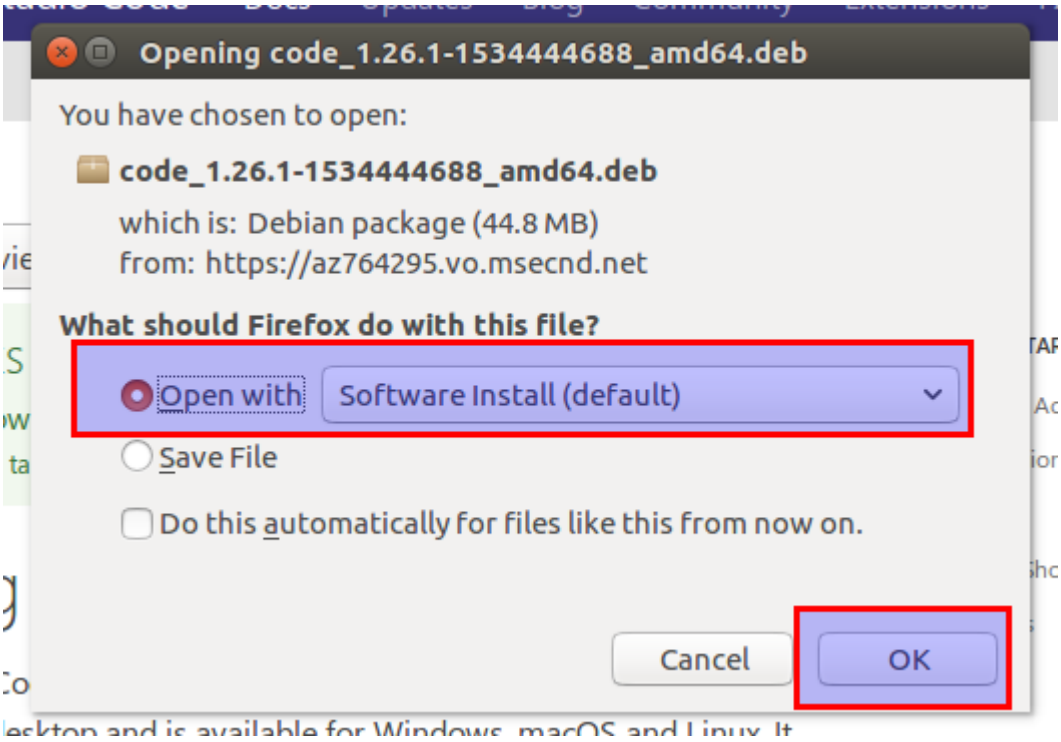


Figure 264 - Use the Software Install utility to handle the download.

When the "Ubuntu Software" application loads, select the "Install" option.

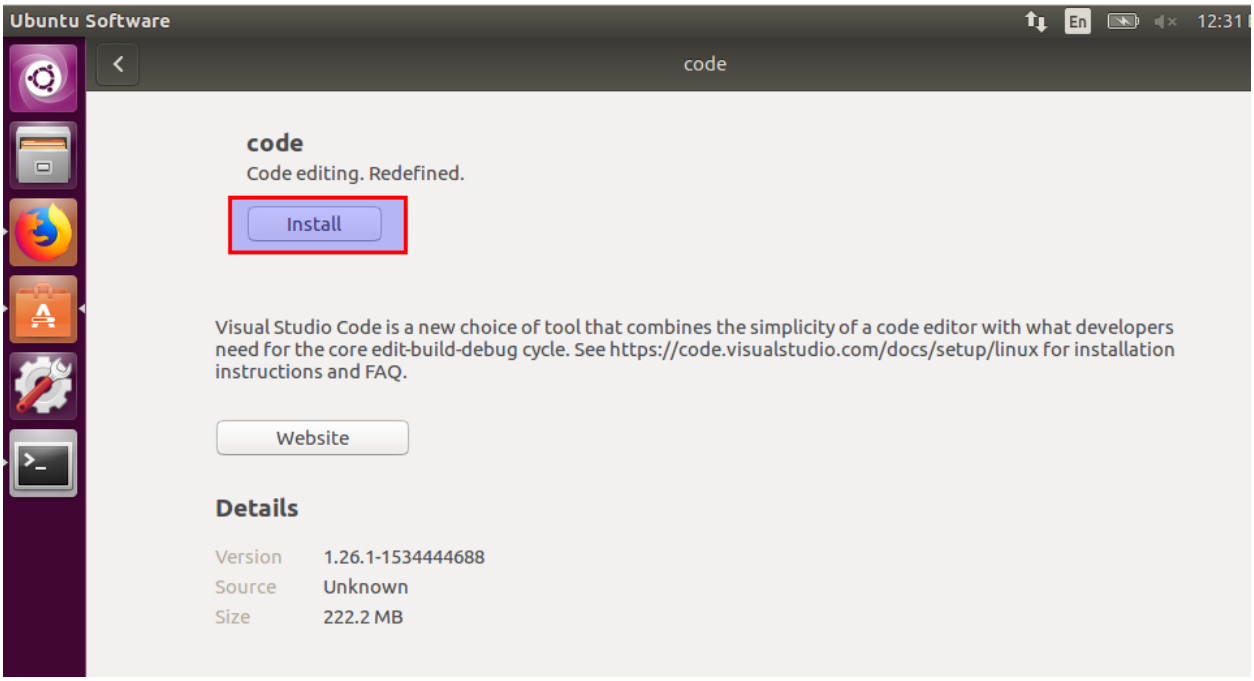


Figure 265 - Start the Code installation.

Provide the password if prompted and select "Authenticate", then close the "Ubuntu Software" application and the web browser.

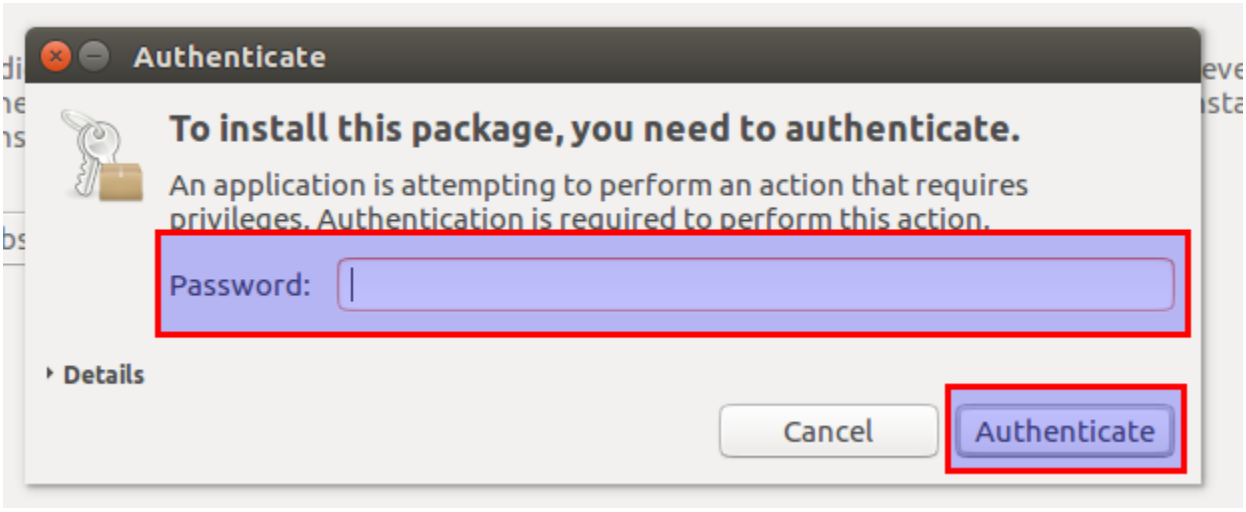


Figure 266 - Provide admin access.

Step 8 – Start Visual Studio Code

Start Visual Studio Code by clicking on the "Search your computer" button and searching for "code". Visual Studio Code should appear in the search results. Click the icon to launch the IDE.

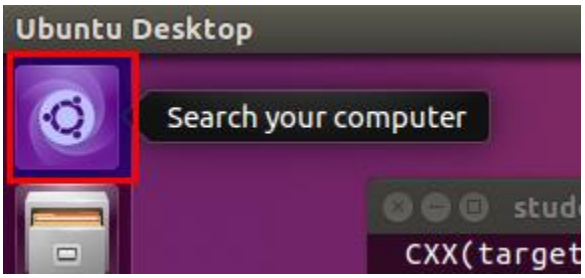


Figure 267 - Open up the applications menu.

Figure 268 - Search for Code

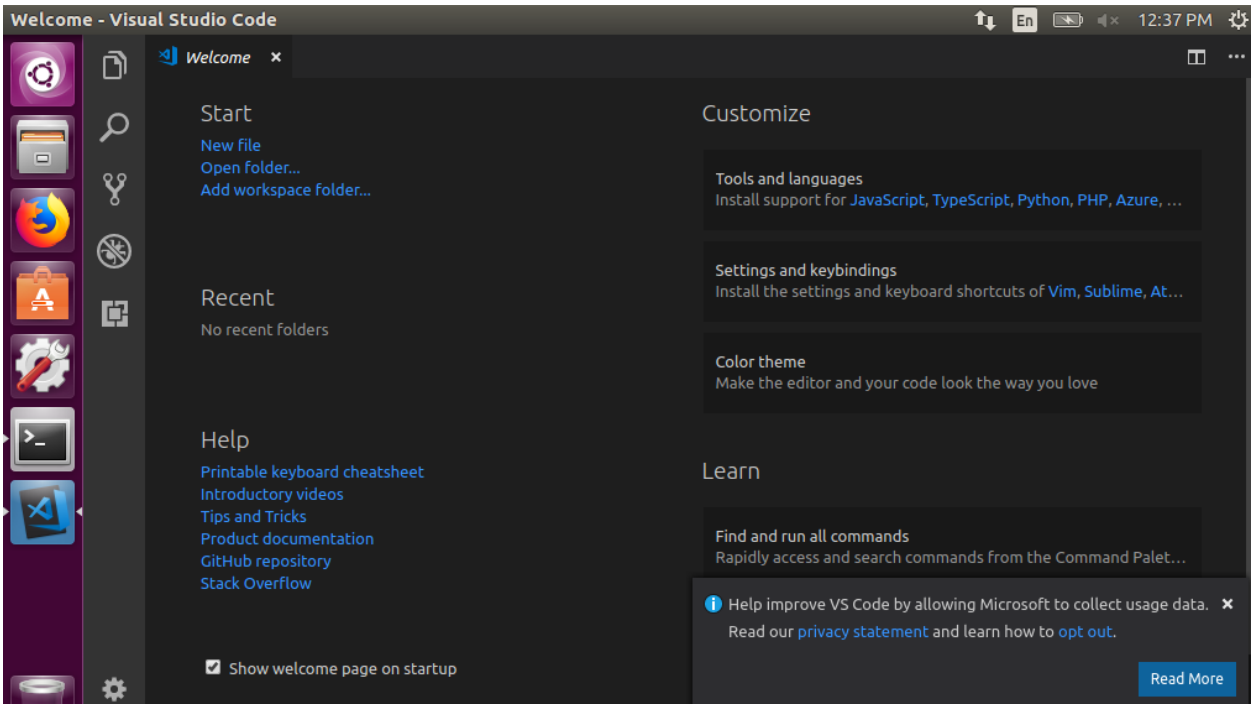


Figure 269 - Launch Visual Studio Code

Step 9 – Add the Hyperledger Composer extension to Code

An extension for Hyperledger Composer solutions is available to add to Visual Studio Code. This will give you Intellisense and code formatting suggestions when using Code to work with Composer solutions. To install the extension, follow these steps:

From Visual Studio Code, click on the Extensions button

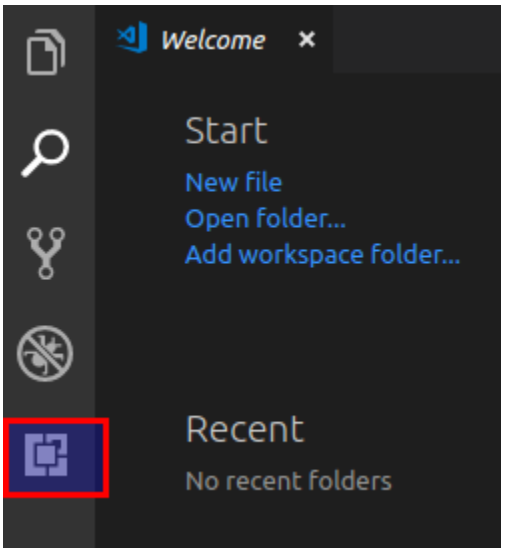


Figure 270 - Click on the extensions button in Code.

Search for "Hyperledger Composer" and click the "Install" button for the "Hyperledger Composer" extension.

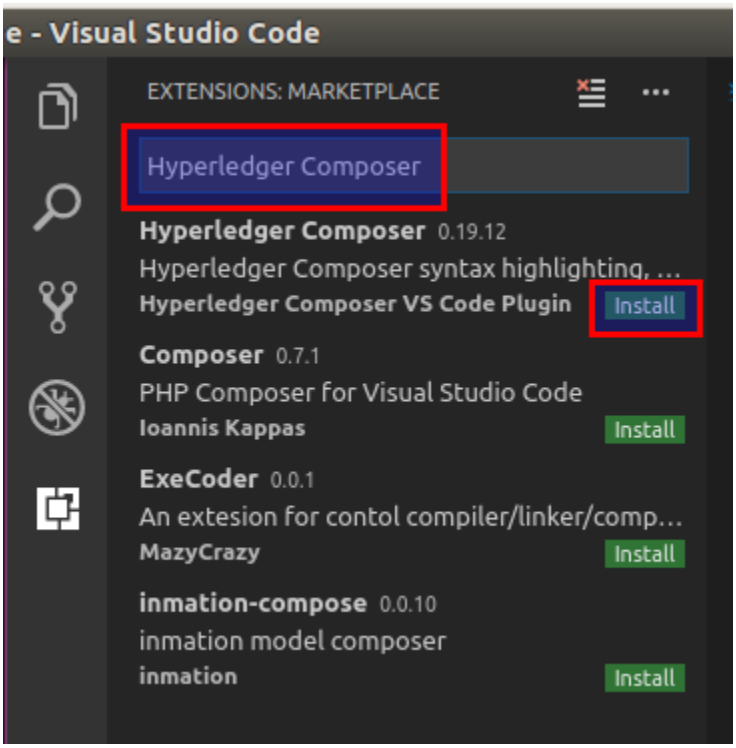


Figure 271 - Search for the Composer extension.

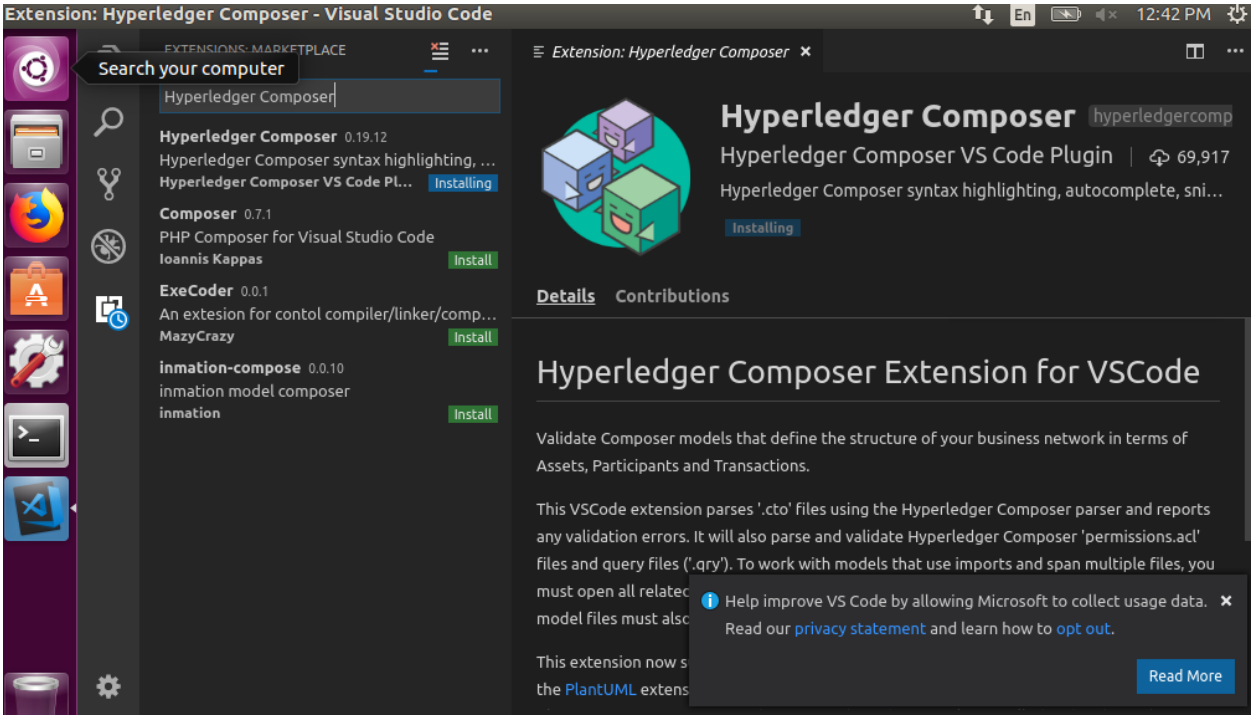


Figure 272 - Installing the Composer extension.

Step 10 - Install Hyperledger Fabric

This step will create a local Hyperledger Fabric runtime to deploy your business networks to. Start by running the following command to create a new directory to hold all the Fabric components:

```
mkdir ~/fabric-dev-servers && cd ~/fabric-dev-servers
```

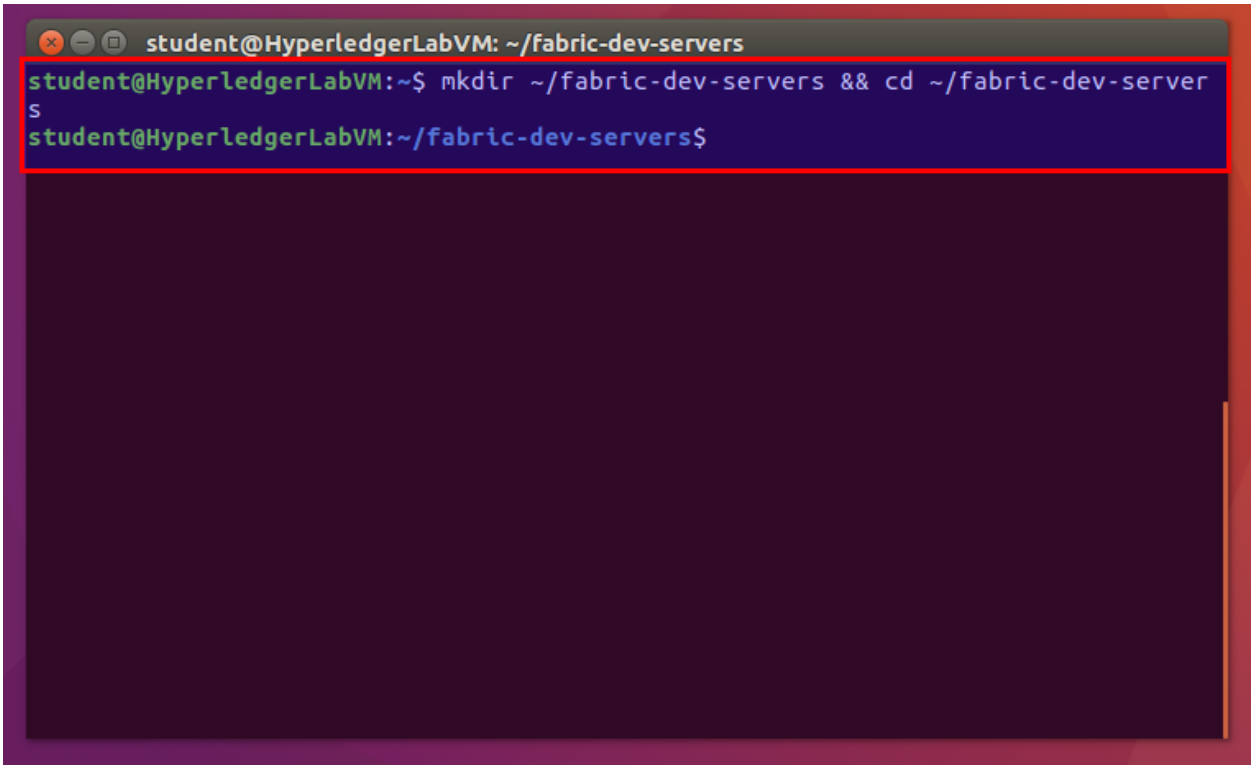


Figure 273 - Creating a new components directory for Fabric.

