

MANUAL

BUILDING BLOCKCHAIN APPLICATIONS USING HYPERLEDGER FABRIC WORKSHOP

**ORGANIZED BY:
KHAWLA HASSAN
MANAR ABU TALIB
QASSIM NASIR**

Installation Guide (Ubuntu)



Skip this section and head over to the *Project objective* section as we've installed all the requirements on your machine

Installing Hyperledger Composer's Pre-requisites

Operating Systems: Ubuntu Linux 14.04 / 16.04 LTS (both 64-bit) ✓

Docker Engine: Version 17.03 or higher

Docker-Compose: Version 1.8 or higher

Node: 8.9 or higher (note version 9 and higher is not supported)

npm: v5.x

Python: 2.7.x

We can download the pre-requisites listed above (except for the OS, of course) by opening our terminal and executing the following set of commands:

```
# download the prereqs-ubuntu script from the composer repository
curl -O https://hyperledger.github.io/composer/latest/prereqs-ubuntu.sh

# grant the user 'u' execute permission 'x'
chmod u+x prereqs-ubuntu.sh

# run the script
./prereqs-ubuntu.sh
```

Close the terminal window in order for these changes to take effect.

Installing the development environment



You should not use **su or **sudo** for the following npm commands**

Step 1: Install CLI tools

```
# essential CLI tools
npm install -g composer-cli@0.20

# utility for running a REST Server on your machine to expose your
# business networks as RESTful APIs
npm install -g composer-rest-server@0.20

# useful utility for generating application assets
npm install -g generator-hyperledger-composer@0.20

# Yeoman is a tool for generating applications, which utilises
# generator-hyperledger-composer
npm install -g yo
```

Step 2: Install Playground

```
# Browser app for simple editing and testing business networks
npm install -g composer-playground@0.19
```

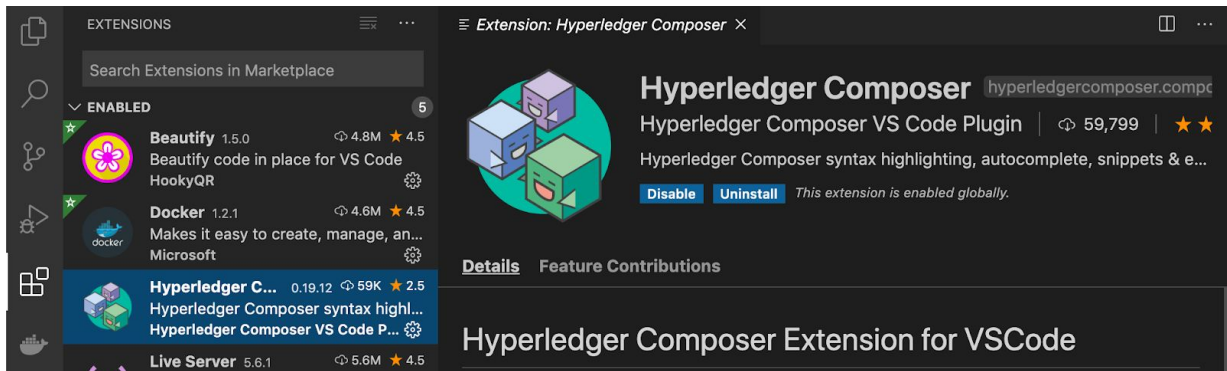
Step 3: Set up your IDE (VSCode)

VSCode is recommended here as it has a Composer Extension available which helps in supporting the development workflow.

1. Install VSCode from this URL: <https://code.visualstudio.com/download>
2. Navigate from your terminal to the directory where the executable code had been saved and run the following command:

```
sudo dpkg -i <file>.deb
```

3. Open VSCode, go to Extensions, then search for and install the **Hyperledger Composer** extension from the Marketplace.



Step 4: Install Insomnia

You can install Insomnia from this URL: <https://insomnia.rest/download/> which we'll be using later to test our REST API.

Step 5: Install Hyperledger Fabric

```
# create a fabric-dev-servers directory
mkdir ~/fabric-dev-servers

# navigate to the newly created directory
cd ~/fabric-dev-servers

# get the .tar.gz file that contains the tools to install Fabric
curl -O https://raw.githubusercontent.com/hyperledger/composer-tools/master/packages/fabric-dev-servers/fabric-dev-servers.tar.gz

# extract all files in the archive
tar -xvf fabric-dev-servers.tar.gz
```



Run **ls** to view the scripts downloaded in your current directory after executing the above commands.

You can start and stop your runtime by using the **startFabric** and **stopFabric** scripts, respectively.

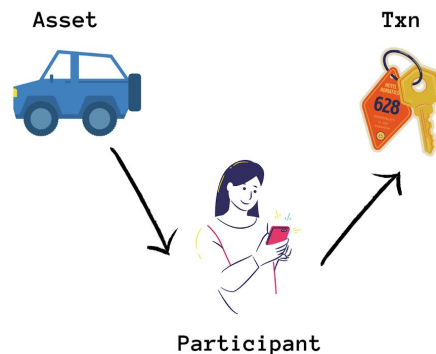
Use the scripts you just downloaded and extracted to download a local Hyperledger Fabric **v1.2** runtime:

```
cd ~/fabric-dev-servers  
  
export FABRIC_VERSION=hlfv12  
  
./downloadFabric.sh
```

We're ready to start creating blockchain solutions!

Project objective

We're going to set up a network for a car rental company. The rental company is going to create and host the fabric network. Within the network, participants can act as owners of the cars, or renters that submit transactions to rent a car of their choice and return it once they're done.



Let's define the main components of our network and the attributes associated with each.

Asset

Asset	Asset attributes
Car	vin, brand, model, year, ownerID, renterID, carStatus

carStatus property can take two values:

- Available
- Not Available

Participant

Participant	Participant attributes
Member	emiratesID, name, phoneNum, role

role property can take two values:

- Owner
- Renter

Transactions

We're going to define two transactions on our network,

1. Rent
2. Return

Setting up the network in Playground

Hyperledger Composer offers this cool UI called Playground that we can use to define and test our network.

However, let us first start fabric by running the command:

```
./startFabric.sh
```

Launching the fabric environment creates a simple network that is made up of a single organization called **Org1**. The organization uses the domain name **org1.example.com**. Additionally, the Membership Services Provider (MSP) for this organization is called **Org1MSP**.

In this tutorial, We'll deploy a blockchain network that only the organization **Org1** – car rental company – can interact with.

The network is made of several components:

- A single peer node for Org1, named **peer0.org1.example.com**.
 - The request port is 7051.
 - The event hub port is 7053.
- A single Certificate Authority (CA) for Org1, named **ca.org1.example.com**.
 - The CA port is 7054.
- A single orderer node, named **orderer.example.com**.
 - The orderer port is 7050.

These components are running inside Docker containers which we can list by running the command:

```
docker ps
```

