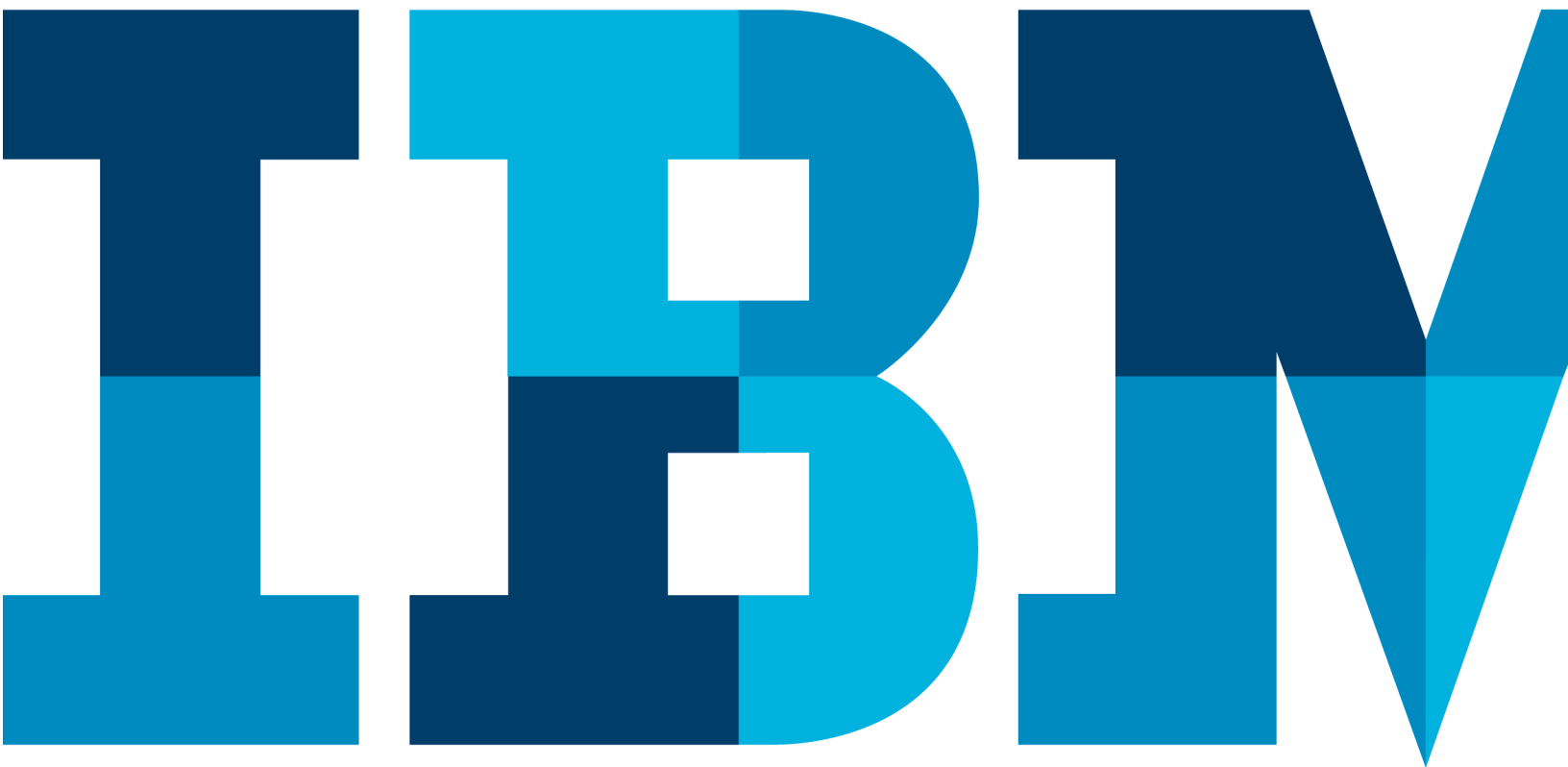


# IBM Blockchain Hands-On

IBM Blockchain Platform Visual Studio Code Extension:

Using an Existing Contract

Lab Three



# Table of Contents

- Disclaimer..... 3**
- 1 Overview of the lab 3 environment and scenario ..... 5
  - 1.1 Lab 3 Scenario..... 6
- 2 Lab 3: Using an Existing Contract..... 8

## Disclaimer

IBM's statements regarding its plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision.