Next, download the Fabric Dev Server components by running the following command:

```
curl -O https://raw.githubusercontent.com/hyperledger/composer-tools/master/packages/fabric-dev-servers/fabric-dev-servers.tar.gz
```

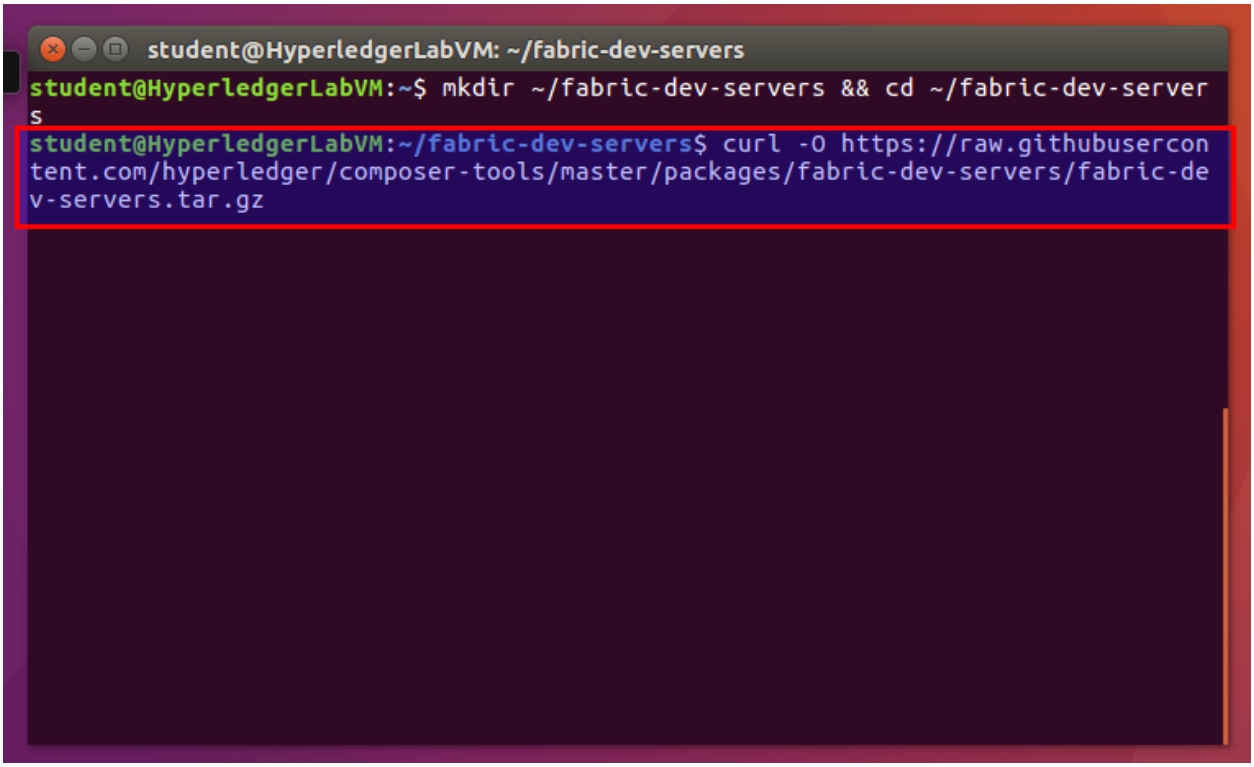


Figure 274 - Running the command to download the Fabric runtime components.

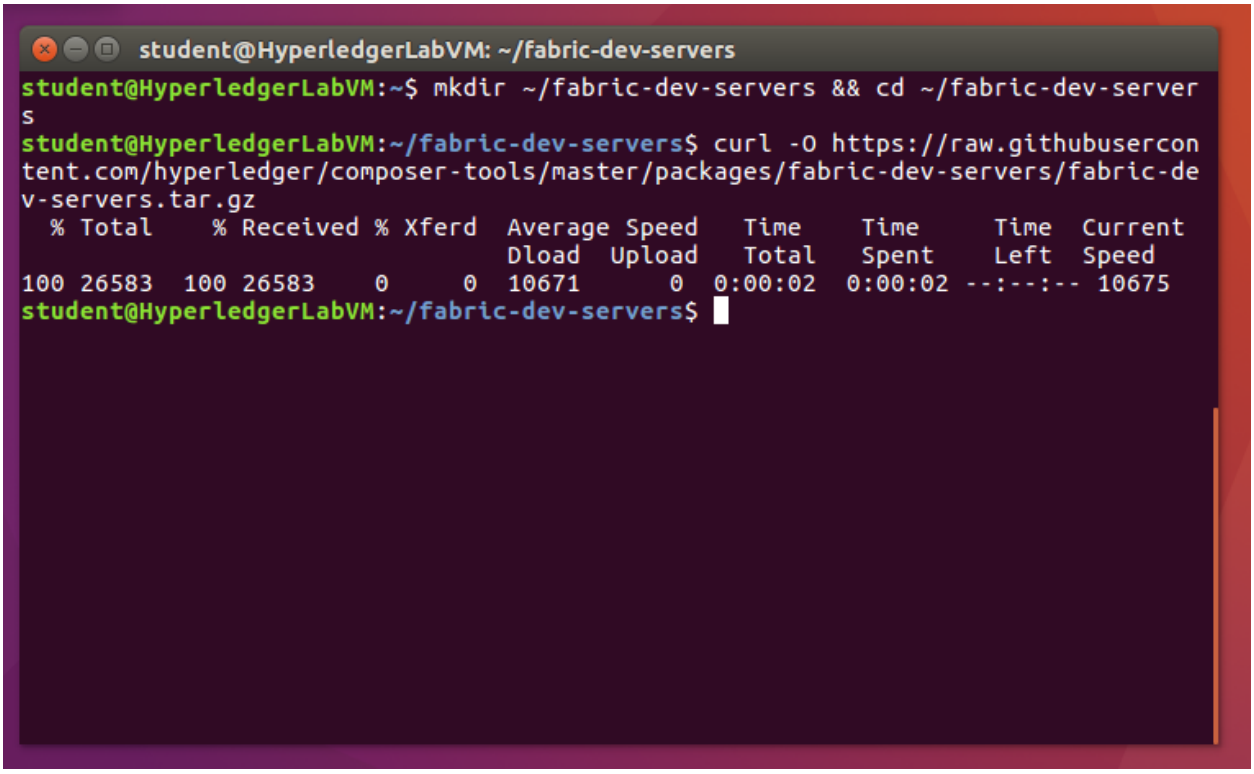


Figure 275 - Download complete.

Next, unpack the archive you downloaded using the following command:

```
tar -xvf fabric-dev-servers.tar.gz
```

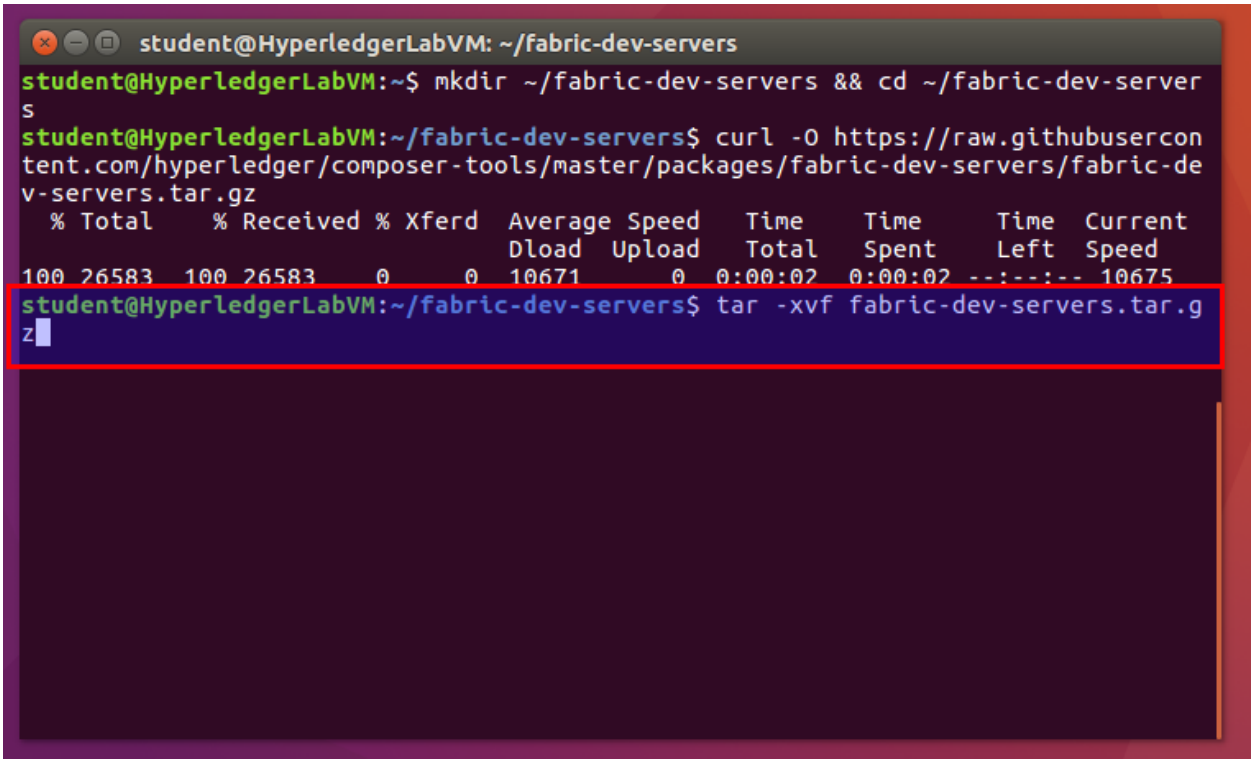


Figure 276 - Unpacking the archive.

