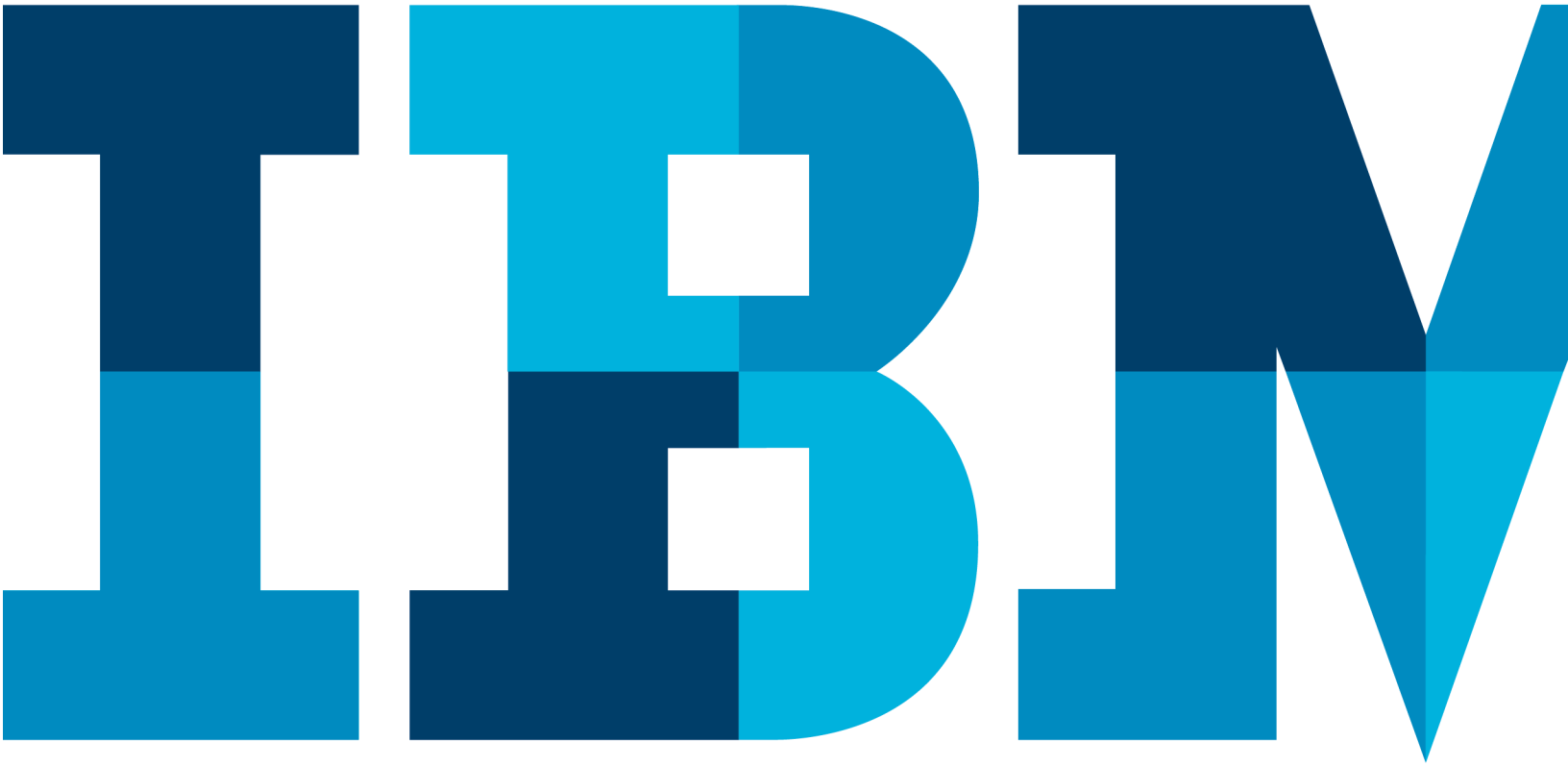
IBM Blockchain Hands-On

Vehicle Lifecycle Lab

Lab One



IBM

Contents

Introduction to the Vehicle Lifecycle Lab

This lab allows you to experience a blockchain solution from the point of view of a set of end-users, and in doing so learn about key blockchain concepts. It is not meant to be a technical introduction to blockchain, but will instead focus on the properties of the business network and value that blockchain brings.

The use-case we will work through is one that demonstrates the **lifecycle of a new car**, from the manufacturing and purchasing through to delivery and insurance. It is a good blockchain use-case because there is a defined business network and an identifiable need for trust between the participants of the network.

In this lab, you will be playing the role of the four personas who use the vehicle lifecycle system:

- *Paul* - the buyer/owner of a car
- *Mike* - an employee for the car manufacturer ("Arium")
- *Debbie* - an administrator for the regulator called the Vehicle & Drivers Authority (VDA)
- *Tommen* - an Insurer from an insurance company called Prince

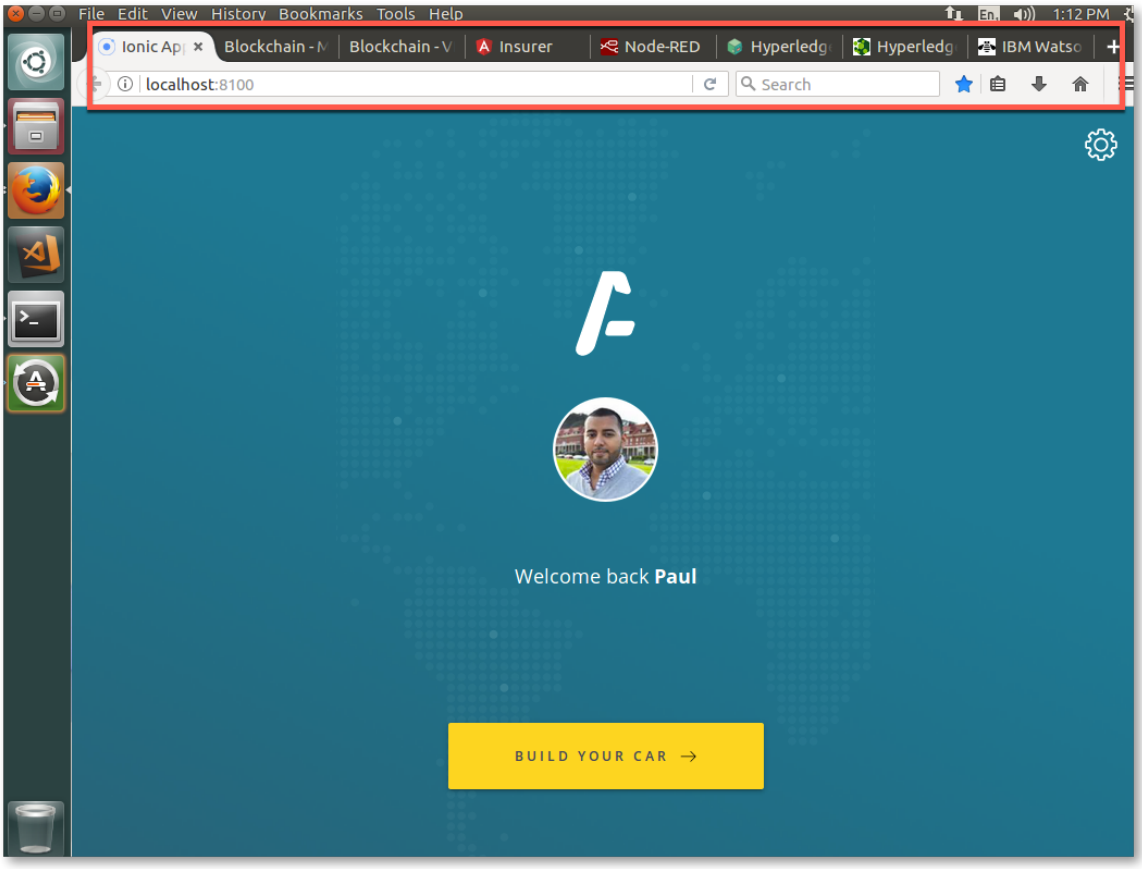
These personas together work on ordering, building, transferring ownership of a vehicle while keeping all the other parties in the network updated and building the trust between them to allow them to work together efficiently.

In this lab, each user's application will be represented by a separate tab in our web browser; of course, in a real blockchain network each user will be running on different systems in different locations, although all connecting in to the same (but distributed) blockchain network.

Start here. Instructions are always shown on numbered lines.

1. If it is not already running, start the virtual machine for the lab. The instructor will tell you how to do this if you are unsure.

2. Wait for the image to boot, and for the blockchain application and associated services to start. This happens automatically but might take several minutes. The image is ready to use when the web browser is visible and eight tabs fully loaded, as per the screenshot below. **Note: Sometimes the insurance tab is not fully loaded, so click reload to refresh the page.**



While the virtual machine is starting, let's recap a few blockchain concepts and introduce the scenario.

The most generally accepted definition of a blockchain is of a ***shared, replicated ledger***.

Ledgers have been around for hundreds of years and are records of what a business has done. They're important systems of record because they describe a business's inputs and outputs and thereby give an indication of its worth. Essentially, they are a log of transactions – a transaction being a change in state of an asset.

The problem with ledgers is that each one is owned by a single business, which means that when one business transacts with another, ledgers can get out of sync. What happens when the transaction I've recorded on my ledger doesn't tally with the transaction you've recorded on your ledger? Disputes inevitably occur which need to be resolved through a reconciliation process. This can be slow and expensive.

By having a *shared* ledger it means that all participants of the business network see the same ledger. By *replicating* it across the business network, it means that the ledger is not held in any one single place, which would otherwise make it vulnerable to outages and malicious attack.

Consider the business network that surrounds the purchase and ownership of a car. Today, each participant (for example, manufacturer, insurer or regulator) has their own ledger and the processes that allow them to interact with each other varies from company to company, and can be time consuming to complete. Connectivity between participants is typically done point-to-point using a variety of processes – some manual or slow (e.g. sending a letter), and some automated (e.g. file, REST API, B2B messaging). This plethora of processes is expensive to maintain and can be vulnerable to attack.

In this scenario we will replace these disparate ledgers with a single blockchain, and the individual business processes with a single shared business process. By doing this we will make the overall process much quicker and less prone to error.

We will experience the solution through the eyes of four key members of the business network: a private purchaser, the manufacturer, the regulator and the insurer.

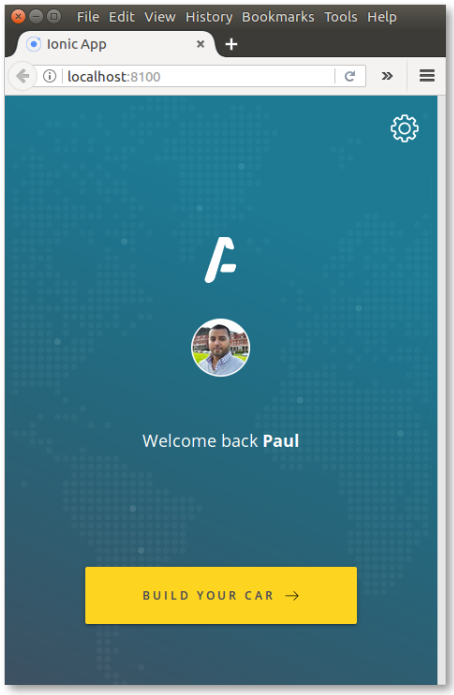
We will start by looking at the ordering process, as experienced by the buyer.

Running the lab

1.1. Ordering the car

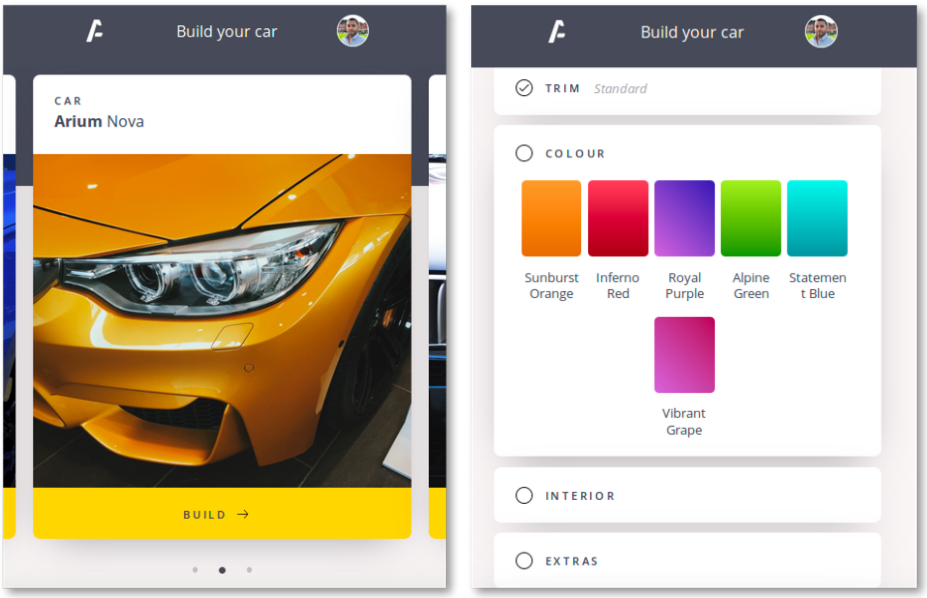
3. If it is not already selected, Switch to the “Ionic App” tab on the web browser (running at localhost:8100).

   Note that Paul’s web page is intended to be viewed as a mobile app; you might want to ungroup this tab from the others by dragging the tab’s title bar off from the others, and resizing the window to make it easier to view and navigate as a mobile app.

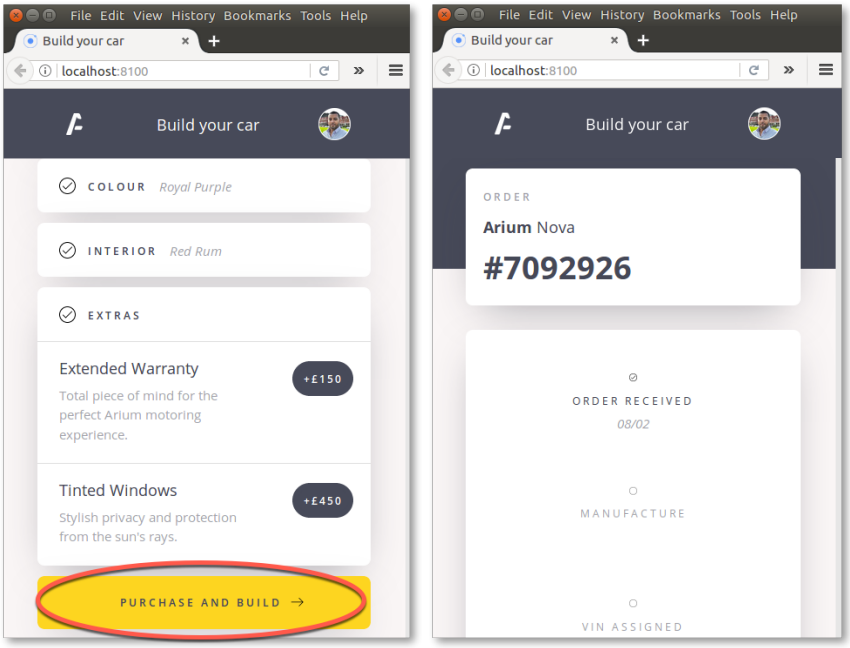


4. In Paul’s app, click **‘Build Your Car’**.

5. Swipe left and right to decide which car to build, and then decide the options on Paul's car.



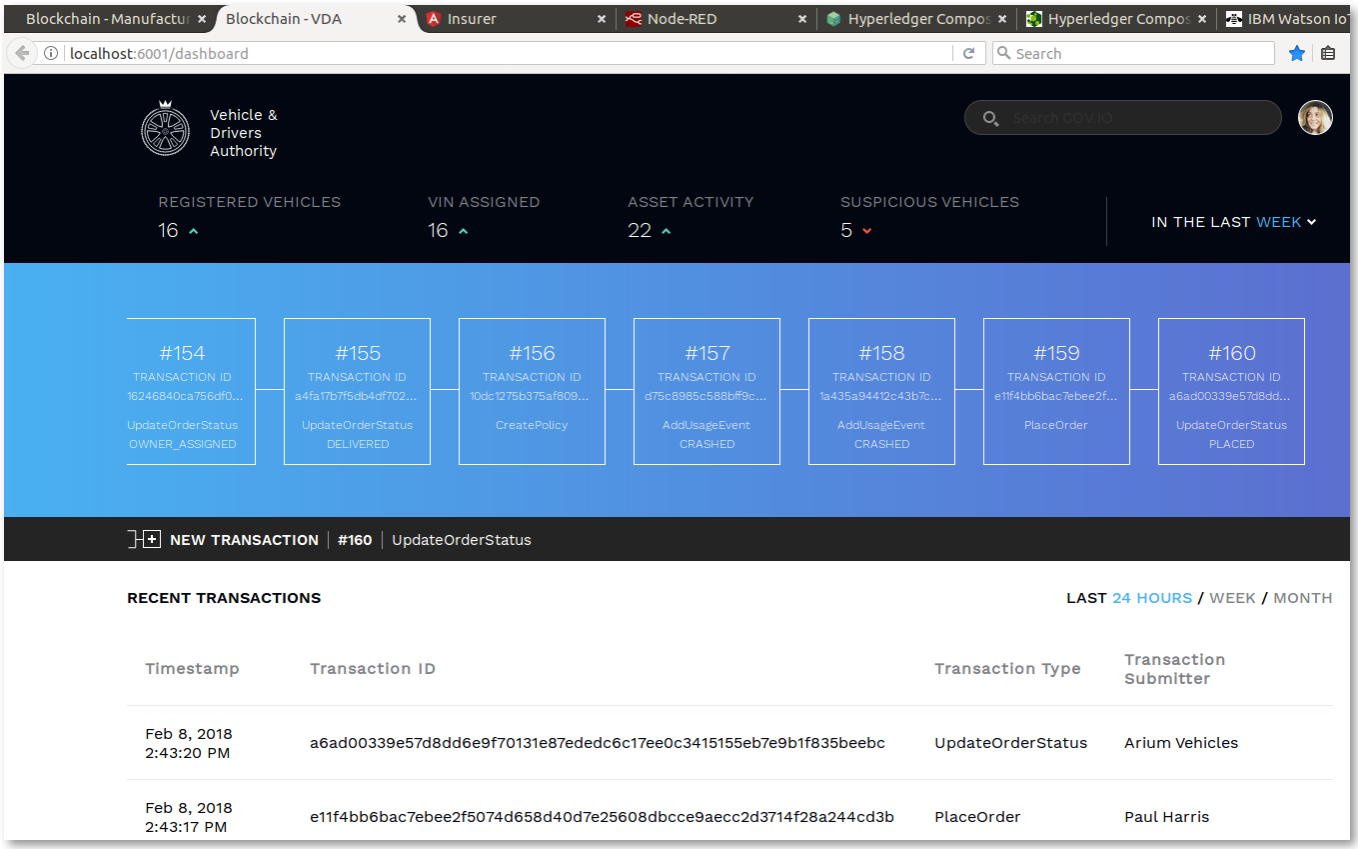
6. Once you have decided on each of Paul's options, click 'Purchase and Build'.



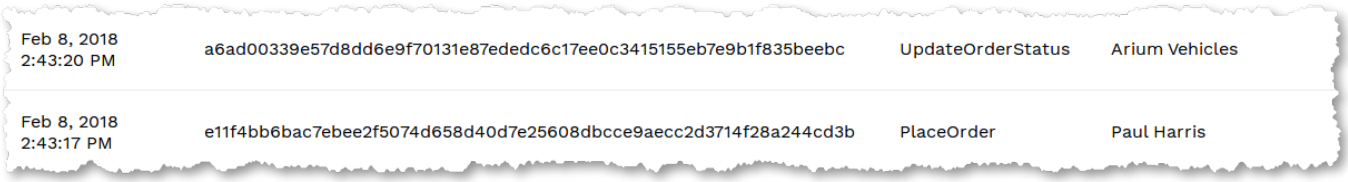
7. Once you place the order, switch to the "Blockchain – VDA" tab (localhost:6001/dashboard).

   As you will recall, the VDA is the regulator who requires notification of all transactions that occur within the business network. Debbie, who works for the regulator, has a dashboard running on her PC that shows all transactions as they occur.

   You will immediately see the VDA dashboard update itself with the latest transaction.



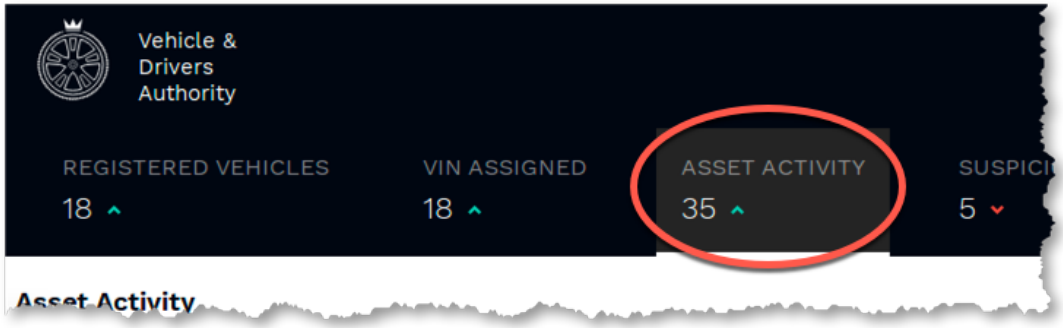
If you look at the “Recent Transactions” log at the bottom of the page, you will see two new transactions listed: a “PlaceOrder” transaction submitted by Paul Harris, and an “UpdateOrderStatus” acknowledgement from Arium Vehicles, our manufacturer.



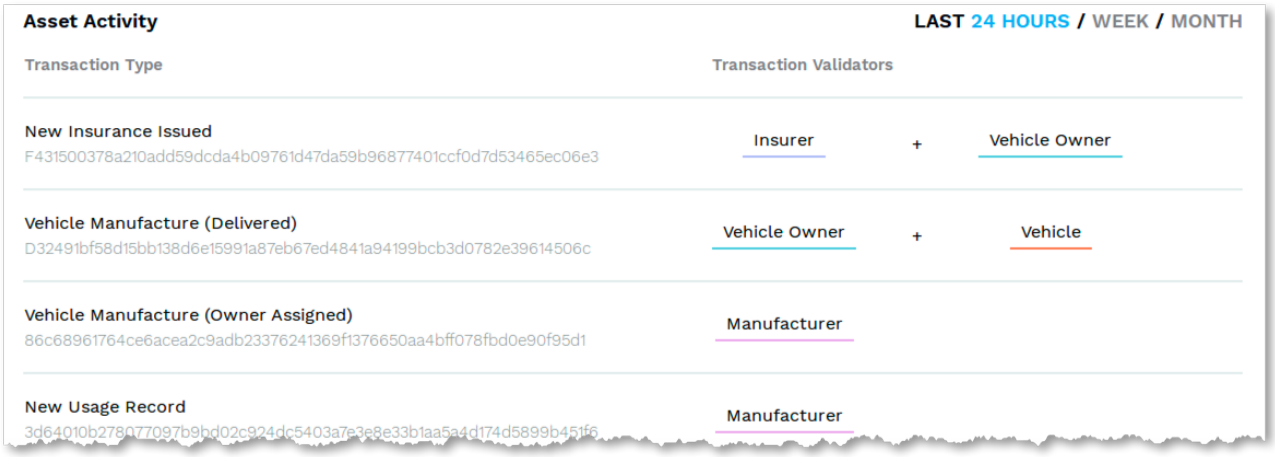
Look in the blue section above this log and you will see those transactions represented graphically as a chain, with the most recent transactions on the right. This is a representation of our blockchain, and the regulator can see everything that is stored on it.

As you will recall, the blockchain is our transaction log which is shared (with appropriate privacy and permissioning) between the participants of our business network. Each block in this chain could potentially actually contain multiple transactions, but here you'll just see each unique transaction inside its own block.

8. Click on 'Asset activity' within the VDA dashboard.



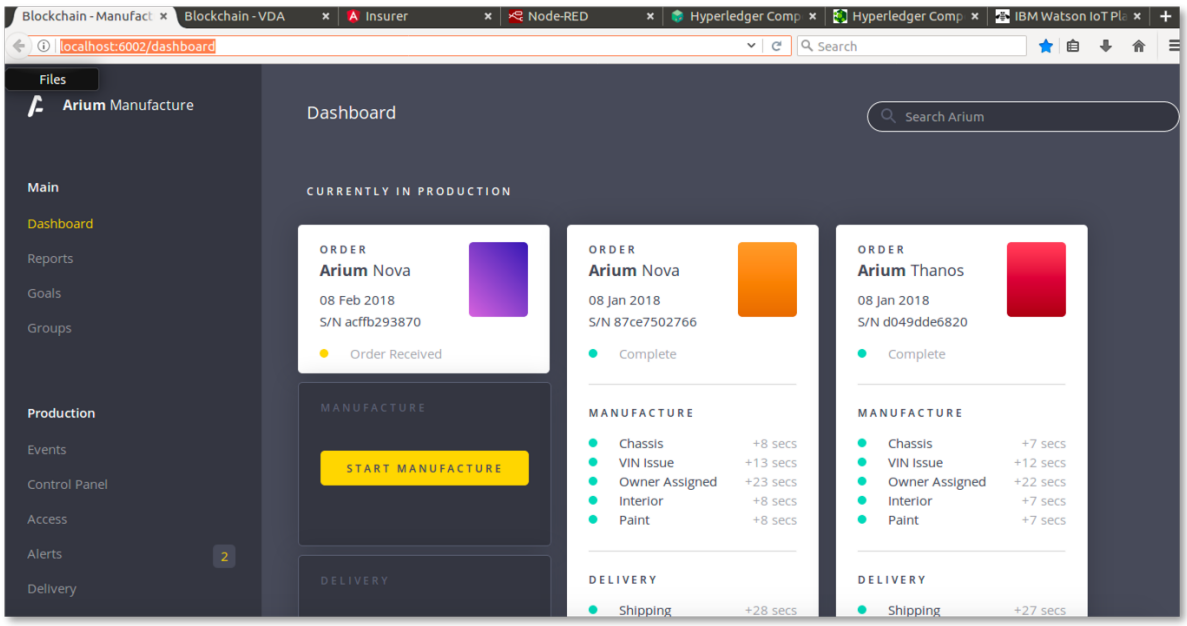
This is an alternative view of the ledger that shows all the transactions that have occurred, and the participants involved.



1.2. Manufacturing the car

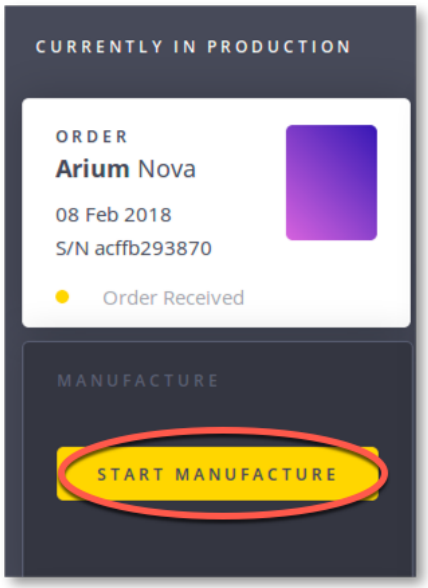
9. Switch to the "Blockchain – Manufacturer" tab (localhost:6002/dashboard).

This is the dashboard that Mike, who works for Arium, uses. He does not have full visibility into the entire blockchain that the regulator requires, but can see the parts of it that pertain to Arium: specifically, he has visibility into all the orders that are coming in so that he can control the manufacturing process.



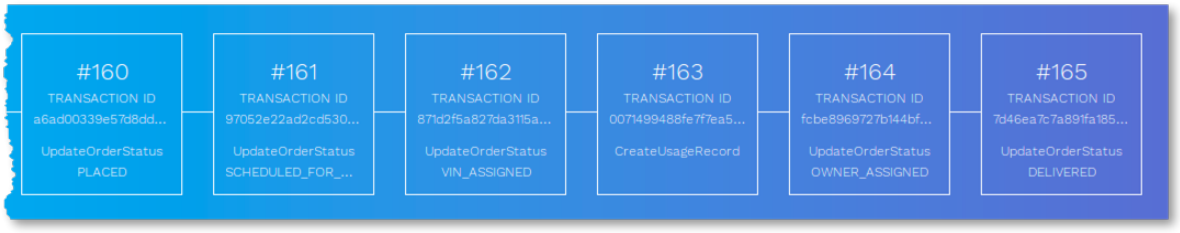
The “Currently in Production” section of this page shows those orders that have been received and the cars that have recently been built. The left-most order in this section will be the car that Paul recently ordered.

10. Click Start Manufacture underneath Paul’s order to start the business process to build a car.

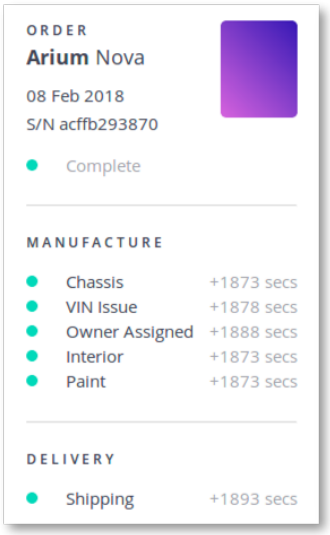


The production process has (of course) been simulated and will take place over the next several seconds; the vehicle will be ‘built’ and blockchain transactions submitted that record status at key milestones of the production process. In a real network, different automated plant systems will trigger these events, which are signed off by the manufacturer, and the issuance of the Vehicle Identification Number might be signed off by the regulator.

11. As the car is being built, switch back to the VDA dashboard to see these key milestones being represented on the unfiltered blockchain.



12. Also note how the Manufacturer's view changes as the vehicle is being built, with icons changing to green as those parts of the process are completed.

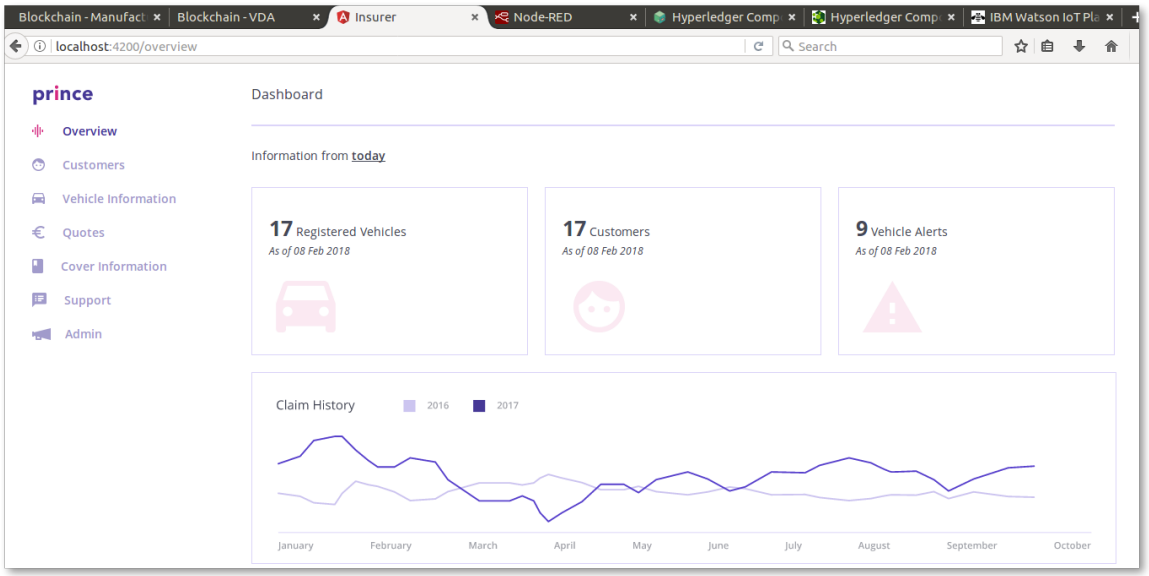


1.3. Insuring the car

As Paul takes ownership of his new car, we will give him the option to insure it. His insurance company offers a discount if he chooses to provide the insurance company with frequent details of the car's location and other things.

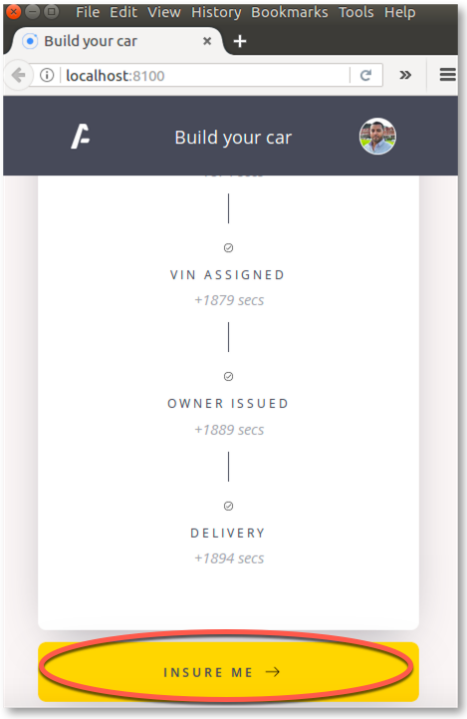
The manufacturer fits the car with a collection of IoT devices, including GPS, air and engine temperature sensors, acceleration information and light information, which can give the insurer information on how the car is being driven, and potentially alert relevant parties if the car is involved in a crash.

13. Switch to the Insurer dashboard (localhost:4200/overview). Ensure that the 'Overview' tab is selected.

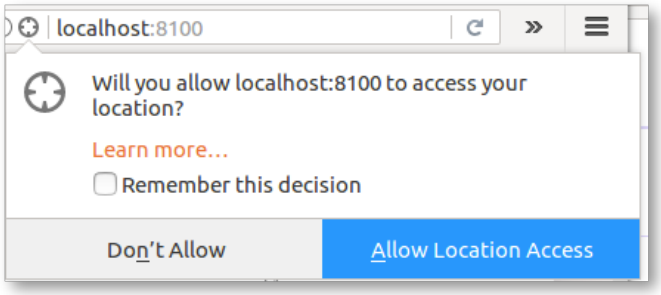


Tommen works for Prince Insurance and this is his dashboard. He requires another subset of information from the blockchain and this view is represented here. He can see information on the cars for which his company is an insurer, and can also approve new polices. (In reality, this latter part can be automated.)