The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract.

The development, release, and timing of any future features or functionality described for our products remains at our sole discretion I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve results like those stated here.

Information in these presentations (including information relating to products that have not yet been announced by IBM) has been reviewed for accuracy as of the date of initial publication and could include unintentional technical or typographical errors. IBM shall have no responsibility to update this information. **This document is distributed "as is" without any warranty, either express or implied. In no event, shall IBM be liable for any damage arising from the use of this information, including but not limited to, loss of data, business interruption, loss of profit or loss of opportunity.** IBM products and services are warranted per the terms and conditions of the agreements under which they are provided.

IBM products are manufactured from new parts or new and used parts. In some cases, a product may not be new and may have been previously installed. Regardless, our warranty terms apply."

**Any statements regarding IBM's future direction, intent or product plans are subject to change or withdrawal without notice.**

Performance data contained herein was generally obtained in controlled, isolated environments. Customer examples are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual performance, cost, savings or other results in other operating environments may vary.

References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business.

Workshops, sessions and associated materials may have been prepared by independent session speakers, and do not necessarily reflect the views of IBM. All materials and

discussions are provided for informational purposes only, and are neither intended to, nor shall constitute legal or other guidance or advice to any individual participant or their specific situation.

It is the customer's responsibility to insure its own compliance with legal requirements and to obtain advice of competent legal counsel as to the identification and interpretation of any relevant laws and regulatory requirements that may affect the customer's business and any actions the customer may need to take to comply with such laws. IBM does not provide legal advice or represent or warrant that its services or products will ensure that the customer follows any law.

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 about this publication 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. IBM does not warrant the quality of any third-party products, or the ability of any such third-party products to interoperate with IBM's products. **IBM expressly disclaims all warranties, expressed or implied, including but not limited to, the implied warranties of merchantability and fitness for a purpose.**

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents, copyrights, trademarks or other intellectual property right.

IBM, the IBM logo, ibm.com and [names of other referenced IBM products and services used in the presentation] are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at: [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

© 2019 International Business Machines Corporation. No part of this document may be reproduced or transmitted in any form without written permission from IBM.


**U.S. Government Users Restricted Rights — use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM.**

## 1 Overview of the lab 3 environment and scenario

This lab is a technical introduction to blockchain, specifically smart contract development using the latest developer enhancements in the Linux Foundation's Hyperledger Fabric v1.4 and shows you how IBM's Blockchain Platform's developer experience can accelerate your pace of development.

**Note:** The screenshots in this lab guide were taken using version **1.31.1** of **VSCode**, and version **0.3.0** of the **IBM Blockchain Platform** plugin. If you use different versions, you may see differences those shown in this guide.