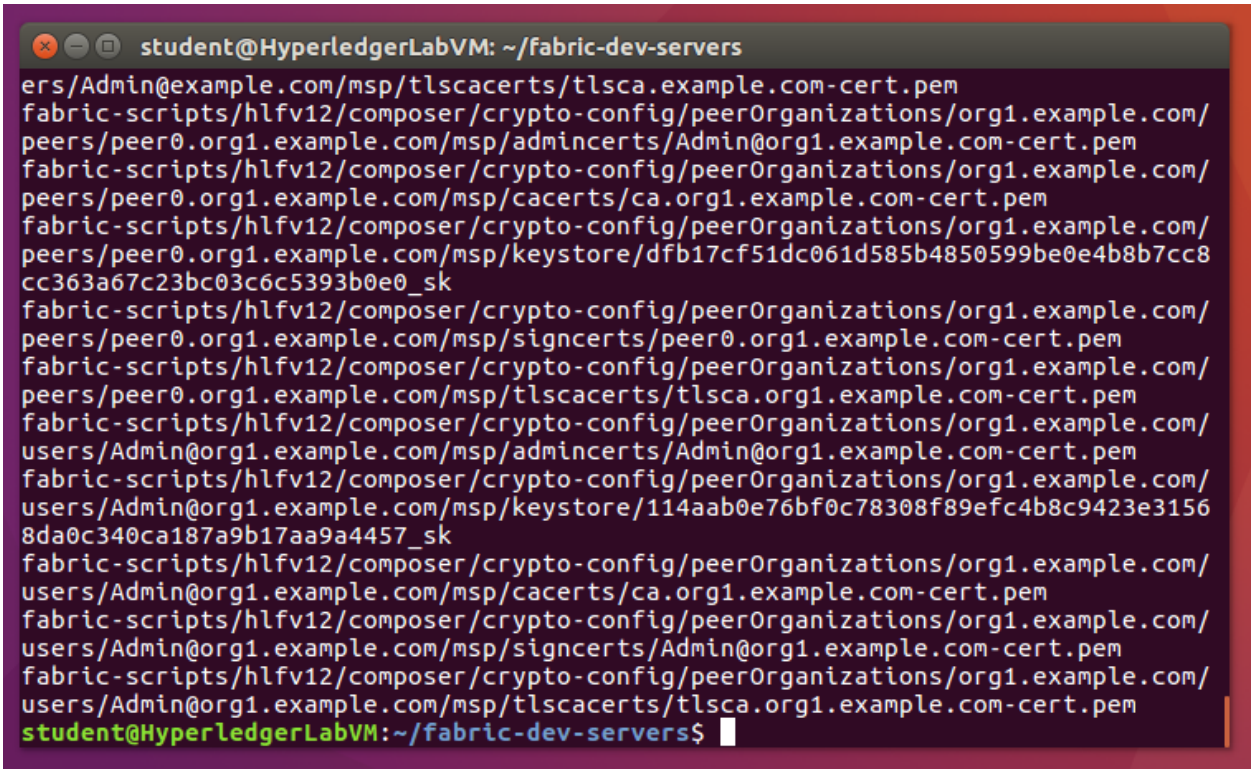


Figure 277 - Unpacking complete.

Navigate to the fabric dev servers folder you created earlier by running the following command:

```
cd ~/fabric-dev-servers
```

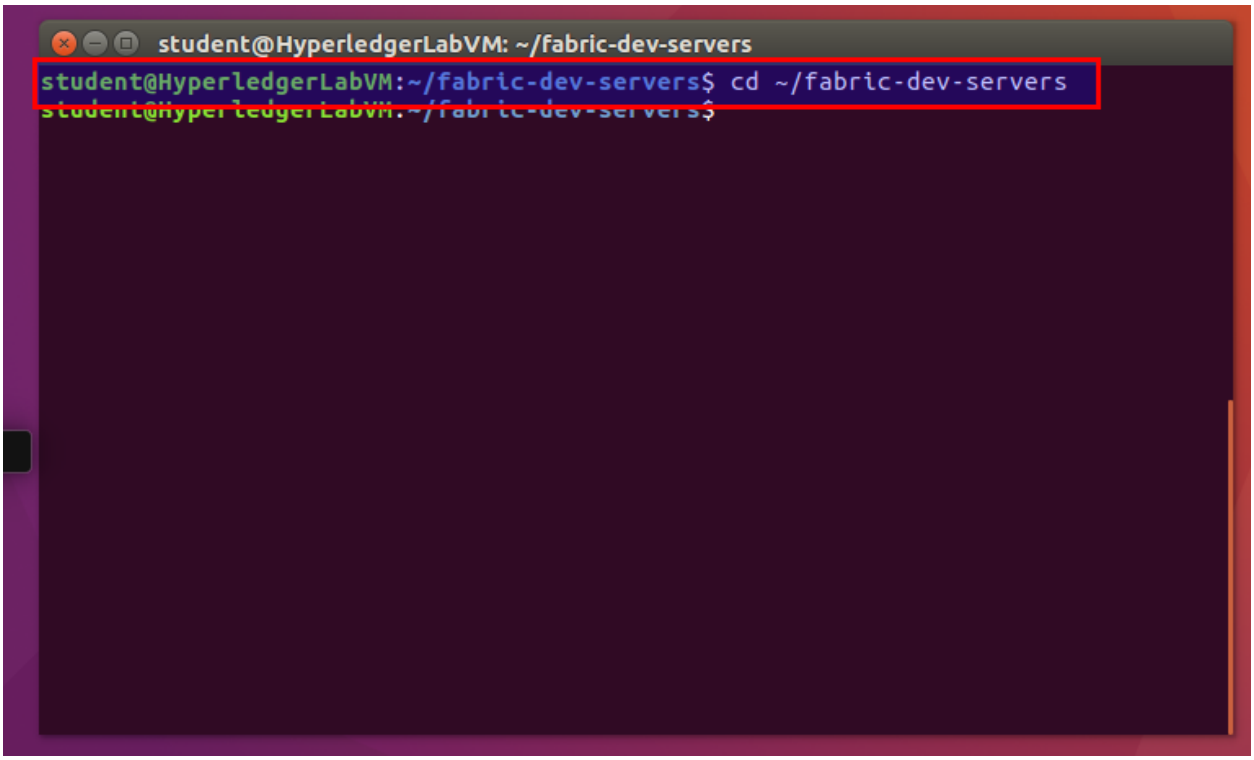


Figure 278 - Navigate to the dev servers folder.

Set the Fabric version using the following command:

```
export FABRIC_VERSION=hlfv12
```

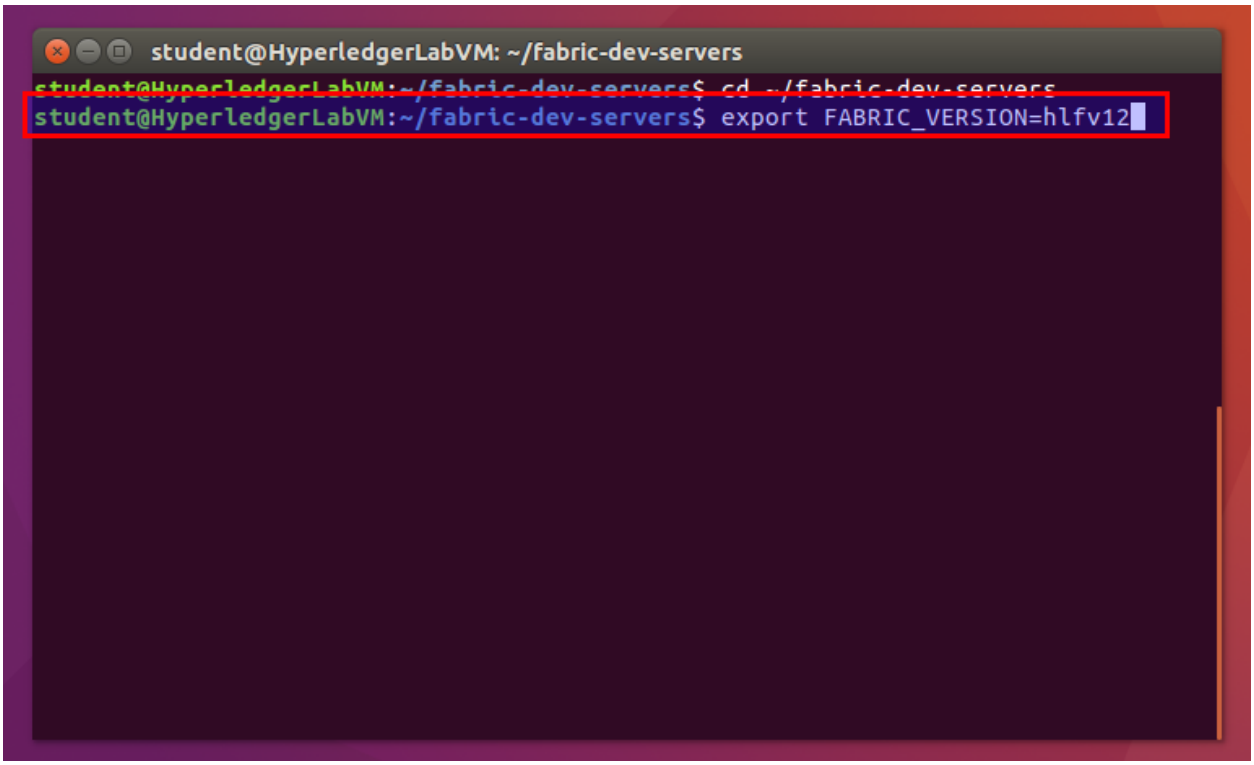


Figure 279 - Setting the Fabric version.

Next, the current user needs to be added to the docker group. The can be accomplished with the following command:

```
sudo usermod -a -G docker student
```

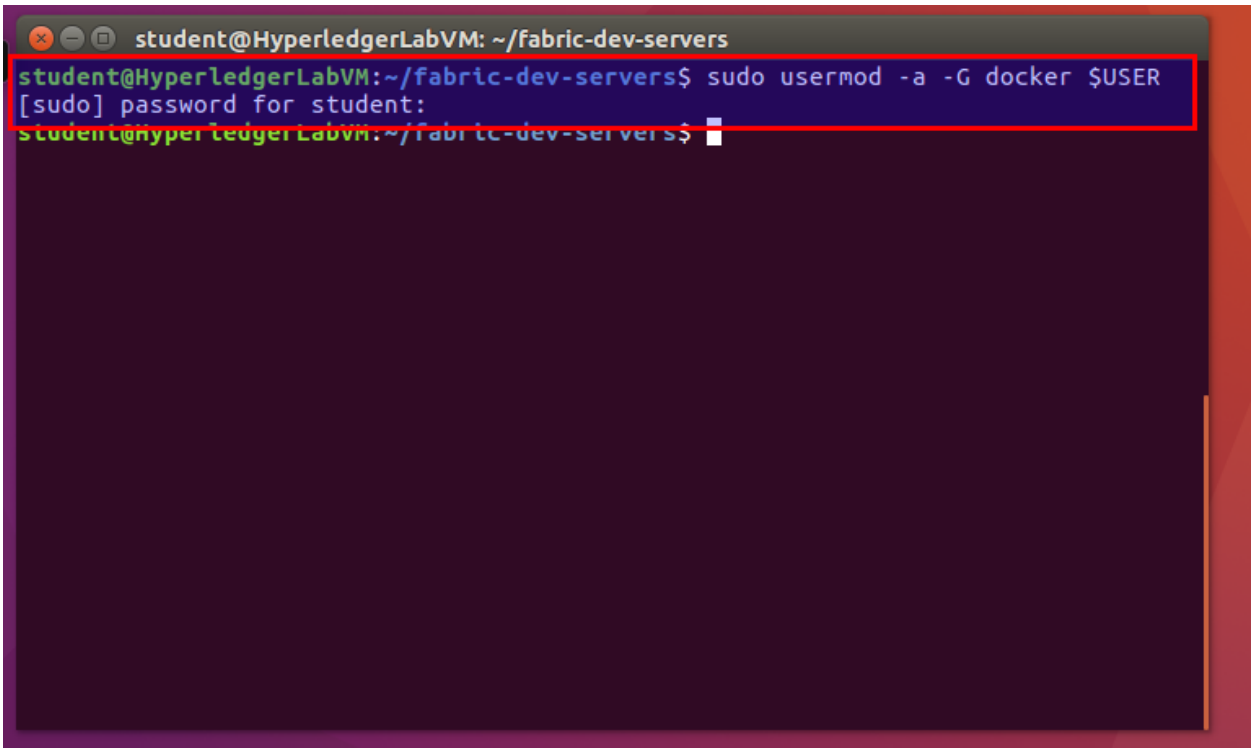


Figure 280 - Adding the current user to the Docker group.