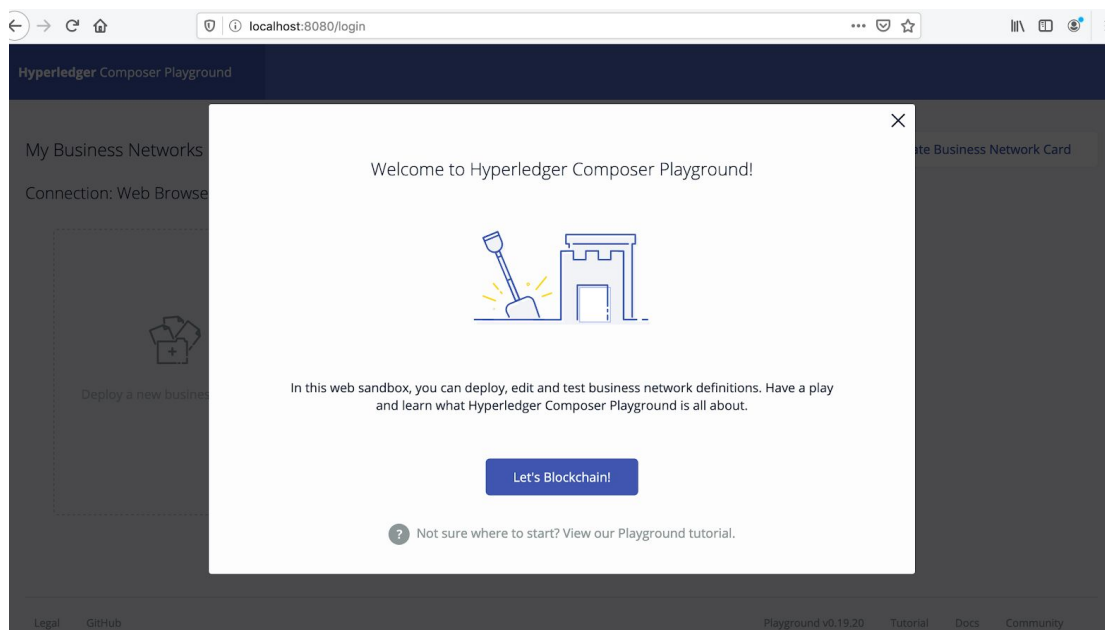
```
Khawlas-MacBook-Pro:fabirc-dev-servers khawlaalaa$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  NAMES                  CREATED
STATUS        PORTS
6da143752e88   hyperledger/fabric-peer:1.2.1      "peer node start"      peer0.org1.example.com 43 seconds ago
Up 41 seconds  0.0.0.0:7051->7051/tcp, 0.0.0.0:7053->7053/tcp
a07ffc9231f9   hyperledger/fabric-orderer:1.2.1   "orderer"              orderer.example.com    48 seconds ago
Up 43 seconds  0.0.0.0:7050->7050/tcp
1c6804889fb2   hyperledger/fabric-couchdb:0.4.10  "tini -- /docker-ent... couchdb                 48 seconds ago
Up 43 seconds  4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp
3d9ab6be1c19   hyperledger/fabric-ca:1.2.1        "sh -c 'fabric-ca-se... ca.org1.example.com    48 seconds ago
Up 43 seconds  0.0.0.0:7054->7054/tcp
```

Also, notice that we have a couchDB container running alongside the Peer which represents our state database.

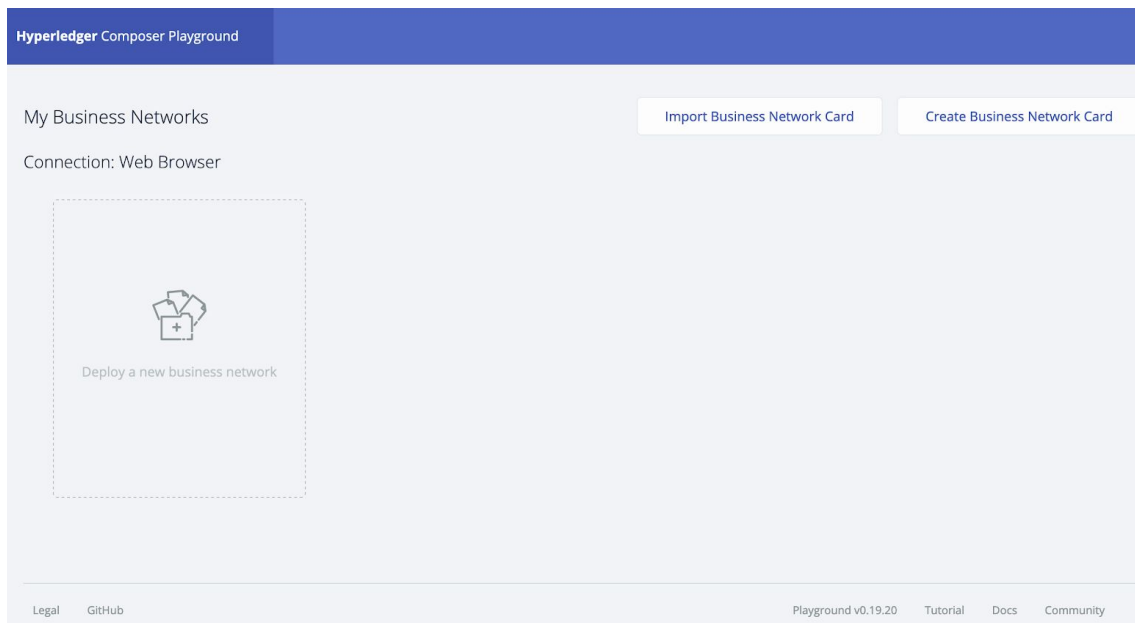
Now, let us start the Hyperledger Composer Playground:

```
composer-playground
```

This will fire up playground on your localhost at the following address:
<http://localhost:8080/login>



Click on the **Let's Blockchain** button and **Deploy a new business network**.



Let's add information about our network;

Deploy New Business Network

1. BASIC INFORMATION




Give your new Business Network a name:	car-rental-network
Describe what your Business Network will be used for:	Provide a trustworthy platform for renting cars
Give the network admin card that will be created a name	admin@car-rental-network

Because we want to build our network from scratch, we'll select the **empty-business-network** template;

2. MODEL NETWORK STARTER TEMPLATE

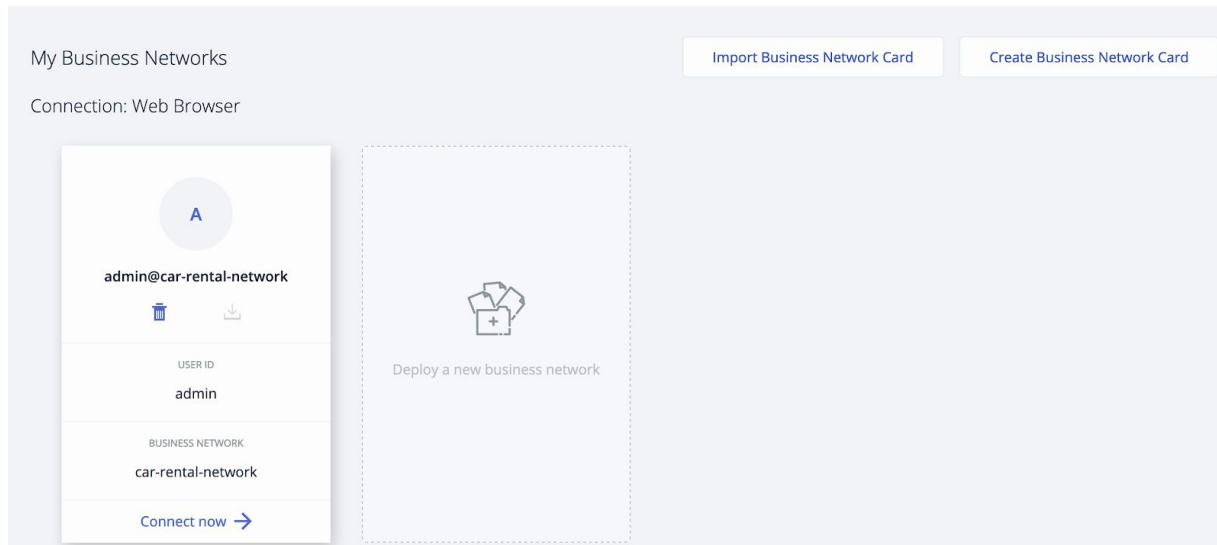
Choose a Business Network Definition to start with:

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

 basic-sample-network	 empty-business-network	 Drop here to upload or browse
---	---	--

Once you're done, Click **Deploy**.

Now that we've created and deployed our network, we should see a new business network card called **admin** for our business network **car-rental-network** in our wallet. The wallet contains business network cards to connect to multiple deployed business networks.



Let's start CODING!

Defining the business model

The key concept for Hyperledger Composer is the **Business Network Definition (BND)**. It defines the data model, transaction logic, access control rules, events, and queries for your blockchain solution. BND is then packaged into a BNA – Business Network Archive – to be deployed on fabric.

Model file (.cto)

Hyperledger Composer includes an object-oriented modeling language that is used to define the data model – a CTO file – for a BND. The CTO file includes the definitions for our resources.

Resources in Hyperledger Composer include:

- Assets, Participants, Transactions, and Events.
- Enumerated Types.
- Concepts.

Let's develop an understanding of the Composer Modeling Language's syntax.