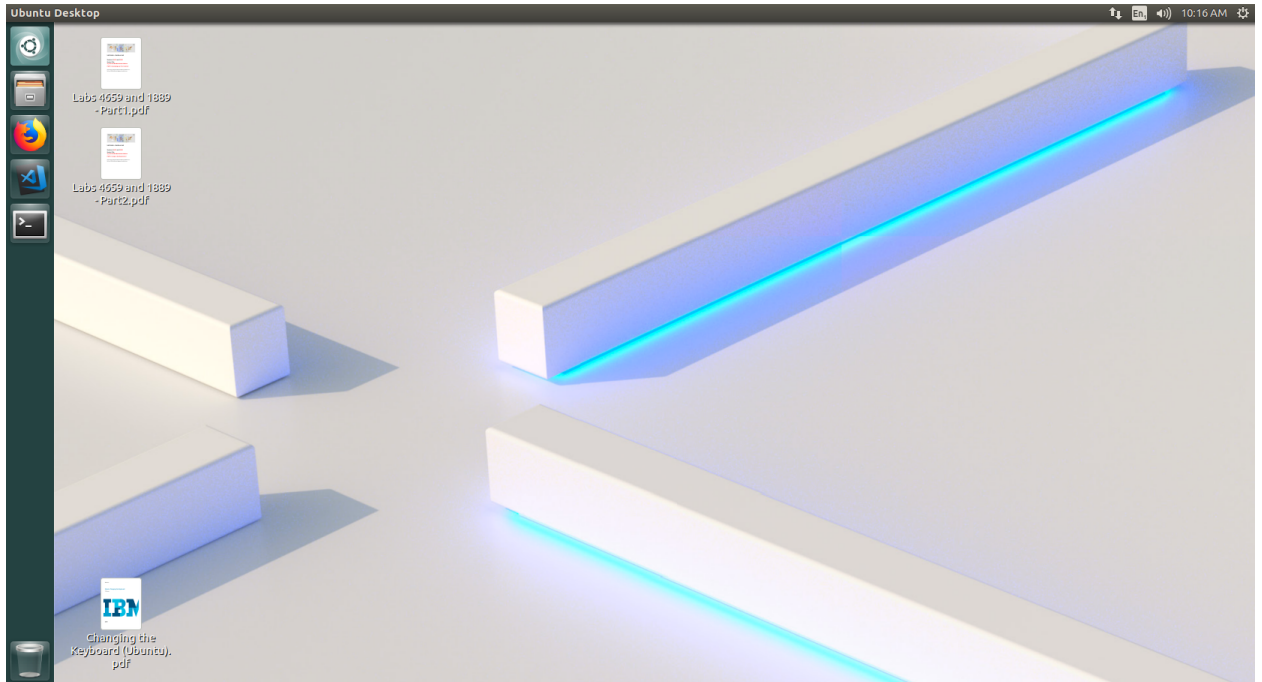
**Start here. Instructions are always shown on numbered lines like this one:**

- 
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.

## IBM Blockchain

- \_\_\_ 2. Wait for the image to boot and for the associated services to start. This happens automatically but might take several minutes. The image is ready to use when the desktop is visible as per the screenshot below.

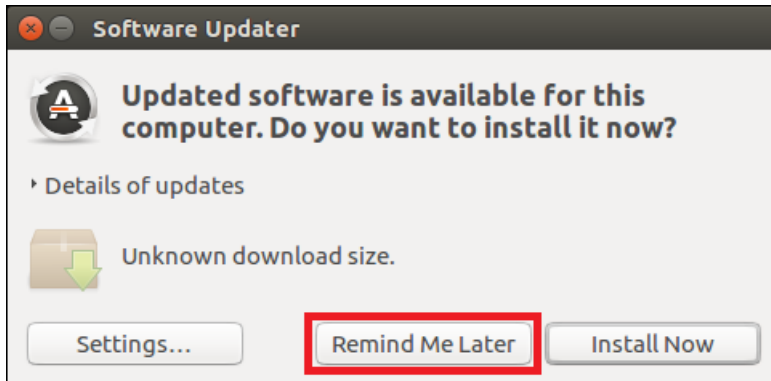
If it asks you to login, the userid and password are both “blockchain”.