14. Switch back to our car buyer's "Ionic App". After the car has been delivered scroll down to the bottom, and click the **"Insure Me"** button.

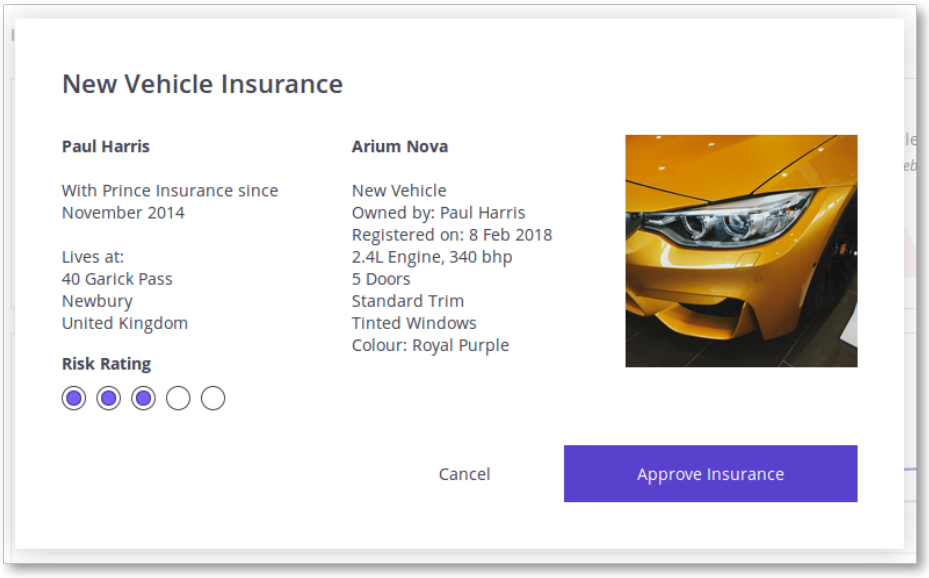


15. Click "**Allow Location Access"** if a popup appears; Paul is willing to share the IoT device's location with the insurance company.



16. Switch back to the "Insurer" view (localhost:4200/overview).

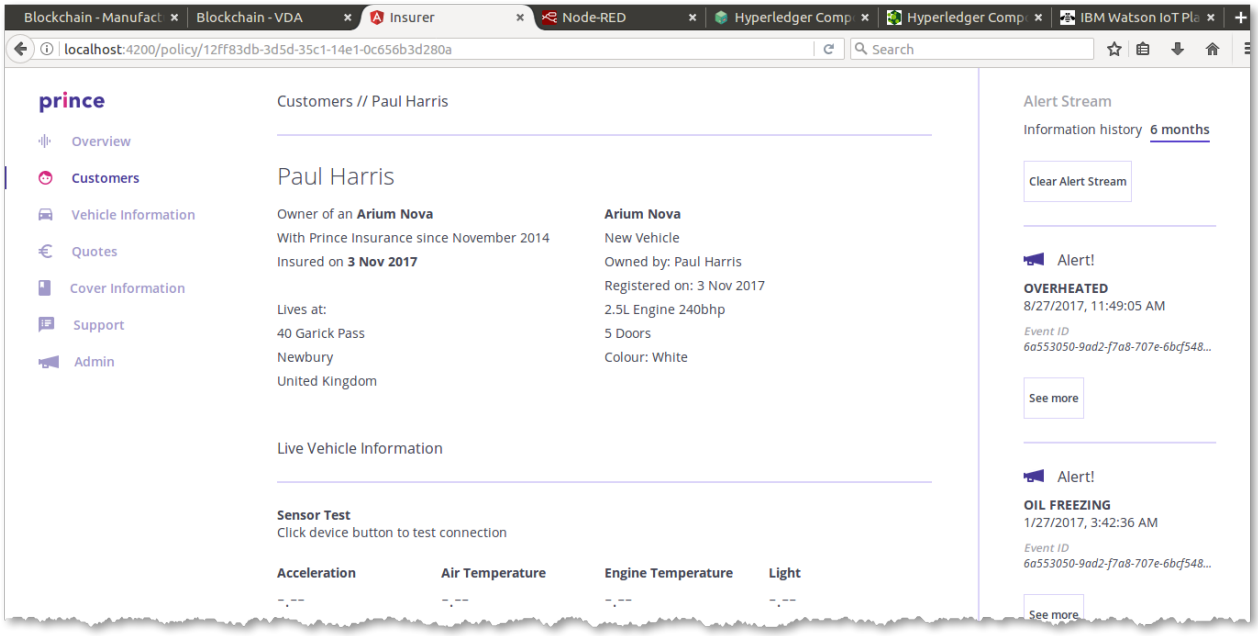
17. Click the "Approve Insurance" button. If the button does not appear, refresh the Insurer view tab.



18. Wait for the approval to be logged on the blockchain.

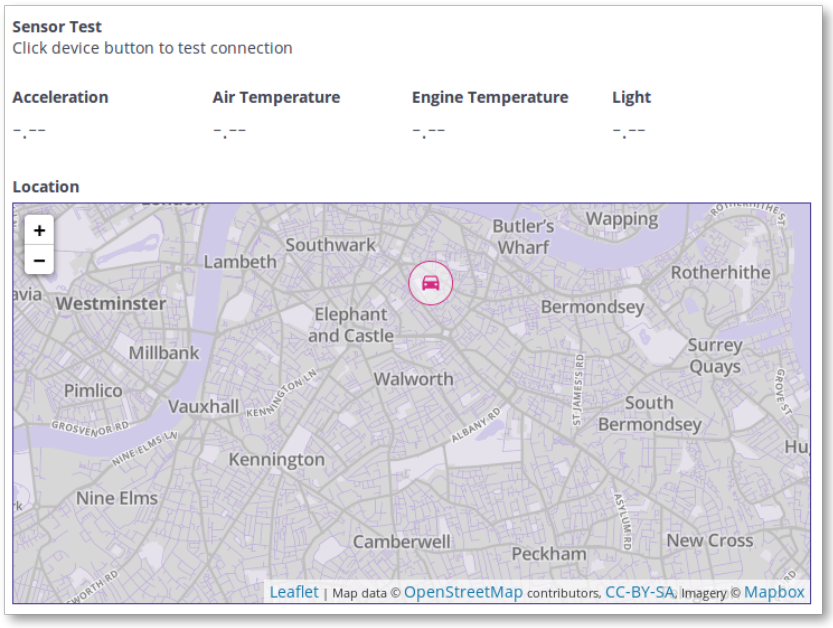
    Paul is now insured by the insurance company.

19. Review the 'Customers' tab to see details of Paul's policy.



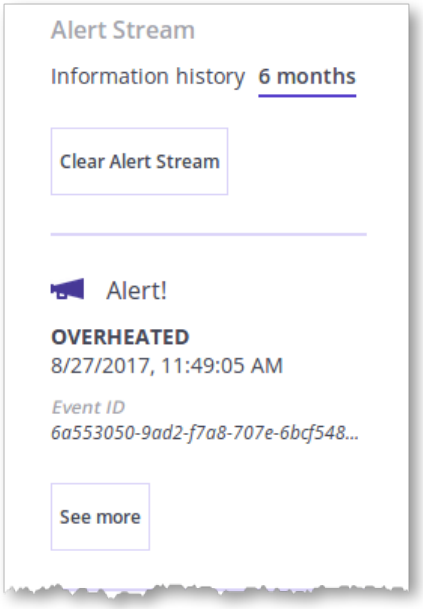
At the top of the page you can see basic details of Paul's policy including his address and car information.

Underneath this is the set of raw readings from the IoT devices attached to Paul's car. This is useful information for debugging; in reality the blockchain is not used to share complete data streams from the IoT sensors as the amount of data is too great and is not relevant to be shared in its entirety.



However, what would be relevant is the analysis of key events in the IoT stream. For example, if the acceleration is shown to be greater than (for example) 2G, this might indicate a crash event that the insurer might care about.

This is shown as a set of alerts on the right hand side of the insurer's customer view:



Without a real sensor tag connected into the application, the information displayed here is either blank or mocked up. In the next section we will inject data into the application using internet of things integration.

It is possible to connect a real sensor tag so that its information is displayed in the Insurer view; we have tested using a Texas Instruments SimpleLink Bluetooth SensorTag. To use this, you need to download the TI SimpleLink Starter app to a nearby mobile device, use it to discover the sensor via Bluetooth, note down the unique address of the tag and enable the “Push to cloud” option to submit the sensor readings so that they can be read by the IBM Watson IoT platform. Then you need to update the “IBM IoTP Test Device” node in the Node-RED flow to monitor the readings from the unique address of the tag from the cloud. Remember to redeploy the Node.RED flow.

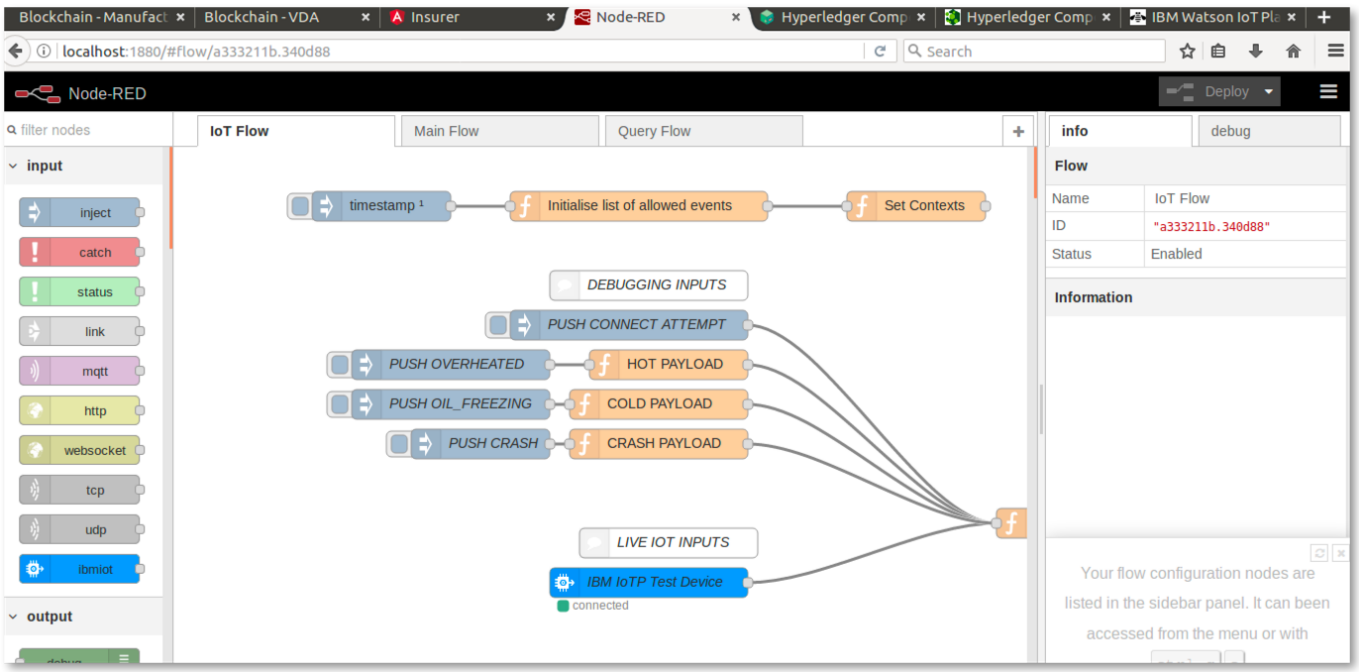
Cool Stuff

In this section, we're going to look at how the scenario can be enhanced by bringing blockchain together with internet of things and analytics.

1.4. Internet of Things Integration

We will start by looking at how sensor data from the car makes its way into the blockchain. To do this we will use an integration tool called Node-RED. This includes a graphical interface to describe how data flows from input sources (e.g. a sensor) to output sources (e.g. the blockchain).
Node-RED has connectors for sending data to, and receiving data from, a blockchain running Hyperledger Composer. It also has connectors for receiving data from the IBM Watson IoT platform (for sensor tag integration). We can also generate fake sensor data for testing, in the absence of a physical sensor device.

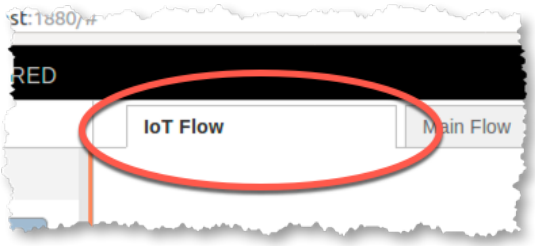
20. Switch to the Node-RED tab (localhost:1880).



The main window shows the flow of how data from devices is mapped to blockchain events. The tabs along the top show the different flows that are deployed. Down the left hand side you can see the available connectors for wiring into the flow. The right hand-side contains the properties of the selected connector and debug information.

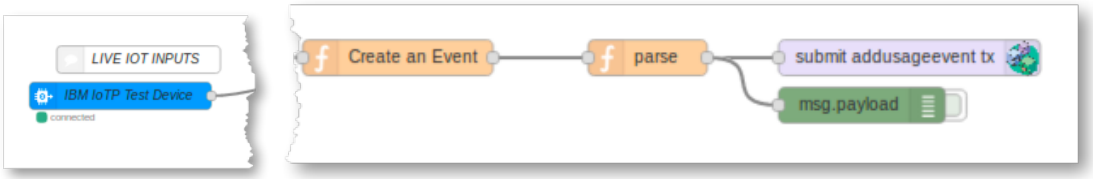
(Note that if you make any changes to the flow, you need to press the "Deploy" button to let them take effect.)

21. Ensure that the "IoT Flow" tab in Node-RED is selected.

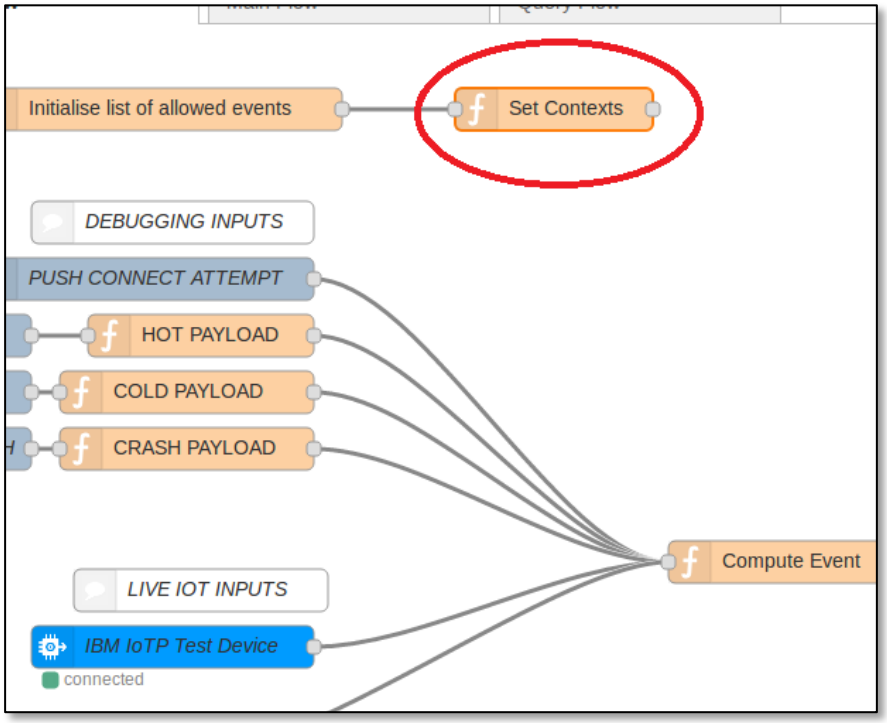


There are some interesting things to note in this flow.

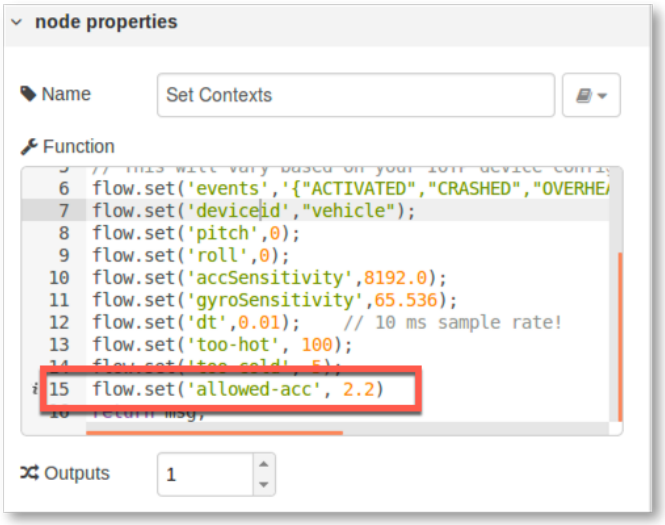
22. Look at the flow path that runs between the “IBM IoTP Test Device” node to the “submit addusagevent tx” flow. This takes readings from a real sensor device and publishes any interesting events to the blockchain.



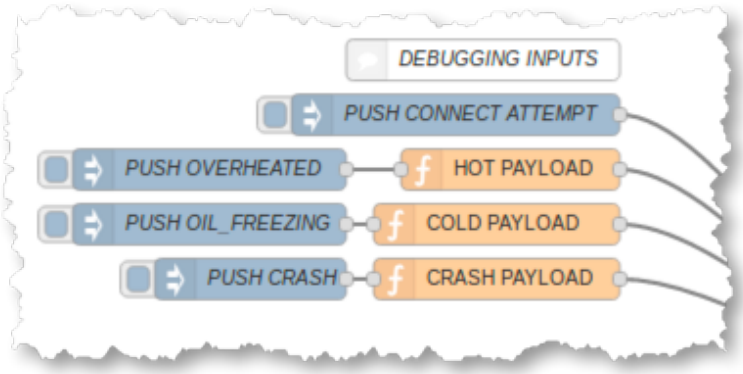
23. Double click the “Set Contexts” node.



This shows the thresholds for sending interesting events to the blockchain. For example, if the acceleration is greater than 2.2G, this causes a crash event to be sent to the blockchain.

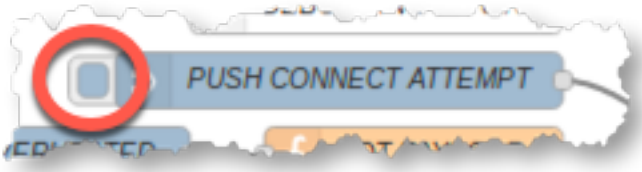


24. Look at the set of connectors next to the "DEBUGGING INPUTS" section: PUSH CONNECT ATTEMPT, PUSH OVERHEATED, PUSH OIL FREEZING and PUSH CRASH.



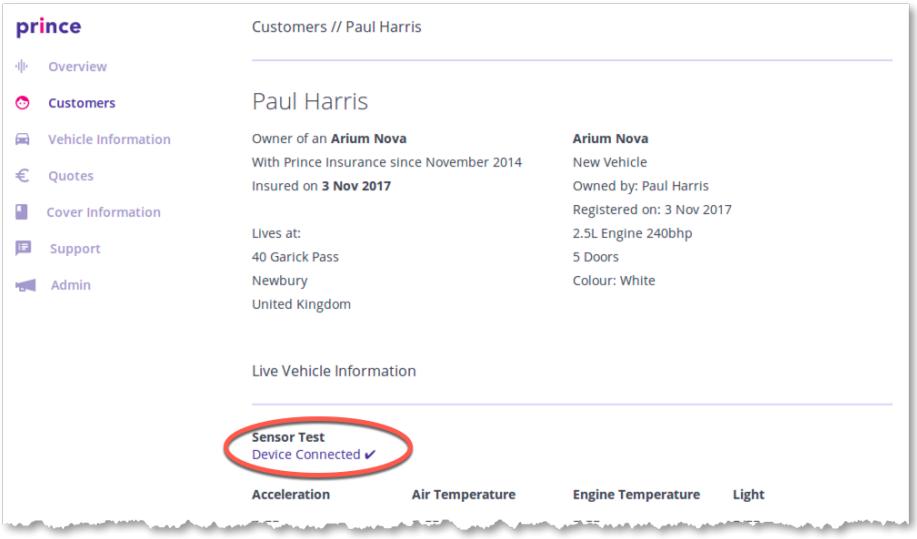
These connectors allow us to simulate an interesting event occurring, in the absence of a real device.

25. Click the rounded square button next to the PUSH CONNECT ATTEMPT connector.



You should see a message saying that data was successfully injected into the flow.

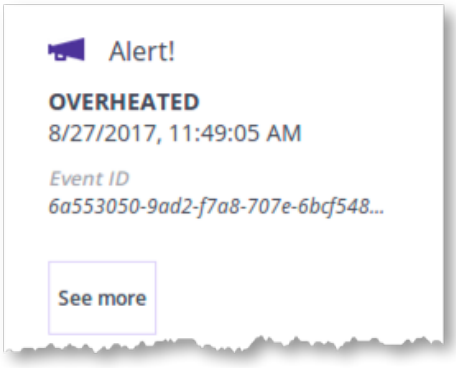
26. Switch to the Insurer tab (localhost:4200), and notice under "Sensor Test" that the vehicle sensor is now connected.



27. Switch back to the Node-RED tab (localhost:1800), and click the button next to the PUSH OVERHEATED node to send an event to the blockchain which denotes Paul's engine overheating.

You should again see a "Successfully injected" message.

28. In the Insurer view you should see an alert that reveals this event to the insurer.



29. Try invoking the other events too (OIL FREEZING, CRASH) to see their effect.

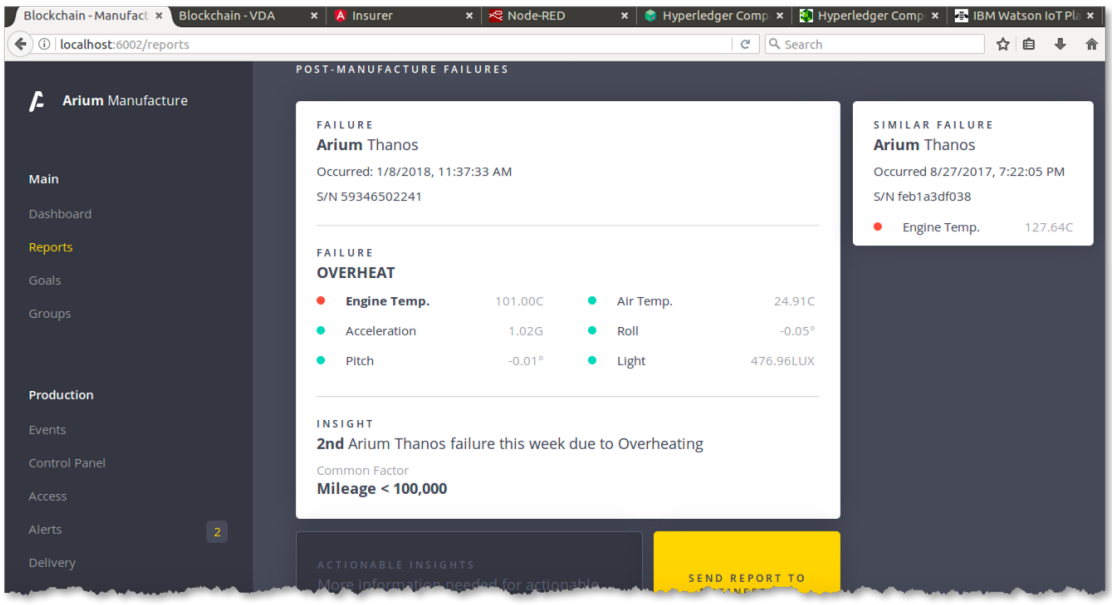
    More details on the IBM Watson IoT Platform can be found on a pre-loaded tab in the web browser (https://i5l9uv.internetofthings.ibmcloud.com/dashboard/#/ibmssolanding).

1.5. Analytics

It is possible to use the information stored on the blockchain to provide insight on aggregate usage patterns to interested authorized parties. This gives the power of data analytics on top of the benefits of a blockchain, as a verifiably clean source of information to analyze.

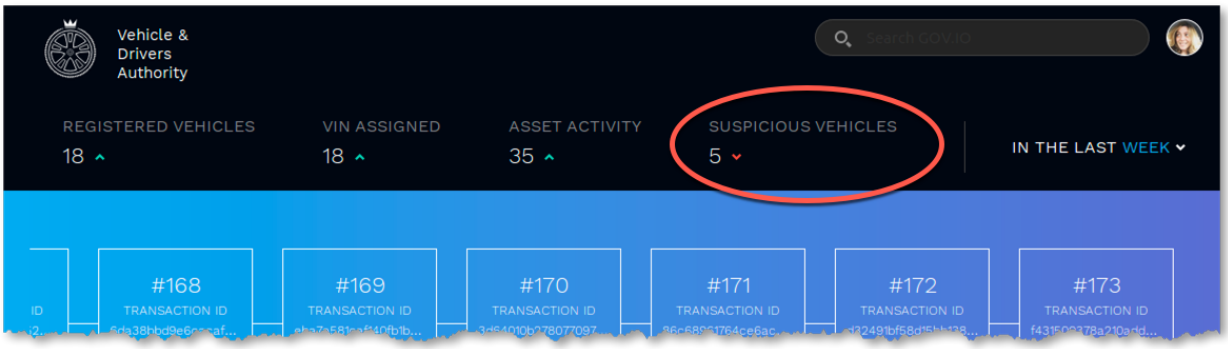
30. Switch back to the Manufacturer view tab (localhost:6002) and click the “Reports” link.

    The engine overheated events show in this view. These events were captured in the blockchain and the manufacturer role has permission see this type of event. The manufacturer wishes to detect trends in engine overheated failures in order to determine if a factory defect is causing this condition.



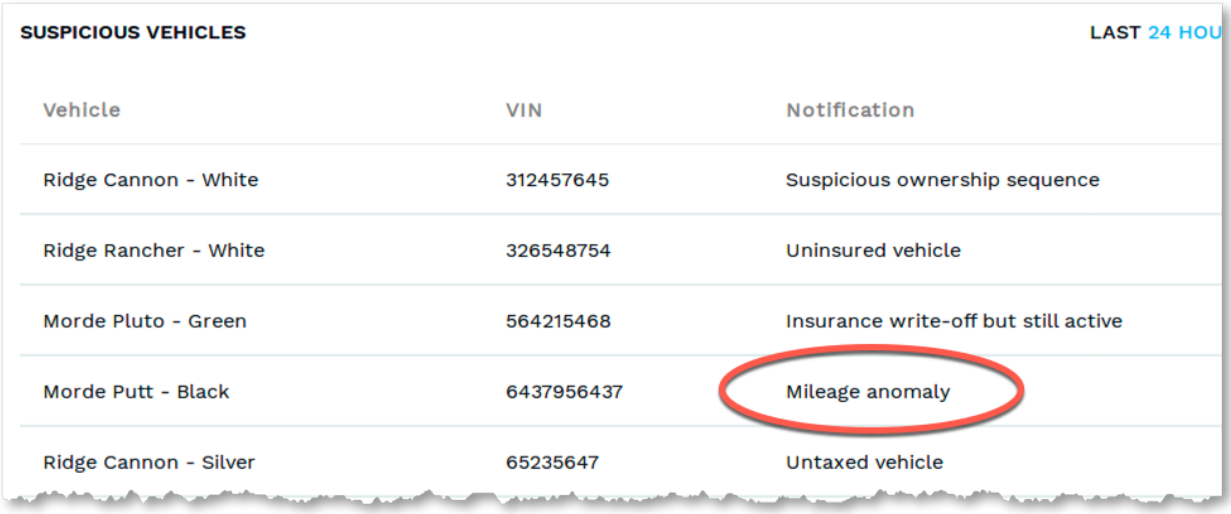
    The regulator in this scenario can also run analytics on the transactions on the blockchain to look for suspicious behavior that the smart contract was not designed to prevent from a single invocation.

31. In the Vehicle & Driver Authority dashboard (localhost:6001/dashboard) click the “Suspicious Vehicles” tab near the top.



Here we can see that by performing analytics on the blockchain dataset, we have found a number of vehicles with associated suspicious transactions that may warrant further investigation.

32. Click on 'Mileage anomaly' in the list of suspicious vehicles.



This shows a list of the transactions that are associated with this anomaly. In this instance, the mileage of the vehicle may not match with insurance records - or has even has decreased from previous records.

Under the Hood

In the final section of the lab, we will briefly consider how the scenario was put together. If you wish to find out more about the tools used to create this application, consider completing a follow-on lab; ask the instructor for details.

1.6. Modelling the Scenario

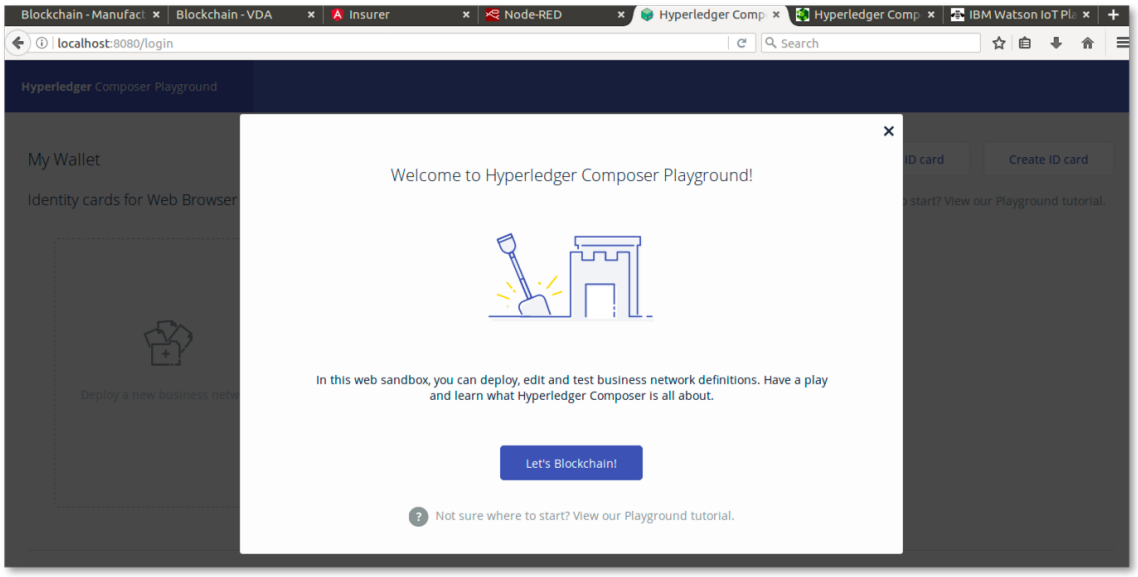
All blockchain use-cases can be described in terms of a set of *assets* (the digital representation of some tangible or intangible object that holds value), *participants* (who wish share information with other participants in a trustworthy way) and *transactions* (which cause the assets to change state).

In our example, the primary asset is the car (obviously), the participants are the owner, manufacturer, regulator and insurer, and as we've seen, there are several transaction types as the car moves through its lifecycle.

These assets, participants and transactions can be modelled in a Linux Foundation tool called Hyperledger Composer, and leveraged through the IBM Blockchain Platform.

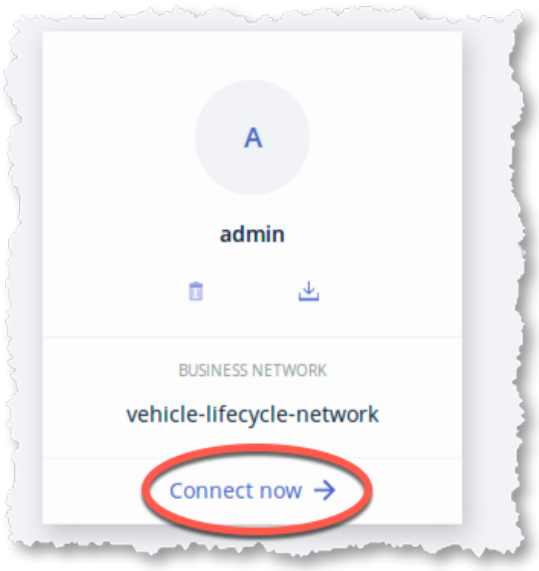
It is useful to develop a model of these concepts as it provides a neat abstraction layer between the business problem being solved and the technical complexities of the underlying blockchain – in much the same way as a compiler shields the programmer from the details of the underlying machine code.

33. Switch to the Hyperledger Composer Playground tab in the web browser (localhost:8080/login).

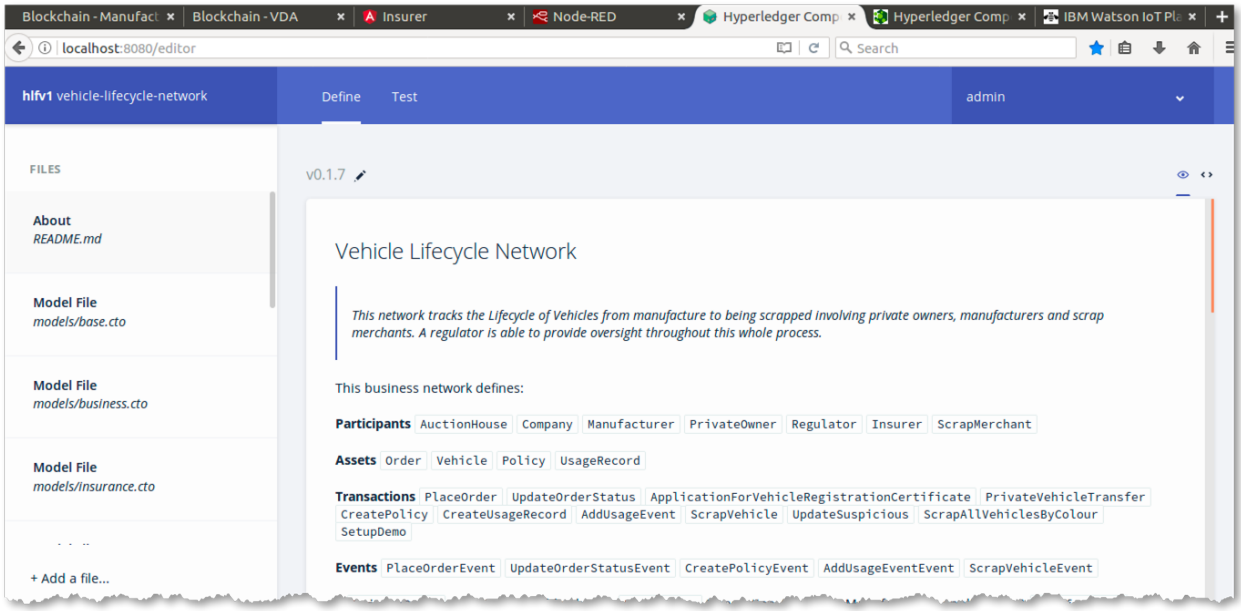


34. Dismiss the welcome dialog by clicking "Let's Blockchain!".

35. Scroll to the bottom of the "My Wallet" screen to see details of our deployed blockchain network (vehicle-lifecycle-network. Click 'Connect now'.