Resource definition has the following properties:

1. A namespace defined by the namespace of its parent file. The namespace of a **.cto** file implicitly applies to all resources created in it.
2. A name, and an identifying field. If the resource is an asset or participant, the name is followed by the identifying field, *if the resource is an event or transaction, the identifying field is set automatically.*

Here's an example:

```
/**
 * A vehicle asset.
 */
asset Vehicle identified by vin {
  o String vin
}
```

3. A set of named properties. The properties must be named, and the primitive data type defined.

Primitive types in Composer Modeling Language are

- **String:** a UTF8 encoded String.
- **Double:** a double precision 64 bit numeric value.
- **Integer:** a 32 bit signed whole number.
- **Long:** a 64 bit signed whole number.
- **DateTime:** an ISO-8601 compatible time instance, with optional time zone and UTZ offset.
- **Boolean:** a Boolean value, either true or false.

Adding a set of properties to our vehicle example:

```
/**
 * A vehicle asset.
 */
asset Vehicle identified by vin {
  o String vin
  o String make
  o Integer year
}
```

4. A set of relationships to other Composer types that are not owned by the resource but that may be referenced from the resource. Relationships are unidirectional and are indicated by a rightwards arrow (**-->**).

```

/**
 * A Field asset. A Field is related to a list of animals
 */
asset Field identified by fieldId {
  o String fieldId
  o String name
  --> Animal[] animals
}

```

Notice how the field asset is referencing an array of animals.

- Enumerated types are used to specify a type that may have 1 or N possible values.

```

/**
 * An enumerated type
 */
enum ProductType {
  o DAIRY
  o BEEF
  o VEGETABLES
}

```

When another resource is created, for example, a participant, a property of that resource can be defined in terms of an enumerated type.

```

participant Farmer identified by farmerId {
  o String farmerId
  o ProductType primaryProduct
}

```

Rental network's CTO file

```

namespace org.car.rental

// Car asset
enum Status {
  o AVAILABLE
  o NOT_AVAILABLE
}

asset Car identified by vin {
  o String vin
}

```

```

    o String brand
    o String model
    o String year
    --> Member owner
    o String renterID default="NONE"
    o Status carStatus
}

// Participants
enum Role {
    o OWNER
    o RENTER
}

participant Member identified by emiratesID {
    o String emiratesID
    o String name
    o String phoneNum
    o Role role
}

// Rent transaction
transaction rentCar {
    --> Member renter
    --> Car car
    o Integer durationInDays
}

// Return transaction
transaction returnCar {
    --> Member renter
    --> Car car
}

```

Script file (.js)

Transaction processor functions implement the transactions defined in the model file. It gets automatically invoked by the runtime when transactions are submitted.

The script file consists of two main parts, both are required for the transaction processor function to work:

1. Decorators and metadata:

```
/**
 * A transaction processor function description
 * @param {org.example.basic.SampleTransaction} parameter-name A human
description of the parameter
 * @transaction
 */
```

The first line of comments above a transaction processor function contains a human readable description of what the transaction processor function does.

The second line must include the `@param` tag to indicate the parameter definition. The `@param` tag is followed by the resource name of the transaction which triggers the transaction processor function, this takes the format of the *namespace of the business network*, followed by the *transaction name*. After the resource name, is the parameter name which will reference the resource, this parameter **must** be supplied to the JavaScript function as an argument.

The third line must contain the `@transaction` tag, this tag identifies the code as a transaction processor function and is required.

2. JavaScript function:

```
function transactionProcessor(parameter-name) {
  //Do some things.
}
```

Each resource has an associated registry that we can perform the following operations on,

Name	Returns	Description
get	Promise	Get a specific resource in the registry
getAll	Promise	Get all of the resources in the registry
update	Promise	Updates a resource in the registry
updateAll	Promise	Updates a list of resources in the registry
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
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
remove	Promise	Remove an asset with a given type and id from the registry
removeAll	Promise	Removes a list of resources from the registry

Rental network's script file

We'll create two script files for each transaction – rent, and return – in our project.

rentCar.js

```
/**
 * A network participant can rent a car from its owner
 * @param {org.car.rental.rentCar} rentTxn sets the status of the
 * requested car to NOT_AVAILABLE
 * @transaction
 */
async function rentCar(rentTxn) {

  // get the car registry
  const carRegistry = await getAssetRegistry('org.car.rental.Car');

  // get the car from the car registry using carVIN
  const car = await carRegistry.get(rentTxn.car.vin);

  if(car.carStatus === "AVAILABLE"){

    // change the car status to NOT_AVAILABLE
    car.carStatus = "NOT_AVAILABLE";
  }
}
```

```

    // set the renterID property
    car.renterID = rentTxn.renter.emiratesID;
    // update the car registry
    await carRegistry.update(car);
  }
  else // car had been rented by someone else who hasn't returned it yet
    throw new Error("CAR IS NOT AVAILABLE FOR RENT");
}

```

returnCar.js

```

/**
 * A network participant can return a car to its owner
 * @param {org.car.rental.returnCar} returnTxn sets the status of the
 * requested car to AVAILABLE
 * @transaction
 */
async function returnCar(returnTxn) {
  // get the car registry
  const carRegistry = await getAssetRegistry('org.car.rental.Car');

  // get the car from the car registry using carVIN
  const car = await carRegistry.get(returnTxn.car.vin);

  // change the car status to AVAILABLE
  car.carStatus = "AVAILABLE";

  // set the renterID property
  car.renterID = "NONE";

  // update the car registry
  await carRegistry.update(car);
}

```

Access Control Rules (.acl)

Hyperledger Composer includes an access control language (ACL) that provides declarative access control over the elements of the domain model. By defining ACL rules you can determine which users/roles are permitted to create, read, update or delete elements in a business network's domain model.

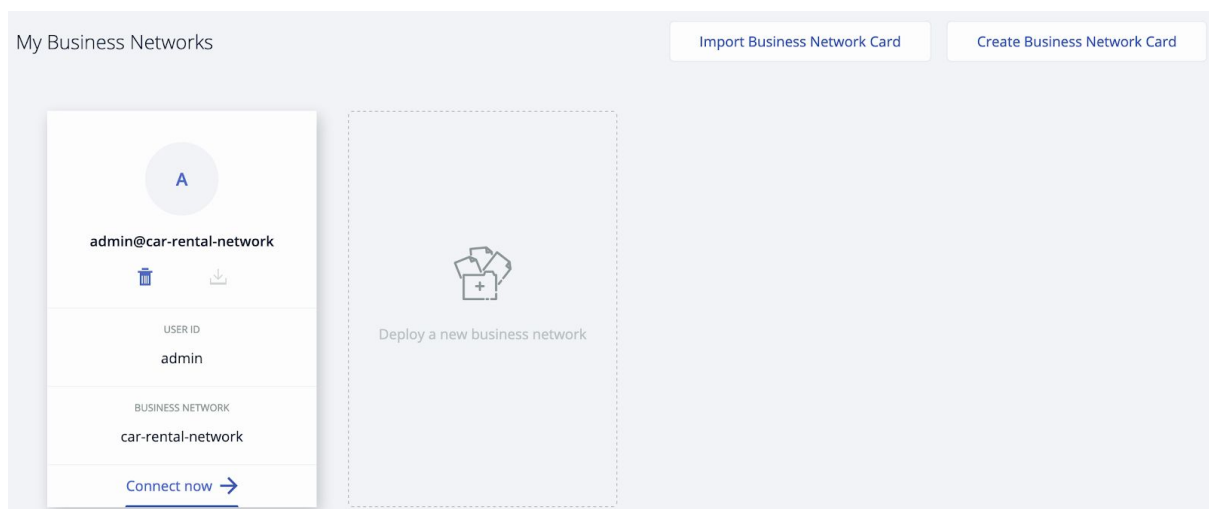
Our network is simple, so the default access control file doesn't need editing. The basic file gives the current participant networkAdmin full access to business network and system-level operations.