In order for the changes you've made to take effect, the image needs to be rebooted. You can reboot the image using the command below. Wait a few minutes for the reboot to complete, then log back into the machine.

```
sudo reboot now
```

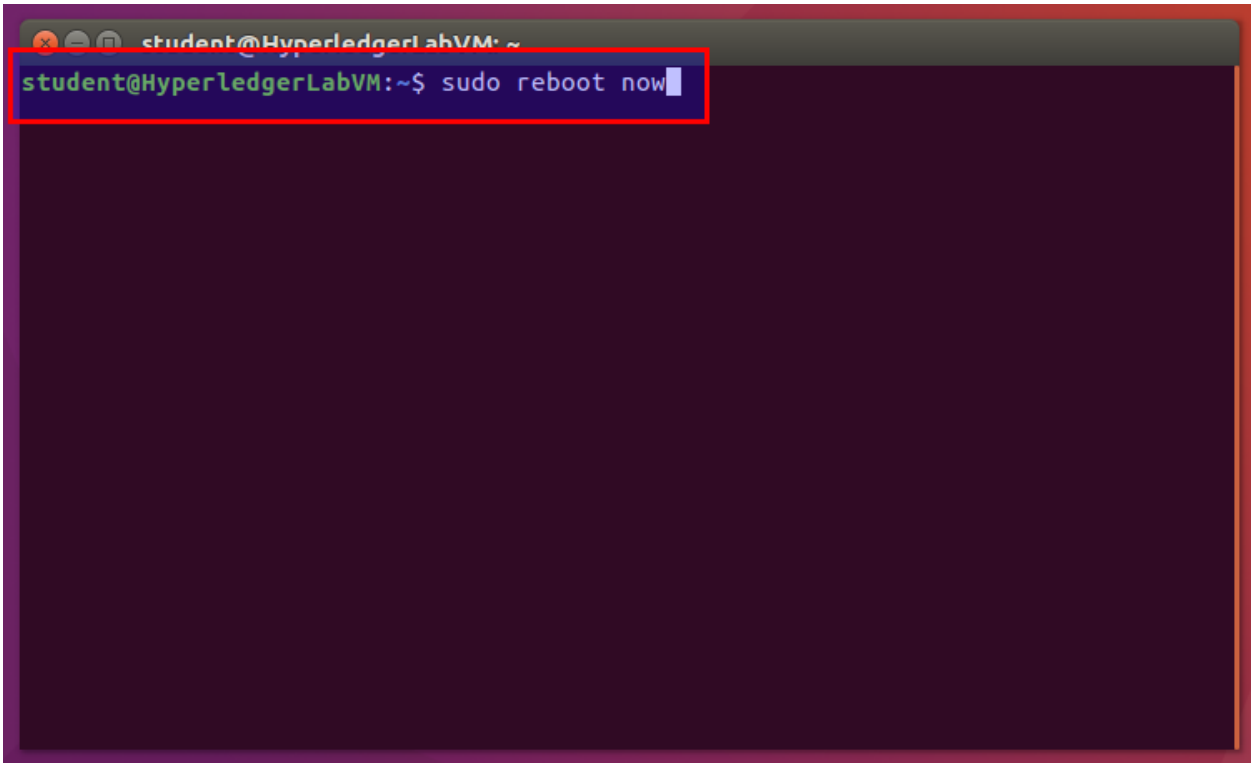


Figure 281 - Rebooting the machine.

Once the machine has rebooted and you've logged back in, open up a terminal window navigate to the Fabric dev servers directory using the following command:

```
cd ~/fabric-dev-servers
```

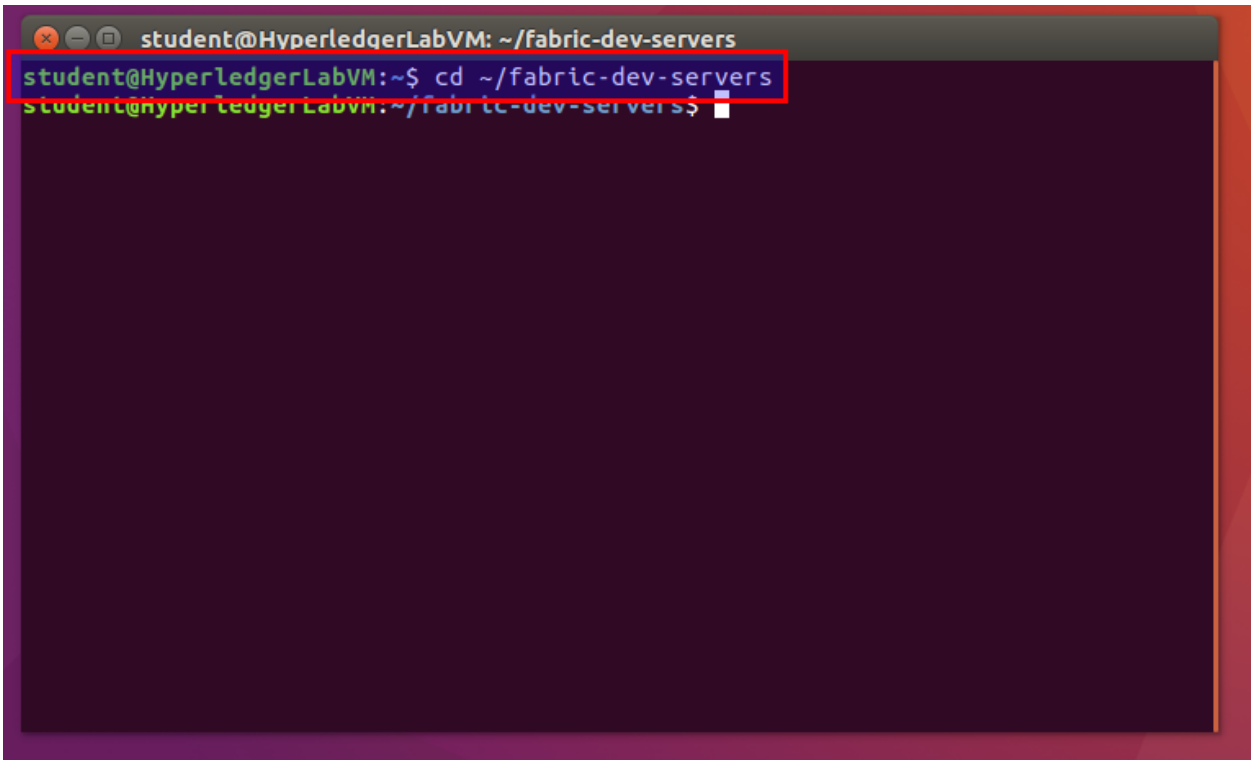


Figure 282 - Navigate to the dev servers folder.

Run the downloadfabric script to download and install required Fabric components. To run this script, issue the following command:

```
./downloadFabric.sh
```

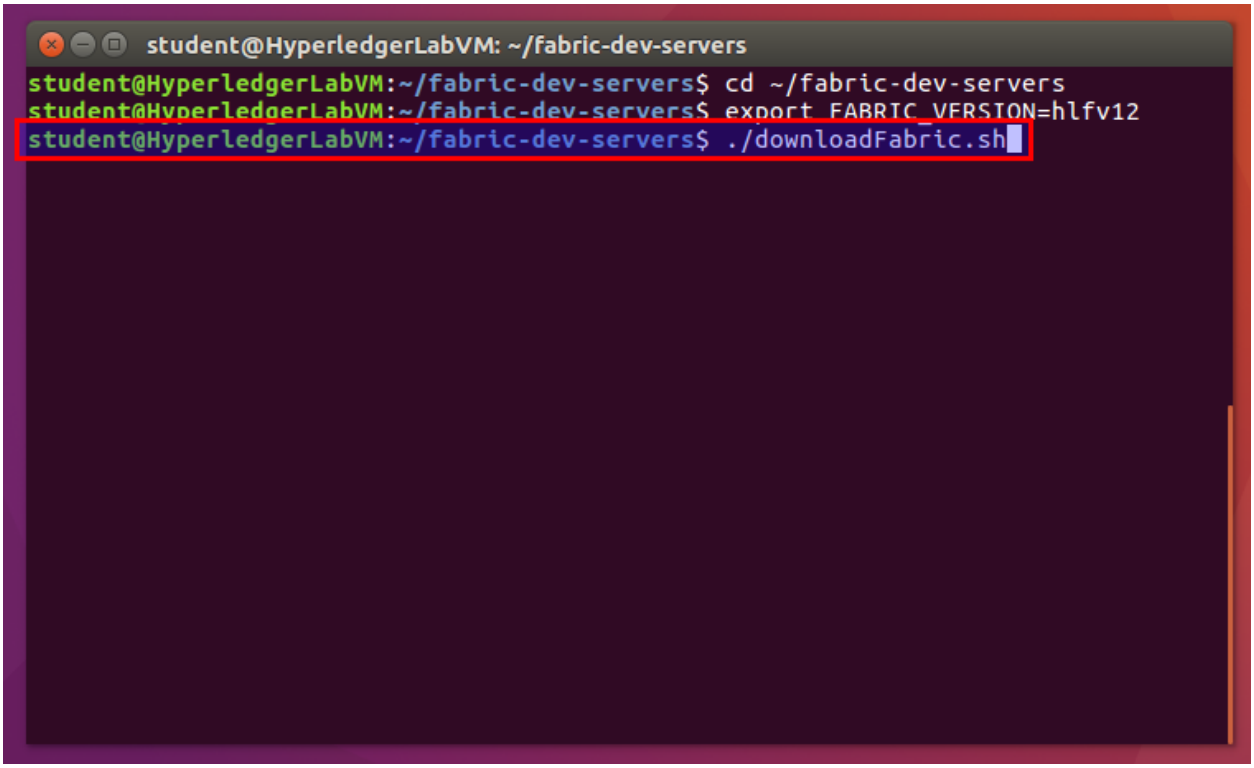


Figure 283 - Running the script.