Once the Playground has connected to the blockchain, you will see details of the vehicle lifecycle network.



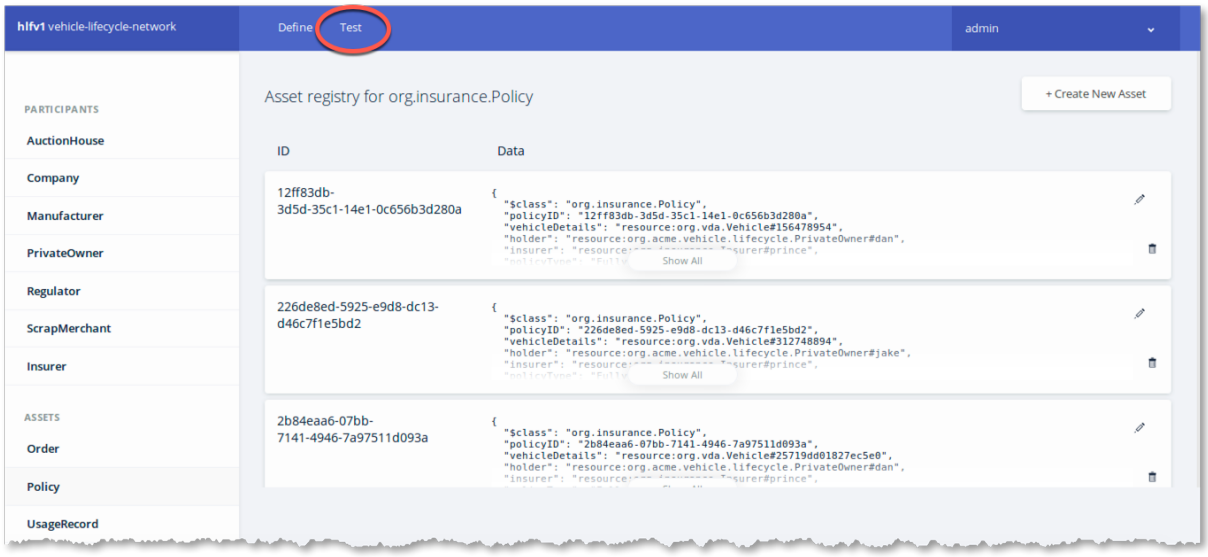
Along the top of the screen are two tabs: “Define” which shows the files used to model the network, and “Test” that allows authorized users (an administrator “admin” - by default) to invoke transactions.

36. With the Define tab selected, click the filenames down the left hand side of the screen to view the contents of the files that comprise the model, transaction logic, access control lists and documentation.

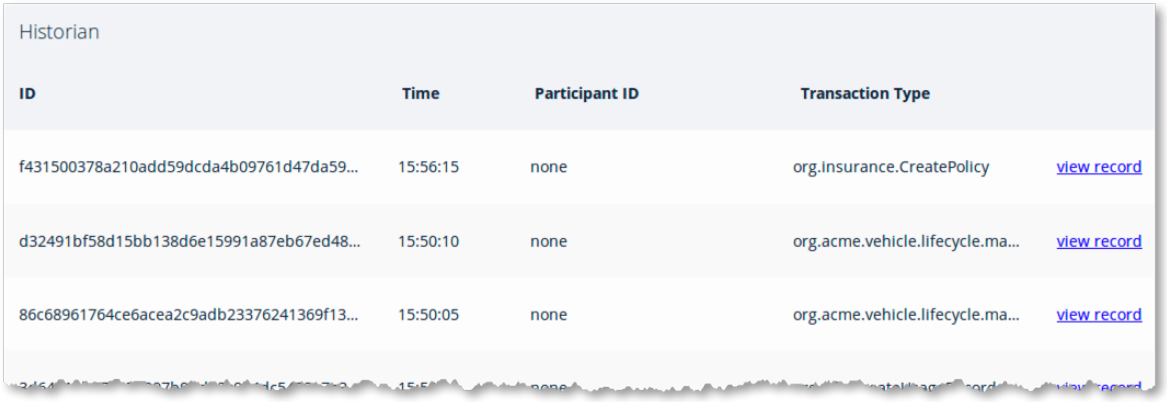
```
122  asset Vehicle identified by vin {
123    o String vin
124    o VehicleDetails vehicleDetails
125    o VehicleStatus vehicleStatus
126    --> Person owner optional
127    o String numberPlate optional
128    o String suspiciousMessage optional
129    o VehicleTransferLogEntry[] logEntries optional
130  }
```

We will go into details of what these files do in a follow-on lab.

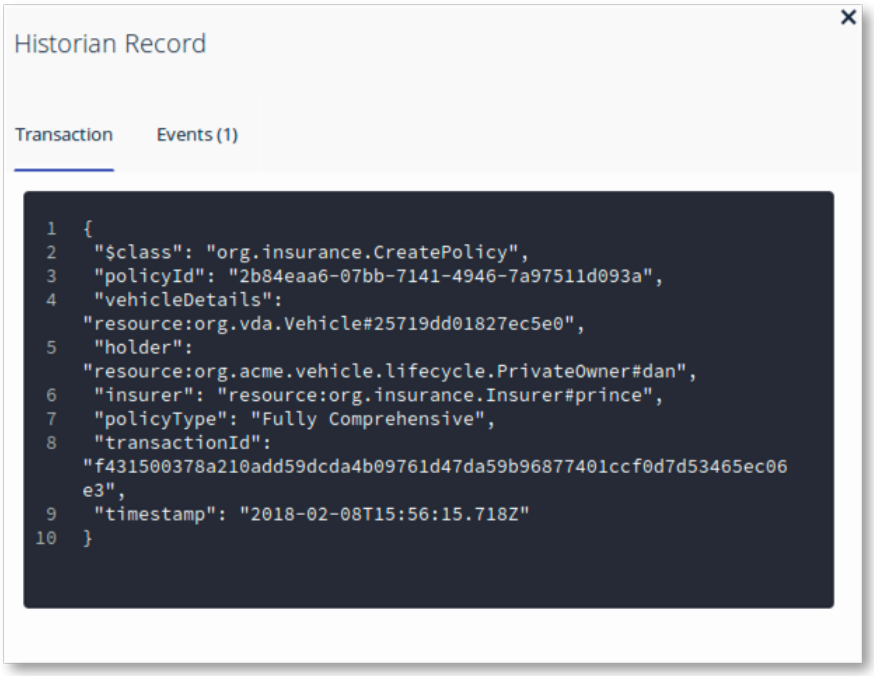
37. With the Test tab selected, click the registries down the left hand side of the screen to view the instances of the assets, participants and transactions that have been created, and their current state.



38. Click 'All Transactions' to view the Transaction Historian. This shows you every transaction that has been recorded on the blockchain that the current user ('admin') has authority to see.



39. Click on any transaction to view details of it.



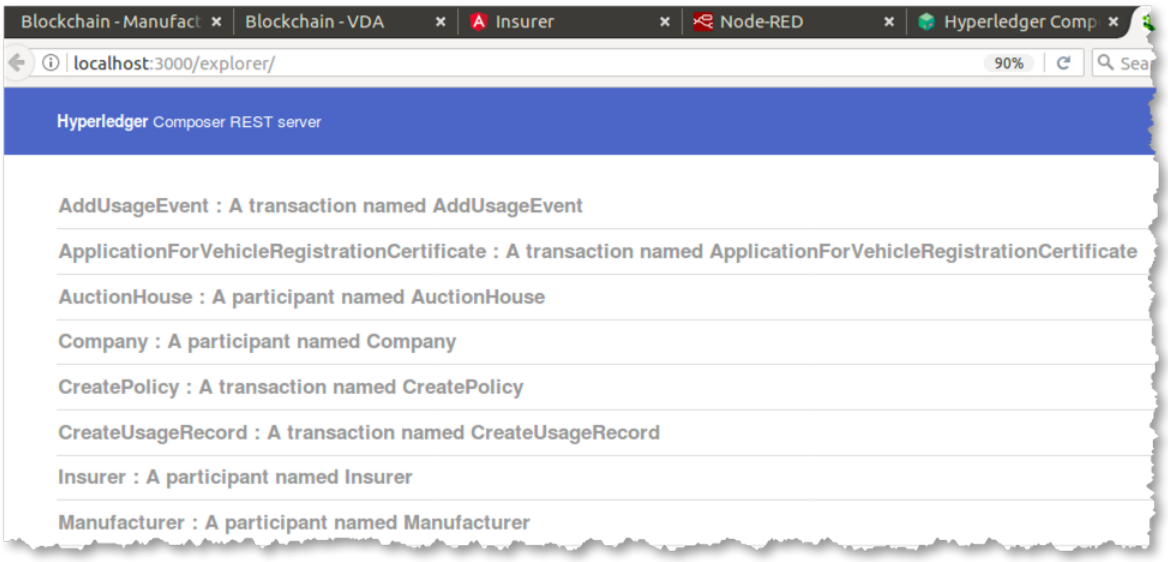
1.7. How the Applications work

While the Playground can be used to test our blockchain scenario, our end-users use mobile apps and dashboards to interact with the running blockchain.

From the files that model this network and implement the transactions, Hyperledger Composer can help this in two ways. Firstly, the models can be used to create skeleton applications that make it easier to develop the end-user applications. Secondly, the models can also be used to generate RESTful APIs that allow client applications and integration middleware to interact with the blockchain.

We will now look at the set of RESTful APIs that have been generated for this scenario.

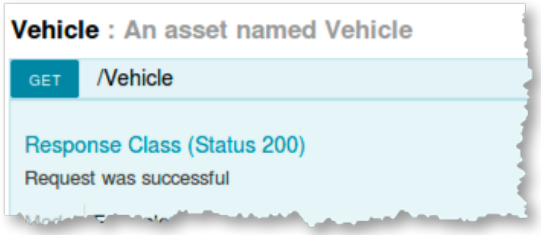
40. Select the Hyperledger Composer REST server tab (localhost:3000/explorer).



This view shows the REST interface that has been generated from the deployed vehicle lifecycle model. End-user applications and integration middleware can invoke these applications by sending HTTP requests that invoke these APIs.

This is how the Node-RED flows interact with the blockchain. Our end-user applications (Paul's mobile app, the VDA view, Insurer dashboard etc.) can also interact in this way, although it is possible for Javascript client applications to instead import (*require*) a Javascript module that interacts the blockchain in a similar way.

41. Review the different APIs available; feel free to try invoking them from the web front end to see what effect it has on the blockchain, on end-user applications and on Playground views. For example… scroll to the bottom of the list, and click on "Vehicle" to expand its available REST calls - then click on "Get /Vehicle".



Next Steps

In this lab you have experienced a live blockchain solution through the eyes of four participants of a vehicle network: a buyer/owner, manufacturer, regulator and insurer. A blockchain can be used to great effect in this business network because there is a clear need to share information and value in participants being able to trust the information they see.

Where you go from here is up to you.

If you have a technical background, consider finding out more about the Hyperledger Fabric and Composer technologies and the blockchain development experience.

If you are interested in the potential benefits of blockchain in your business, IBM has a bunch of services that can help. Start by going to www.ibm.com/blockchain.

Important! Be sure to cleanup the hyperledger fabric environment for subsequent labs. Perform the following at the command prompt in the VLD directory:

a. Open a terminal window right clicking on the terminal icon  on the left-hand navigation pane of the Ubuntu Linux desktop. Select **New Terminal** and a new terminal window will appear.
b. cd VLD
c. ***./stopAll.sh***

Appendix A. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements

will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental. All references to fictitious companies or individuals are used for illustration purposes only.

Appendix B. Trademarks and copyrights

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	AIX	CICS	ClearCase	ClearQuest	Cloudscape
Cube Views	DB2	developerWorks	DRDA	IMS	IMS/ESA
Informix	Lotus	Lotus Workflow	MQSeries	OmniFind	
Rational	Redbooks	Red Brick	RequisitePro	System i	
System z	*Tivoli*	*WebSphere*	*Workplace*	*System p*	

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of The Minister for the Cabinet Office, and is registered in the U.S. Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

Please Recycle

IBM Blockchain Hands-On
Hyperledger Fabric SDK Development:
Writing Your First Application

Lab Two

Contents

Overview

The purpose of this lab is to enable you to write your first blockchain application by introducing you to the Hyperledger Fabric SDK. This lab is based on the “fabcar” Hyperledger Fabric sample: http://hyperledger-fabric.readthedocs.io/en/release-1.3/write_first_app.html

Writing your First Hyperledger Fabric Application

At the most basic level, applications on a blockchain network are what enable users to **query** a ledger (asking for specific records it contains), or to **update** it (adding records to it).

Our application, composed in JavaScript, leverages the Node.js SDK to interact with the network (where our ledger exists). This tutorial will guide you through the three steps involved in writing your first application.

1. Install Samples, Binaries, and Docker Images. While we work on developing real installers for the Hyperledger Fabric binaries, we provide a script that will download and install samples and binaries to your system. We think that you'll find the sample applications installed useful to learn more about the capabilities and operations of Hyperledger Fabric.

2. Starting a test Hyperledger Fabric blockchain network. We need some basic components in our network in order to query and update the ledger. These components – a peer node, ordering node and Certificate Authority – serve as the backbone of our network; we'll also have a CLI container used for a few administrative commands. A single script will launch this test network.

3. Learning the parameters of the sample smart contract our app will use. Our smart contracts contain various functions that allow us to interact with the ledger in different ways. For example, we can read data holistically or on a more granular level.

4. Developing the application to be able to query and update records. We provide two sample applications – one for querying the ledger and another for updating it. Our apps will use the SDK APIs to interact with the network and ultimately call these functions.

After completing this tutorial, you should have a basic understanding of how an application, using the Hyperledger Fabric SDK for Node.js, is programmed in conjunction with a smart contract to interact with the ledger on a Hyperledger Fabric network.

1.1. Install Samples, Binaries, and Docker Images

1 Start a terminal
2 cd ~
3 Your instructor has already performed this step. However for future reference, you can enter the following command to pull down binaries, docker images, and samples. You do not need to enter the command below as this has already been performed for you on the VMWare image used for these labs. If for some reason, you do not have **a /home/blockchain/fabric-samples** directory, then, issue the command below to pull down the samples.

```
curl -sSL http://bit.ly/2ysbOFE | bash -s 1.3.0
```

4 cd ~/fabric-samples/first-network
5 Issue the following command to shutdown any existing blockchain networks. Answer 'Y' to the Continue prompt.

```
./byfn.sh down
```

6 Issue the following command to kill any active or stale containers:

```
docker rm -f $(docker ps -aq)
```

7 Issue the following command to clear any cached networks. Answer 'Y' to the Continue prompt.

```
docker network prune
```

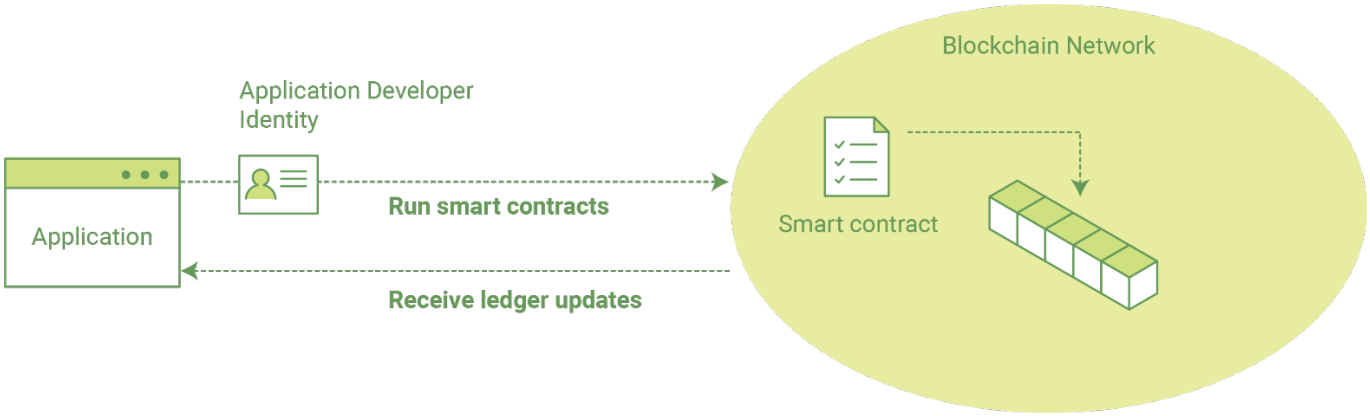
1.2. Start the Test Network

1 Start a terminal
2 `cd ~/fabric-samples/fabcar/`
3 Issue the following command which can take a minute or so to complete. The 'node' parameter will start the Node.js chaincode container vs. the Go chaincode container.

```
./startFabric.sh node
```

4 For the sake of brevity, we won't delve into the details of what's happening with this command. Here's a quick synopsis:
- launches a peer node, ordering node, Certificate Authority and CLI container
- creates a channel and joins the peer to the channel
- installs smart contract (i.e. chaincode) onto the peer's file system and instantiates said chaincode on the channel; instantiate starts a chaincode container
- calls the `initLedger` function to populate the channel ledger with 10 unique cars

5 These operations will typically be done by an organizational or peer admin. The script uses the CLI to execute these commands, however there is support in the SDK as well. Refer to the Hyperledger Fabric Node SDK repo for example scripts.
6 Issue a "`docker ps`" command to reveal the processes started by the `startFabric.sh` script. You can learn more about the details and mechanics of these operations in the Building Your First Network section. Here we'll just focus on the application. The following picture provides a simplistic representation of how the application interacts with the Hyperledger Fabric network.



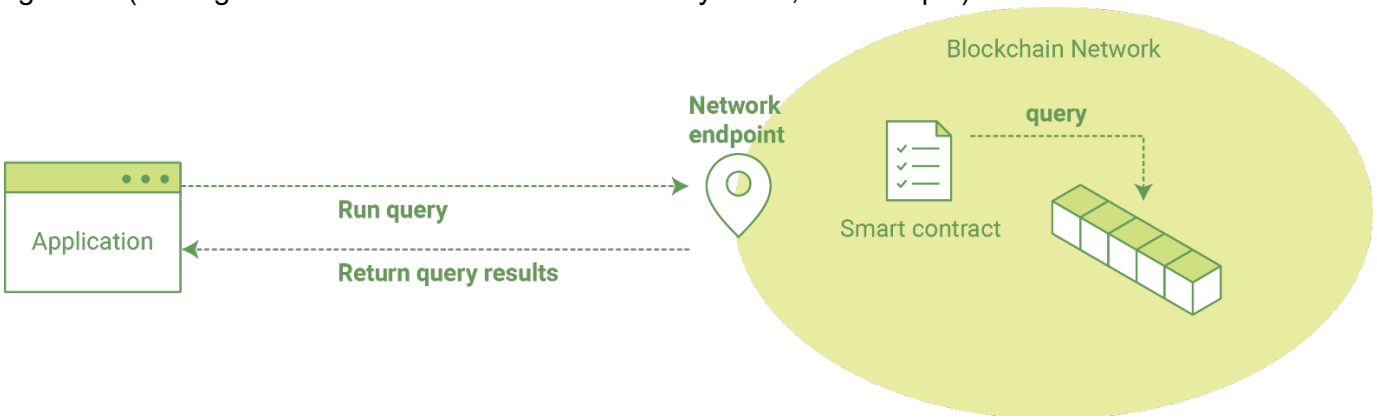
1.3. How Applications Interact with the Network

Applications use **APIs** to invoke smart contracts (referred to as "chaincode"). These smart contracts are hosted in the network and identified by name and version. For example, our chaincode container is titled - `dev-peer0.org1.example.com-fabcar-1.0` - where the name is `fabcar`, the version is `1.0` and the peer it is running against is `dev-peer0.org1.example.com`.

APIs are accessible with a software development kit (SDK). For purposes of this exercise, we'll be using the Hyperledger Fabric Node SDK though there is also a Java SDK and CLI that can be used to develop applications.

Querying the Ledger

Queries are how you read data from the ledger. You can query for the value of a single key, multiple keys, or – if the ledger is written in a rich data storage format like JSON – perform complex searches against it (looking for all assets that contain certain keywords, for example).



As we said earlier, our sample network has an active chaincode container and a ledger that has been primed with 10 different cars. We also have some sample Javascript code - `query.js` - in the `fabcar` directory that can be used to query the ledger for details on the cars.

Before we take a look at how that app works, we need to install the SDK node modules in order for our program to function. From your `fabcar` directory, issue the following:

```
npm install
```

You will issue all subsequent commands from the `fabcar` directory.

1 The following two commands involve communication with the Certificate Authority. You may find it useful to view a log of activity with the Certificate Authority. Open a new terminal window and enter the following command.

   docker logs -f ca.example.com

2 When we launched our network, an admin user – admin – was registered with our Certificate Authority. Now we need to send an enrollment call to the CA server and retrieve the enrollment certificate (eCert) for this user. We won't delve into enrollment details here, but suffice it to say that the SDK and by extension our applications need this cert in order to form a user object for the admin. We will then use this admin object to subsequently register and enroll a new user. Send the admin

enroll call to the CA server.

```
node enrollAdmin.js
```

This program will invoke a certificate signing request (CSR) and ultimately output an eCert and key material into a newly created folder – hfc-key-store – at the root of this project. Our apps will then look to this location when they need to create or load the identity objects for our various users.

3 With our newly generated admin eCert, we will now communicate with the CA server once more to register and enroll a new user. This user – user1 – will be the identity we use when querying and updating the ledger. It's important to note here that it is the admin identity that is issuing the registration and enrollment calls for our new user (i.e. this user is acting in the role of a registrar). Send the register and enroll calls for user1:

```
node registerUser.js
```

Similar to the admin enrollment, this program invokes a CSR and outputs the keys and eCert into the hfc-key-store subdirectory. So now we have identity material for two separate users – admin & user1. Time to interact with the ledger…

4 Now we can run our javascript programs. First, let's run our `query.js` program to return a listing of all the cars on the ledger. A function that will query all the cars, `queryAllCars`, is pre-loaded in the app, so we can simply run the program as is:

```
node query.js
```

5 It should return something like this:

```
Query result count =  1
Response is  [{"Key":"CAR0",
"Record":{"colour":"blue","make":"Toyota","model":"Prius","owner":"Tomoko"}},
{"Key":"CAR1",   "Record":{"colour":"red","make":"Ford","model":"Mustang","owner":"Brad"}},
{"Key":"CAR2", "Record":{"colour":"green","make":"Hyundai","model":"Tucson","owner":"Jin Soo"}},
{"Key":"CAR3", "Record":{"colour":"yellow","make":"Volkswagen","model":"Passat","owner":"Max"}},
{"Key":"CAR4", "Record":{"colour":"black","make":"Tesla","model":"S","owner":"Adriana"}},
{"Key":"CAR5", "Record":{"colour":"purple","make":"Peugeot","model":"205","owner":"Michel"}},
{"Key":"CAR6", "Record":{"colour":"white","make":"Chery","model":"S22L","owner":"Aarav"}},
{"Key":"CAR7", "Record":{"colour":"violet","make":"Fiat","model":"Punto","owner":"Pari"}},
{"Key":"CAR8", "Record":{"colour":"indigo","make":"Tata","model":"Nano","owner":"Valeria"}},
{"Key":"CAR9", "Record":{"colour":"brown","make":"Holden","model":"Barina","owner":"Shotaro"}}]
```

These are the 10 cars. A black Tesla Model S owned by Adriana, a red Ford Mustang owned by Brad, a violet Fiat Punto owned by someone named Pari, and so on. The ledger is key/value based and in our implementation the key is `CAR0` through `CAR9`. This will become particularly important in a moment.

6 Now let's see what it looks like under the hood (if you'll forgive the pun). Use VSCode to open the `query.js` program with "`code query.js`".

7 The initial section of the application defines certain variables such as channel name and network endpoints:

```
// setup the fabric network
```

```
var channel = fabric_client.newChannel('mychannel');
var peer = fabric_client.newPeer('grpc://localhost:7051');
channel.addPeer(peer);
```

8 This is the chunk where we construct our query:

```
const request = {
   //targets : --- letting this default to the peers assigned to the
   channel
   chaincodeId: 'fabcar',
   fcn: 'queryAllCars',
   args: ['']
```

We define the `chaincodeId` variable as `fabcar` – allowing us to target this specific chaincode – and then call the `queryAllCars` function defined within that chaincode.
When we issued the `node query.js` command earlier, this specific function was called to query the ledger. However, this isn't the only function that we can pass.

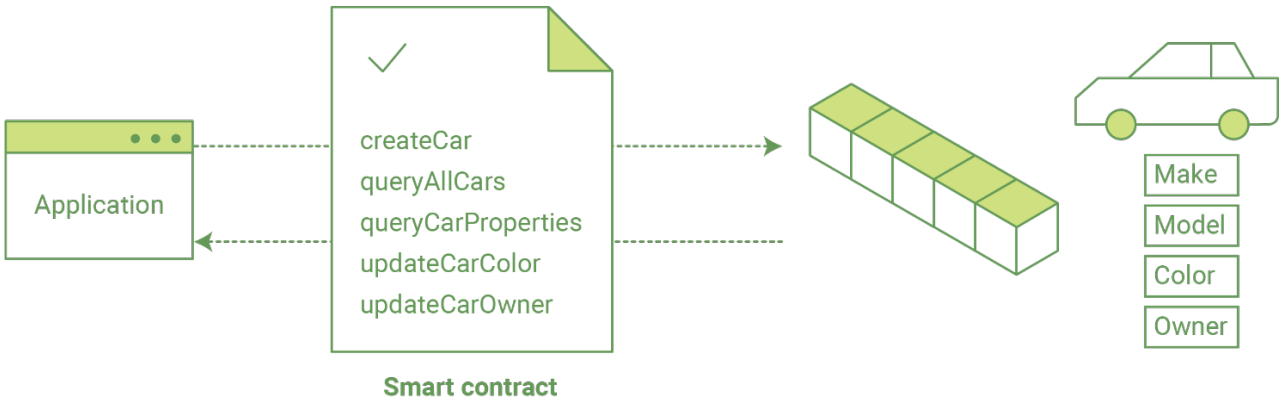
9 To take a look at the others, open `fabcar.js` in VSCode with "`code ../chaincode/fabcar/node/fabcar.js`". You'll see that we have the following functions available to call - `initLedger`, `queryCar`, `queryAllCars`, `createCar` and `changeCarOwner`. Let's take a closer look at the `queryAllCars` function to see how it interacts with the ledger.

```
async queryAllCars(stub, args) {

    let startKey = 'CAR0';
    let endKey = 'CAR999';

    let iterator = await stub.getStateByRange(startKey, endKey);

    let allResults = [];
    while (true) {
      let res = await iterator.next();

      if (res.value && res.value.value.toString()) {
        let jsonRes = {};
        console.log(res.value.value.toString('utf8'));

        jsonRes.Key = res.value.key;
        try {
          jsonRes.Record =
JSON.parse(res.value.value.toString('utf8'));
        } catch (err) {
          console.log(err);
          jsonRes.Record = res.value.value.toString('utf8');
        }
        allResults.push(jsonRes);
      }
      if (res.done) {
        console.log('end of data');
        await iterator.close();
```

```
            console.info(allResults);
            return Buffer.from(JSON.stringify(allResults));
        }
    }
}
```

The function uses the shim interface function `getStateByRange` to return ledger data between the args of `startKey` and `endKey`. Those keys are defined as `CAR0` and `CAR999` respectively. Therefore, we could theoretically create 1,000 cars (assuming the keys are tagged properly) and a `queryAllCars` would reveal every one.

Below is a representation of how an app would call different functions in chaincode.



10 We can see our `queryAllCars` function up there, as well as one called `createCar` that will allow us to update the ledger and ultimately append a new block to the chain. But first, let's do another query.

11 Go back to the `query.js` program and edit the request to query a specific car. We'll do this by changing the function from `queryAllCars` to `queryCar` and passing a specific "Key" to the args parameter. Let's use `CAR4` here. So our edited `query.js` program should now contain the following:

```
const request = {
      chaincodeId: 'fabcar',
      fcn: 'queryCar',
      args: ['CAR4']
```

12 Save the program and navigate back to your `fabcar` directory. Now run the program again:

```
node query.js
```

13 You should see the following:

```
{"colour":"black","make":"Tesla","model":"S","owner":"Adriana"}
```

So we've gone from querying all cars to querying just one, Adriana's black Tesla Model S. Using the `queryCar` function, we can query against any key (e.g. `CAR0`) and get whatever make, model, color, and owner correspond to that car.

Great. Now you should be comfortable with the basic query functions in the chaincode, and the handful of parameters in the query program. Time to update the ledger...

1.4. Updating the Ledger

Now that we've done a few ledger queries and changes a bit of code, we're ready to update the ledger. There are a lot of potential updates we could make, but let's just create a new car for starters.

Ledger updates start with an application generating a transaction proposal. Just like query, a request is constructed to identify the channel ID, function, and specific smart contract to target for the transaction. The program then calls the `channel.sendTransactionProposal` API to send the transaction proposal to the peer(s) for endorsement.

The network (i.e. endorsing peer) returns a proposal response, which the application uses to build and sign a transaction request. This request is sent to the ordering service by calling the `channel.sendTransaction` API. The ordering service will bundle the transaction into a block and then "deliver" the block to all peers on a channel for validation. (In our case we have only the single endorsing peer.)

Finally the application uses the `channel.newChannelEventHub` API to connect to the peer's event listener port, and calls `event_hub.registerTxEvent` to register events associated with a specific transaction ID. This API allows the application to know the fate of a transaction (i.e. successfully committed or unsuccessful). Think of it as a notification mechanism.

We don't go into depth here on a transaction's lifecycle. Consult the Transaction Flow documentation for lower level details on how a transaction is ultimately committed to the ledger.

1 The goal with our initial invoke is to simply create a new asset (car in this case). We have a separate javascript program - `invoke.js` - that we will use for these transactions. Just like `query.js`, use VSCode to open the file and navigate to the code block where we construct our invocation:

```
var request = {
            //targets: let default to the peer assigned to the client
            chaincodeId: 'fabcar',
            fcn: '',
            args: [''],
            chainId: 'mychannel',
            txId: tx_id
      };
```

2 You'll see that we can call one of two functions - `createCar` or `changeCarOwner`. Let's create a red Chevy Volt and give it to an owner named Nick. We're up to `CAR9` on our ledger, so we'll use `CAR10` as the identifying key here. The updated code block should look like this:

```
var request = {
    targets: targets,
    chaincodeId: 'fabcar',
    fcn: 'createCar',
    args: ['CAR10', 'Chevy', 'Volt', 'Red', 'Nick'],
    chainId: 'mychannel',
    txId: tx_id
```

3 Save it and run the program:

```
node invoke.js
```

4 There will be some output in the terminal about Proposal Response and Transaction ID. However, all we're concerned with is this message:

```
The transaction has been committed on peer localhost:7051
```

5 The peer emits this event notification, and our application receives it thanks to our `event_hub.registerTxEvent` API. So now if we go back and edit our `query.js` program and call the `queryCar` function against an arg of `CAR10`, we should see the following:

```
Response is {"colour":"Red","make":"Chevy","model":"Volt","owner":"Nick"}
```

6 Finally, let's call our last function - `changeCarOwner`. Nick is feeling generous and he wants to give his Chevy Volt to a man named Barry. So, we simply edit `invoke.js` to reflect the following:

```
var request = {
   //targets: let default to the peer assigned to the client
   chaincodeId: 'fabcar'
   fcn: 'changeCarOwner',
   args: ['CAR10', 'Barry'],
   chainId: 'myChannel',
   txId: tx_id
```

7 Execute the program again - `node invoke.js` - and then run the query app one final time. We are still querying against `CAR10`, so we should see:

```
Response is {"colour":"Red","make":"Chevy","model":"Volt","owner":"Barry"}
```

1.5. Lab Cleanup

1 Start a terminal
2 cd ~/fabric-samples/first-network
3 Issue the following command to shutdown any existing blockchain networks. Answer 'Y' to the Continue prompt.

```
./byfn.sh down
```

4 Issue the following command to kill any active or stale containers:

```
docker rm -f $(docker ps -aq)
```

5 Issue the following command to clear any cached networks. Answer 'Y' to the Continue prompt.

```
docker network prune
```

Appendix A. Appendix A. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be

the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental. All references to fictitious companies or individuals are used for illustration purposes only.

Appendix B. Trademarks and copyrights

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	AIX	CICS	ClearCase	ClearQuest	Cloudscape
Cube Views	DB2	developerWorks	DRDA	IMS	IMS/ESA
Informix	Lotus	Lotus Workflow	MQSeries	OmniFind	
Rational	Redbooks	Red Brick	RequisitePro	System i	
System z	Tivoli	WebSphere	Workplace	System p	

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of The Minister for the Cabinet Office, and is registered in the U.S. Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

Please Recycle

IBM Blockchain Hands-On

Hyperledger Fabric SDK Development: Modify Chaincode

Lab Three

Contents

Overview

The purpose of this lab is to enable you to modify existing smart contract code invoked by a blockchain application that uses the Hyperledger Fabric SDK.

Modify Chaincode

At the most basic level, applications on a blockchain network are what enable users to **query** a ledger (asking for specific records it contains), or to **update** it (adding records to it). Chaincode is the business logic that is invoked by the application. Chaincode is the codification of the smart contract.