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

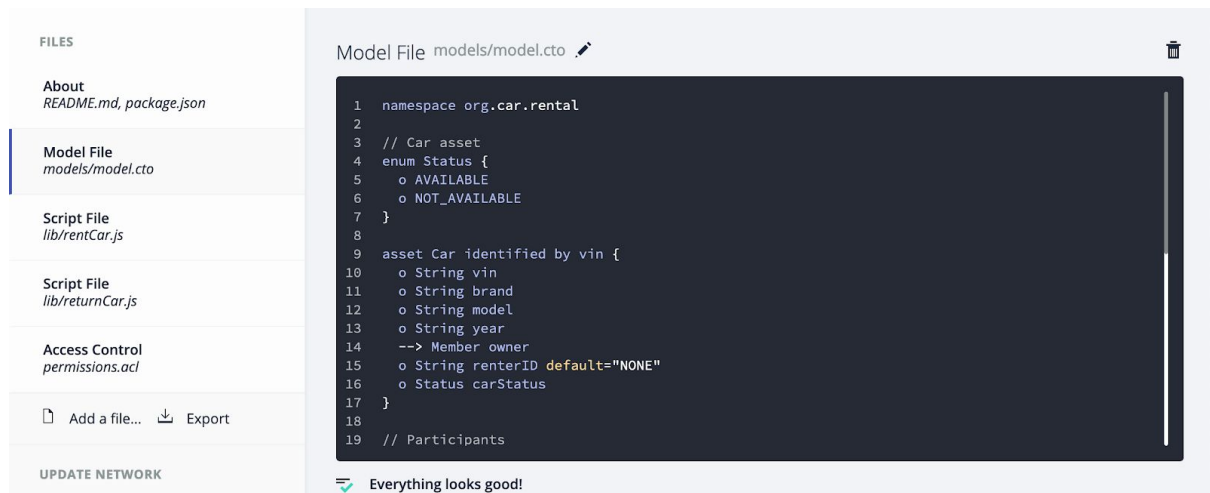
rule NetworkAdminSystem {
  description: "Grant business network administrators full access to
system resources"
  participant: "org.hyperledger.composer.system.NetworkAdmin"
  operation: ALL
  resource: "org.hyperledger.composer.system.*"
  action: ALLOW
}
```

Testing the network in Playground

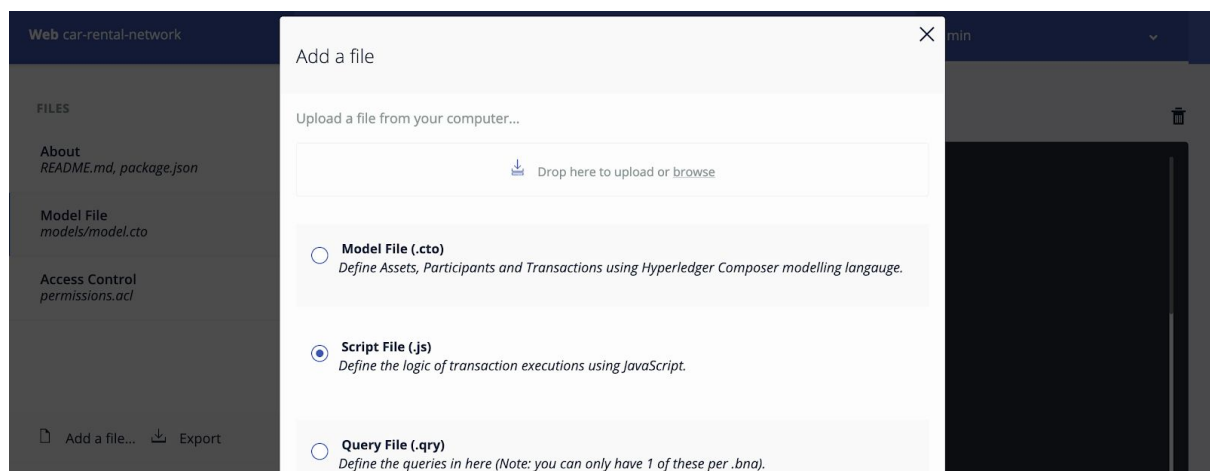
Go to your wallet and click **Connect now** to connect to the network.



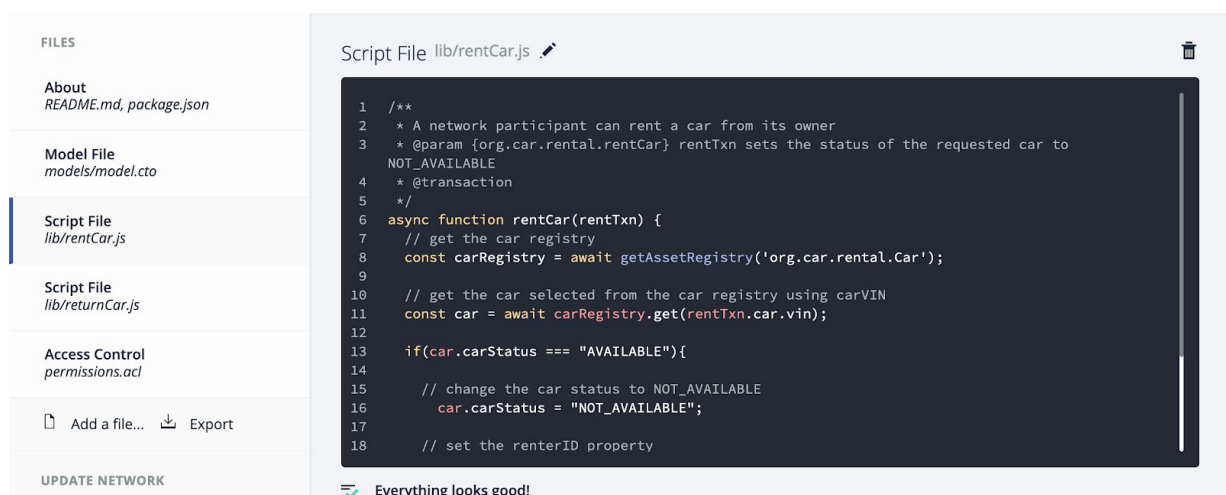
Head over to the Model file then copy and paste the definition we created previously.



We'll add our transaction processor functions by first creating a script file, so click on **Add a file** next to the **Export** button and choose **Script File (.js)**.



Edit the name of the file to be **rentCar.js** then copy and paste the script file we defined above.



Repeat the same process for the second function while giving it a name of **returnCar.js**, this time.

Finally, click on **Deploy changes** so we can start testing our network!

Head over now to the **Test** tab and let's populate our network with assets and participants. Click on **+ Create New Participant**, fill in the details of the participant, and hit **Create New**.

Ahmed is the OWNER of a car and he's making it available for others to rent.

Web car-rental-network

PARTICIPANTS

Member

ASSETS

Car

TRANSACTIONS

All Transactions

Submit Transaction

Create New Participant

In registry: org.car.rental.Member

JSON Data Preview

```
1 {
2   "$class": "org.car.rental.Member",
3   "emiratesID": "7732",
4   "name": "Ahmed salim",
5   "phoneNum": "0501234567",
6   "role": "OWNER"
7 }
```

☐ Optional Properties

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

Cancel Create New

Tutorial Docs Community

Sara joined the network as a RENTER.

Web car-rental-network

PARTICIPANTS

Member

ASSETS

Car

TRANSACTIONS

All Transactions

Submit Transaction

Create New Participant

In registry: org.car.rental.Member

JSON Data Preview

```
1 {
2   "$class": "org.car.rental.Member",
3   "emiratesID": "9239",
4   "name": "Sara bader",
5   "phoneNum": "0501234567",
6   "role": "RENTER"
7 }
```

☐ Optional Properties

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

Cancel Create New

Tutorial Docs Community

Ahmed has a Nissan Sunny 2016 car so let's add that as an asset,

The screenshot shows the 'Create New Asset' dialog box. On the left is a sidebar with navigation links: 'PARTICIPANTS' (with 'Member' selected), 'ASSETS' (with 'Car' selected), and 'TRANSACTIONS' (with 'All Transactions' selected). Below these is a 'Submit Transaction' button. The main area of the dialog is titled 'Create New Asset' and shows 'In registry: org.car.rental.Car'. Below this is a 'JSON Data Preview' section with a dark background and white text showing a JSON object:

```
1 {
2   "$class": "org.car.rental.Car",
3   "vin": "1181",
4   "brand": "Nissan",
5   "model": "Sunny",
6   "year": "2016",
7   "owner": "resource:org.car.rental.Member#2422",
8   "renterID": "NONE",
9   "carStatus": "AVAILABLE"
10 }
```

 Below the JSON preview is an unchecked checkbox labeled 'Optional Properties'. At the bottom of the dialog are three buttons: 'Generate Random Data', 'Cancel', and 'Create New'. On the right side of the image, a partial view of the application interface shows a '+ Create New Asset' button and some text.

Notice how the hashtag in the "owner" property is referencing Ahmed's emiratesID.

Sara would like to rent Ahmed's car for 3 days so let's submit a **rentCar** transaction.

The screenshot shows the 'Submit Transaction' dialog box. The sidebar is the same as in the previous screenshot. The main area is titled 'Submit Transaction' and shows 'Transaction Type' as 'rentCar' with a dropdown arrow. Below this is a 'JSON Data Preview' section with a dark background and white text showing a JSON object:

```
1 {
2   "$class": "org.car.rental.rentCar",
3   "renter": "resource:org.car.rental.Member#9239",
4   "car": "resource:org.car.rental.Car#1181",
5   "durationInDays": 3
6 }
```

 Below the JSON preview is an unchecked checkbox labeled 'Optional Properties'. At the bottom of the dialog are three buttons: 'Generate Random Data', 'Cancel', and 'Create New'. On the right side of the image, a partial view of the application interface shows a '+ Create New Asset' button and some text.

If we check the asset registry, we'll see that the **renterID** and **carStatus** have both changed their values accordingly.

Asset registry for org.car.rental.Car

+ Create New Asset

ID	Data
1181	<div><pre>{ "\$class": "org.car.rental.Car", "vin": "1181", "brand": "Nissan", "model": "Sunny", "year": "2016", "owner": "resource:org.car.rental.Member#2422", "renterID": "9239", "carStatus": "NOT_AVAILABLE" }</pre><div>Collapse</div></div>

Try creating a new participant and submit a transaction to rent Ahmed's car. You'll receive an error indicating that the car is not available for rent as Sara hasn't returned it, yet.

Web car-rental-network

PARTICIPANTS

Member

ASSETS

Car

TRANSACTIONS

All Transactions

Submit Transaction

Transaction Type

rentCar

JSON Data Preview

```
1 {
2   "$class": "org.car.rental.rentCar",
3   "renter": "resource:org.car.rental.Member#1516",
4   "car": "resource:org.car.rental.Car#1181",
5   "durationInDays": 2
6 }
```

☐ Optional Properties

Error: CAR IS NOT AVAILABLE FOR RENT

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

Cancel

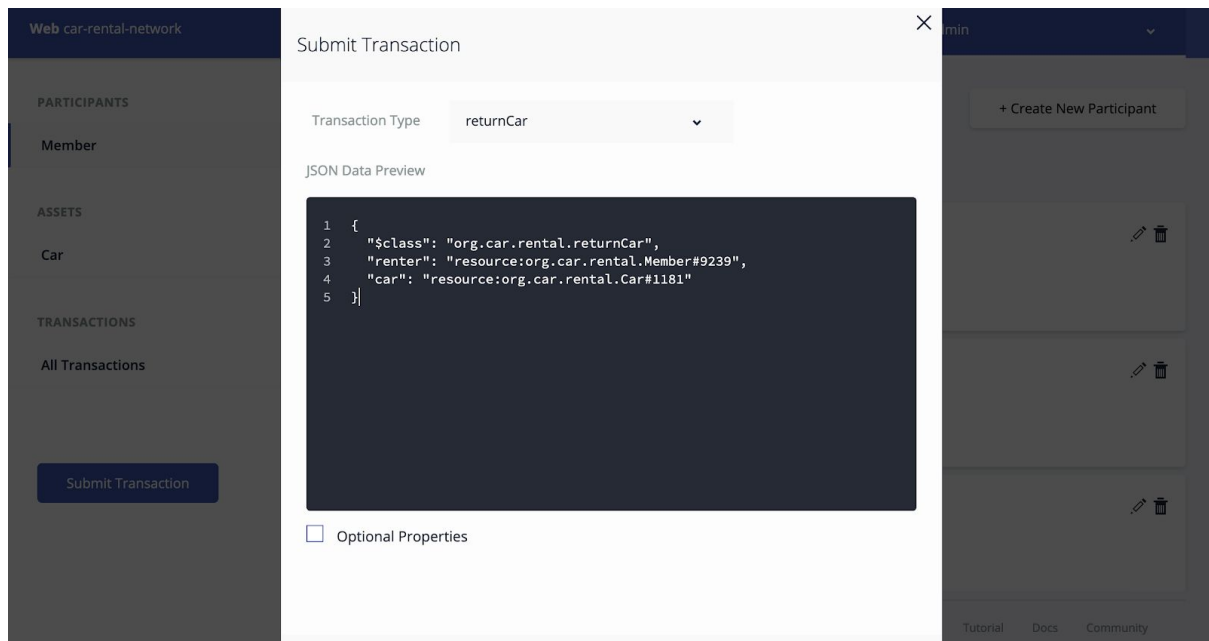
Submit

min

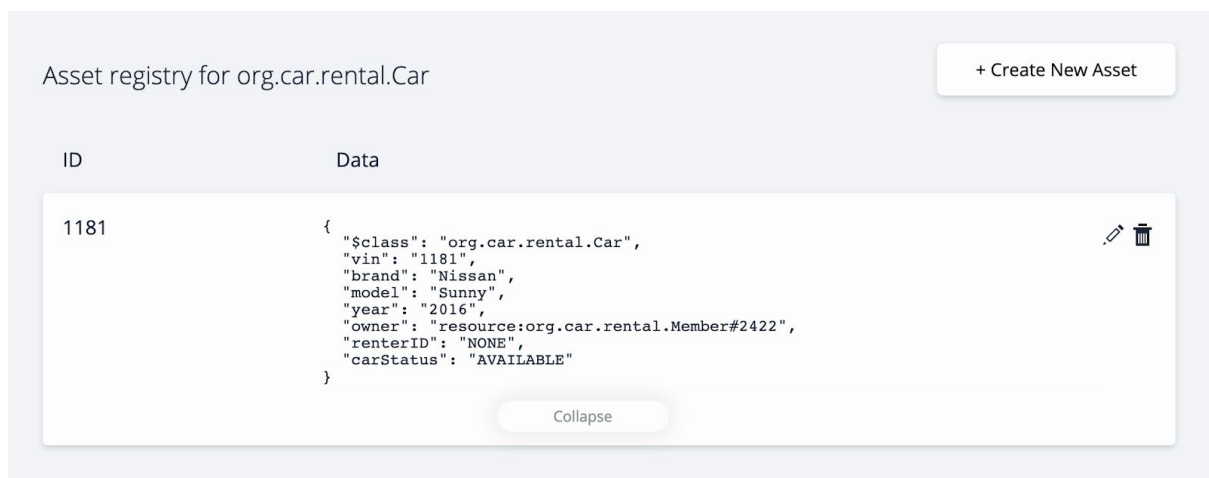
+ Create New Participant

Tutorial Docs Community

Let's return the car now to make it available for rent again by submitting a **returnCar** transaction.



If you head over to the Asset registry, you'll notice that Ahmed's car is AVAILABLE for rent.

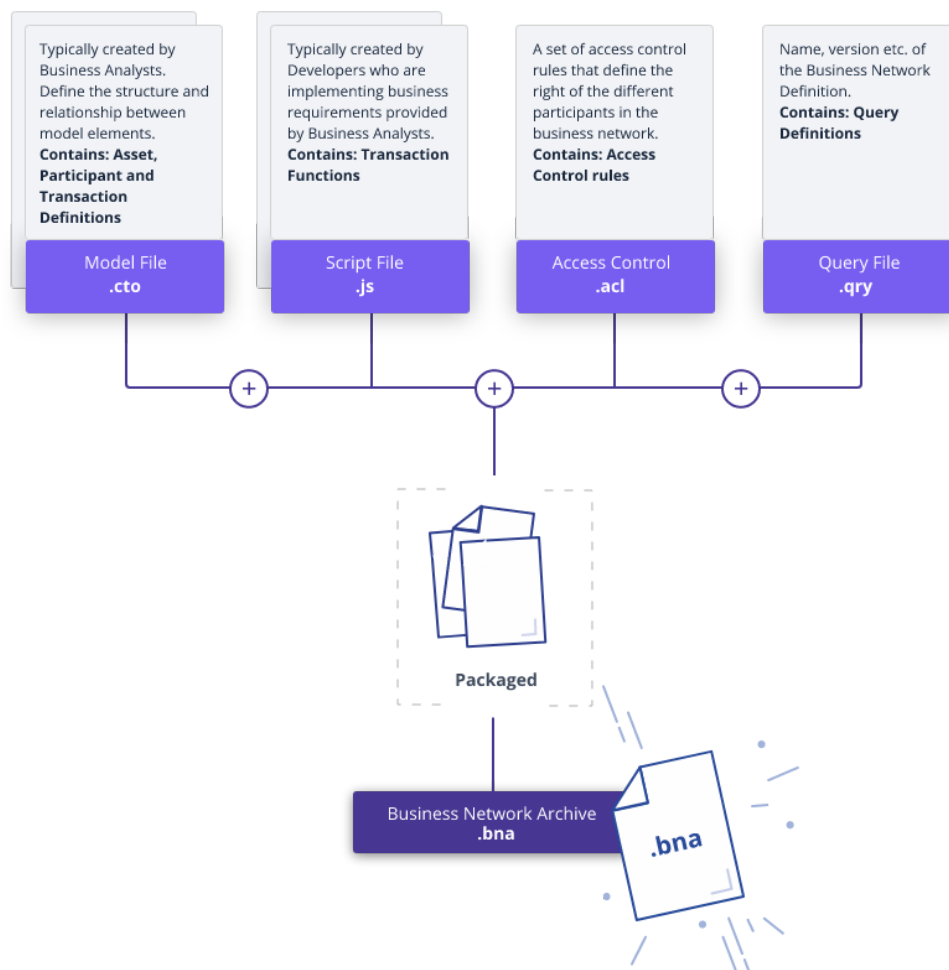


To view the full transaction history of our network, click **All Transactions** on the left. What you see is called the Historian Registry, a specialised Hyperledger Composer registry which records successful transactions, including the participants and identities that submitted them.


Okay, our network seems to be working exactly the way we wanted it to be. Let's move to the next step.

Generating a REST API

Before generating a REST API, we'll have to transfer our BND to a folder and package it into an archive using composer CLI.



The easiest way to create a skeleton network is to use Yeoman generator. Open up your terminal and execute the following command:

 Make sure you're inside the **fabric-dev-servers** folder before executing the next command

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

This command will prompt you to enter the following:

```
Business network name: car-rental-network
Description: Provide a trustworthy platform for renting cars
Author name: [insert]
Author email: [insert]
License: Apache-2.0
Namespace: org.car.rental
Do you want to generate an empty template network? No: generate a
populated sample network
```

Navigate to your newly created network folder and explore its contents:

```
cd car-rental-network
ls
```

Inside the **models** folder,

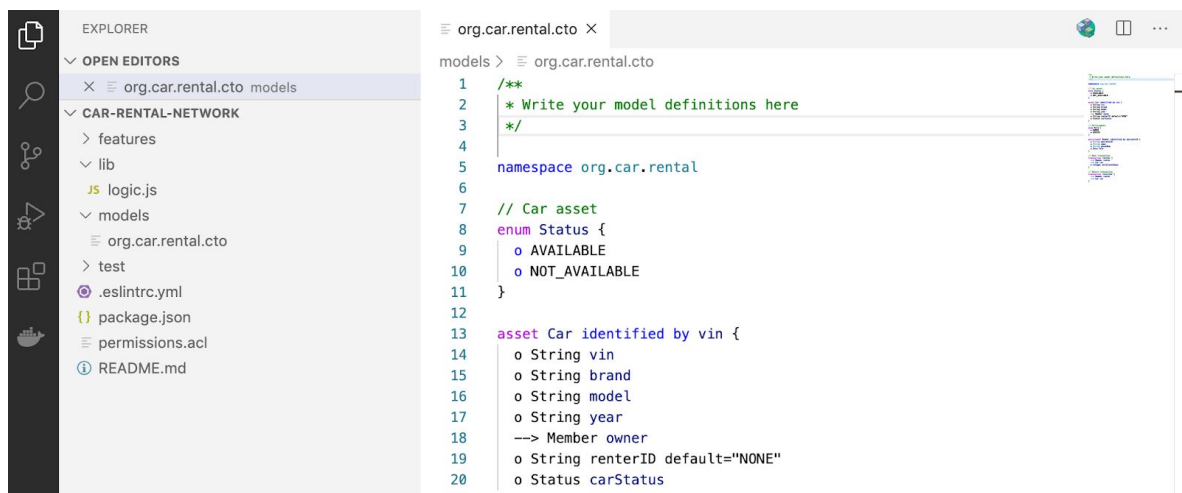
```
# to list the contents of the models folder
ls models
```

You'll find your CTO file with the name **org.car.rental.cto**. Whereas, in the **lib** folder,

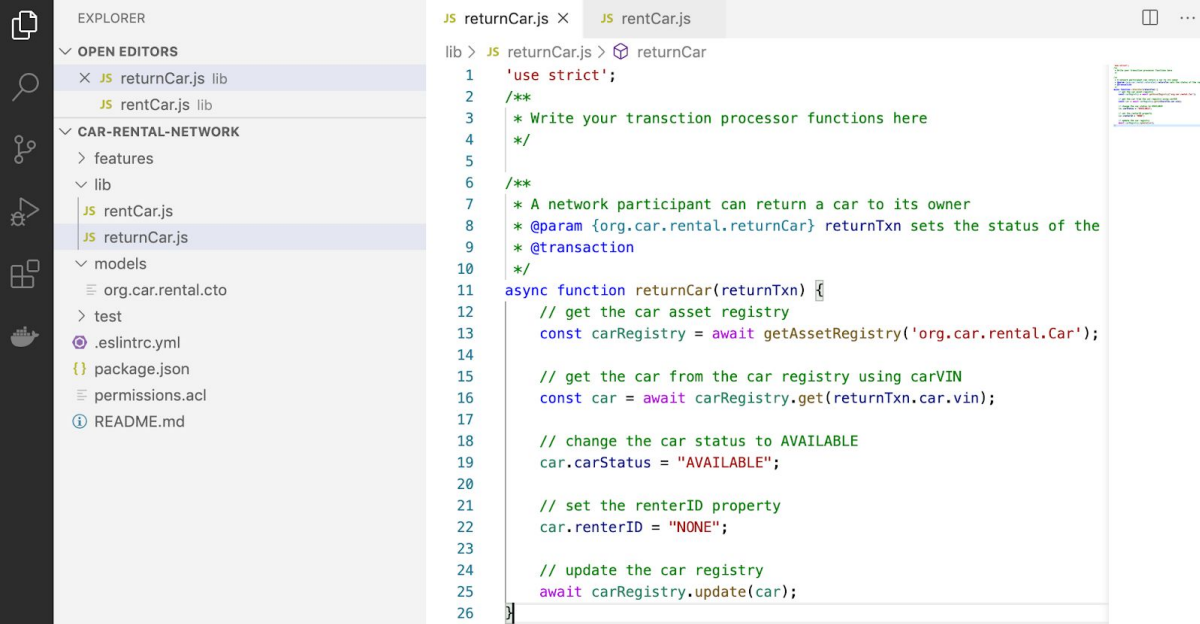
```
# to list the contents of the lib folder
ls lib
```

You'll find the transaction logic script with the name **logic.js**.

Now, open the **car-rental-network** folder in VSCode then copy and paste our BND into the appropriate files.

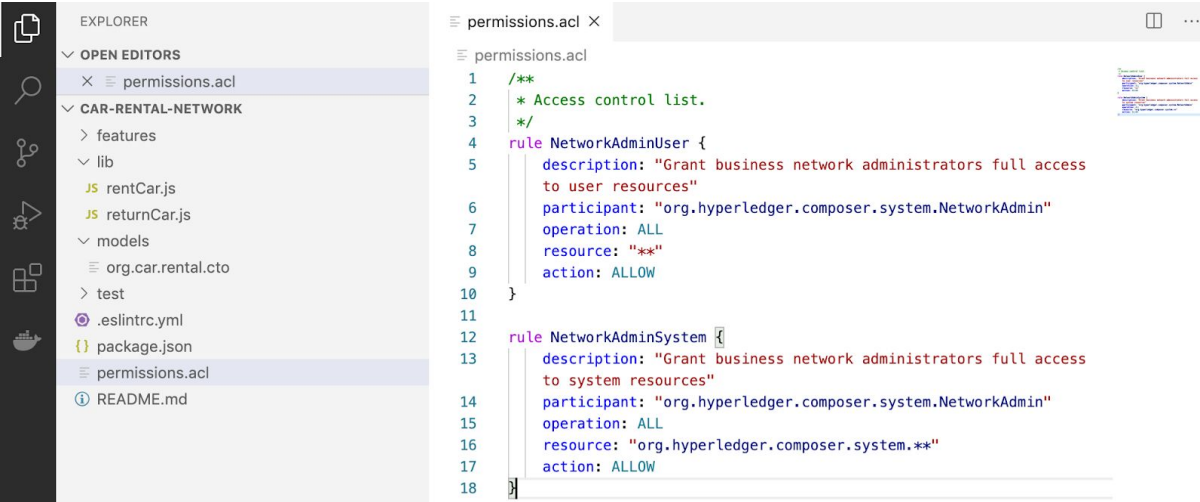


We'll rename **logic.js** to be **rentCar.js** then add an extra script file to our **lib** folder and name it **returnCar.js**.