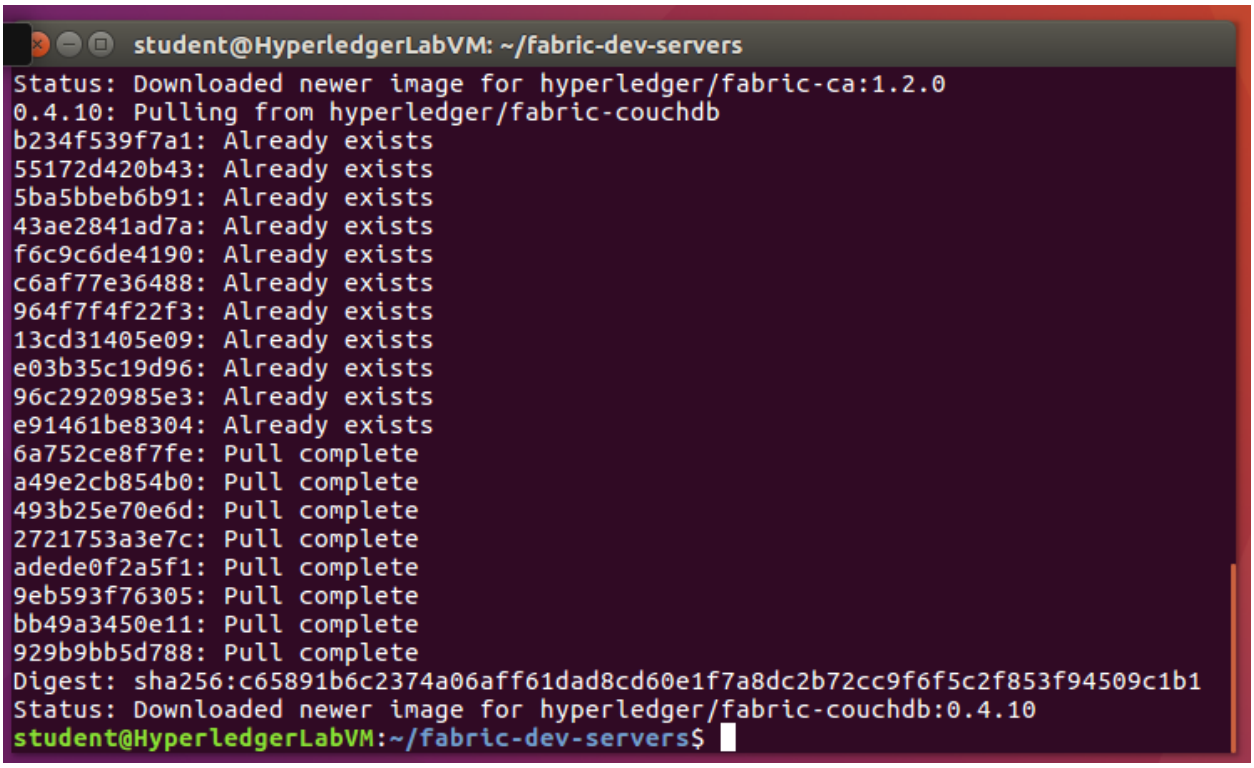


Figure 284 - Script complete.

Step 11 – Starting up a new Fabric runtime

To start a new Fabric runtime, you'll need to run the start script, then generate a PeerAdmin network card. Start by navigating to the fabric dev servers folder using the following command:

```
cd ~/fabric-dev-servers
```

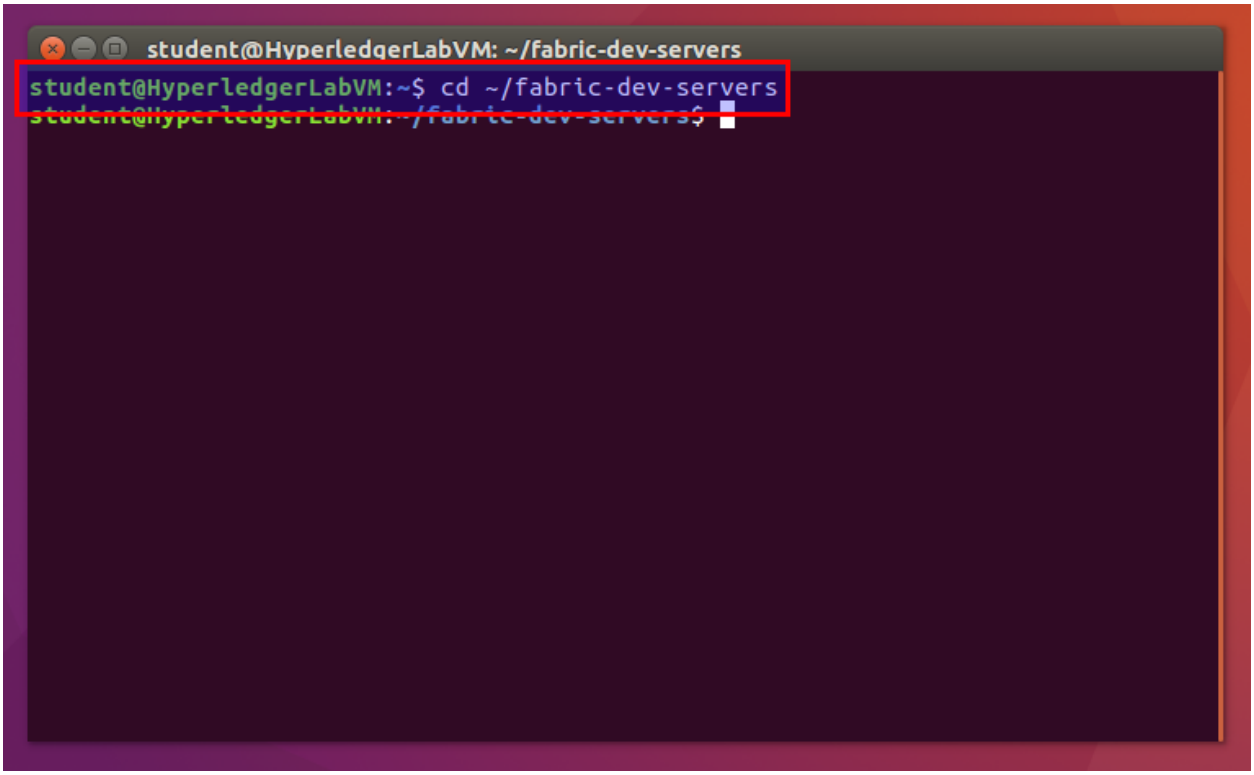


Figure 285 - Navigating to the directory.

Then, ensure the correct Fabric version is set using the command below:

```
export FABRIC_VERSION=hlfv12
```

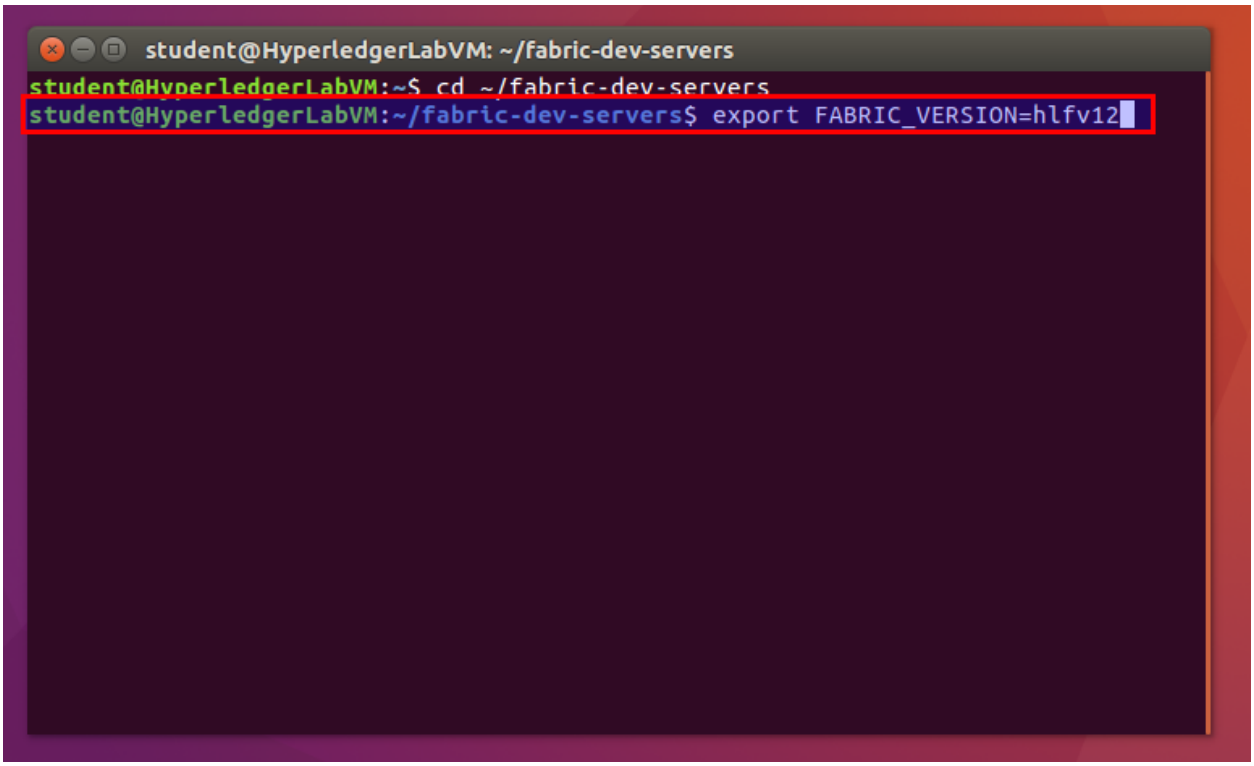


Figure 286 - Set the Fabric version.