Our application, composed in JavaScript, leverages the Node.js SDK to interact with the network (where our ledger exists). The chaincode also is composed in Node.js. This tutorial will guide you through the steps involved in modifying the chaincode to add a new method and then using the application to invoke the new chaincode method.

1. Perform prerequisite lab cleanup. Cleanup docker containers and existing networks from the previous lab.

2. Modify the sample smart contract our app will use. Our smart contracts contain various functions that allow us to interact with the ledger in different ways. For example, we can read data holistically or on a more granular level. We will modify the smart contract to add a method to query the transaction history of an asset.

3. Starting a test Hyperledger Fabric blockchain network. We need some basic components in our network in order to query and update the ledger. These components – a peer node, ordering node and Certificate Authority – serve as the backbone of our network; we'll also have a CLI container used for a few administrative commands. A single script will launch this test network.

4. Using the application to be able to query and update records. We provide two sample applications – one for querying the ledger and another for updating it. Our apps will use the SDK APIs to interact with the network and ultimately call these functions. We will modify the query application to invoke the method to query the transaction history of a car.

After completing this tutorial, you should have a basic understanding of how an application, using the Hyperledger Fabric SDK for Node.js is programmed in conjunction with how to modify a smart contract to interact with the ledger on a Hyperledger Fabric network.

1.1. Perform prerequisite lab cleanup

1 Start a terminal
2 cd ~/fabric-samples/first-network
3 Issue the following command to shutdown any existing blockchain networks. Answer 'Y' to the Continue prompt.

```
./byfn.sh down
```

4 Issue the following command to kill any active or stale containers:

```
docker rm -f $(docker ps -aq)
```

5 Issue the following command to clear any cached networks. Answer 'Y' to the Continue prompt.

```
docker network prune
```

6 Delete the underlying chaincode image for the fabcar smart contract.

```
docker rmi dev-peer0.org1.example.com-fabcar-1.0-5c906e402ed29f20260ae42283216aa75549c571e2e380f3615826365d8269ba
```

1.2. Modify the smart contract to add a car history method

Modify the **fabcar.js** Node.js code in the following steps to add a method to obtain the history of a car. The chaincode API GetHistoryForKey() will return history of values for a key. Subsequent steps in this lab will create a new CAR asset and then modify the asset so we have some historical activity for the asset.

1 Invoke the VS Code editor for the fabcar.js chaincode.

```
code ../chaincode/fabcar/node/fabcar.js
```

2 Add the following **getCarHistory** method code snippet after the **queryAllCars** method. Be sure to save your changes in the editor.

```
async getCarHistory(stub, args) {
  console.info('============= START : getCarHistory ===========');
  if (args.length != 1) {
    throw new Error('Incorrect number of arguments. Expecting 1');
  }


  let iterator = await stub.getHistoryForKey(args[0]);

  let allResults = [];
  while (true) {
    let res = await iterator.next();

    if (res.value && res.value.value.toString()) {
      let jsonRes = {};
      console.log(res.value.value.toString('utf8'));

      jsonRes.Key = res.value.key;
      try {
        jsonRes.Record = JSON.parse(res.value.value.toString('utf8'));
      } catch (err) {
        console.log(err);
        jsonRes.Record = res.value.value.toString('utf8');
      }
      allResults.push(jsonRes);
    }
    if (res.done) {
      console.log('end of data');
      await iterator.close();
      console.info(allResults);
      return Buffer.from(JSON.stringify(allResults));
    }
```

```
    }

  }
```

The function uses the shim interface function `getHistoryForKey` to return ledger asset history data for the argument passed to the getCarHistory function. getHistoryForKey returns a history of key values across time. The History db is enabled in core.yaml file (fabric-samples/config/core.yaml. You can find it enabled in part of the ledger section

```
history:
  # enableHistoryDatabase - options are true or false
  # Indicates if the history of key updates should be stored.
  # All history 'index' will be stored in goleveldb, regardless if using
  # CouchDB or alternate database for the state.
  enableHistoryDatabase: true
```

1.3. Start the Test Network

1 Start a terminal

2 `cd ~/fabric-samples/fabcar/`

3 Issue the following command which can take a minute or so to complete. The 'node' parameter will start the Node.js chaincode container vs. the Go chaincode container.

```
./startFabric.sh node
```

4 Issue the following command to download any required Node.js packages.

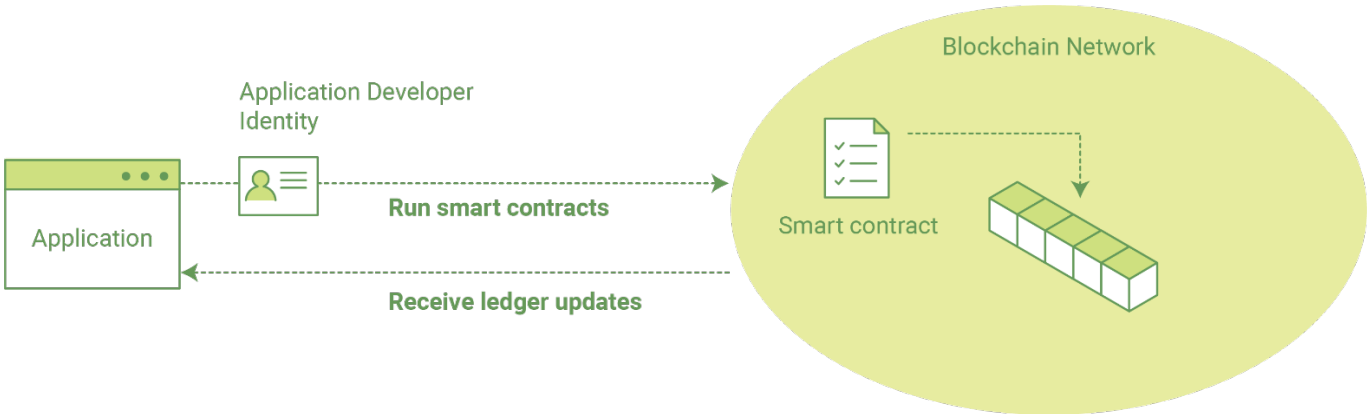
```
npm install
```

5 For the sake of brevity, we won't delve into the details of what's happening with this command. Here's a quick synopsis:

- launches a peer node, ordering node, Certificate Authority and CLI container
- creates a channel and joins the peer to the channel
- installs smart contract (i.e. chaincode) onto the peer's file system and instantiates said chaincode on the channel; instantiate starts a chaincode container
- calls the `initLedger` function to populate the channel ledger with 10 unique cars

6 These operations will typically be done by an organizational or peer admin. The script uses the CLI to execute these commands, however there is support in the SDK as well. Refer to the Hyperledger Fabric Node SDK repo for example scripts.

7 Issue a "`docker ps`" command to reveal the processes started by the `startFabric.sh` script. You can learn more about the details and mechanics of these operations in the Building Your First Network section. Here we'll just focus on the application. The following picture provides a simplistic representation of how the application interacts with the Hyperledger Fabric network.



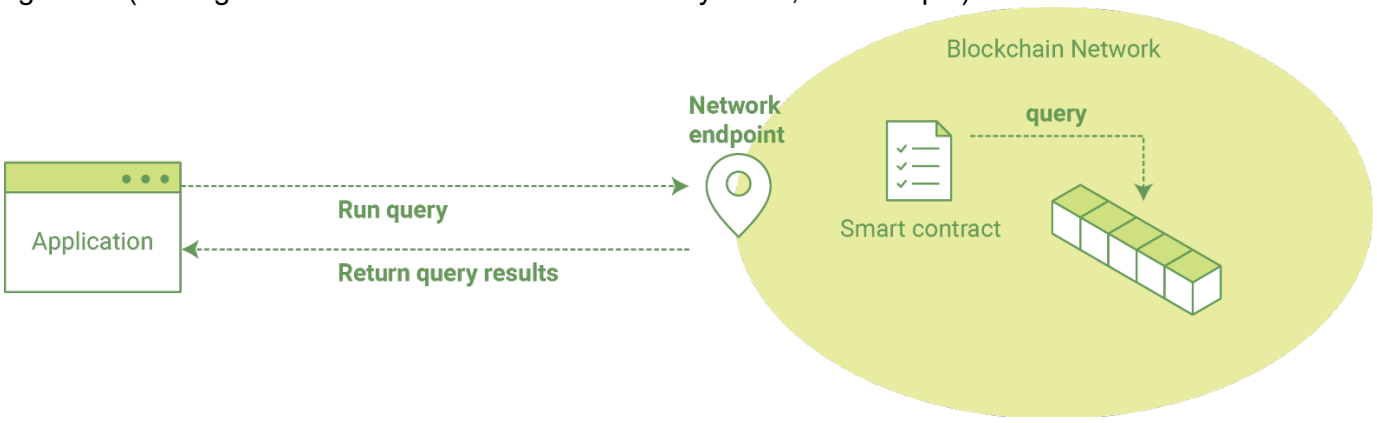
1.4. How Applications Interact with the Network

Applications use **APIs** to invoke smart contracts (referred to as "chaincode"). These smart contracts are hosted in the network and identified by name and version. For example, our chaincode container is titled - `dev-peer0.org1.example.com-fabcar-1.0` - where the name is `fabcar`, the version is `1.0` and the peer it is running against is `dev-peer0.org1.example.com`.

APIs are accessible with a software development kit (SDK). For purposes of this exercise, we'll be using the Hyperledger Fabric Node SDK though there is also a Java SDK and CLI that can be used to develop applications.

Querying the Ledger

Queries are how you read data from the ledger. You can query for the value of a single key, multiple keys, or – if the ledger is written in a rich data storage format like JSON – perform complex searches against it (looking for all assets that contain certain keywords, for example).



As we said earlier, our sample network has an active chaincode container and a ledger that has been primed with 10 different cars. We also have some sample JavaScript code - `query.js` - in the `fabcar` directory that can be used to query the ledger for details on the cars.

1 When we launched our network, an admin user – admin – was registered with our Certificate Authority. Now we need to send an enrollment call to the CA server and retrieve the enrollment certificate (eCert) for this user. We won't delve into enrollment details here, but suffice it to say that the SDK and by extension our applications need this cert in order to form a user object for the admin. We will then use this admin object to subsequently register and enroll a new user. Send the admin

enroll call to the CA server.

```
node enrollAdmin.js
```

This program will invoke a certificate signing request (CSR) and ultimately output an eCert and key material into a newly created folder – hfc-key-store – at the root of this project. Our apps will then look to this location when they need to create or load the identity objects for our various users.

2 With our newly generated admin eCert, we will now communicate with the CA server once more to register and enroll a new user. This user – user1 – will be the identity we use when querying and updating the ledger. It's important to note here that it is the admin identity that is issuing the registration and enrollment calls for our new user (i.e. this user is acting in the role of a registrar). Send the register and enroll calls for user1:

```
node registerUser.js
```

Similar to the admin enrollment, this program invokes a CSR and outputs the keys and eCert into the hfc-key-store subdirectory. So now we have identity material for two separate users – admin & user1. Time to interact with the ledger…

3 Use VSCode to open the `query.js` program with "`code query.js`" and edit the request in query.js by entering *node query.js* to look as the follows in order to query all cars. This request was changed in the previous lab. Be sure to save the file.

```
const request = {
   //targets : --- letting this default to the peers assigned to the
   channel
   chaincodeId: 'fabcar',
   fcn: 'queryAllCars',
   args: ['']
```

4 Now we can run our javascript programs. First, let's run our `query.js` program to return a listing of all the cars on the ledger. A function that will query all the cars, `queryAllCars`, is pre-loaded in the app, so we can simply run the program as is:

```
node query.js
```

5 It should return something like this:

```
Query result count =  1
Response is  [{"Key":"CAR0",
"Record":{"colour":"blue","make":"Toyota","model":"Prius","owner":"Tomoko"}},
{"Key":"CAR1",   "Record":{"colour":"red","make":"Ford","model":"Mustang","owner":"Brad"}},
{"Key":"CAR2", "Record":{"colour":"green","make":"Hyundai","model":"Tucson","owner":"Jin Soo"}},
{"Key":"CAR3", "Record":{"colour":"yellow","make":"Volkswagen","model":"Passat","owner":"Max"}},
{"Key":"CAR4", "Record":{"colour":"black","make":"Tesla","model":"S","owner":"Adriana"}},
{"Key":"CAR5", "Record":{"colour":"purple","make":"Peugeot","model":"205","owner":"Michel"}},
{"Key":"CAR6", "Record":{"colour":"white","make":"Chery","model":"S22L","owner":"Aarav"}},
{"Key":"CAR7", "Record":{"colour":"violet","make":"Fiat","model":"Punto","owner":"Pari"}},
{"Key":"CAR8", "Record":{"colour":"indigo","make":"Tata","model":"Nano","owner":"Valeria"}},
{"Key":"CAR9", "Record":{"colour":"brown","make":"Holden","model":"Barina","owner":"Shotaro"}}]
```

These are the 10 cars. A black Tesla Model S owned by Adriana, a red Ford Mustang owned by Brad, a violet Fiat Punto owned by someone named Pari, and so on. The ledger is key/value based

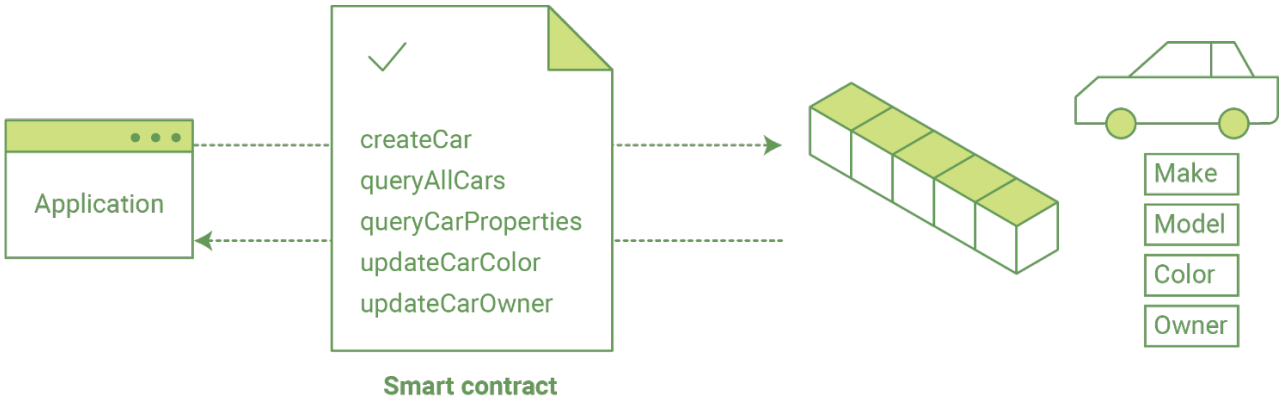
and in our implementation the key is `CAR0` through `CAR9`. This will become particularly important in a moment.

6 To take a look at the other chaincode functions, open `fabcar.js` in VSCode with "`code ../chaincode/fabcar/nodefabcar.js`". You'll see that we have the following functions available to call - `initLedger`, `queryCar`, `queryAllCars`, `createCar`, `changeCarOwner`, and getCarHistory. Let's take a closer look at the `queryAllCars` function to see how it interacts with the ledger.

```
async queryAllCars(stub, args) {

    let startKey = 'CAR0';
    let endKey = 'CAR999';

    let iterator = await stub.getStateByRange(startKey, endKey);

    let allResults = [];
    while (true) {
      let res = await iterator.next();

      if (res.value && res.value.value.toString()) {
        let jsonRes = {};
        console.log(res.value.value.toString('utf8'));

        jsonRes.Key = res.value.key;
        try {
          jsonRes.Record =
JSON.parse(res.value.value.toString('utf8'));
        } catch (err) {
          console.log(err);
          jsonRes.Record = res.value.value.toString('utf8');
        }
        allResults.push(jsonRes);
      }
      if (res.done) {
        console.log('end of data');
        await iterator.close();
        console.info(allResults);
        return Buffer.from(JSON.stringify(allResults));
      }
    }
  }
```

The function uses the shim interface function `getStateByRange` to return ledger data between the args of `startKey` and `endKey`. Those keys are defined as `CAR0` and `CAR999` respectively. Therefore, we could theoretically create 1,000 cars (assuming the keys are tagged properly) and a `queryAllCars` would reveal every one.

Below is a representation of how an app would call different functions in chaincode.



7 We can see our `queryAllCars` function up there, as well as one called `createCar` that will allow us to update the ledger and ultimately append a new block to the chain. But first, let's do another query.

8 Go back to the `query.js` program and edit the request to query a specific car. We'll do this by changing the function from `queryAllCars` to `queryCar` and passing a specific "Key" to the args parameter. Let's use `CAR4` here. So our edited `query.js` program should now contain the following:

```
const request = {
      chaincodeId: 'fabcar',
      fcn: 'queryCar',
      args: ['CAR4']
```

9 Save the program and navigate back to your `fabcar` directory. Now run the program again:

```
node query.js
```

10 You should see the following:

```
{"colour":"black","make":"Tesla","model":"S","owner":"Adriana"}
```

So we've gone from querying all cars to querying just one, Adriana's black Tesla Model S. Using the `queryCar` function, we can query against any key (e.g. `CAR0`) and get whatever make, model, color, and owner correspond to that car.

Great. Now you should be comfortable with the basic query functions in the chaincode, and the handful of parameters in the query program. Time to update the ledger...

1.5. Updating the Ledger

Now that we've done a few ledger queries and changes a bit of code, we're ready to update the ledger. There are a lot of potential updates we could make, but let's just create a new car for starters.

Ledger updates start with an application generating a transaction proposal. Just like query, a request is constructed to identify the channel ID, function, and specific smart contract to target for the transaction. The program then calls the `channel.sendTransactionProposal` API to send the transaction proposal to the peer(s) for endorsement.

The network (i.e. endorsing peer) returns a proposal response, which the application uses to build and sign a transaction request. This request is sent to the ordering service by calling the `channel.sendTransaction` API. The ordering service will bundle the transaction into a block and then "deliver" the block to all peers on a channel for validation. (In our case we have only the single endorsing peer.)

Finally the application uses the `channel.newChannelEventHub` API to connect to the peer's event listener port, and calls `event_hub.registerTxEvent` to register events associated with a specific transaction ID. This API allows the application to know the fate of a transaction (i.e. successfully committed or unsuccessful). Think of it as a notification mechanism.

We don't go into depth here on a transaction's lifecycle. Consult the Transaction Flow documentation for lower level details on how a transaction is ultimately committed to the ledger.

1 The goal with our initial invoke is to simply create a new asset (car in this case). We have a separate javascript program - `invoke.js` - that we will use for these transactions. Just like `query.js`, use VSCode to open the file and navigate to the code block where we construct our invocation:

```
var request = {
        //targets: let default to the peer assigned to the client
        chaincodeId: 'fabcar',
        fcn: '',
        args: [''],
        chainId: 'mychannel',
        txId: tx_id
   };
```

2 You'll see that we can call one of two functions - `createCar` or `changeCarOwner`. Let's create a red Chevy Volt and give it to an owner named Nick. We're up to `CAR9` on our ledger, so we'll use `CAR10` as the identifying key here. The updated code block should look like this:

```
var request = {
    chaincodeId: 'fabcar',
    fcn: 'createCar',
    args: ['CAR10', 'Chevy', 'Volt', 'Red', 'Nick'],
    chainId: 'mychannel',
    txId: tx_id
```

3 Save it and run the program:

```
node invoke.js
```

4 There will be some output in the terminal about Proposal Response and Transaction ID. However, all we're concerned with is this message:

```
The transaction has been committed on peer localhost:7051
```

5 The peer emits this event notification, and our application receives it thanks to our `event_hub.registerTxEvent` API. So now if we go back and edit our `query.js` program and call the `queryCar` function against an arg of `CAR10`, we should see the following:

```
Response is {"colour":"Red","make":"Chevy","model":"Volt","owner":"Nick"}
```

6 Finally, let's change ownership of the car to establish a history for the vehicle - `changeCarOwner`. Nick is feeling generous and he wants to give his Chevy Volt to a man named Barry. So, we simply edit `invoke.js` to reflect the following:

```
var request = {
   //targets: let default to the peer assigned to the client
   chaincodeId: 'fabcar'
   fcn: 'changeCarOwner',
   args: ['CAR10', 'Barry'],
   chainId: 'myChannel',
   txId: tx_id
```

7 Execute the program again - `node invoke.js` - and then run the query app one final time. We are still querying against `CAR10`, so we should see:

```
Response is {"colour":"Red","make":"Chevy","model":"Volt","owner":"Barry"}
```

8 Finally, go back to the query.js program and edit the request to query the history for a specific car. We'll do this by changing the function from queryCar to getCarHistory and passing a specific "Key" to the args parameter. Let's use CAR10 here. So our edited query.js program should now contain the following:

```
const request = {
      chaincodeId: 'fabcar',
      fcn: 'getCarHistory',
      args: ['CAR10']
```

Run node query.js to rerun the query once again. This time the results will show the complete history for the CAR10 asset. The history includes an initial record for the asset creation and an additional record for when we modified CAR10 to have an owner of Barry.

```
Response is
[{"Record":{"docType":"car","make":"Chevy","model":"Volt","color":"Red","owner":"Nick
"}},{"Record":{"color":"Red","docType":"car","make":"Chevy","model":"Volt","owner":"B
arry"}}]
```

1.6. Lab Cleanup

1 Start a terminal and change to the first-network project.

```
cd ~/fabric-samples/first-network
```

2 Issue the following command to shutdown any existing blockchain networks. Answer 'Y' to the

Continue prompt.

```
./byfn.sh down
```

3 Issue the following command to kill any active or stale containers:

```
docker rm -f $(docker ps -aq)
```

4 Issue the following command to clear any cached networks. Answer 'Y' to the Continue prompt.

```
docker network prune
```

5 Close VSCode by choosing File->Exit from the VSCode menu bar.

Appendix A. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be

the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental. All references to fictitious companies or individuals are used for illustration purposes only.

Appendix B. Trademarks and copyrights

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	AIX	CICS	ClearCase	ClearQuest	Cloudscape
Cube Views	DB2	developerWorks	DRDA	IMS	IMS/ESA
Informix	Lotus	Lotus Workflow	MQSeries	OmniFind	
Rational	Redbooks	Red Brick	RequisitePro	System i	
System z	*Tivoli*	*WebSphere*	*Workplace*	*System p*	

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of The Minister for the Cabinet Office, and is registered in the U.S. Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

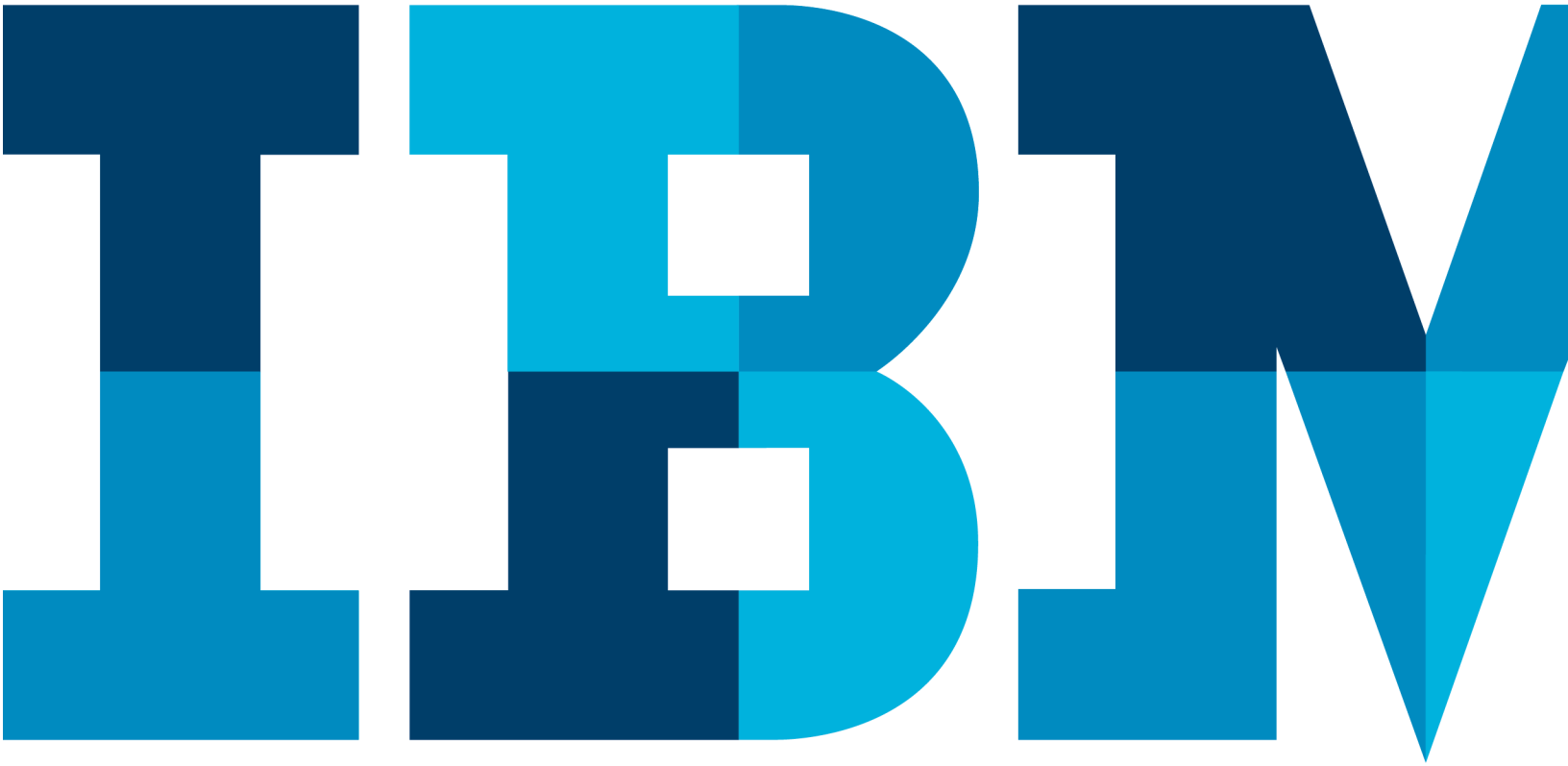
Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

Please Recycle

IBM Blockchain Hands-On

VSCode Extension for IBM Blockchain Platform: Generate, Deploy, Instantiate, and Test Smart Contracts

Lab Four



Contents

Overview

The purpose of this lab is to enable you to become familiar with the new Visual Studio Code (VSCode) IBM Blockchain Platform extension to generate template smart contracts, deploy to a local Hyperledger Fabric instance, and test the smart contract using a generated test Node.js test module.

IBM Blockchain Platform Extension Overview

The IBM Blockchain Platform extension has been created to assist users in developing, testing, and deploying smart contracts; including connecting to Hyperledger Fabric environments. The IBM Blockchain Platform extension provides an explorer and the following commands in the table below accessible from the Command Palette, for developing smart contracts quickly:

Command	Description
Add Connection	Add a Hyperledger Fabric instance connection
Add Identity To Connection	Add an identity to be used when connecting to a Hyperledger Fabric instance
Connect To Blockchain	Connect to a Hyperledger Fabric blockchain using a blockchain connection
Create Smart Contract Project	Create a new JavaScript or TypeScript smart contract project
Debug	Debug a Smart Contract
Delete Connection	Delete a Hyperledger Fabric instance connection
Delete Package	Delete a smart contract package
Disconnect From Blockchain	Disconnect from the blockchain you're currently connected to
Edit Connection	Edit connection to a blockchain
Export Package	Export an already-packaged smart contract package to use outside VSCode
Generate Smart Contract Tests	Create a functional level test file for instantiated smart contracts
Install Smart Contract	Install a smart contract package onto a peer
Instantiate Smart Contract	Instantiate an installed smart contract package onto a channel **Note: This currently doesn't work with IBM Blockchain Platform Enterprise plan - Coming soon!*
Open Fabric Runtime Terminal	Open a terminal with access to the Fabric runtime (peer CLI)
Package a Smart Contract Project	Create a new smart contract package from a project in the Explorer
Refresh Blockchain Connections	Refresh the Blockchain Connections view
Refresh Smart Contract Packages	Refresh the Smart Contract Packages view
Restart Fabric Runtime	Restart a Hyperledger Fabric instance
Start Fabric Runtime	Start a Hyperledger Fabric instance
Submit Transaction	Submit a transaction to a smart contract
Teardown Fabric Runtime	Teardown the local_fabric runtime (hard reset)
Toggle Development Mode	Toggle the Hyperledger Fabric instance development mode
Upgrade Smart Contract	Upgrade an instantiated smart contract

This tutorial is an introduction to the new IBM Blockchain Platform extension which is currently under development. The tutorial will guide you through the steps to steps involved in generating a smart contract, installing and instantiating the smart contract, and then unit testing the smart contract. This lab was tested with the .6 version or higher of the extension. Please note: this extension is available for early

experimentation. There are many features and improvements to come before the v1.0 release. Please bear this in mind, and if you find something you'd like to see added, let the team know by raising a GitHub issue at
https://github.com/IBM-Blockchain/blockchain-vscode-extension/issues

The major steps you will follow for this lab are:

1. Upgrade the IBM Blockchain Platform extension. We will upgrade the existing IBM Blockchain Platform extension to the current version.

2. Starting a local Hyperledger Fabric blockchain instance. The extension enables to connect to an external Hyperledger Fabric instance or an embedded local Hyperledger Fabric instance. We will connect to the local Hyperledger Fabric instance.

3. Use the extension to generate a template smart contract. We will generate a sample smart contract template.

4. Package, deploy, and instantiate the smart contract. We will package, deploy, and instantiate the smart contract using the extension.

5. Unit test the smart contract without blockchain. We will unit test the smart contract using a generated test module.

6. Test smart contract with the embedded Hyperledger Fabric runtime. We will test the smart contract using the embedded Hyperledger Fabric runtime.

7. Generate a test client for the smart contract. We will generate a test client that can be used for regression testing the smart contract or as the basis for your own application client.

8. Debug the smart contract (Optional). We will debug a smart contract setting a breakpoint.

9. Lab Cleanup. Disconnect from the embedded Hyperledger Fabric runtime and tear down the environment.

After completing this tutorial, you should have a basic understanding on how to use the IBM Blockchain Platform VSCode extension.

1.1. Upgrade the IBM Blockchain Platform extension

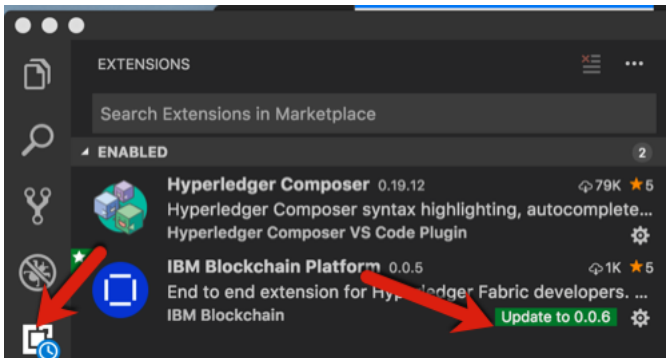
In this section you will upgrade the IBM Blockchain Platform extension to the latest version.

1 Invoke the VS Code editor from the user home directory

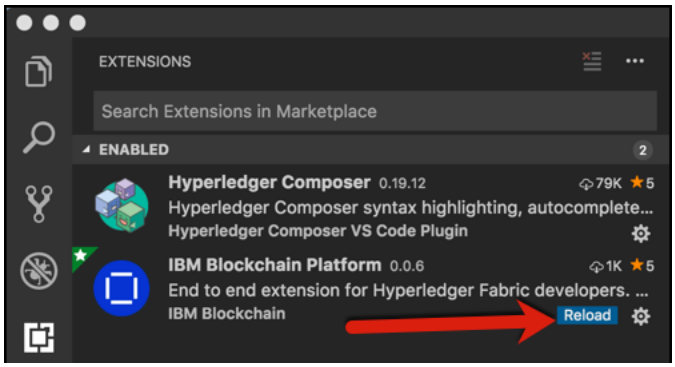
```
cd ~
code
```

2 Close any tabs that may be open from the previous lab(s).

3 Click on the extensions icon as shown below and then click on the **Update to 0.0.6** link.



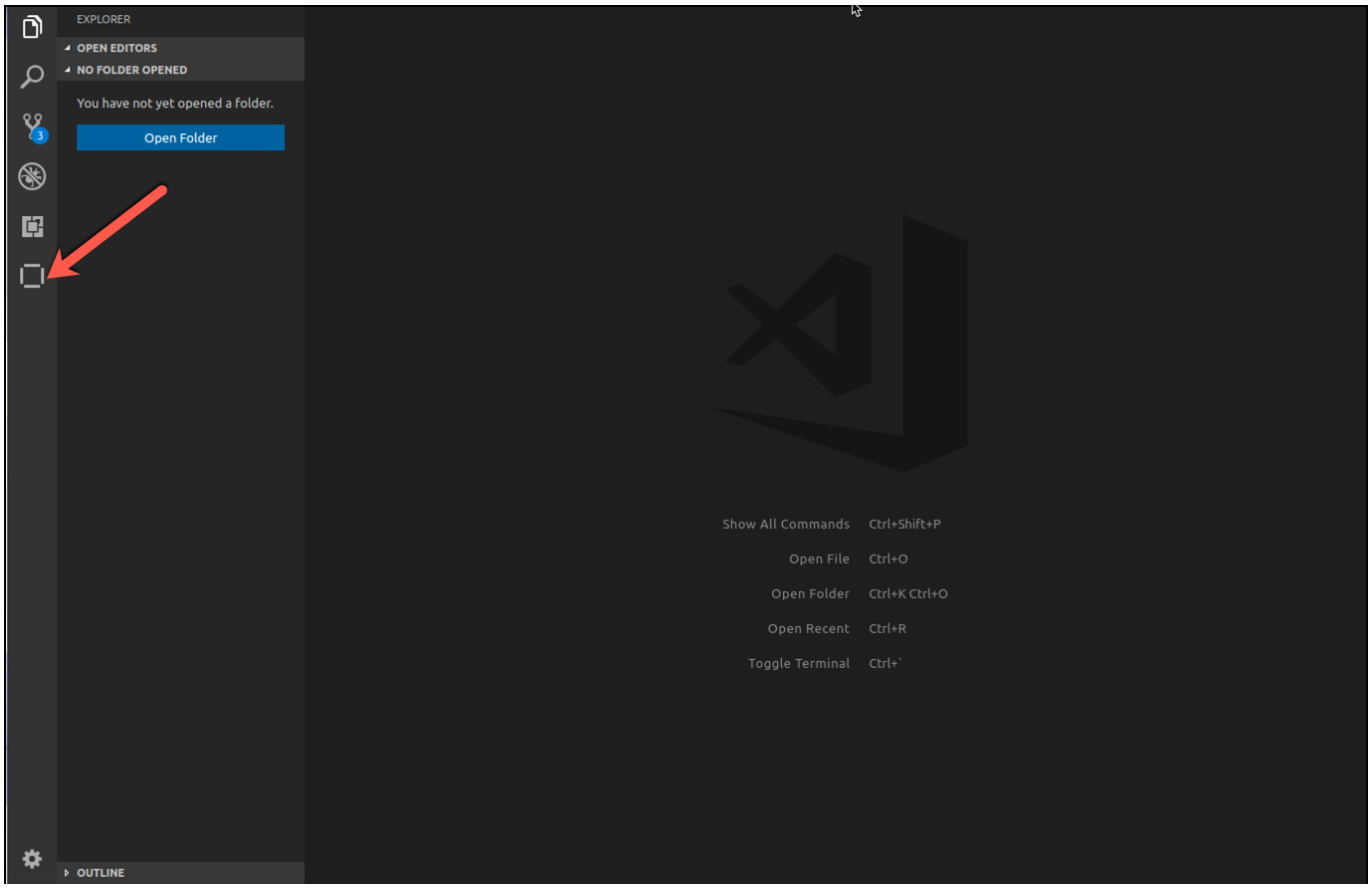
4 Reload Visual Studio Code by clicking on the **Reload** link.