```
lib > JS returnCar.js > returnCar
1  'use strict';
2  /**
3   * Write your transaction processor functions here
4   */
5
6  /**
7   * A network participant can return a car to its owner
8   * @param {org.car.rental.returnCar} returnTxn sets the status of the
9   * @transaction
10  */
11  async function returnCar(returnTxn) {
12    // get the car asset registry
13    const carRegistry = await getAssetRegistry('org.car.rental.Car');
14
15    // get the car from the car registry using carVIN
16    const car = await carRegistry.get(returnTxn.car.vin);
17
18    // change the car status to AVAILABLE
19    car.carStatus = "AVAILABLE";
20
21    // set the renterID property
22    car.renterID = "NONE";
23
24    // update the car registry
25    await carRegistry.update(car);
26  }
```

Copy and paste the access control rules into **permissions.acl**.



```
permissions.acl
1  /**
2   * Access control list.
3   */
4
5  rule NetworkAdminUser {
6    description: "Grant business network administrators full access
7    to user resources"
8    participant: "org.hyperledger.composer.system.NetworkAdmin"
9    operation: ALL
10   resource: "*"
11   action: ALLOW
12 }
13
14 rule NetworkAdminSystem {
15   description: "Grant business network administrators full access
16   to system resources"
17   participant: "org.hyperledger.composer.system.NetworkAdmin"
18   operation: ALL
19   resource: "org.hyperledger.composer.system.*"
20   action: ALLOW
21 }
```

Save your work and head back to the terminal.

To package our BND into a deployable network archive (.bna) file, we'll execute the following command:



Make sure you're inside the **car-rental-network** directory

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



You can run `composer archive create --help` to understand what the options **-t** and **-n** stand for

This will create a BNA file called **car-rental-network@0.0.1.bna** in the **car-rental-network** directory.

```
Khawlas-MacBook-Pro:car-rental-network khawlaalaa$ composer archive create -t dir -n .
Creating Business Network Archive

Looking for package.json of Business Network Definition
Input directory: /Users/khawlaalaa/fabric-dev-servers/car-rental-network

Found:
  Description: Provide a trustworthy platform for renting cars
  Name: car-rental-network
  Identifier: car-rental-network@0.0.1

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

Command succeeded
```

After creating the BNA file, the network can be deployed to the instance of Hyperledger Fabric. This process requires a **PeerAdmin** card as it has administrative privileges to install and deploy the network.

Let's create one by executing the **createPeerAdminCard** script in the **fabric-dev-servers** folder:

```
cd ..

./createPeerAdminCard.sh
```

To install the network to fabric, run the following command from the **car-rental-network** directory:

```
cd car-rental-network

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

The network card used in the command must be a peer admin card in order to install the network to the blockchain peers.

After the network has been installed to the peers, the network can be started. For best practice, a new identity should be created to administer the business network after deployment. This identity is referred to as a network admin.

To start the network, and give a filename for the network admin card to be created, run the following command:

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

To import the network administrator identity as a usable network card, run the following command:

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

The composer card import command requires the filename specified in **composer network start** to create a card.

To check that the business network has been deployed successfully, run the following command to ping the network:

```
composer network ping --card admin@car-rental-network
```

We're finally ready to generate a REST API based on our network that can be used for developing applications.

Let's run the following command from our **car-rental-network** directory:

```
composer-rest-server
```