Start the Fabric network by executing the startFabric script using the commands below:

```
export PATH=/bin:/usr/bin:/usr/local/bin:/sbin:/usr/sbin
./startFabric.sh
```

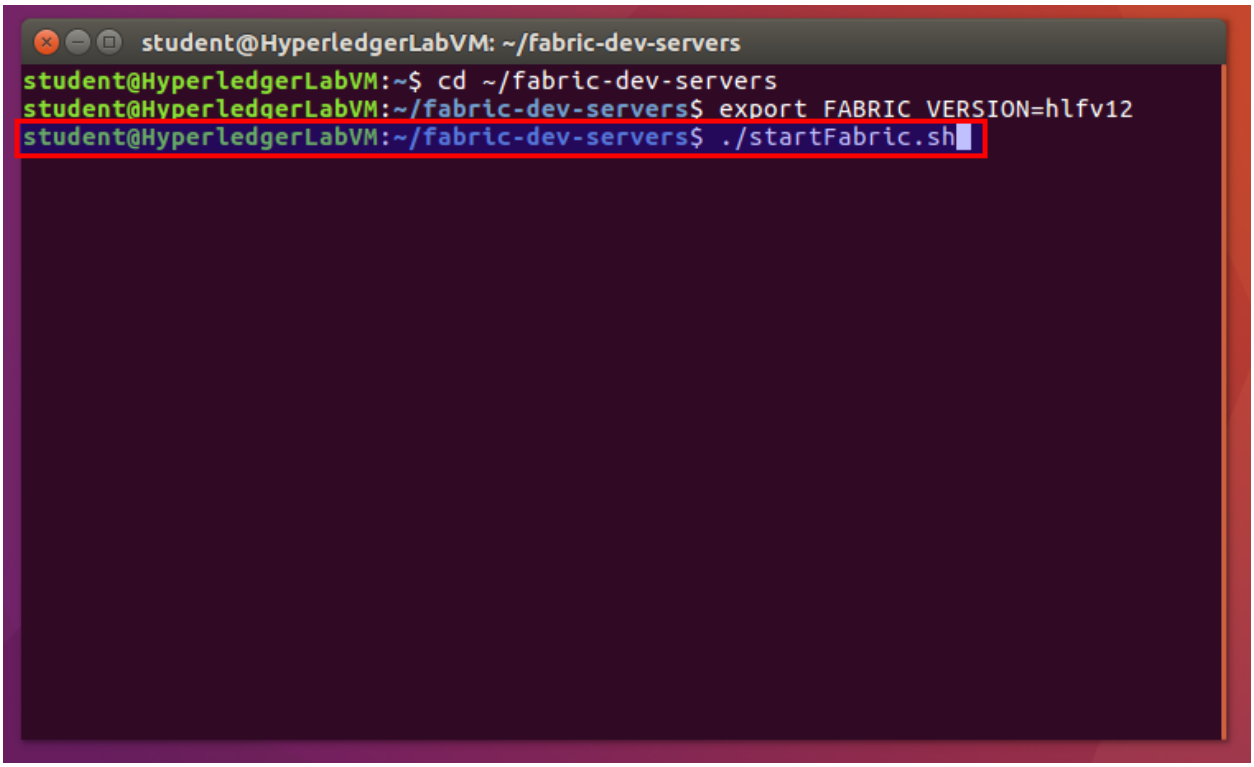


Figure 287 - Run the startFabric script.

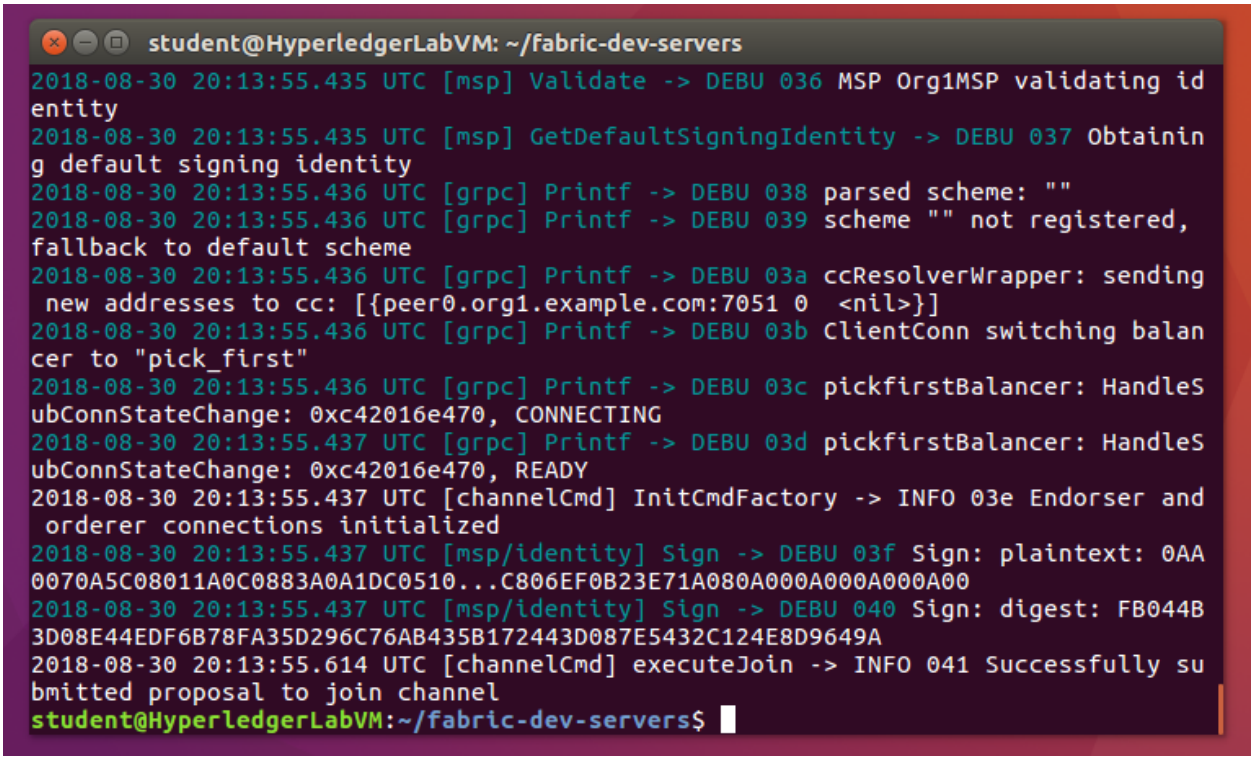


Figure 288 - Script is complete.

Finally, create the PeerAdmin network card by running the createPeerAdminCard script using the following command:

```
./createPeerAdminCard.sh
```

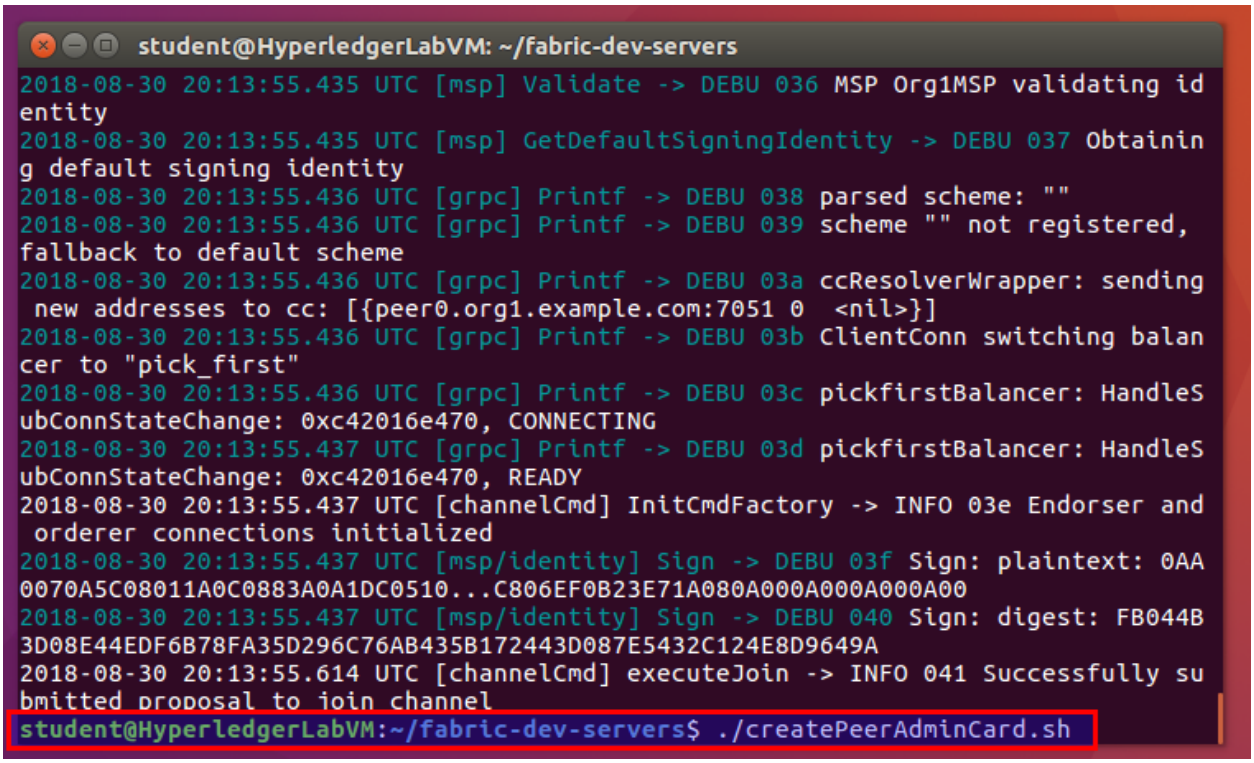


Figure 289 - Running the createPeerAdminCard script.

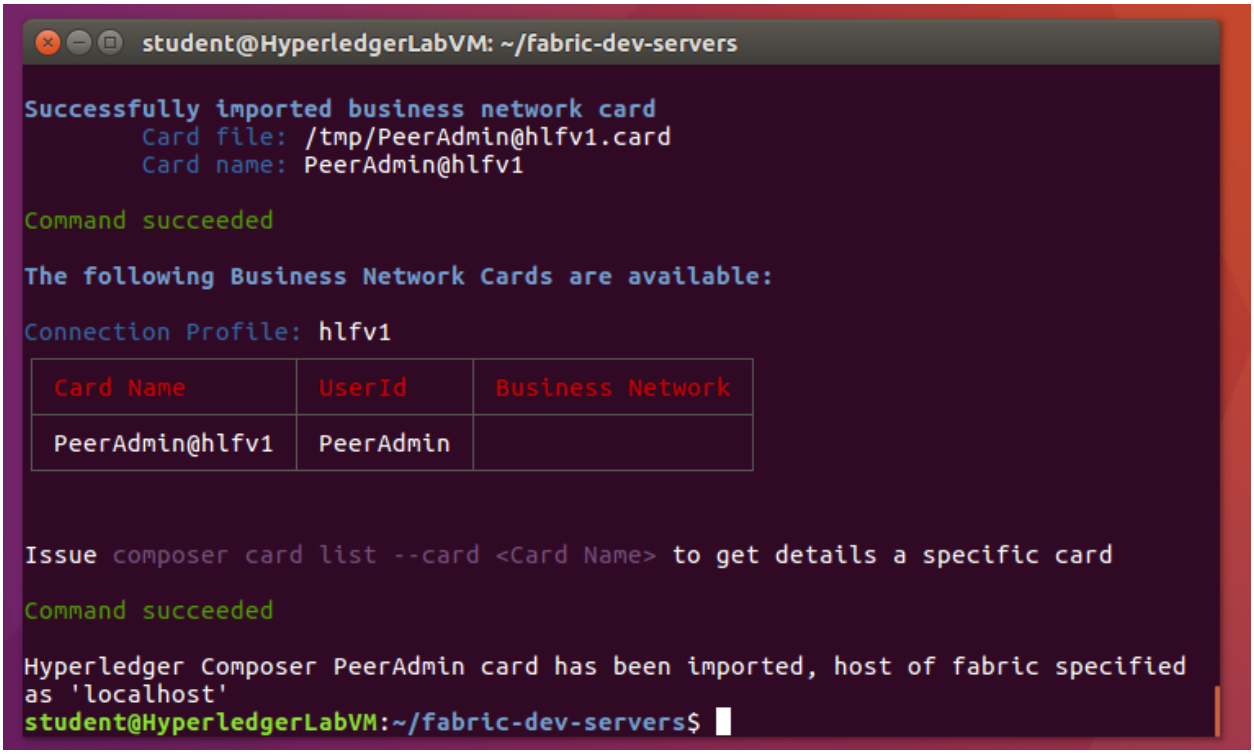


Figure 290 - Script complete.