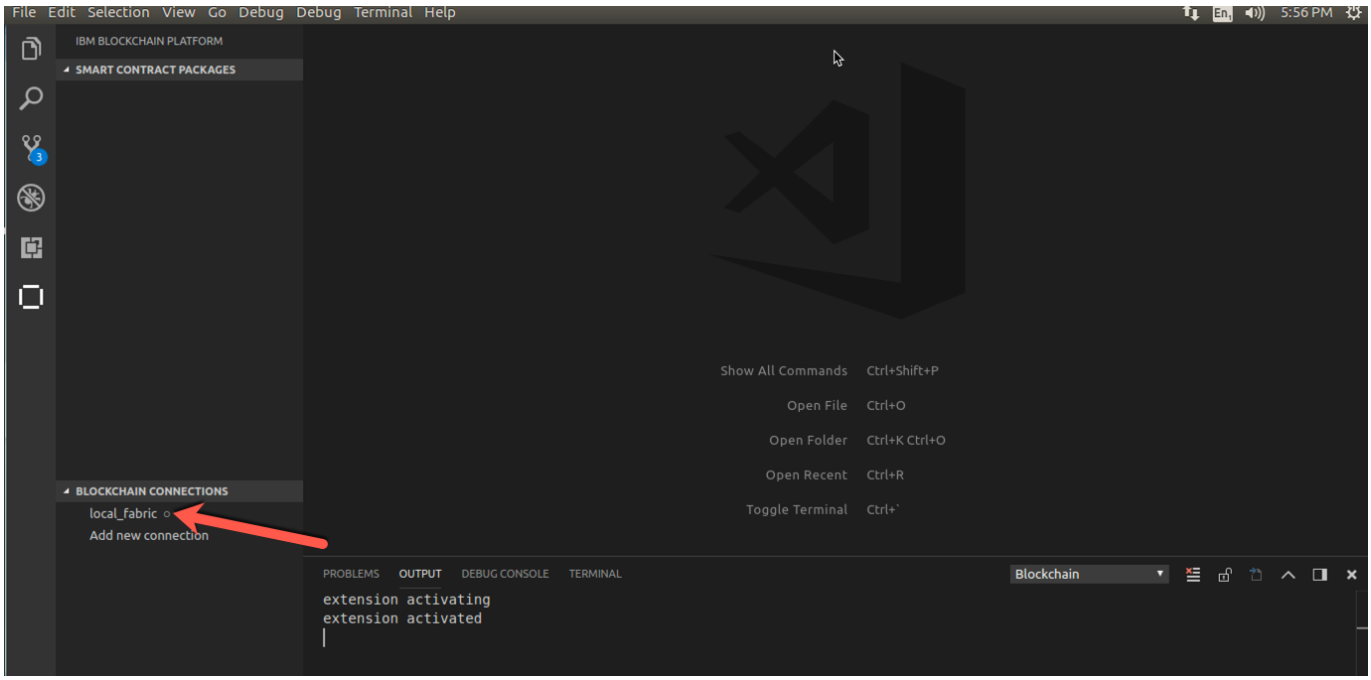
1.2. Start a local Hyperledger Fabric instance

In this section you will start and connect to a local Hyperledger Fabric instance that is automatically configured by the IBM Blockchain Platform extension.

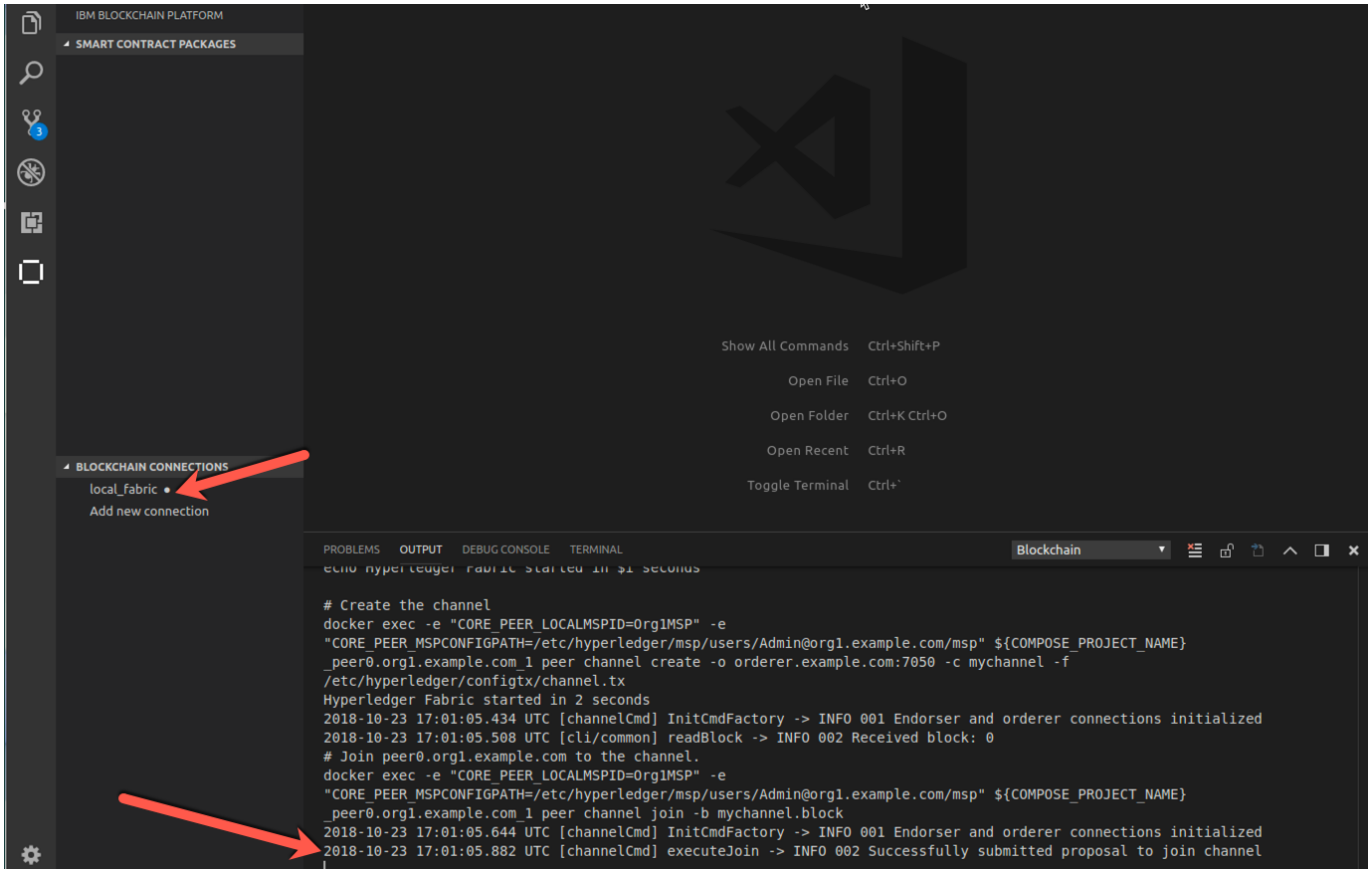
1 Click on the icon as shown below to open the IBM Blockchain Platform view.



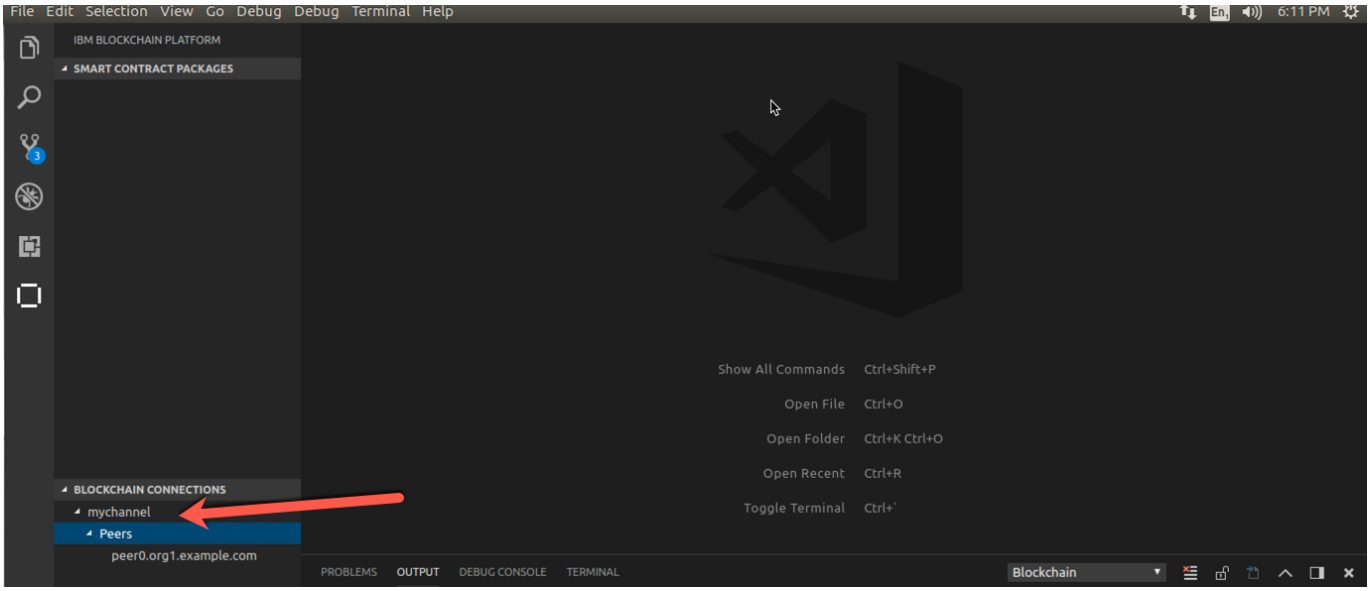
2 You should see the extension activating/activated messages below in the Output pane. Click on the **local_fabric** instance under Blockchain Connections pane to start and connect to a local Hyperledger Fabric instance.



3 The **local_fabric** instance under Blockchain Connections pane should indicate you are connected with a filled in circle to the right of the instance and the output window should indicate the local Hyperledger Fabric instance has been successfully started as indicated by the **INFO 002 Successfully submitted proposal to join channel** message. Click on the **local_fabric** instance to expand it.



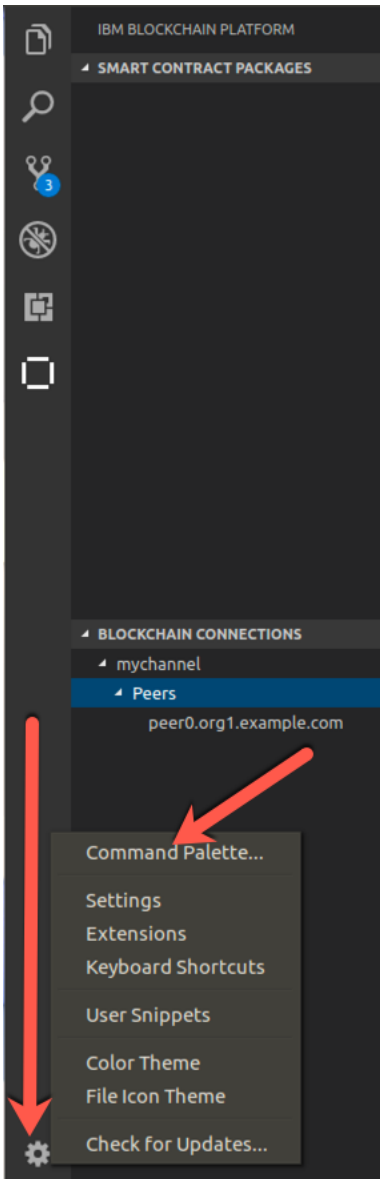
4 The **local_fabric** instance is expanded to display the configured channels and peers. In this case one channel and peer is configured.



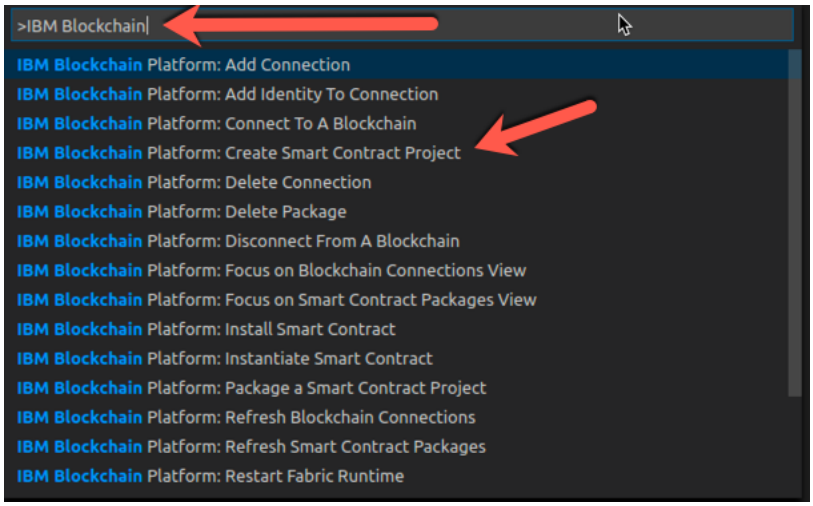
1.3. Generate a template a smart contract

In this section, we will generate a sample template smart contract.

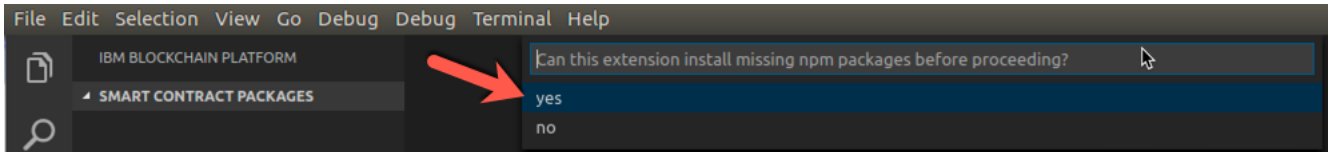
1 Click on the gear icon and then Command Palette to bring up a list of commands.



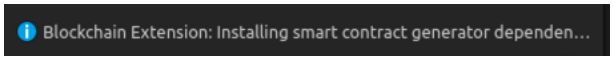
2 Enter **IBM Blockchain** in the command palette dialog to filter the search and then click on **IBM Blockchain Platform: Create Smart Contract Project**.



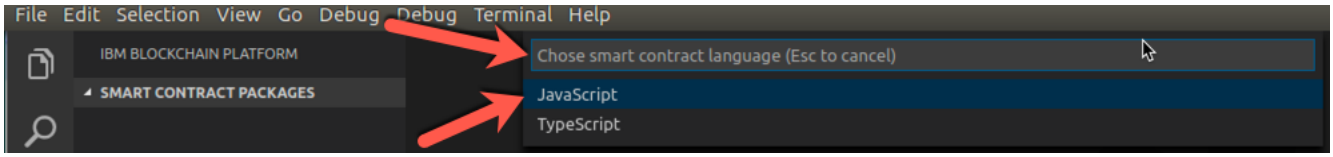
3 Click on **yes** to the **Can this extension install missing npm packages before proceeding?** dialog.



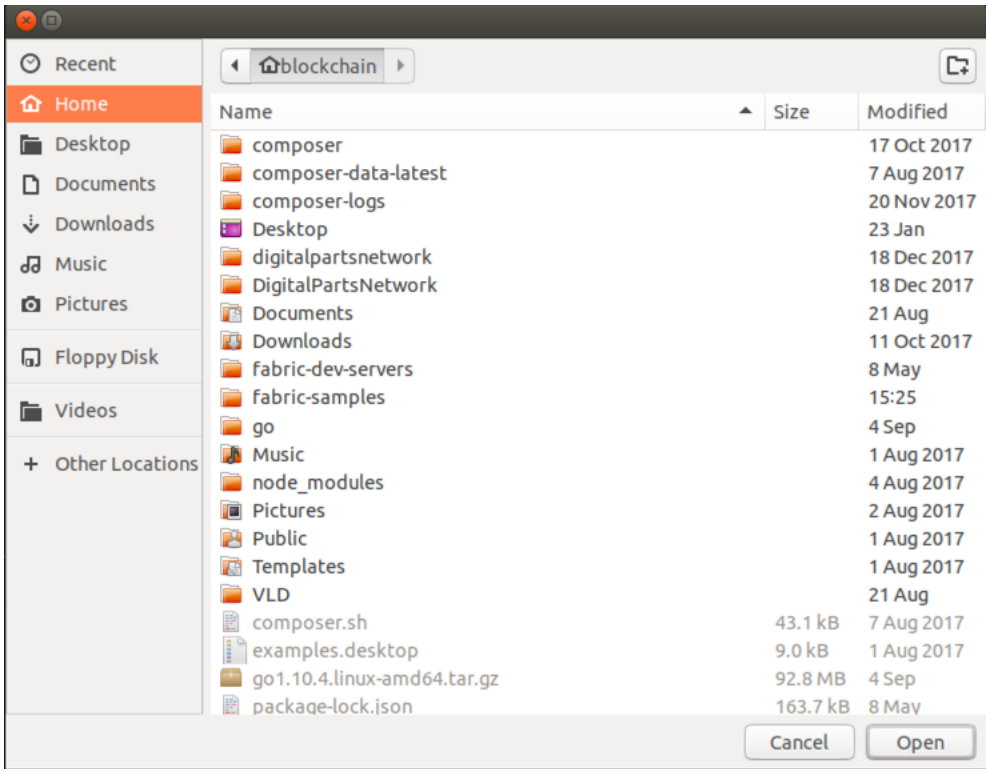
4 In lower right, you will see a message that the extension is installing smart contract generator dependencies.



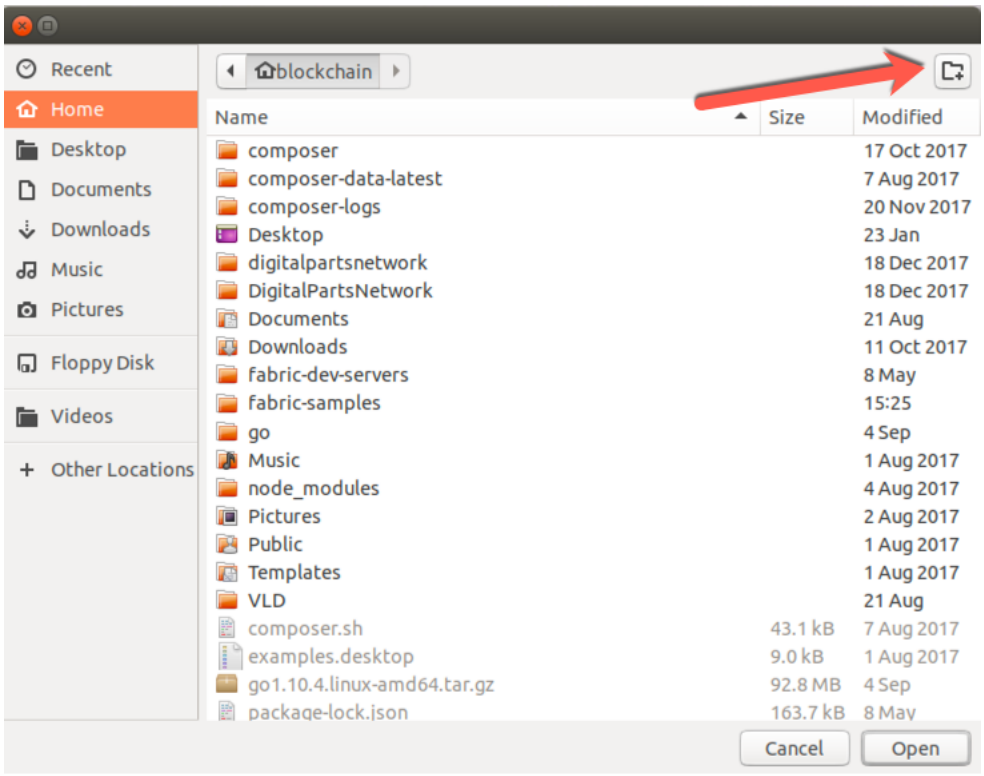
5 Click on **JavaScript** to the **Chose smart contract language (Esc to cancel)** dialog.



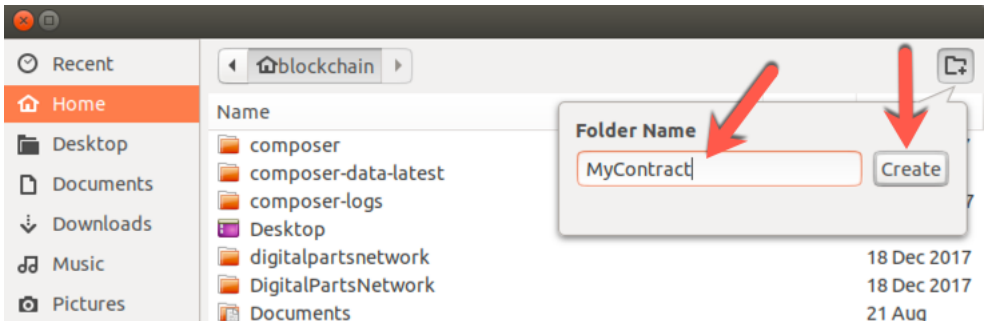
6 A file dialog will appear.



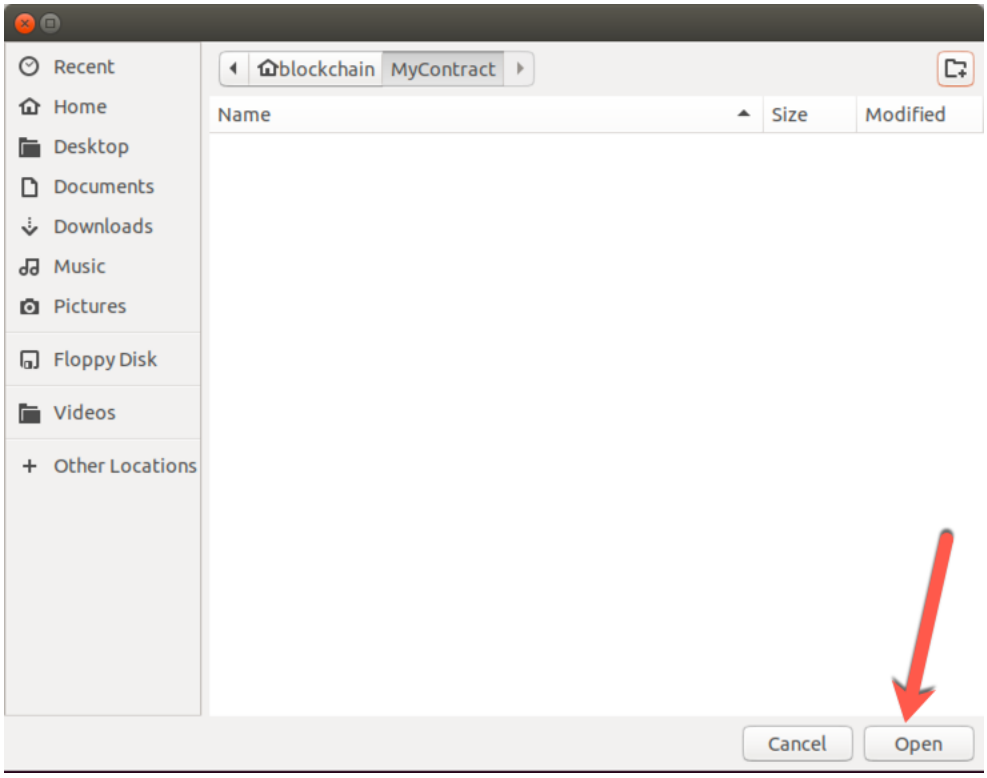
7 On the file dialog, create a new folder by clicking on the new folder icon.



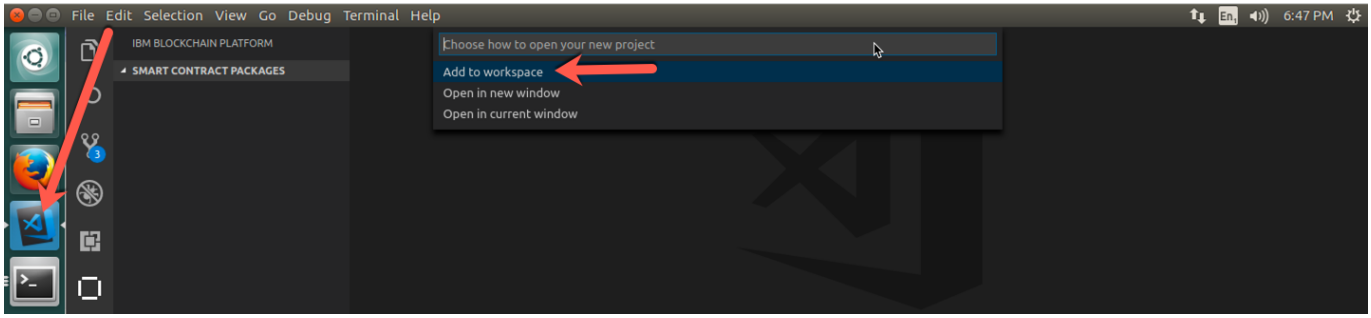
8 In the Folder Name dialog, enter **MyContract** and click the **Create** button.



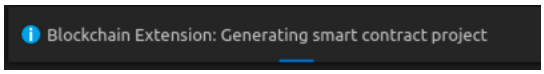
9 Click the **Open** button.



10 Click on **Add to workspace** on the **Choose how to open your new project** dialog.



11 On the lower right within VSCode, you will see the **Generating smart contract project** message. The message will disappear once the smart contract project has been generated.



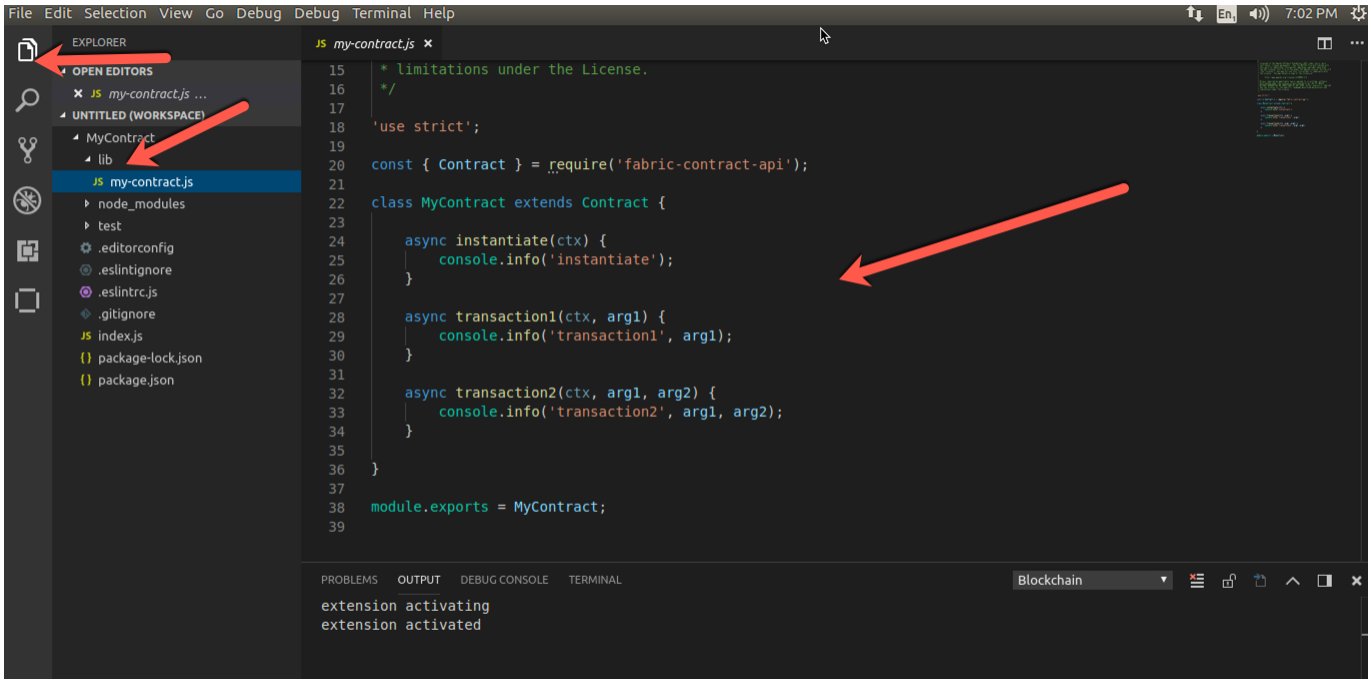
12 Click on the Explorer icon to bring up the Explorer navigation tree. Expand **MyContract->lib** and select my-contract.js to view the generated smart contract.

Notice how **MyContract** extends the Hyperledger Fabric Contract class. This built-in class was brought into scope earlier in the program:

const {Contract} = require('fabric-contract-api');

Our MyContract contract will acquire useful capabilities from the Fabric Contract class, such as automatic method invocation, a per-transaction context, transaction handlers, and class-shared state.

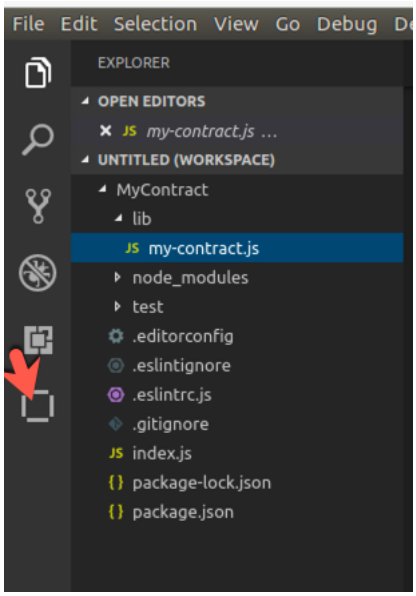
You might have noticed an extra variable in the **instantiate, transaction1, and transaction2** definitions – ctx. It's called the **transaction context**, and it's always the first variable. It contains a per-transaction data area to make it easier for transaction logic to create and recall relevant smart contract information. For example, in this case it would contain a participant's specified transaction identifier and digital identity, as well as access to the ledger API, and a temporary storage area.



1.4. Package, deploy, and instantiate the smart contract

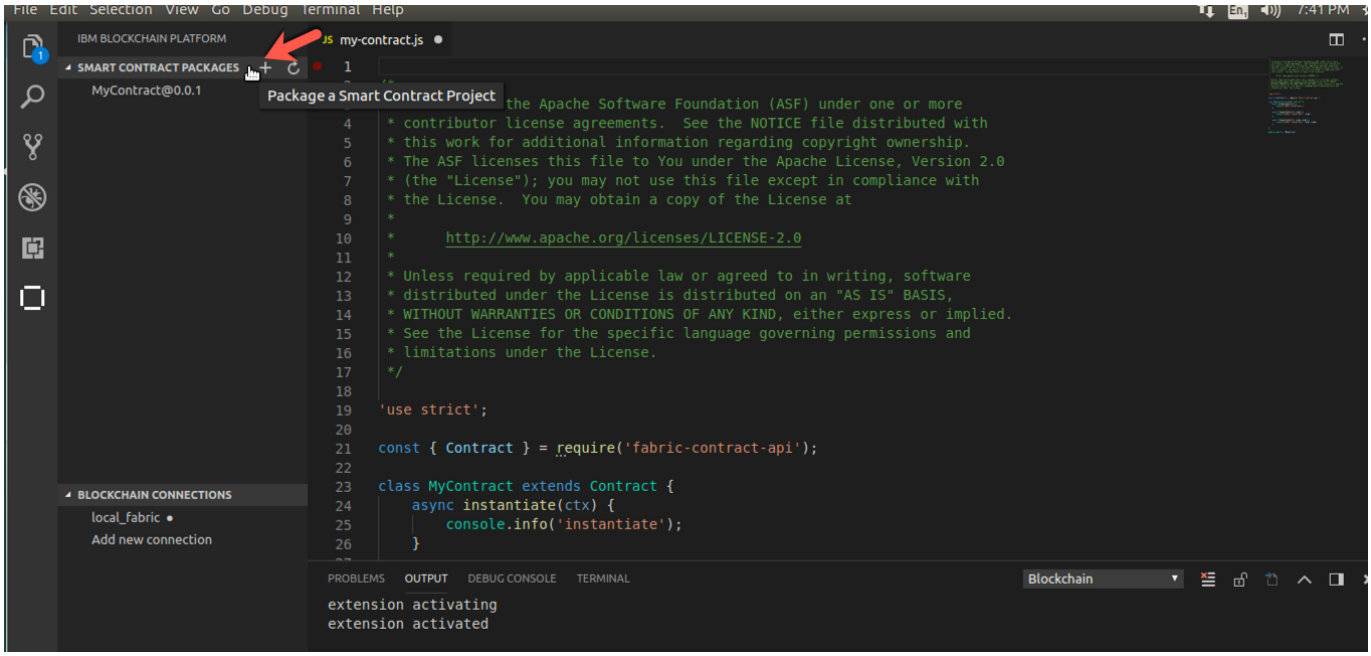
In this section, we will package, deploy, and instantiate the smart contract.