It will prompt you to enter the following:

```
? Enter the name of the business network card to use: admin@car-rental-network
? Specify if you want namespaces in the generated REST API: always use namespaces
? Specify if you want to use an API key to secure the REST API: No
? Specify if you want to enable authentication for the REST API using Passport: No
? Specify if you want to enable the explorer test interface: Yes
? Specify a key if you want to enable dynamic logging: [Press Enter]
? Specify if you want to enable event publication over WebSockets: Yes
? Specify if you want to enable TLS security for the REST API: No
```

```
[Khawlas-MacBook-Pro:car-rental-network khawlaalaa$ composer-rest-server ]
[? Enter the name of the business network card to use: admin@car-rental-network ]
[? Specify if you want namespaces in the generated REST API: always use namespaces ]
[? Specify if you want to use an API key to secure the REST API: No ]
[? Specify if you want to enable authentication for the REST API using Passport: No ]
[? Specify if you want to enable the explorer test interface: Yes ]
[? Specify a key if you want to enable dynamic logging: ]
[? Specify if you want to enable event publication over WebSockets: Yes ]
[? Specify if you want to enable TLS security for the REST API: No ]

To restart the REST server using the same options, issue the following command:
  composer-rest-server -c admin@car-rental-network -n always -u true -w true

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

Navigate to your API Explorer at the address <http://localhost:3000/explorer>, you'll see a similar screen to this.

Hyperledger Composer REST server			
Admin : Rest server methods		Show/Hide	List Operations Expand Operations
org_car_rental_Car : An asset named Car		Show/Hide	List Operations Expand Operations
org_car_rental_Member : A participant named Member		Show/Hide	List Operations Expand Operations
org_car_rental_rentCar : A transaction named rentCar		Show/Hide	List Operations Expand Operations
org_car_rental_returnCar : A transaction named returnCar		Show/Hide	List Operations Expand Operations
System : General business network methods		Show/Hide	List Operations Expand Operations
[BASE URL: /api , API VERSION: 0.0.1]			

API Testing

We'll use Insomnia to test our Composer REST server and get a slight feel of the client-side experience when interacting with our network. This also allows us to ensure that everything's working as expected.

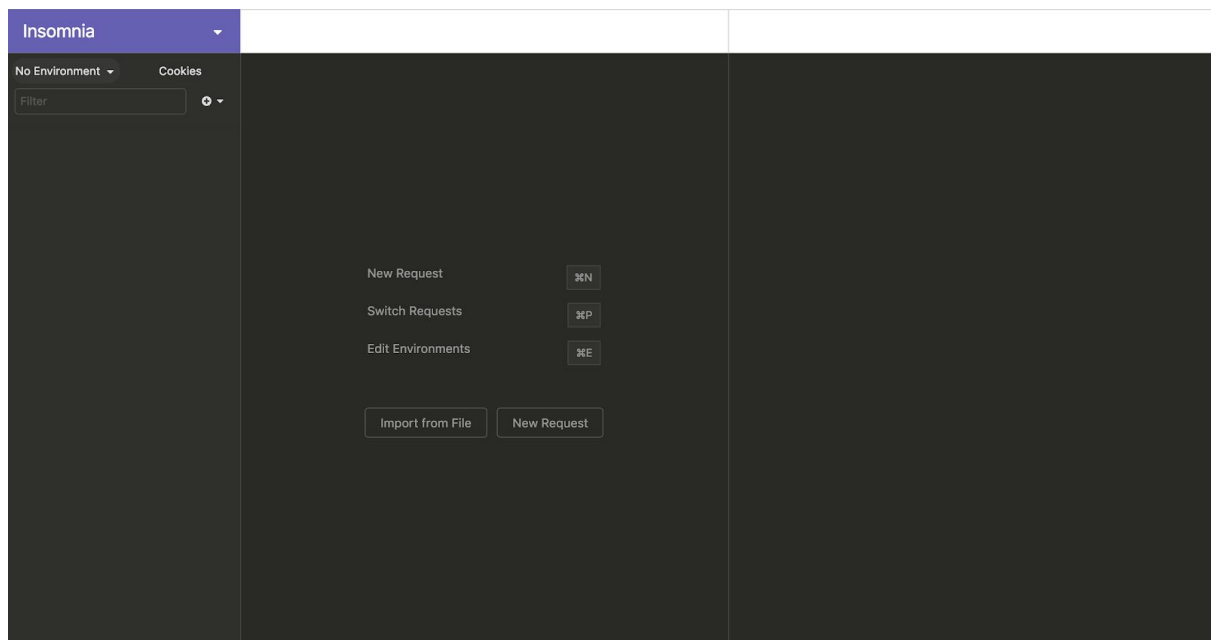
HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.

The target of an HTTP request is called a "resource". Each resource in our network is identified by a URL used throughout HTTP for identifying resources. For example, participants are identified by <http://localhost:3000/api/org.car.rental.Member> and we can perform the following methods on them:

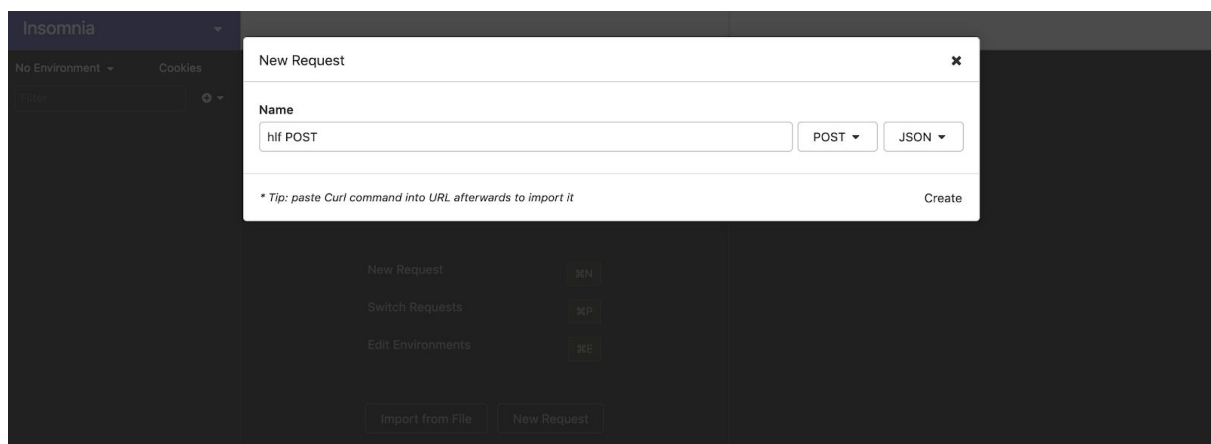
Method	Description
GET	Retrieve a representation of the specified resource
POST	Create a new instance of the model and submit to the specified resource
PUT	Replace attributes for a model instance
HEAD	Check whether a model instance exists
DELETE	Delete a model instance

Okay, let's start testing!

1. Open Insomnia, click on **New Request**, and give it a name of your choice.



2. Select the **POST** method and **JSON** as the format of our resources.



3. We'll create a new participant instance which is why we've selected the **POST** method. The **Request URL** is <http://localhost:3000/api/org.car.rental.Member>.

Enter the same details used previously for Ahmed, the Nissan car owner, from our Playground walkthrough, then click **SEND**.

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** http://localhost:3000/api/org.car.rental.Member
- Status:** 200 OK
- Time:** 3.5 s
- Size:** 114 B
- Response Type:** JSON
- Request Body (JSON):**

```
1 {
2   "emiratesID": "7732",
3   "name": "Ahmed salim",
4   "phoneNum": "0501234567",
5   "role": "OWNER"
6 }
```
- Preview (JSON):**

```
1 {
2   "$class": "org.car.rental.Member",
3   "emiratesID": "7732",
4   "name": "Ahmed salim",
5   "phoneNum": "0501234567",
6   "role": "OWNER"
7 }
```

Notice that the response code is **200**, that means we've successfully created a new instance of the participant resource.

4. Create a new car instance but, this time, change the URL to the appropriate resource.

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** http://localhost:3000/api/org.car.rental.Car
- Status:** 200 OK
- Time:** 2.53 s
- Size:** 179 B
- Response Type:** JSON
- Request Body (JSON):**

```
1 {
2   "vin": "1181",
3   "brand": "Nissan",
4   "model": "Sunny",
5   "year": "2016",
6   "owner": "resource:org.car.rental.Member#7732",
7   "renterID": "NONE",
8   "carStatus": "AVAILABLE"
9 }
```
- Preview (JSON):**

```
1 {
2   "$class": "org.car.rental.Car",
3   "vin": "1181",
4   "brand": "Nissan",
5   "model": "Sunny",
6   "year": "2016",
7   "owner": "resource:org.car.rental.Member#7732",
8   "renterID": "NONE",
9   "carStatus": "AVAILABLE"
10 }
```

5. Add Sara, a member of the network from the Playground walkthrough.

POST http://localhost:3000/api/org.car.rental.Member Send 200 OK 2.55 s 114 B Just Now

JSON Auth Query Header 1 Docs

```
1 {
2   "emiratesID": "9239",
3   "name": "Sara bader",
4   "phoneNum": "0551234567",
5   "role": "RENTER"
6 }
```

Preview Header 11 Cookie Timeline

```
1 {
2   "$class": "org.car.rental.Member",
3   "emiratesID": "9239",
4   "name": "Sara bader",
5   "phoneNum": "0551234567",
6   "role": "RENTER"
7 }
```

6. Sara wants to rent Ahmed's car for 3 days so we'll submit a transaction by using the **POST** method and specifying the URL to the **rentCar** resource.

POST http://localhost:3000/api/org.car.rental.rentCar Send 200 OK 2.54 s 225 B Just Now

JSON Auth Query Header 1 Docs

```
1 {
2   "renter": "resource:org.car.rental.Member#9239",
3   "car": "resource:org.car.rental.Car#1181",
4   "durationInDays": 3
5 }
```

Preview Header 11 Cookie Timeline

```
1 {
2   "$class": "org.car.rental.rentCar",
3   "renter": "resource:org.car.rental.Member#9239",
4   "car": "resource:org.car.rental.Car#1181",
5   "durationInDays": 3,
6   "transactionId":
7     "901f00be363684724e66913c08a50962d6e67034a263e6135786e7a91999f2b5"
8 }
```

7. Send a **GET** request targeting Ahmed's car to ensure that the transaction has taken place. You can do that by adding the car vin at the end of the request URL as follows, <http://localhost:3000/api/org.car.rental.Car/1181>.

GET http://localhost:3000/api/org.car.rental.Car/1181 Send 200 OK 191 ms 183 B A Minute Ago

Body Auth Query Header Docs

```
1 {
2   "$class": "org.car.rental.Car",
3   "vin": "1181",
4   "brand": "Nissan",
5   "model": "Sunny",
6   "year": "2016",
7   "owner": "resource:org.car.rental.Member#7732",
8   "renterID": "9239",
9   "carStatus": "NOT_AVAILABLE"
10 }
```

We can see that the **renterID** has been updated to Sara's id and the car's status indicates that it's no longer available for rent.

Conclusion

In this workshop, you've seen how to create a blockchain business network and deploy it to a Hyperledger Fabric network. You've also seen how to test your API with Insomnia to ensure that it's working the way you want it to be.

Where to go from here?

Visit the Hyperledger Fabric and Hyperledger Composer for more information, or try creating your own blockchain network!

References

[Hyperledger Fabric](#)

[Hyperledger Composer](#)

[Modeling Language | Hyperledger Composer](#)

[Hyperledger Fabric and Composer - First Practical Blockchain | Udemy](#)

[Yeoman](#)

[HTTP request methods | MDN](#)

[Insomnia](#)