Step 12 - Verify Installation

In this step, we'll quickly verify all required components were installed correctly by ensuring Composer Playground can start locally. Start Composer Playground by running the following command:

```
composer-playground
```

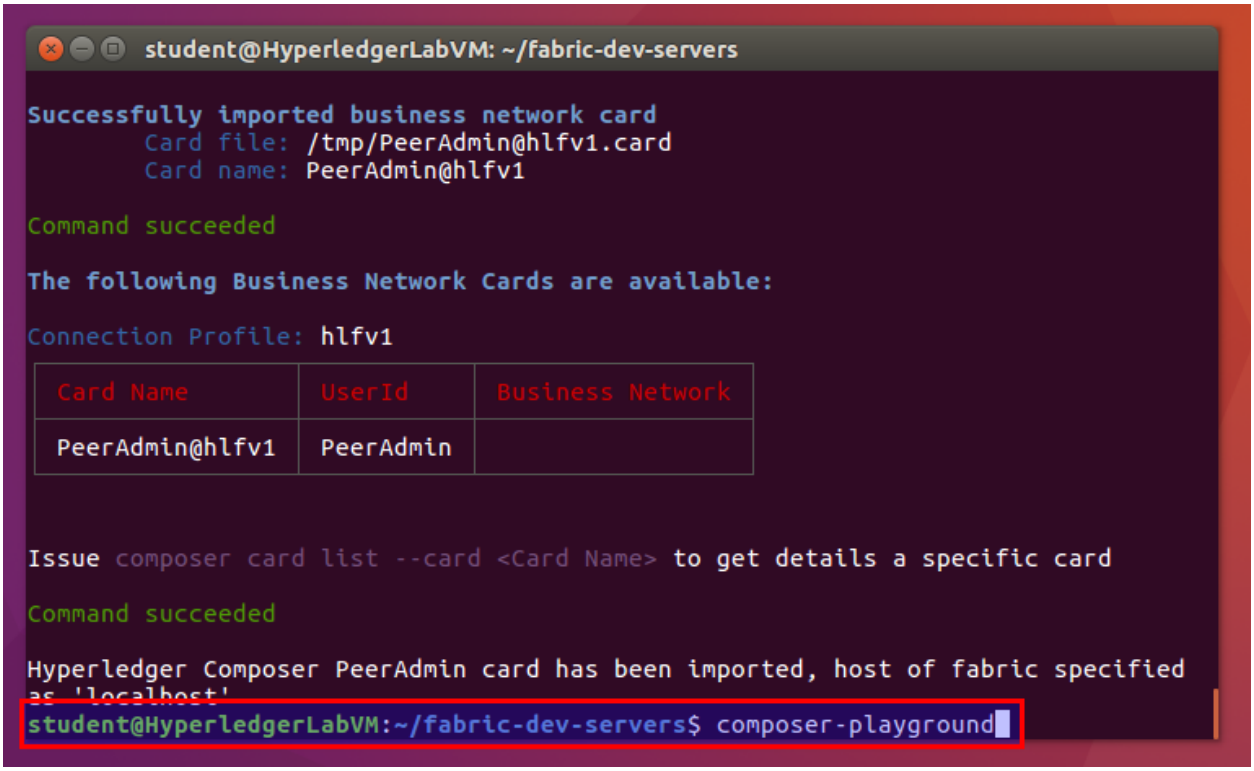


Figure 291 - Staring Composer Playground.

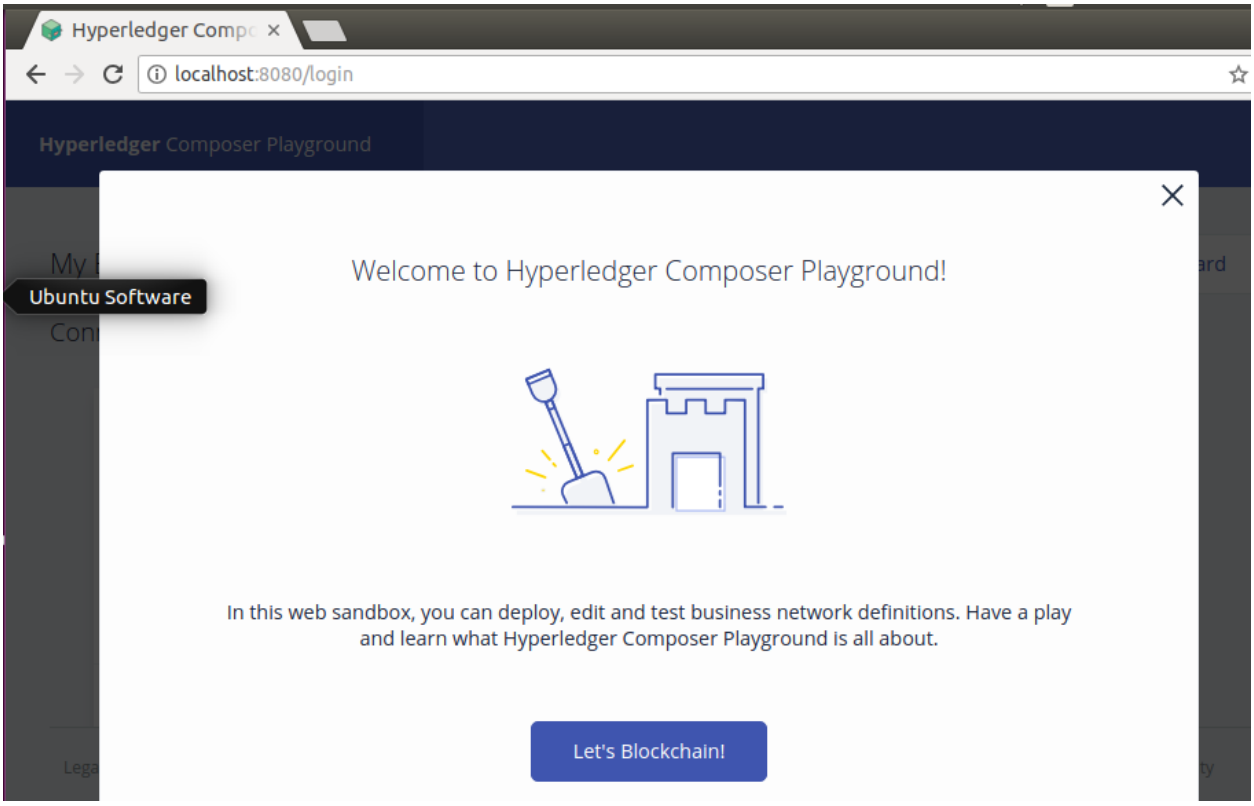


Figure 292 - Your browser should open to the Composer Playground welcome page.

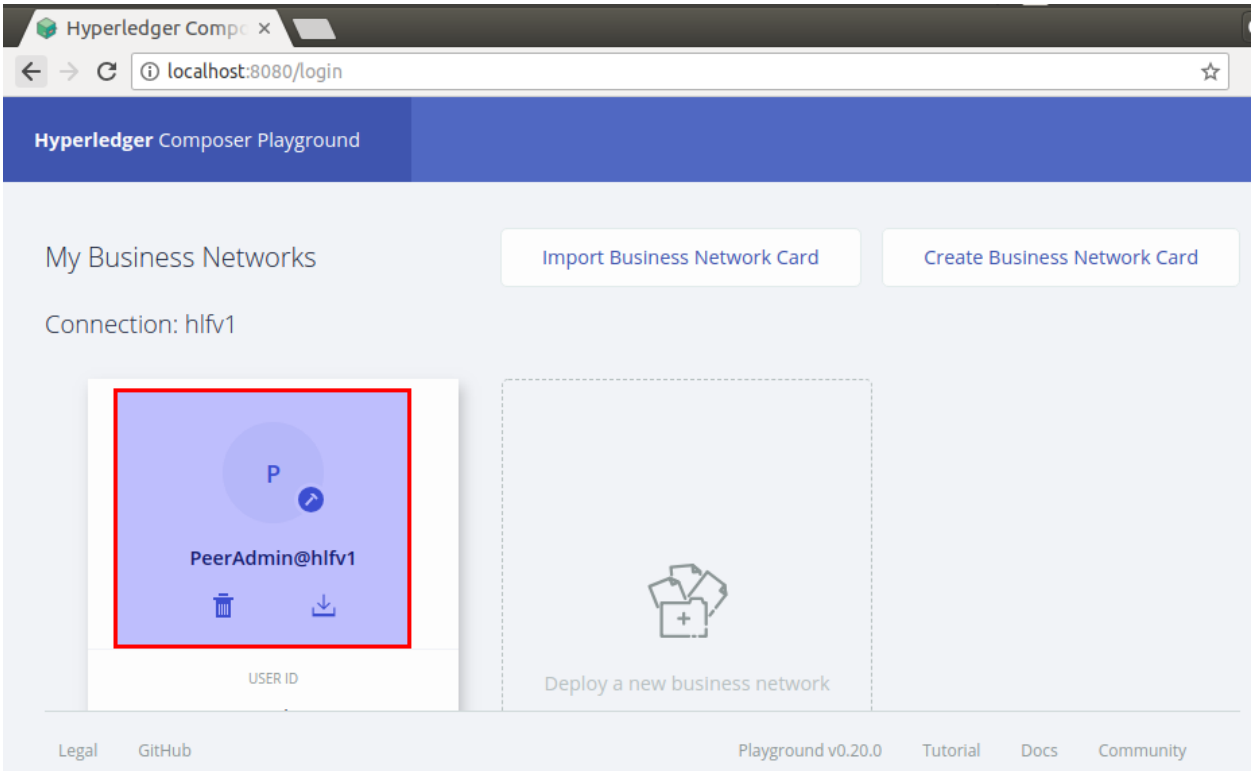


Figure 293 - You should see the PeerAdmin card you previously created.