14 Click on the icon as shown below to switch back to the IBM Blockchain Platform view.

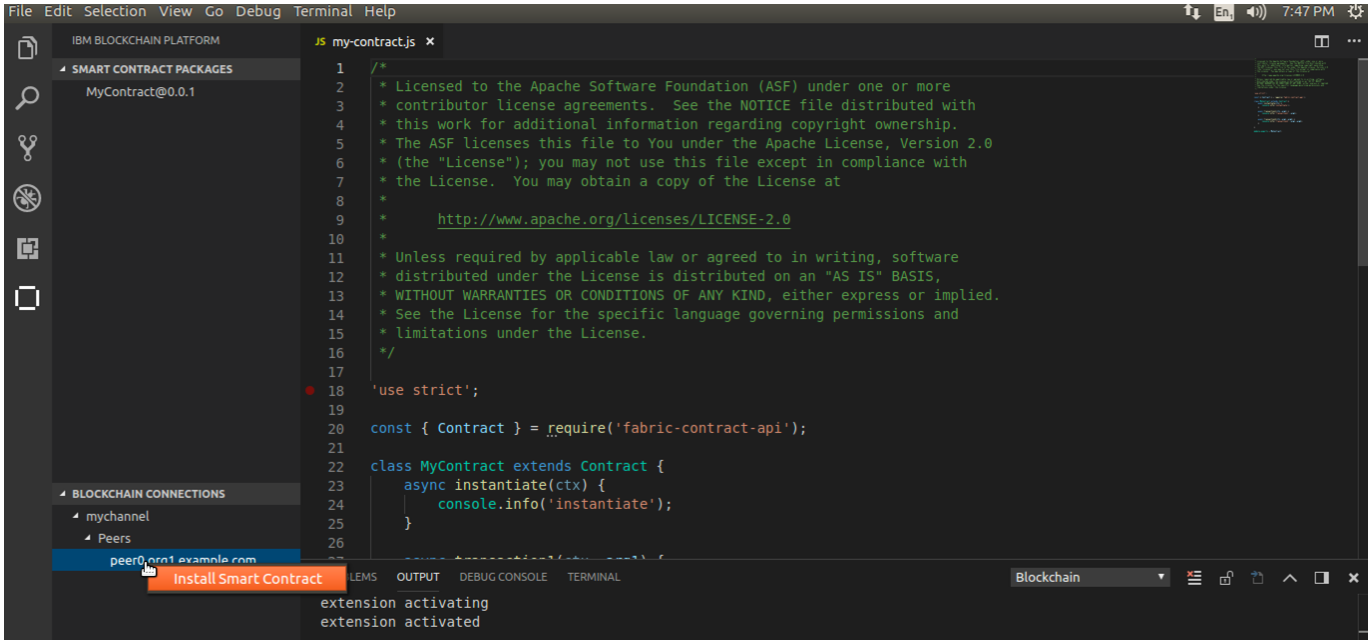


15 Click on the Plus **+** sign next to **SMART CONTRACT PACKAGES**. MyContract@0.0.1 will automatically be added.

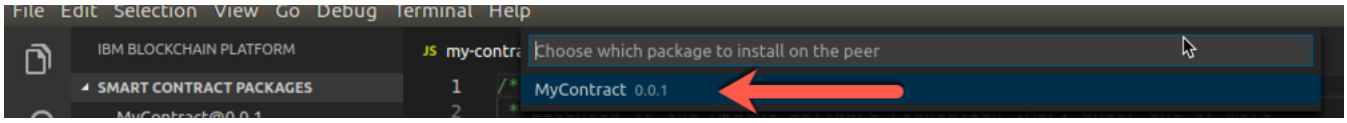
The package consists of three parts: **1.** the chaincode as defined by ChaincodeDeploymentSpec. This defines the code and other meta properties such as name and version, **2.** an instantiation policy which can be syntactically described by the same policy used for endorsement and described in endorsement-policies.rst, and **3.** a set of signatures by the entities that “own” the chaincode.



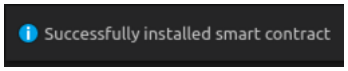
16 Deploy the smart contract. Click on **local_fabric** and expand **mychannel->Peers>peer0.org1.example.com**. Right click on the peer and select **Install Smart Contract**.



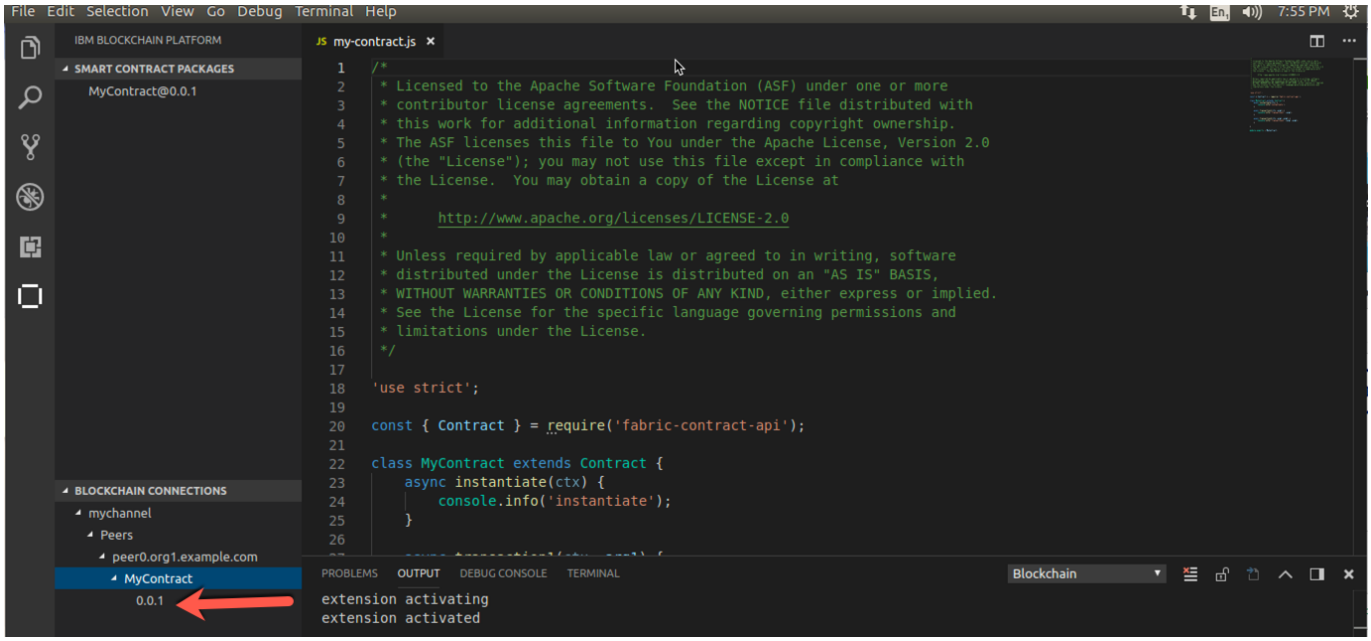
17 Select **MyContract 0.0.1** on the **Choose which package to install on the peer** dialog.



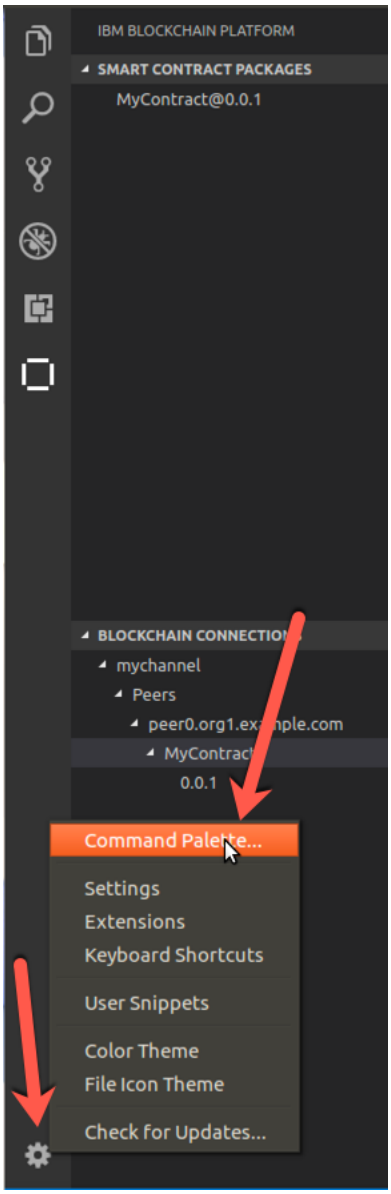
18 You will see a **Successfully installed smart contract** message on the lower right.



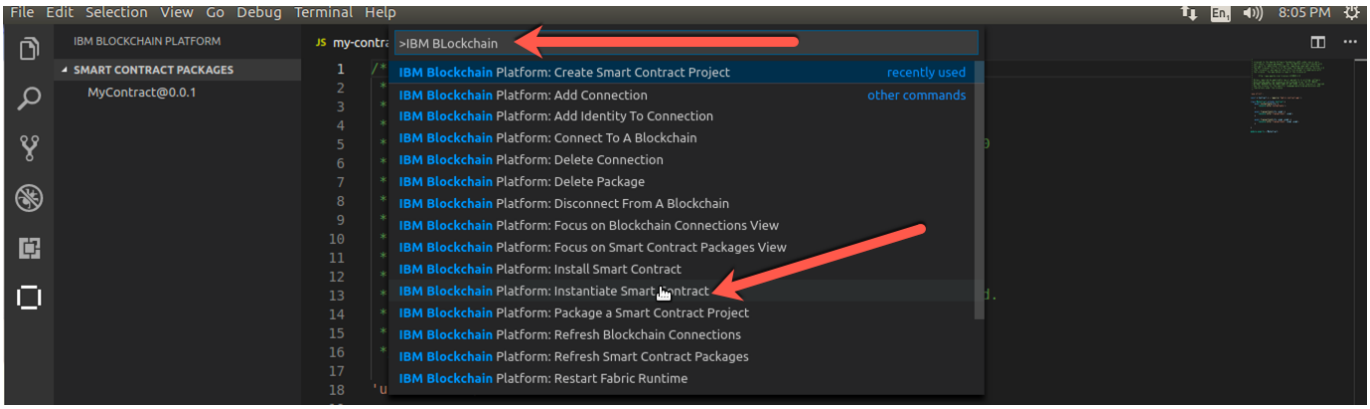
19 Expanding the peer, will show the deployed smart contract as shown below.



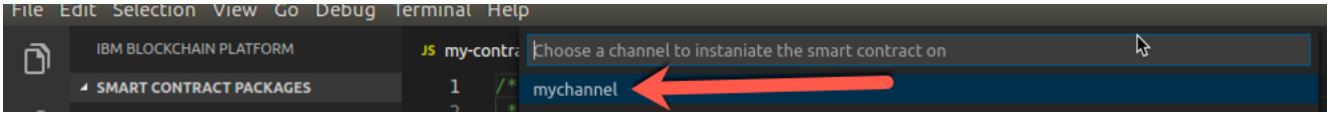
20 Click on the gear icon and then Command Palette to bring up a list of commands.



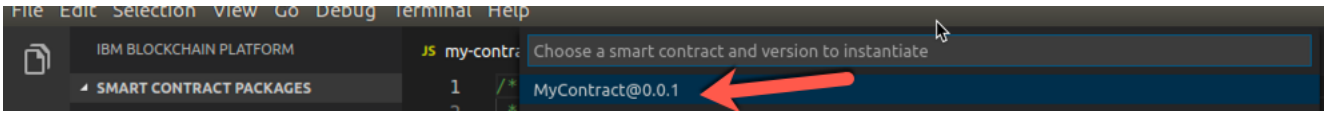
21 Instantiate the smart contract. Enter **IBM Blockchain** in the command palette dialog to filter the search and then click on **IBM Blockchain Platform: Instantiate Smart Contract**. Alternatively you can also right click on the channel under the **local-fabric** instance and choose to **Instantiate/Upgrade the Smart Contract**.



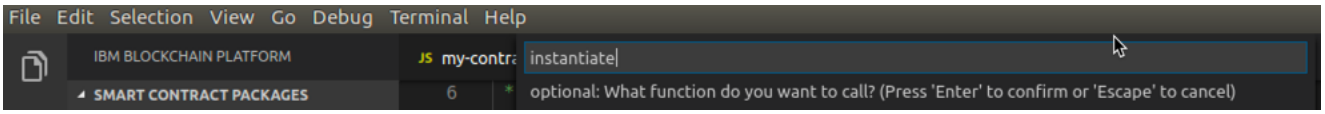
22 Click on **mychannel** to the **Choose a channel to instantiate the smart contract on smart** dialog.



23 Click on **MyContract@0.0.1** to the **Choose a smart contract and version to instantiate** dialog.

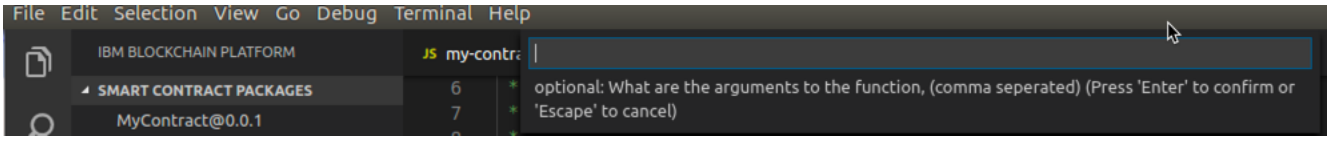


24 Enter **instantiate** to the **optional: What function to you want to call?** dialog. Then press enter.

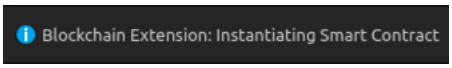


.

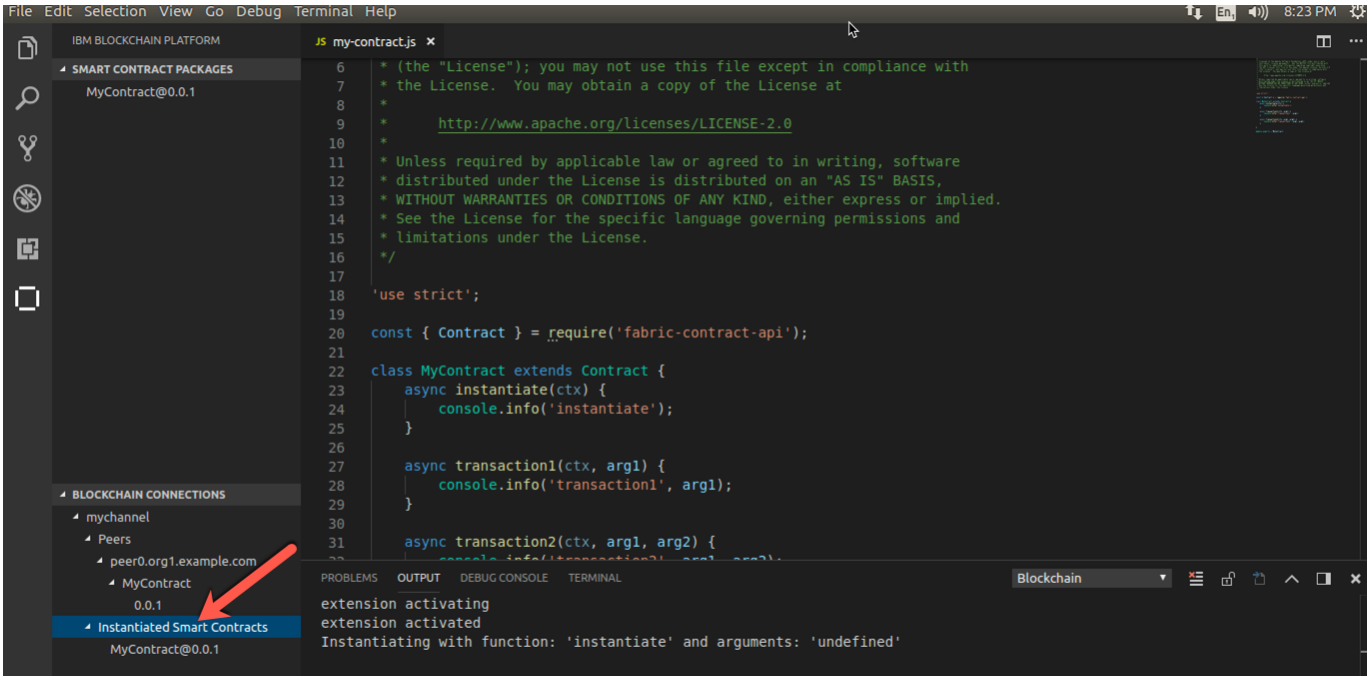
25 Press enter to the **optional: What are the arguments to the function …** dialog.



26 You will see the Blockchain Extension: Instantiating Smart Contract message on the lower right.



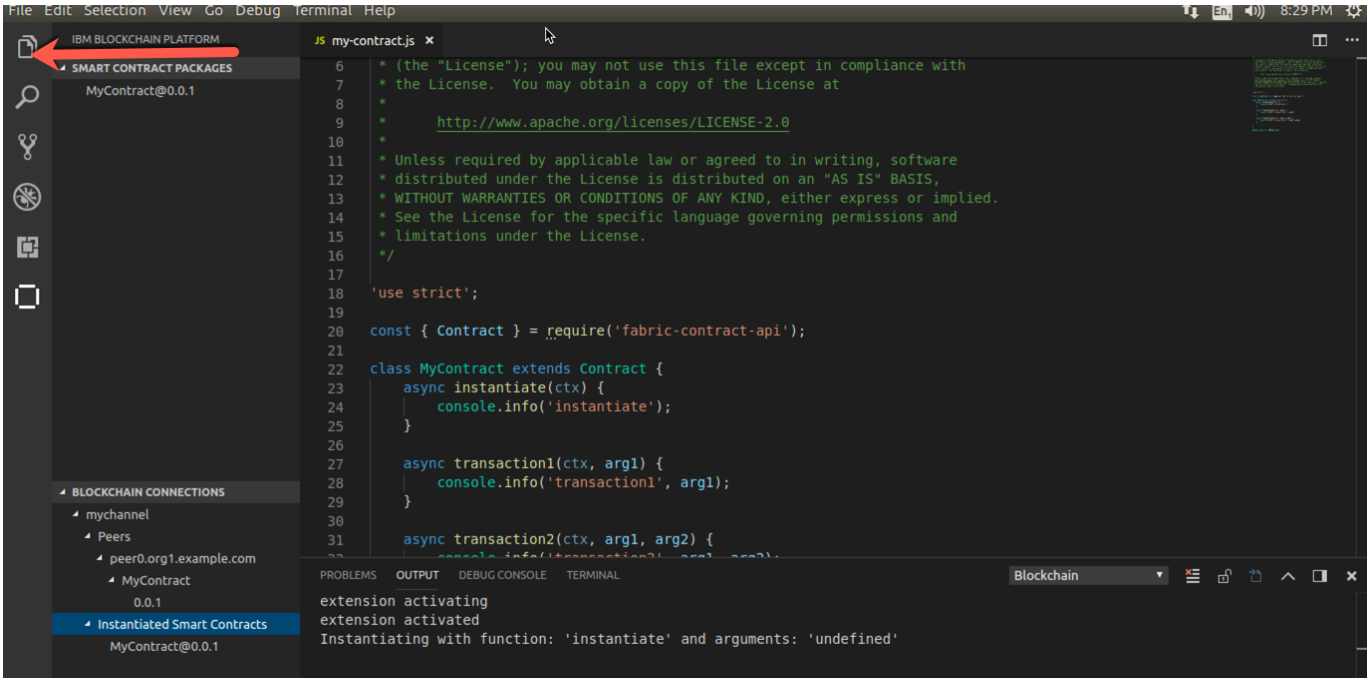
27 Once the smart contract has been successfully instantiated, you will see **Instantiated Smart Contracts** and MyContract@0.0.1 show up under **mychannel**.



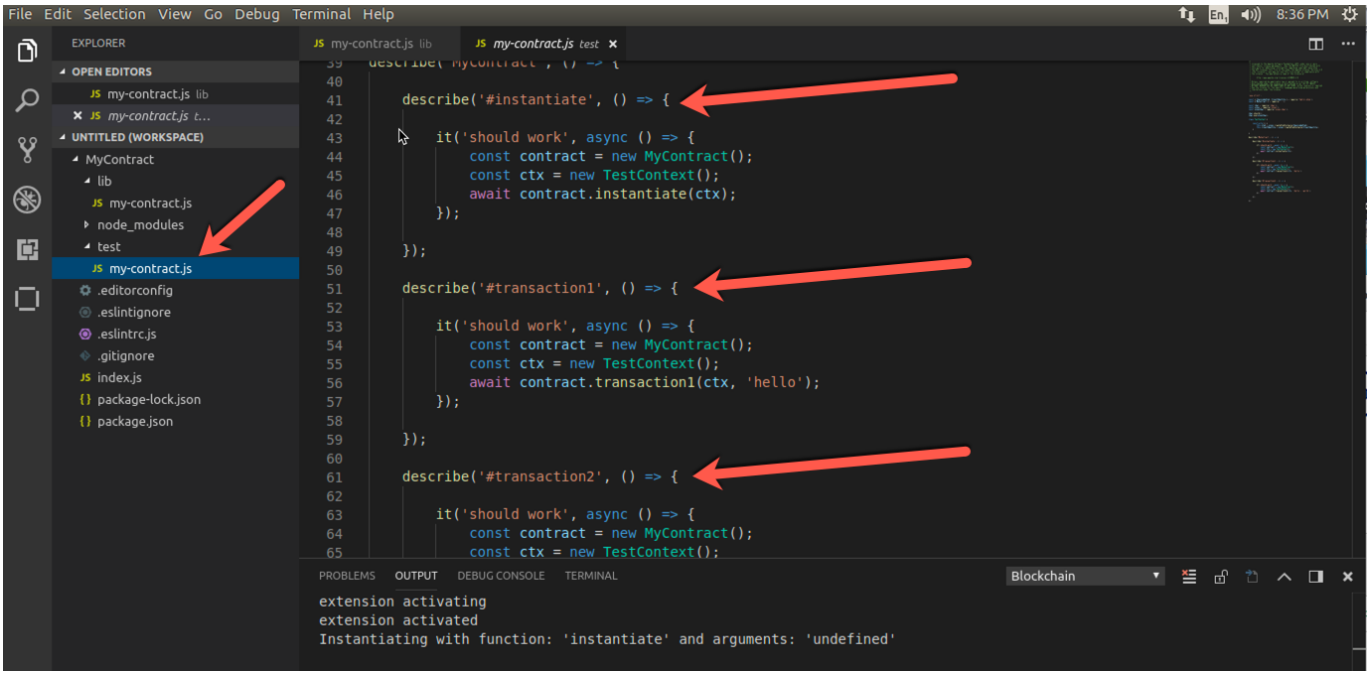
1.5. Unit test the smart contract without blockchain

In this section we will unit test the smart contract that has been generated using npm test. This will run the Node.js package's test script using the Mocha test framework as defined in the package.json file. The test module uses a stub interface to unit test the functions. This is useful for initial unit testing before connecting to real systems or blockchain network. Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

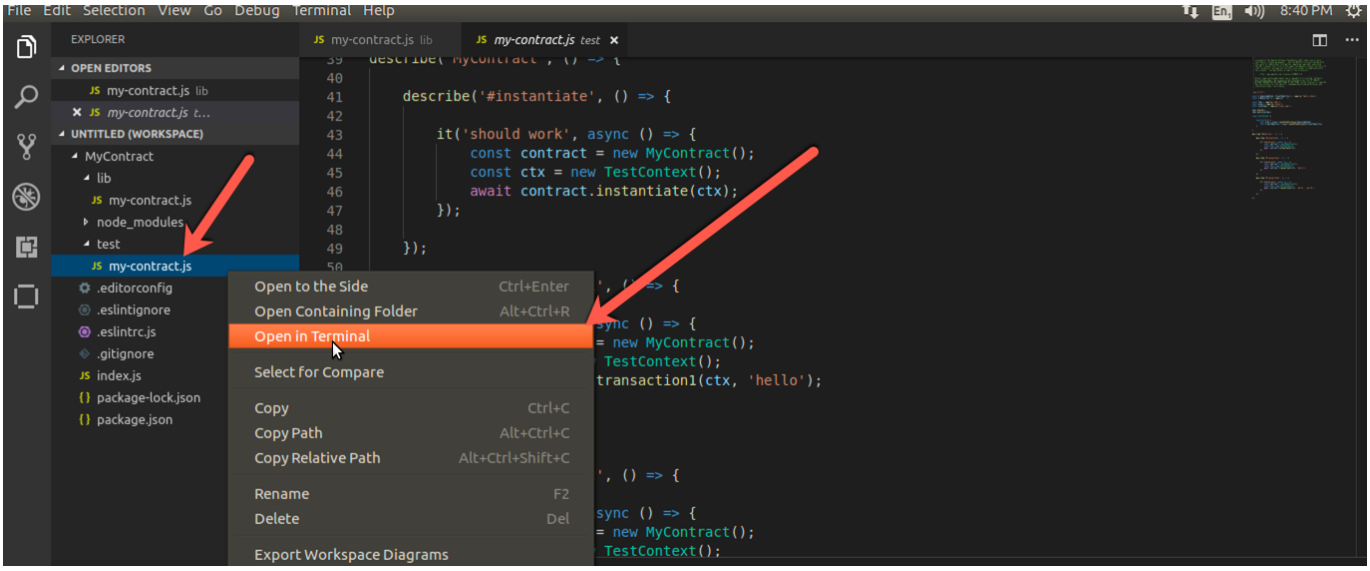
1 Switch back to the Explorer view by clicking on the Explorer icon.



2 In the smart contract navigation tree, expand **test** and click on **my-contract.js**. This is a Node.js module we will invoke with the npm test. Note the test module will exercise all three transactions.

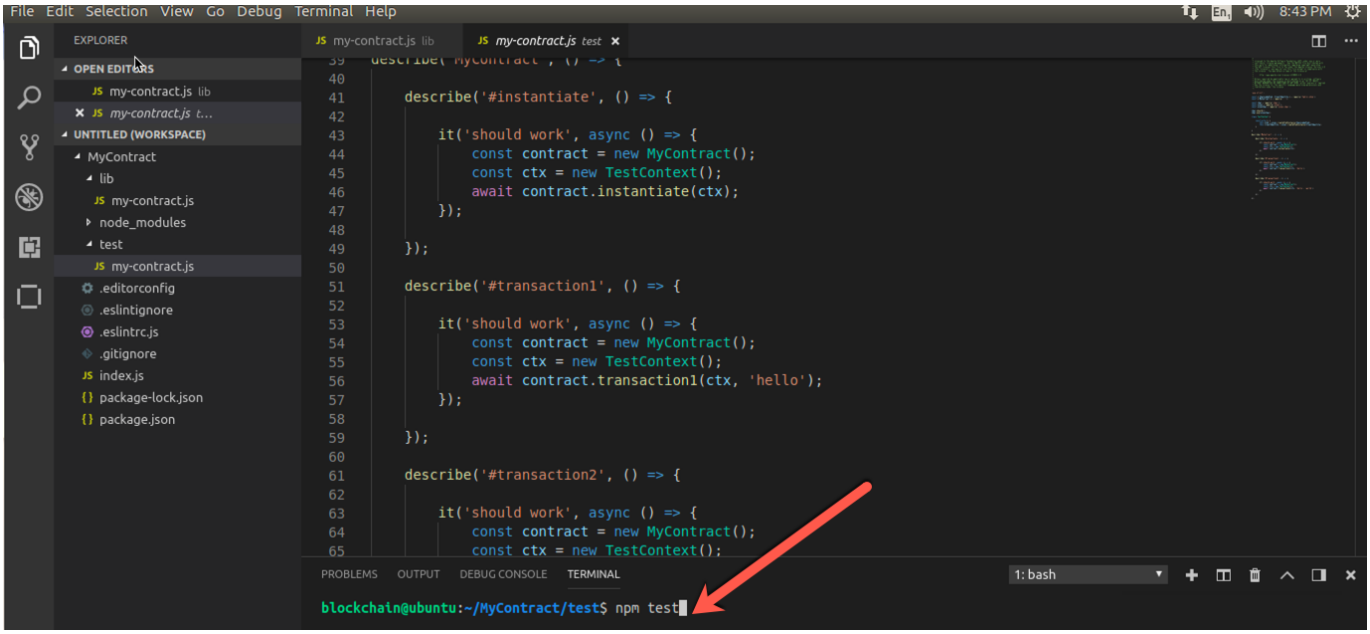


3 Right click on **my-contract.js** and select **Open in Terminal**.

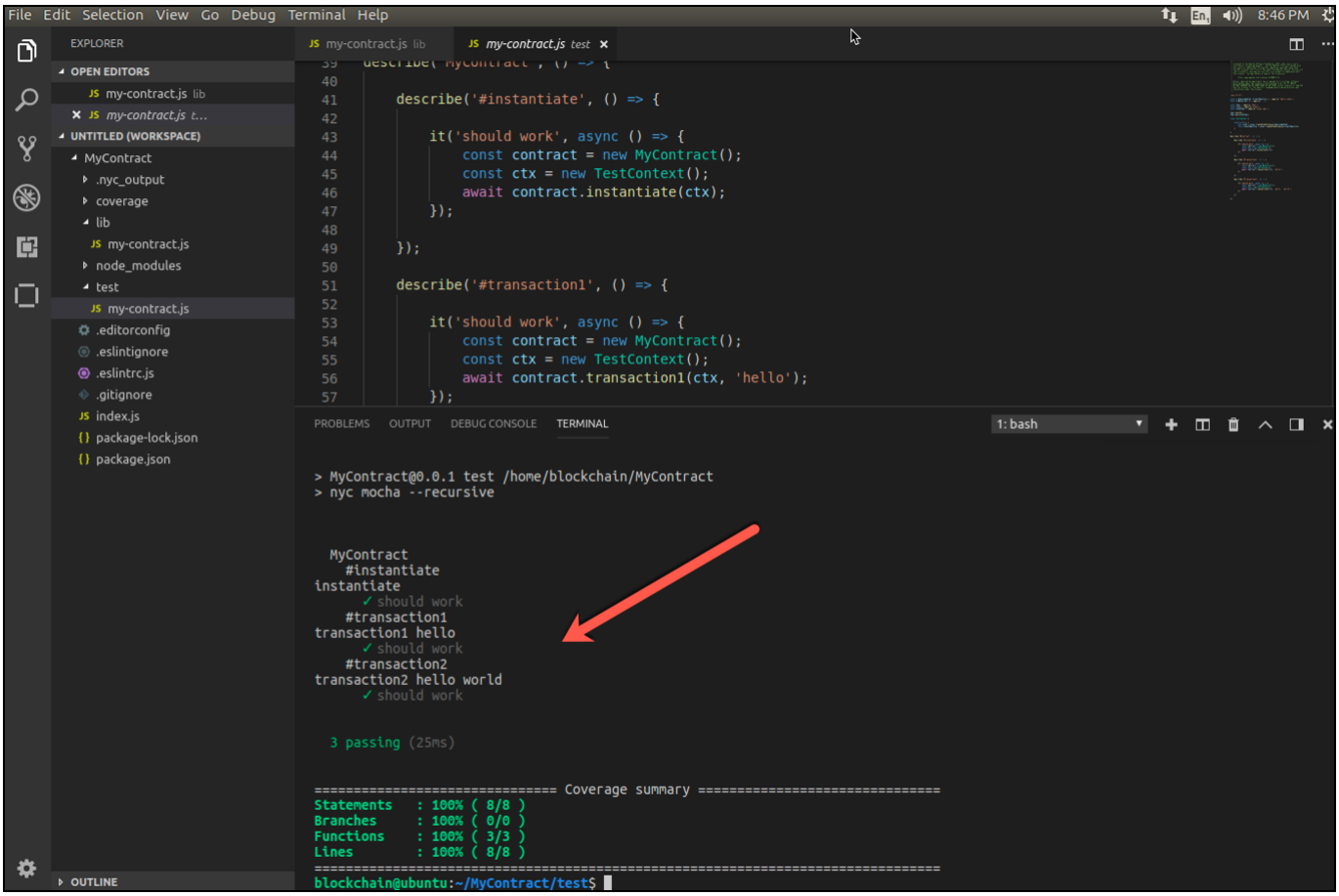


4 Enter the following on the terminal window as show below:

```
npm test
```



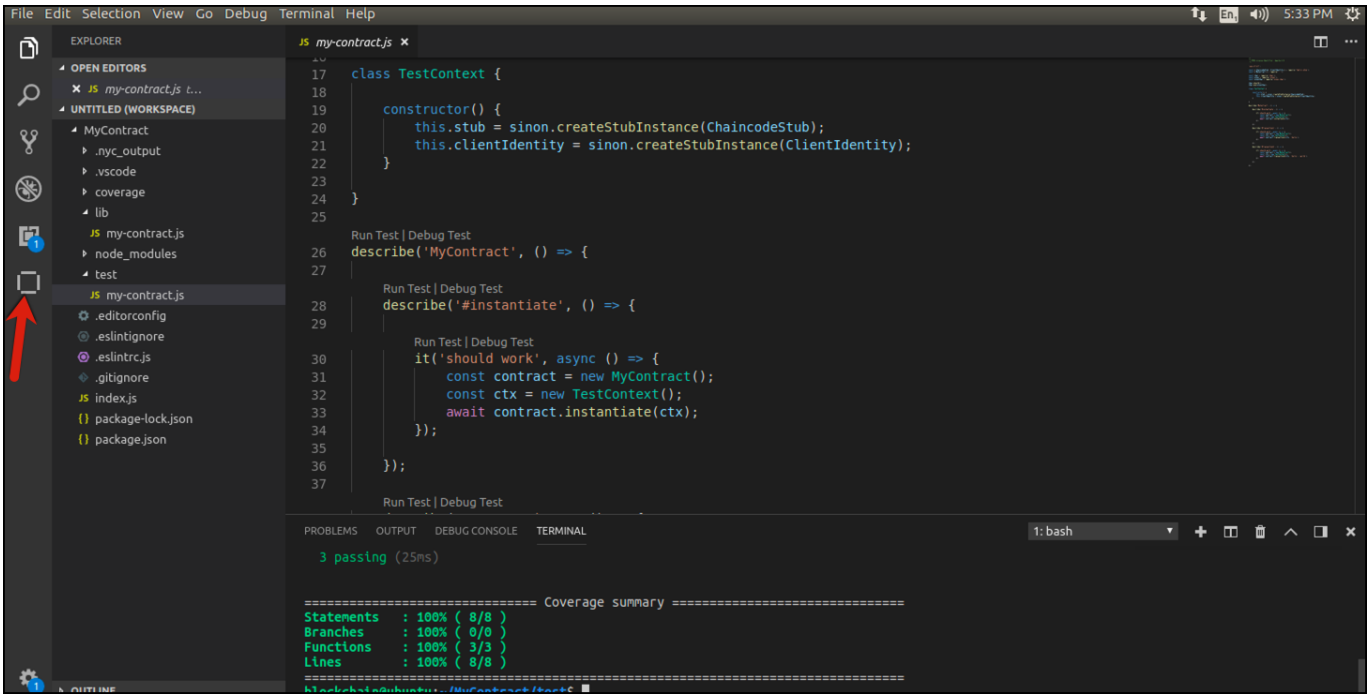
5 You should see the output below in the terminal window. All tests have succeeded.



1.6. Test smart contract using embedded Hyperledger Fabric runtime

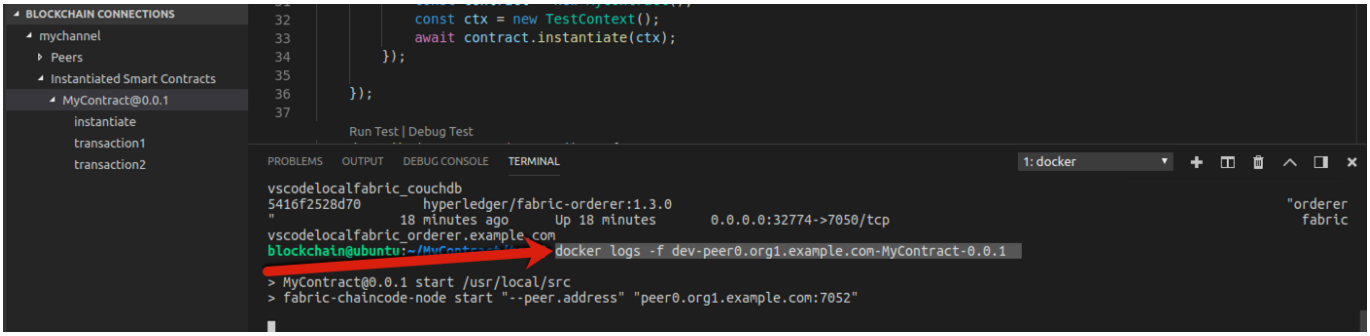
In this section we will test the smart contract that has been generated, deployed, and instantiated using the embedded Hyperledger Fabric runtime.

1 Switch back to the IBM Blockchain Platform view by clicking on the icon as shown below.

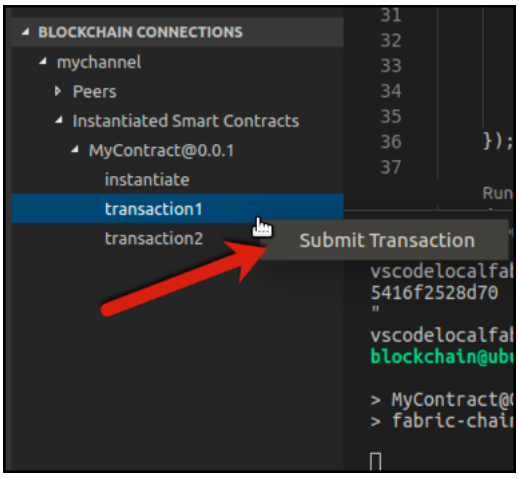


2 View the log file output from the chaincode container for the **MyContract-0.0.1** Smart Contract. In the terminal window open from the last section, enter the command below. Note, the name of the Docker chaincode container is easily determined using the docker ps command.

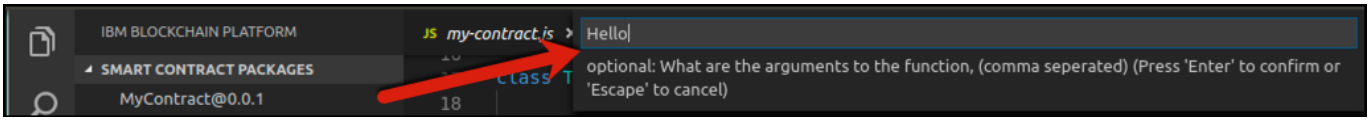
docker logs -f dev-peer0.org1.example.com-MyContract-0.0.1



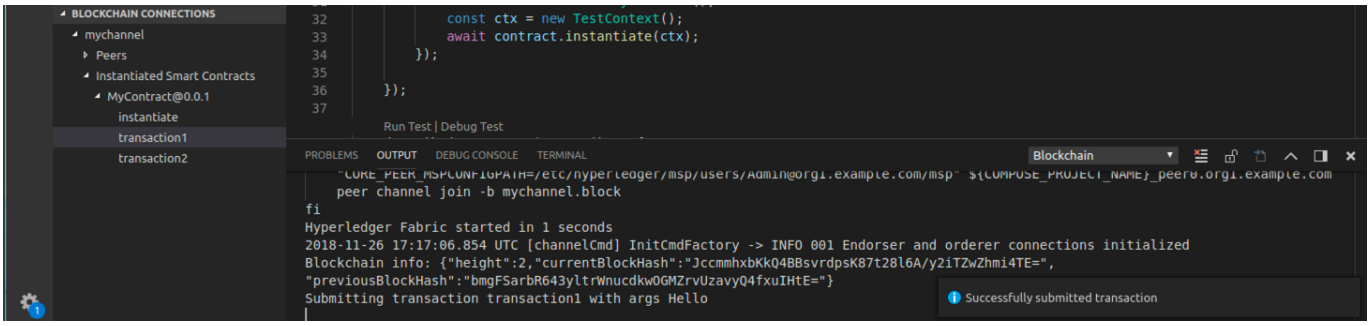
3 Right click on **transaction1** and select **Submit Transaction**.



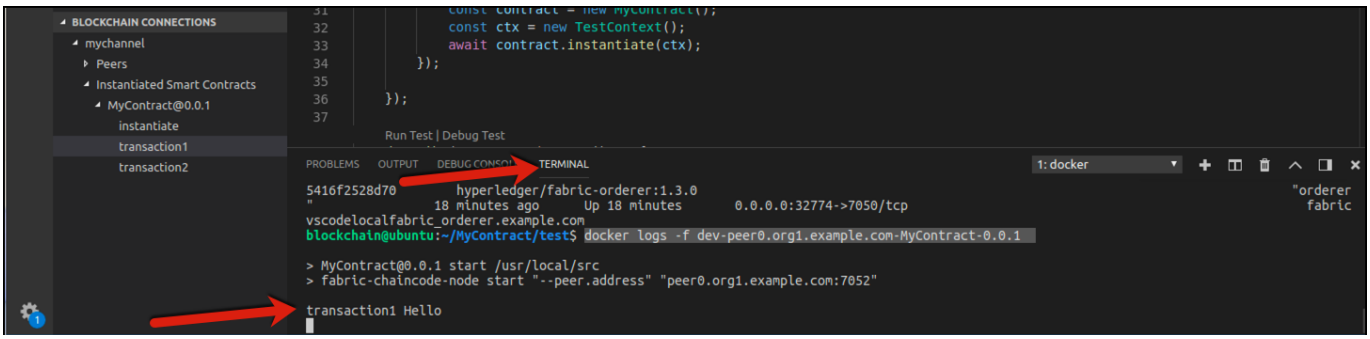
4 Type **Hello** and press enter/return at the argument prompt at the top of the view as follows.



5 You should see the output below in the terminal window. The transaction has succeeded.



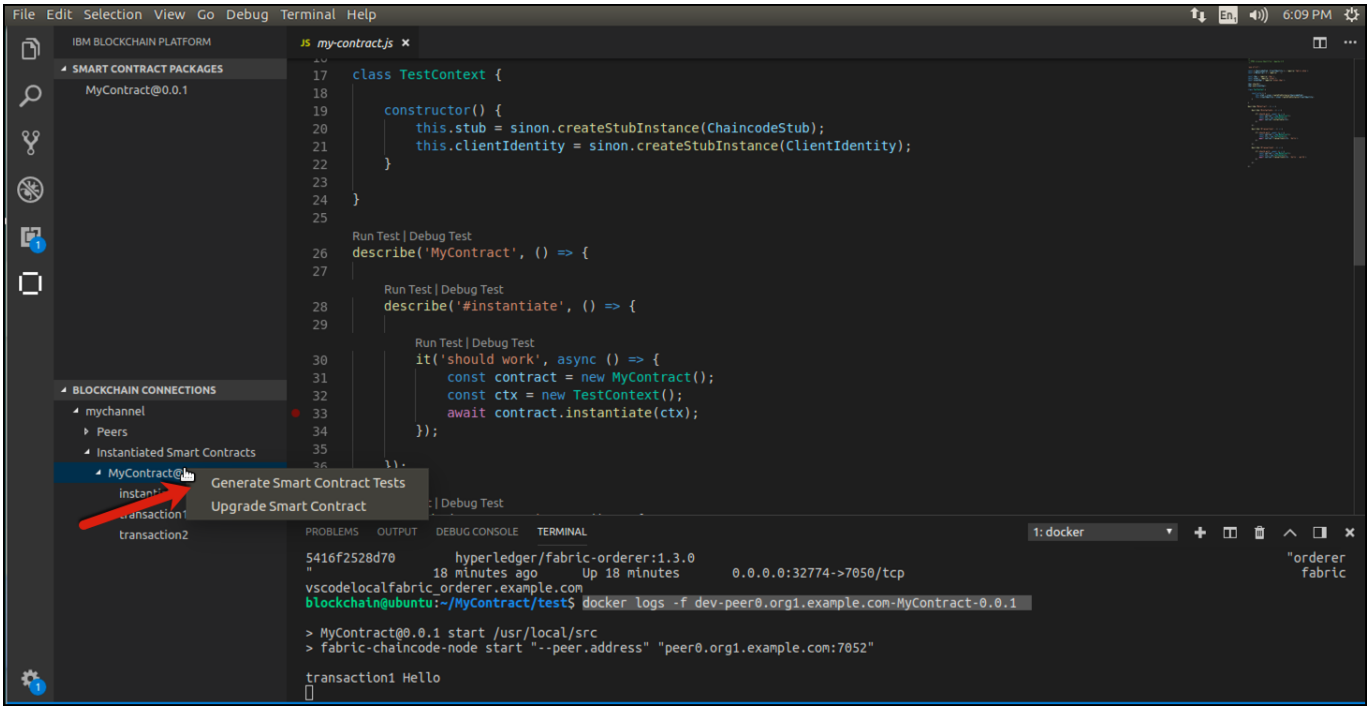
6 We will now look at the output displayed from the chaincode terminal. Click the Terminal pane and you should see the output below in the terminal window. The name of the transaction and the text you entered for the argument is displayed.



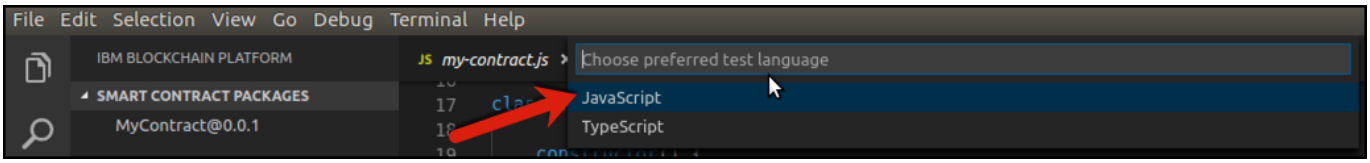
1.7. Generate a test client for the smart contract

In this section we will generate a test client for the smart contract that has been generated, deployed, and instantiated using the embedded Hyperledger Fabric runtime.

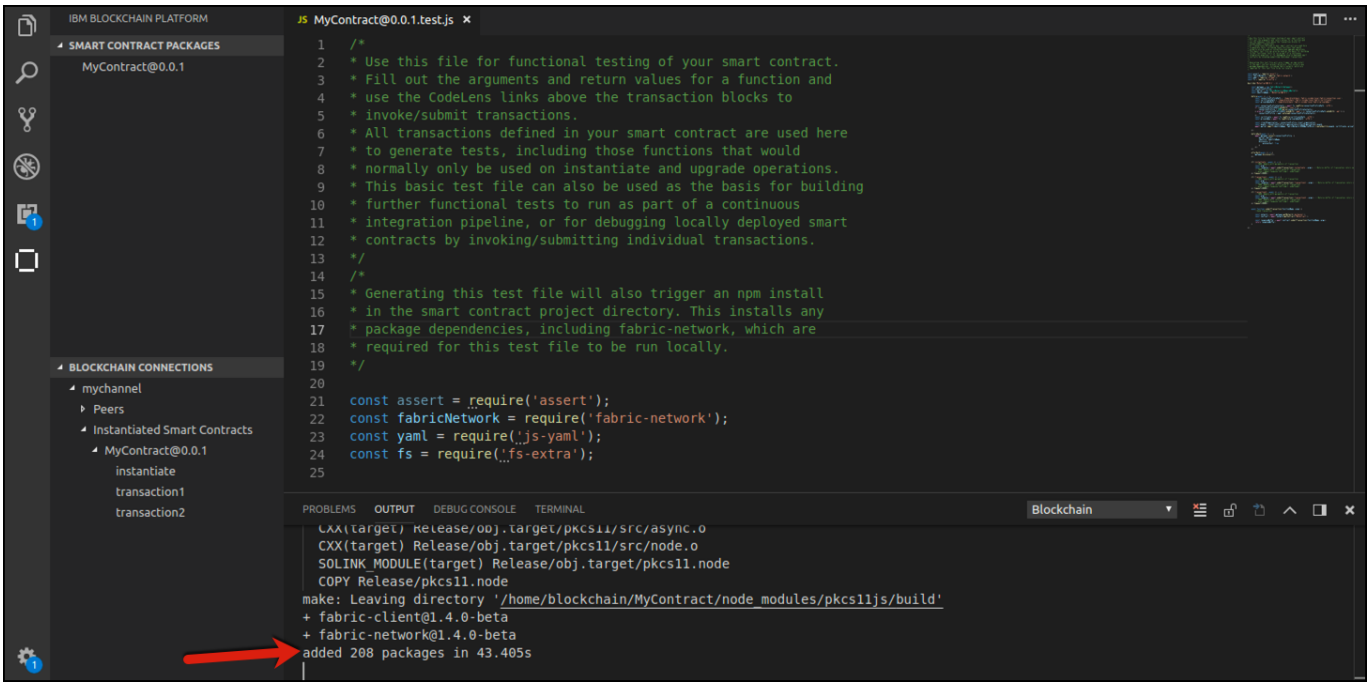
1 Right click on MyContract@0.0.1 and select **Generate Smart Contract Tests**.



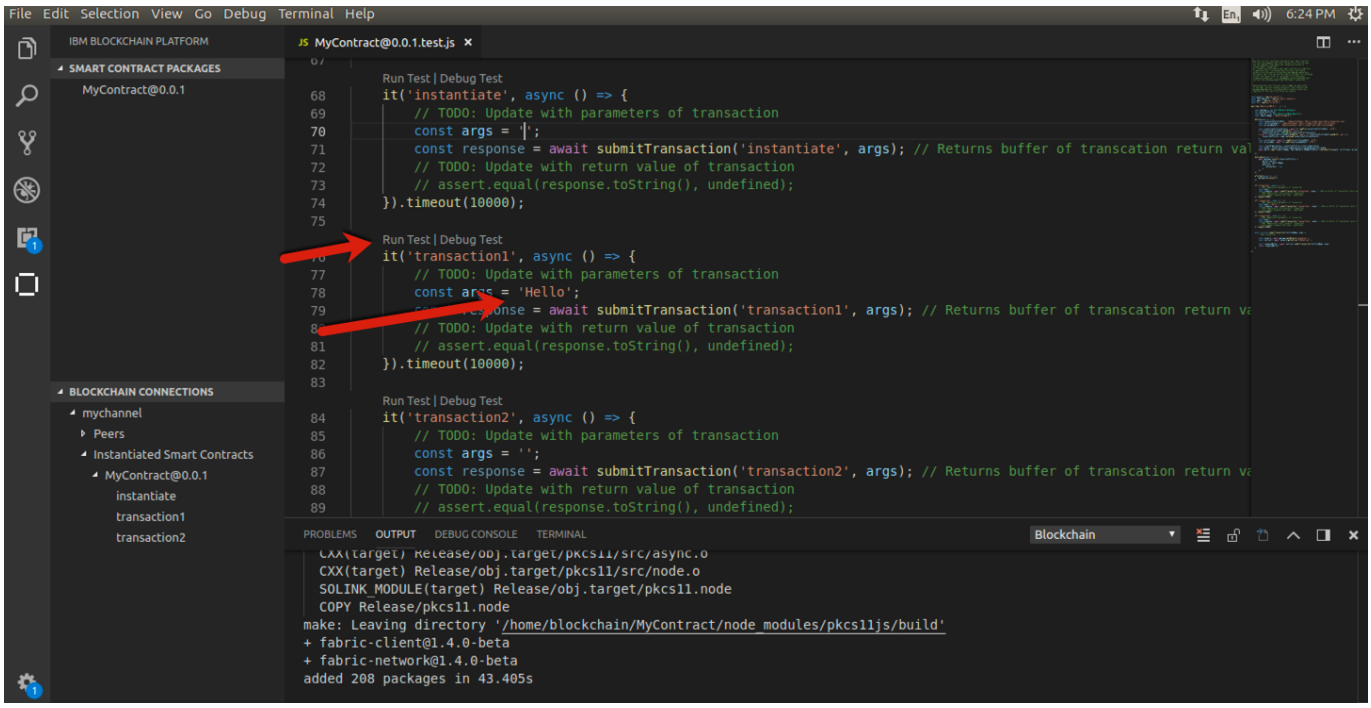
2 Select JavaScript as the preferred test language.



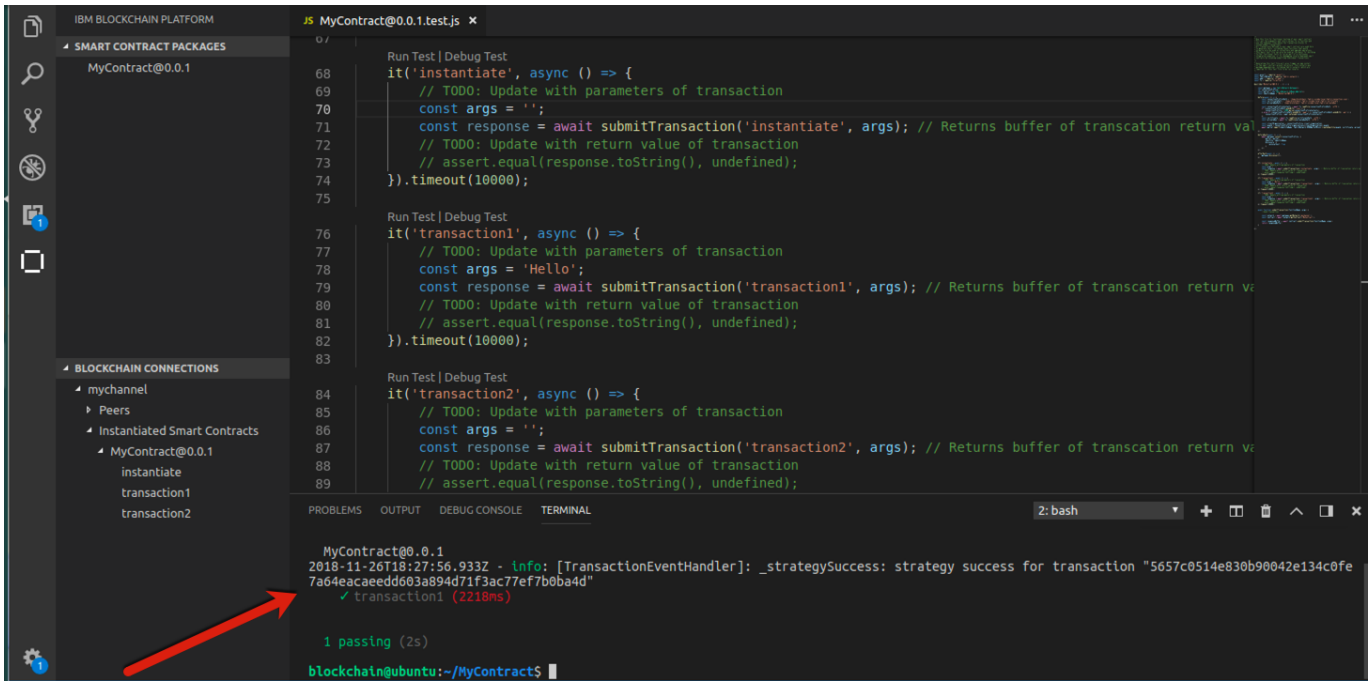
3 Wait a few minutes for the npm packages to install. A new JavaScript test module is created and opened. You should see a message that packages have been installed as follows.



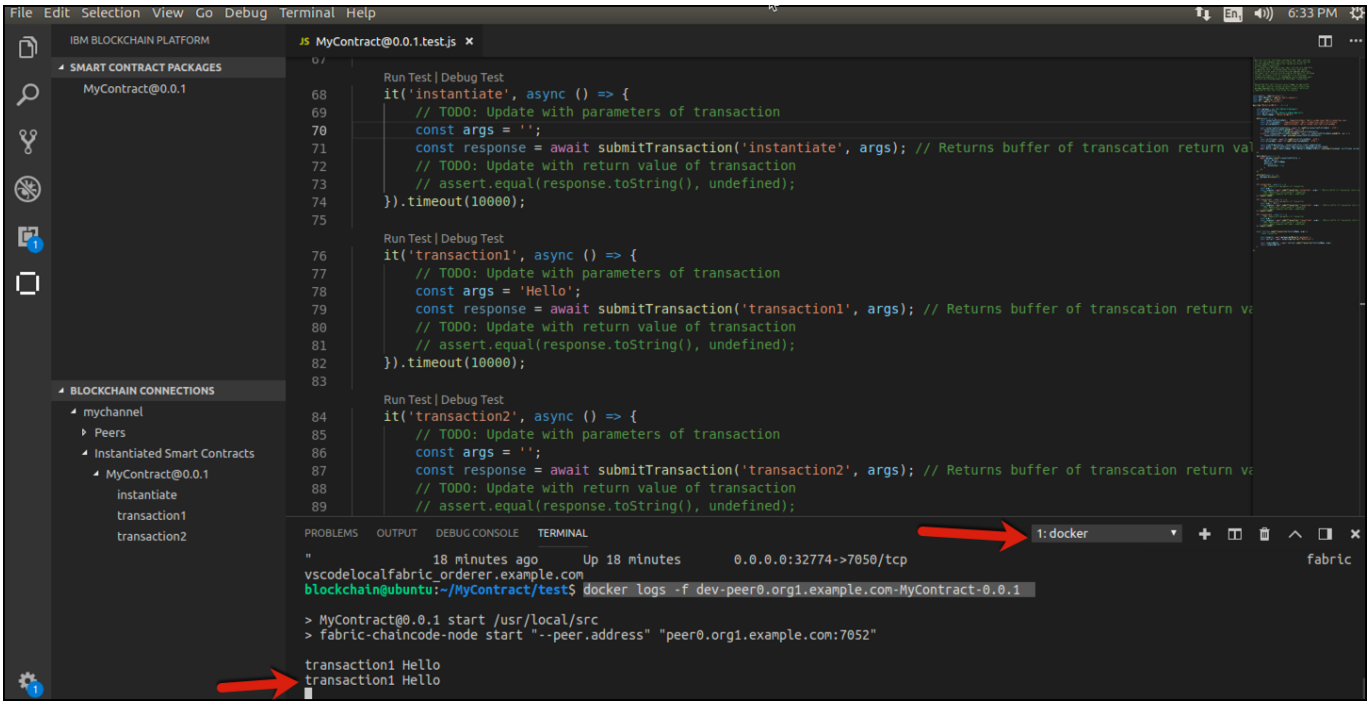
4 Scroll down in the test module to the test for transaction1. Replace the null argument with the string **Hello**, save the file, and click **Run Test**.



5 You should see the test passed as shown below in the terminal pane.



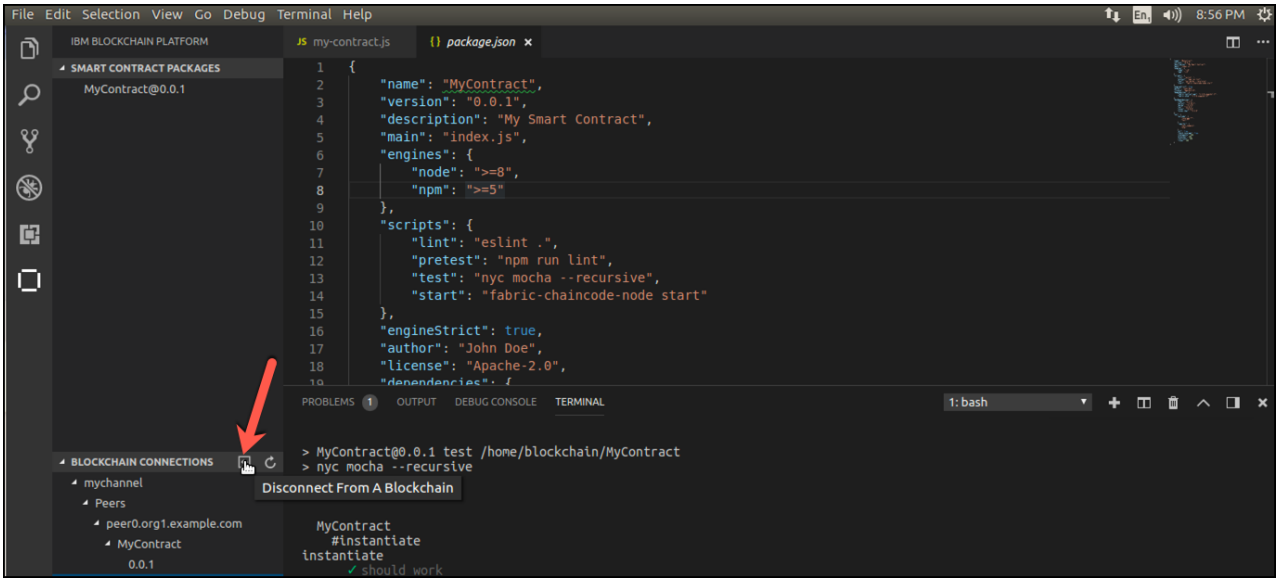
6 You can also view the last output from the chaincode container by selecting 1: docker as shown to go back to the first terminal window pane. You will see you now have an additional line displaying **transaction1 Hello.**



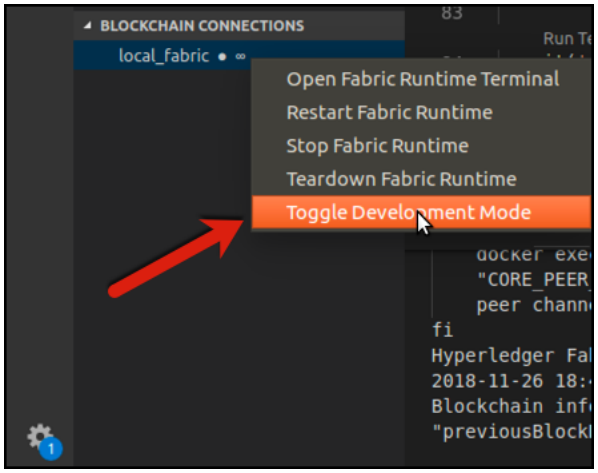
1.8. Debug the smart contract (Optional)

In this section we will debug the smart contract that has been generated, deployed, and instantiated using the embedded Hyperledger Fabric runtime.

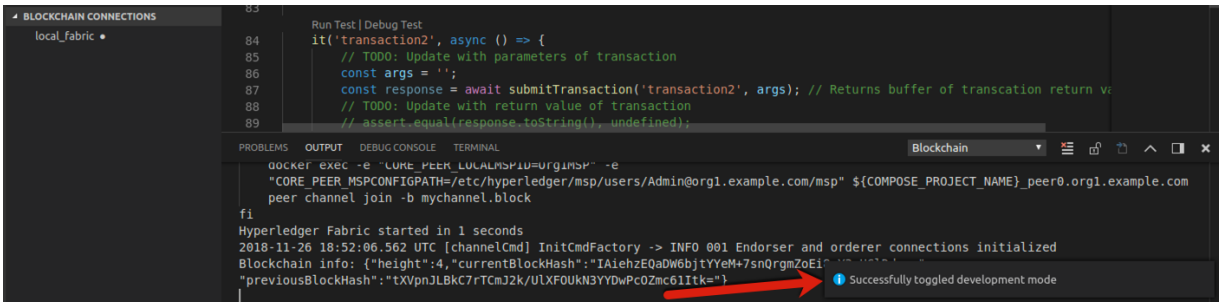
6 Click on the icon as shown below to **Disconnect from a Blockchain**.



7 Right click on **local_fabric** and select **Toggle Development Mode.**

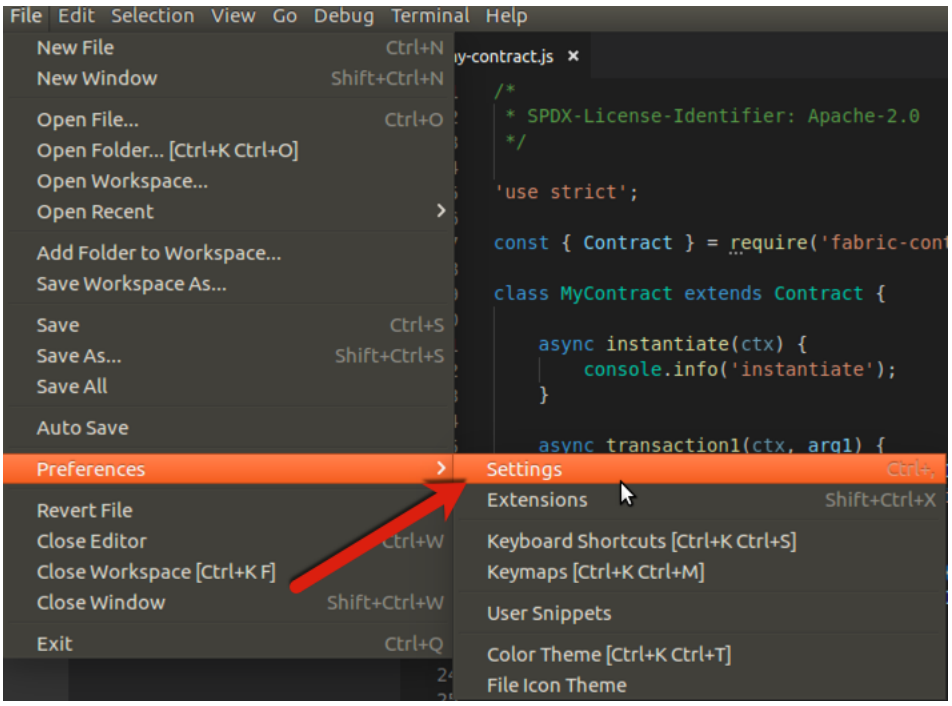


8 You should see the **Successfully Toggled Development Mode** message below.

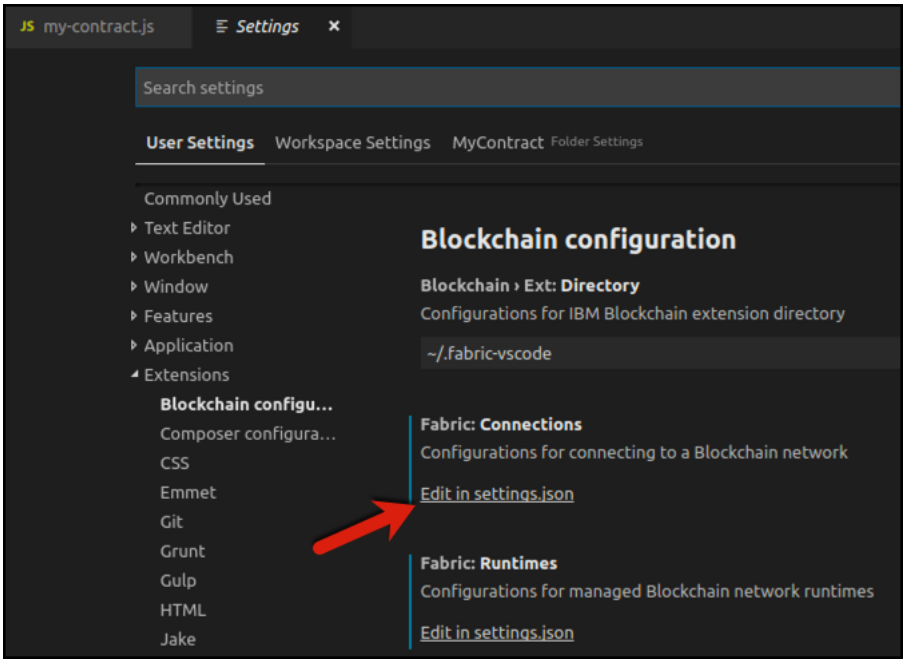


9 A workaround is needed to correct an issue with an undefined connection when using the

debugger. Edit the user preference settings for the connection to the local Hyperledger Fabric settings by selecting **File->Preferences->Settings** from the menu.



a. Under **Fabric: Connections**, select **Edit in settings.json**.



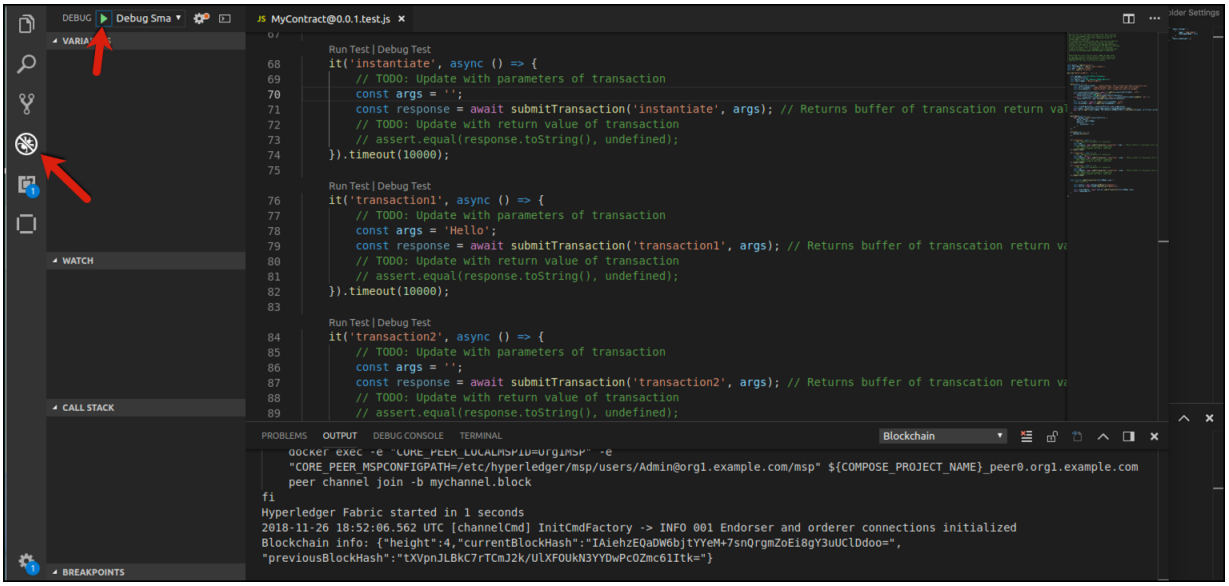
b. Edit **fabric.connections** to look as follows:

```
{
```

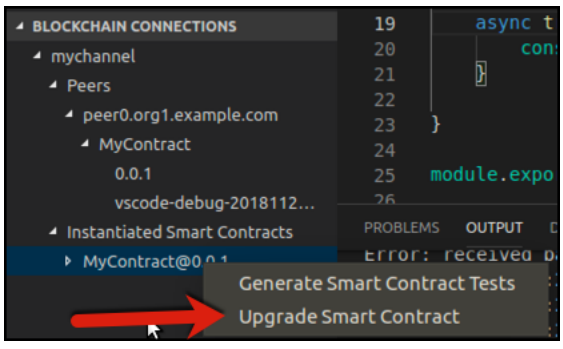
```
    "fabric.runtimes": [
        {
            "name": "local_fabric",
            "developmentMode": true
        }
    ],
    "fabric.connections": [
        {
            "name": "local_fabric"
        }
    ],
    "extensions.autoUpdate": false
}
```

c. Save the file by selecting **File->Save** from the menu.

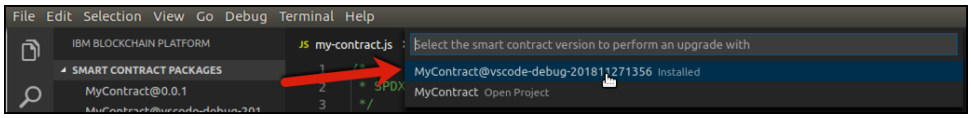
10 Select the Debug icon pane and then click the green **Play** button to package and install the smart contract.



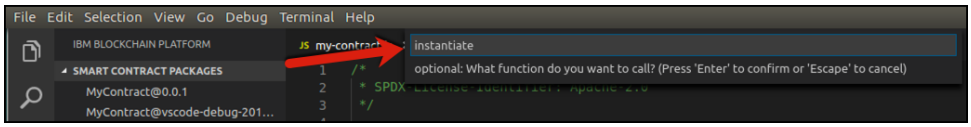
11 Upgrade the Smart Contract with the generated debug Smart Contract. Right click on MyContact@0.0.1 and select **Upgrade Smart Contract.**



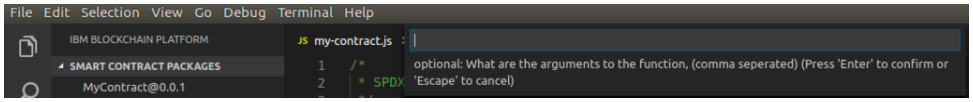
12 Select the debug version of the Smart Contract package as shown below.



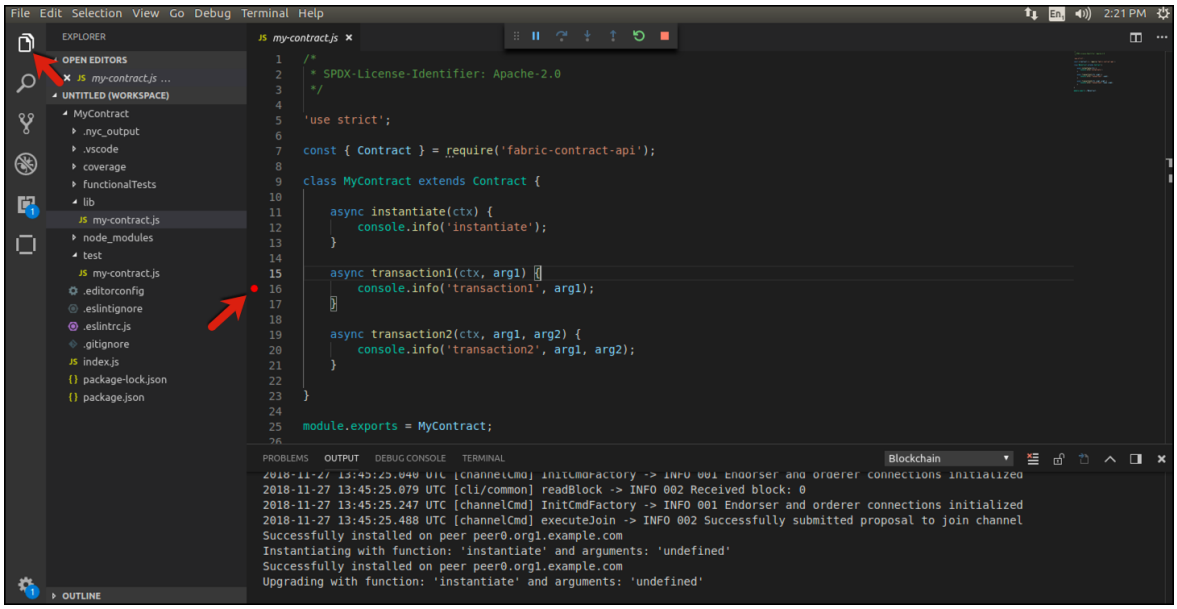
13 Enter **instantiate** and press the enter key.



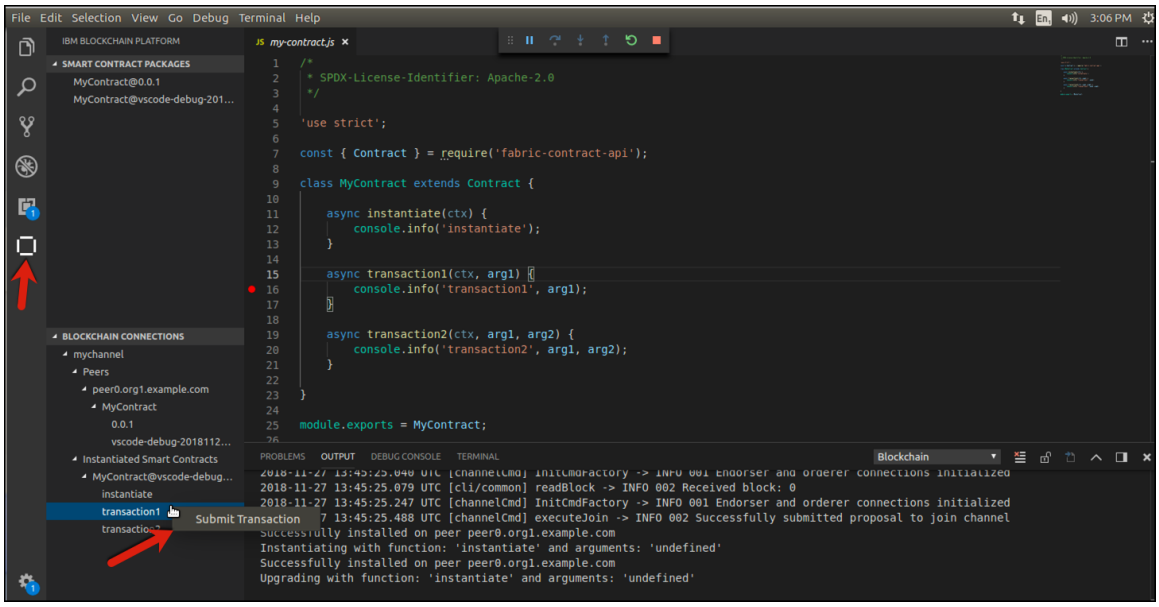
14 Leave arguments empty at the arguments prompt and press the enter key.



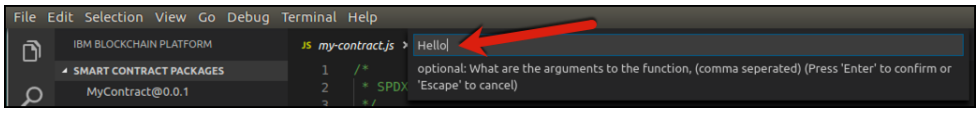
15 Now let's set a break point. Switch back to the explorer view, open the **my-contract.js** file under the **lib** folder if not already open. For **transaction1**, click on the line number next to the **console.info('transaction1', arg1);**



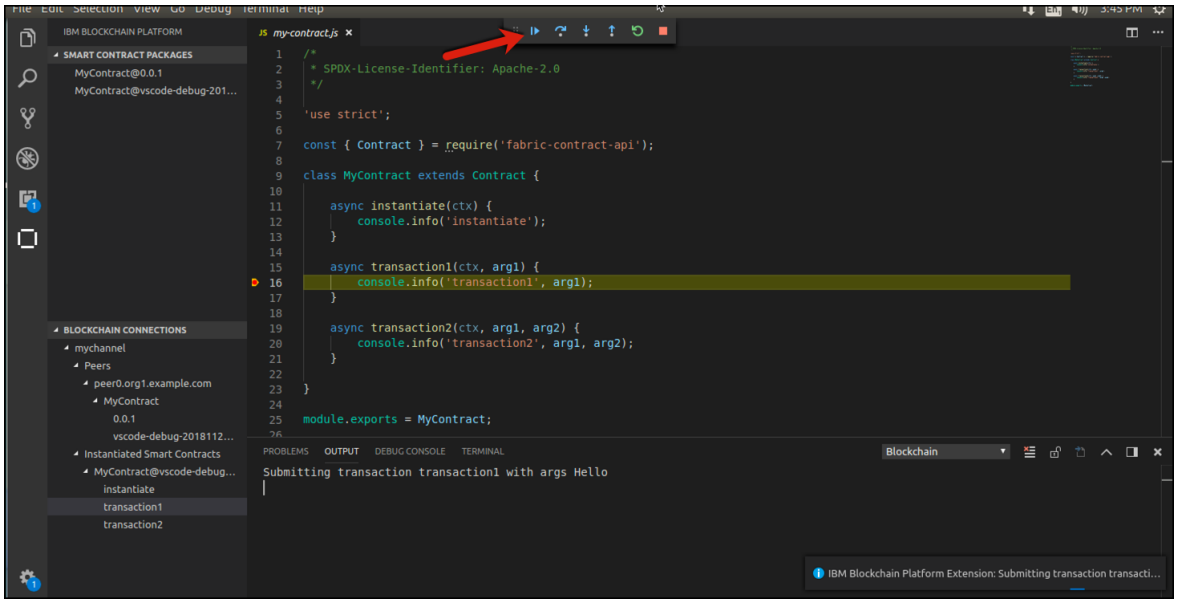
16 Switch back to the IBM Blockchain Platform view and right click on **transaction1** and select **Submit Transaction**.



17 At the argument prompt, enter **Hello** or a string of your choosing and press the enter key.

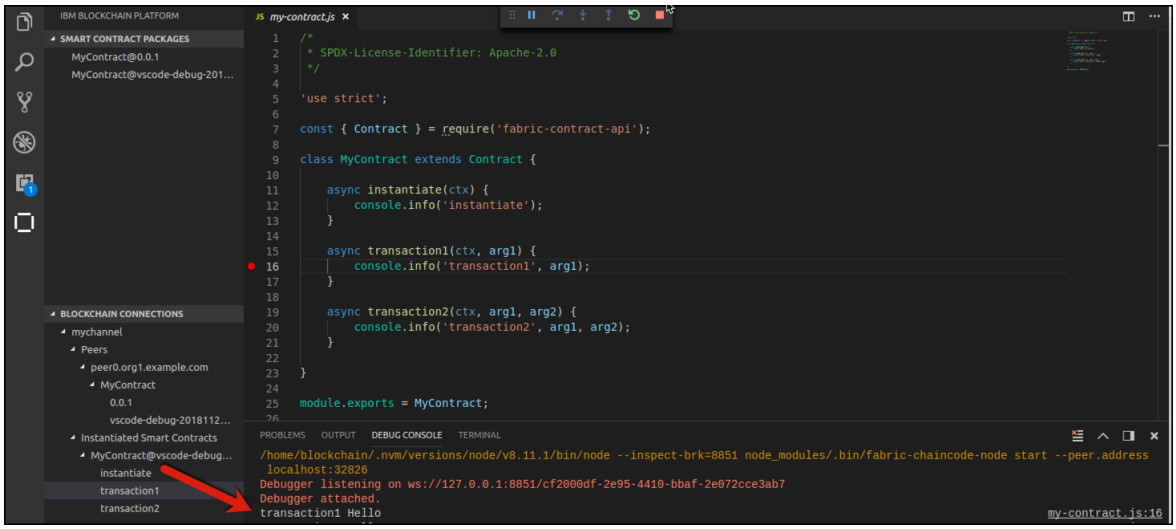


18 Execution of the smart contract stops at the breakpoint you set. Within the debug pane, press the **continue** icon.



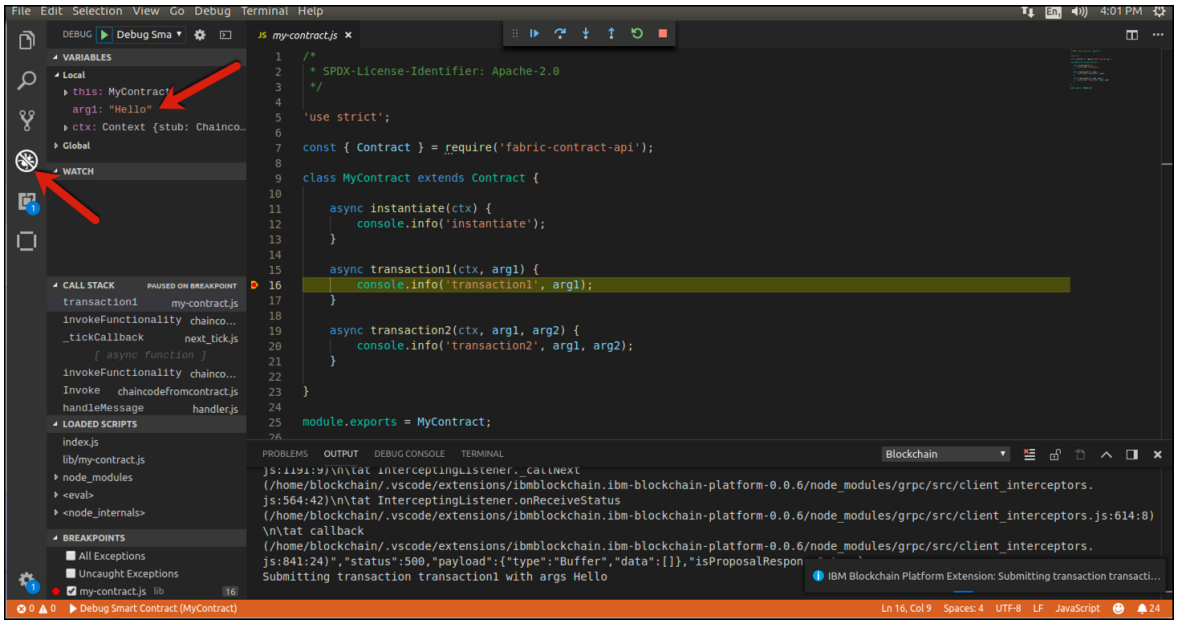
19 Click the **DEBUG CONSOLE** tab to see the output displayed from execution of **transaction1**. Note **transaction1** is displayed along with the argument you provided which was **Hello** in this

example.



20 Repeat steps **11-12** above. This time, click the **DEBUG CONSOLE** pane and enter **arg1** and then click the **continue** icon. Be quick about it as the transaction will time out on you. This is one way to interrogate or set a variable. To set the argument, you would enter **arg1=”YourFirstName”** for example.

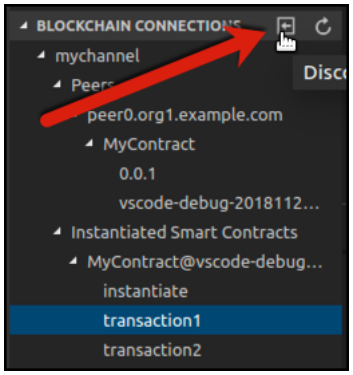
21 Repeat steps **11-12** above. This time, go to the Debug view. You can see the value you set for **arg1** under the **Local Variables** section. Click the **continue** icon



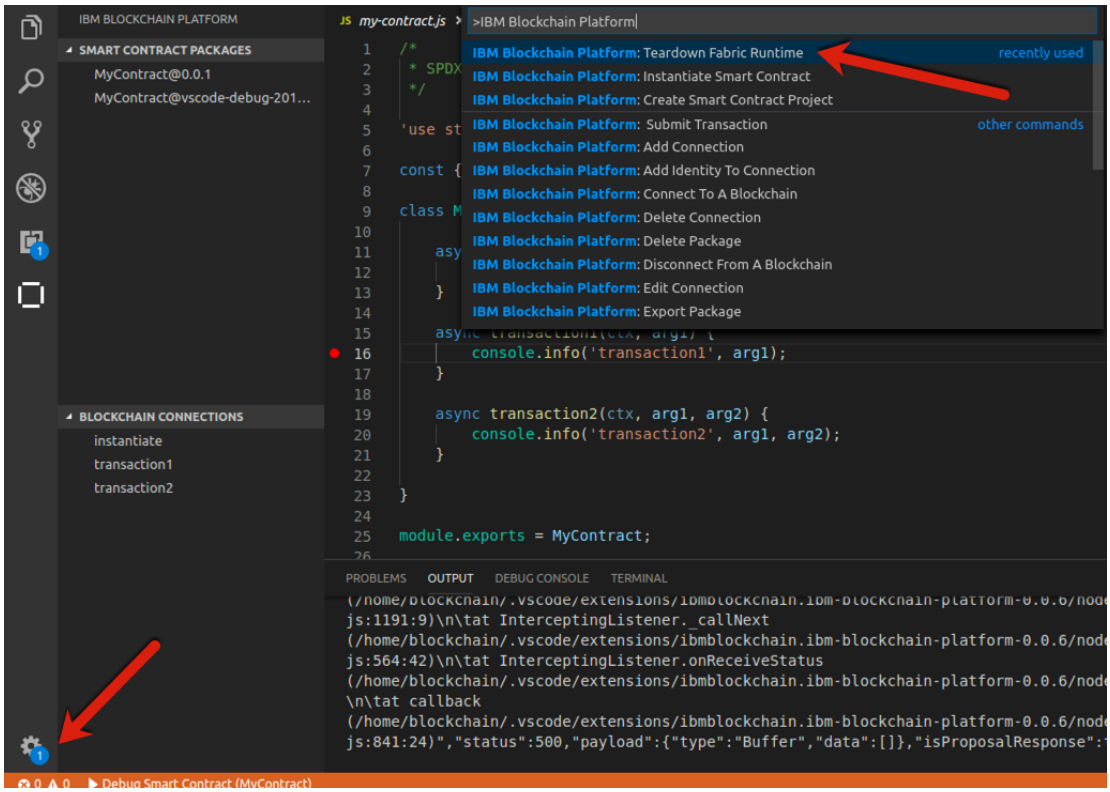
1.9. Lab Cleanup

1 Navigate back to the IBM Blockchain Platform view. You should know how to do this by now.

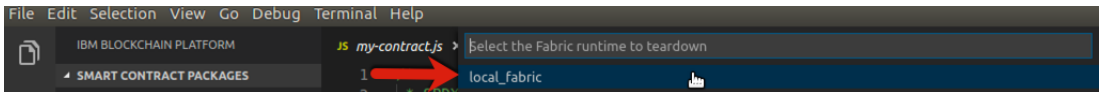
2 Click on the icon as shown below to **Disconnect from a Blockchain**.



3 Select the **Command Palette** and then select **Teardown Fabric Runtime**.



4 Select **local_fabric** as the Fabric runtime to teardown.



5 Click **Yes** to the prompt on the lower right of VSCode to destroy all world state and ledger data.

6 The Docker containers are stopped and removed along with the volumes as shown below.

7 Close VSCode by choosing **File->Exit** from the VSCode menu bar.

Appendix B. Appendix A. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be

the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental. All references to fictitious companies or individuals are used for illustration purposes only.

Appendix B. Trademarks and copyrights

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	AIX	CICS	ClearCase	ClearQuest	Cloudscape
Cube Views	DB2	developerWorks	DRDA	IMS	IMS/ESA
Informix	Lotus	Lotus Workflow	MQSeries	OmniFind	
Rational	Redbooks	Red Brick	RequisitePro	System i	
System z	Tivoli	WebSphere	Workplace	System p	

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of The Minister for the Cabinet Office, and is registered in the U.S. Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

Please Recycle

IBM Blockchain Hands-On

Go Chaincode and Testing in Development Mode

Lab Five

Contents

Overview

What is Chaincode? Chaincode is a program, written in Go, node.js, or Java that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages the ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it is similar to a “smart contract”. A chaincode can be invoked to update or query the ledger in a proposal transaction. Given the appropriate permission, a chaincode may invoke another chaincode, either in the same channel or in different channels, to access its state. Note that, if the called chaincode is on a different channel from the calling chaincode, only read query is allowed. That is, the called chaincode on a different channel is only a `Query`, which does not participate in state validation checks in subsequent commit phase.

Chaincode APIs

In the following sections, we will explore chaincode through the eyes of an application developer. We'll present a simple chaincode sample application and walk through the purpose of each method in the Chaincode Shim API.

There is another set of chaincode APIs that allow the client (submitter) identity to be used for access control decisions, whether that is based on client identity itself, or the org identity, or on a client identity attribute. For example, an asset that is represented as a key/value may include the client's identity, and only this client may be authorized to make updates to the key/value. The client identity library has APIs that chaincode can use to retrieve this submitter information to make such access control decisions. We won't cover that in this tutorial, however it is documented here.

Every chaincode program must implement the `Chaincode` interface:

- Go
- node.js
- Java

whose methods are called in response to received transactions. In particular the `Init` method is called when a chaincode receives an `instantiate` or `upgrade` transaction so that the chaincode may perform any necessary initialization, including initialization of application state. The `Invoke` method is called in response to receiving an `invoke` transaction to process transaction proposals.

The other interface in the chaincode "shim" APIs is the `ChaincodeStubInterface`:

- Go
- node.js
- Java

which is used to access and modify the ledger, and to make invocations between chaincodes.

In this tutorial using Go chaincode, we will demonstrate the use of these APIs by implementing a simple chaincode application that manages simple "assets".

Writing Simple Asset Chaincode

Our application is a basic sample chaincode to create assets (key-value pairs) on the ledger.

1.1. Perform prerequisite lab cleanup

1 Start a terminal
2 cd ~/fabric-samples/first-network
3 Issue the following command to shutdown any existing blockchain networks. Answer 'Y' to the Continue prompt.

```
./byfn.sh down
```

4 Issue the following command to kill any active or stale containers:

```
docker rm -f $(docker ps -aq)
```

5 Issue the following command to clear any cached networks. Answer 'Y' to the Continue prompt.

```
docker network prune
```

6 Delete the underlying chaincode image for the fabcar smart contract.

```
docker rmi dev-peer0.org1.example.com-fabcar-1.0-5c906e402ed29f20260ae42283216aa75549c571e2e380f3615826365d8269ba
```

1.2. Choosing a Location for the Code

If you haven't been doing programming in Go, you may want to make sure that you have Go Programming Language installed and your system properly configured.

Now, you will want to create a directory for your chaincode application as a child directory of `$GOPATH/src/`.

To keep things simple, let's use the following command:

```
mkdir -p $GOPATH/src/sacc && cd $GOPATH/src/sacc
```

Now, let's create the source file that we'll fill in with code:

```
touch sacc.go
```

Housekeeping

First, let's start with some housekeeping. As with every chaincode, it implements theChaincode interface in particular, `Init` and `Invoke` functions. So, let's add the Go import statements for the necessary dependencies for our chaincode. We'll import the chaincode shim package and the peer protobuf package. Next, let's add a struct `SimpleAsset` as a receiver for Chaincode shim functions.

```
package main

import (
    "fmt"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct {
}
```

Initializing the Chaincode

Next, we'll implement the `Init` function.

```
// Init is called during chaincode instantiation to initialize any data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {

}
```

Note

Note that chaincode upgrade also calls this function. When writing a chaincode that will upgrade an existing one, make sure to modify the Init function appropriately. In particular, provide an empty "Init" method if there's no "migration" or nothing to be initialized as part of the upgrade.

Next, we'll retrieve the arguments to the `Init` call using the ChaincodeStubInterface.GetStringArgs function and check for validity. In our case, we are expecting a key-value pair.

```
// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data, so be careful to avoid a scenario where you
// inadvertently clobber your ledger's data!
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
  // Get the args from the transaction proposal
  args := stub.GetStringArgs()
  if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
  }
```

```
}
```

Next, now that we have established that the call is valid, we'll store the initial state in the ledger. To do this, we will call ChaincodeStubInterface.PutState with the key and value passed in as the arguments. Assuming all went well, return a peer.Response object that indicates the initialization was a success.

```
// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data, so be careful to avoid a scenario where you
// inadvertently clobber your ledger's data!
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
  // Get the args from the transaction proposal
  args := stub.GetStringArgs()
  if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
  }

  // Set up any variables or assets here by calling stub.PutState()

  // We store the key and the value on the ledger
  err := stub.PutState(args[0], []byte(args[1]))
  if err != nil {
    return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
  }
  return shim.Success(nil)
}
```

Invoking the Chaincode

First, let's add the `Invoke` function's signature.

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The 'set'
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {

}
```

As with the `Init` function above, we need to extract the arguments from the `ChaincodeStubInterface`. The `Invoke` function's arguments will be the name of the chaincode application function to invoke. In our case, our application will simply have two functions: `set` and `get`, that allow the value of an asset to be set or its current state to be retrieved.

We first call ChaincodeStubInterface.GetFunctionAndParameters to extract the function name and the parameters to that chaincode application function.

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

}
```

Next, we'll validate the function name as being either `set` or `get`, and invoke those chaincode application functions, returning an appropriate response via the `shim.Success` or `shim.Error` functions that will serialize the response into a gRPC protobuf message.

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
            result, err = set(stub, args)
    } else {
            result, err = get(stub, args)
    }
    if err != nil {
            return shim.Error(err.Error())
    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}
```

Implementing the Chaincode Application

As noted, our chaincode application implements two functions that can be invoked via the `Invoke` function. Let's implement those functions now. Note that as we mentioned above, to access the ledger's state, we will leverage the ChaincodeStubInterface.PutState and ChaincodeStubInterface.GetState functions of the chaincode shim API.

```
// Set stores the asset (both key and value) on the ledger. If the key exists,
// it will override the value with the new one
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
            return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// Get returns the value of the specified asset key
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
            return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0], err)
```

```
    }
    if value == nil {
            return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}
```

Pulling it All Together

Finally, we need to add the `main` function, which will call the shim.Start function. Here's the whole chaincode program source.

```
package main

import (
    "fmt"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct {
}

// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    args := stub.GetStringArgs()
    if len(args) != 2 {
            return shim.Error("Incorrect arguments. Expecting a key and a value")
    }

    // Set up any variables or assets here by calling stub.PutState()

    // We store the key and the value on the ledger
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
            return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
    }
    return shim.Success(nil)
}

// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
            result, err = set(stub, args)
    } else { // assume 'get' even if fn is nil
            result, err = get(stub, args)
```

```
    }
    if err != nil {
            return shim.Error(err.Error())
    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}

// Set stores the asset (both key and value) on the ledger. If the key exists,
// it will override the value with the new one
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
            return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// Get returns the value of the specified asset key
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
            return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0], err)
    }
    if value == nil {
            return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}

// main function starts up the chaincode in the container during instantiate
func main() {
    if err := shim.Start(new(SimpleAsset)); err != nil {
            fmt.Printf("Error starting SimpleAsset chaincode: %s", err)
    }
}
```

Building Chaincode

Now let's compile your chaincode.

```
go get -u github.com/hyperledger/fabric/core/chaincode/shim
go build
```

Assuming there are no errors, now we can proceed to the next step, testing your chaincode.

1.3. Testing Using dev mode

Normally chaincodes are started and maintained by peer. However in “dev mode”, chaincode is built and started by the user. This mode is useful during chaincode development phase for rapid code/build/run/debug cycle turnaround.

We start “dev mode” by leveraging pre-generated orderer and channel artifacts for a sample dev network. As such, the user can immediately jump into the process of compiling chaincode and driving calls.

Install Hyperledger Fabric Samples

Navigate to the `chaincode-docker-devmode` directory of the `fabric-samples` clone:

```
cd chaincode-docker-devmode
```

Now open three terminals and navigate to your `chaincode-docker-devmode` directory in each.

Terminal 1 - Start the network

Use the following command to start the network with the `SingleSampleMSPSolo` orderer profile and launch the peer in “dev mode”.

```
docker-compose -f docker-compose-simple.yaml up
```

The command also launches two additional containers - one for the chaincode environment and a CLI to interact with the chaincode. The commands for create and join channel are embedded in the CLI container, so we can jump immediately to the chaincode calls.

Terminal 2 - Build & start the chaincode

Execute the following command in terminal 2:

```
docker exec -it chaincode bash
```

You should see the following:

```
root@d2629980e76b:/opt/gopath/src/chaincode#
```

Now, compile your chaincode:

```
cd sacc
go build
```

Now run the chaincode:

```
CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=mycc:0 ./sacc
```

The chaincode is started with peer and chaincode logs indicating successful registration with the peer. Note that at this stage the chaincode is not associated with any channel. This is done in subsequent steps using the `instantiate` command.

Terminal 3 - Use the chaincode

Even though you are in `--peer-chaincodedev` mode, you still have to install the chaincode so the life-cycle system chaincode can go through its checks normally. This requirement may be removed in the future when in `--peer-chaincodedev` mode.

We'll leverage the CLI container to drive these calls.

```
docker exec -it cli bash
peer chaincode install -p chaincodedev/chaincode/sacc -n mycc -v 0
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a","10"]}' -C myc
```

Now issue an invoke to change the value of "a" to "20".

```
peer chaincode invoke -n mycc -c '{"Args":["set", "a", "20"]}' -C myc
```

Finally, query `a`. We should see a value of `20`.

```
peer chaincode query -n mycc -c '{"Args":["query","a"]}' -C myc
```

1.4. Testing new chaincode

By default, we mount only `sacc`. However, you can easily test different chaincodes by adding them to the `chaincode` subdirectory and relaunching your network. At this point they will be accessible in your `chaincode` container.

1.5. Chaincode encryption

In certain scenarios, it may be useful to encrypt values associated with a key in their entirety or simply in part. For example, if a person's social security number or address was being written to the ledger, then you likely would not want this data to appear in plaintext. Chaincode encryption is achieved by leveraging the entities extension which is a BCCSP wrapper with commodity factories and functions to perform cryptographic operations such as encryption and elliptic curve digital signatures. For example, to encrypt, the invoker of a chaincode passes in a cryptographic key via the transient field. The same key may then be used for subsequent query operations, allowing for proper decryption of the encrypted state values.

For more information and samples, see the Encc Example within the `fabric/examples` directory. Pay specific attention to the `utils.go` helper program. This utility loads the chaincode shim APIs and Entities extension and builds a new class of functions (e.g. `encryptAndPutState` & `getStateAndDecrypt`) that the sample encryption chaincode then leverages. As such, the chaincode can now marry the basic shim APIs of `Get` and `Put` with the added functionality of `Encrypt` and `Decrypt`.

1.6. Managing external dependencies for chaincode written in Go

If your chaincode requires packages not provided by the Go standard library, you will need to include those packages with your chaincode. There are many tools available for managing (or "vendoring") these dependencies. The following demonstrates how to use `govendor`:

```
govendor init
govendor add +external  // Add all external package, or
govendor add github.com/external/pkg // Add specific external package
```

This imports the external dependencies into a local `vendor` directory. `peer chaincode package` and `peer chaincode install` operations will then include code associated with the dependencies into the chaincode package.

Appendix C. Appendix A. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be

the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental. All references to fictitious companies or individuals are used for illustration purposes only.

Appendix B. Trademarks and copyrights

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	AIX	CICS	ClearCase	ClearQuest	Cloudscape
Cube Views	DB2	developerWorks	DRDA	IMS	IMS/ESA
Informix	Lotus	Lotus Workflow	MQSeries	OmniFind	
Rational	Redbooks	Red Brick	RequisitePro	System i	
System z	Tivoli	WebSphere	Workplace	System p	

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of The Minister for the Cabinet Office, and is registered in the U.S. Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

Please Recycle