### 1.1 Lab 3 Scenario

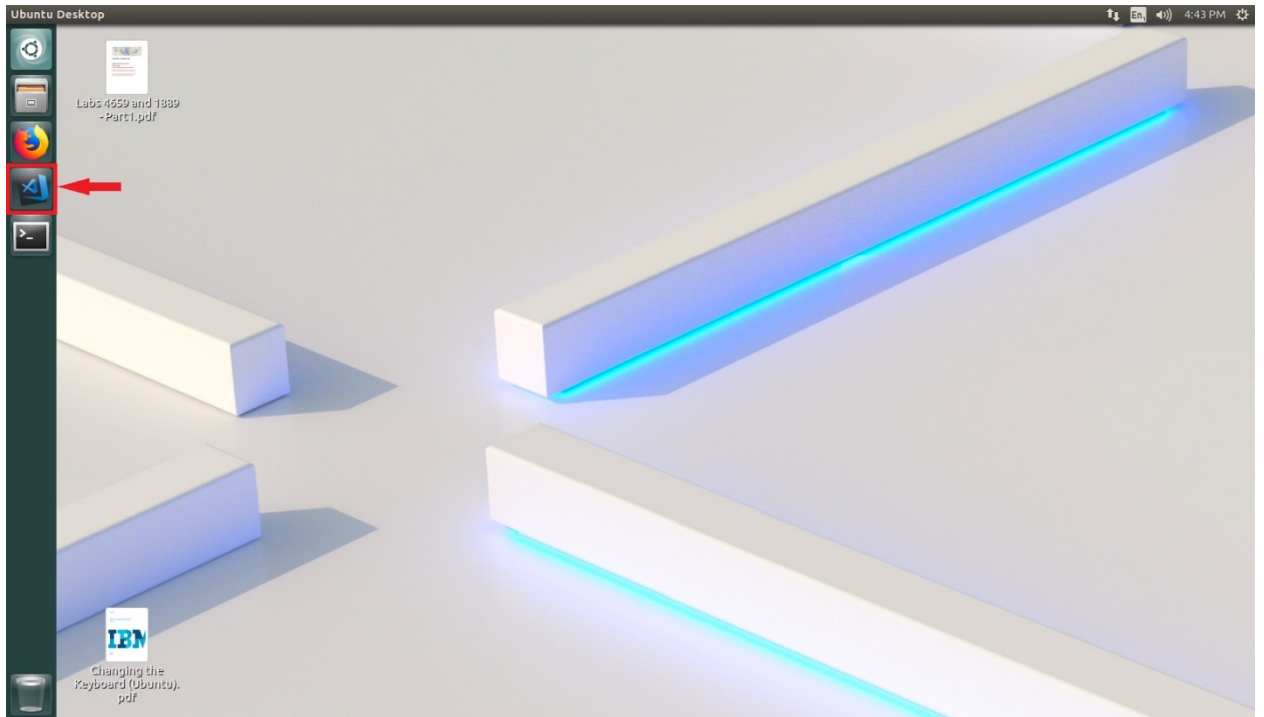
In this lab, we will take you through using a sample smart contract that comes with Hyperledger Fabric’s sample through the VSCode lens, where you will learn how to import contracts and interact with the development environment in more detail. After importing and deploying the contract, you will then use a client application to invoke some of the contract’s transactions.

**Note** that if you get an “Software Updater” pop-up at any point during the lab, please click **“Remind Me Later”**:



## 2 Lab 3: Using an Existing Contract

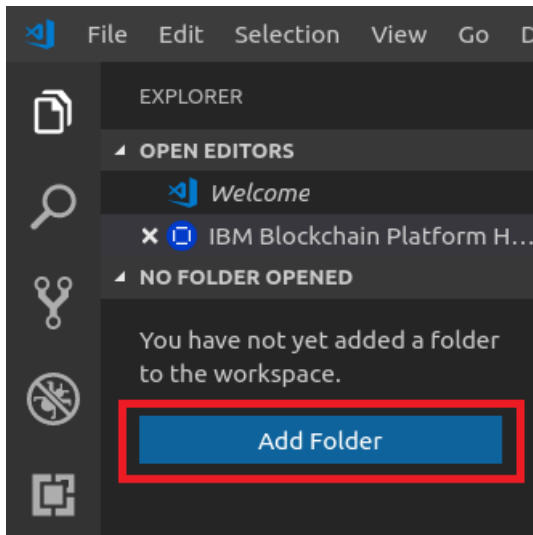
- \_\_ 3. If you have just finished previous VSCode labs, VSCode will already be open so skip this step and move straight to **step \_\_4**. If you are starting with this lab, launch VSCode by clicking on the VSCode Icon in the toolbar.





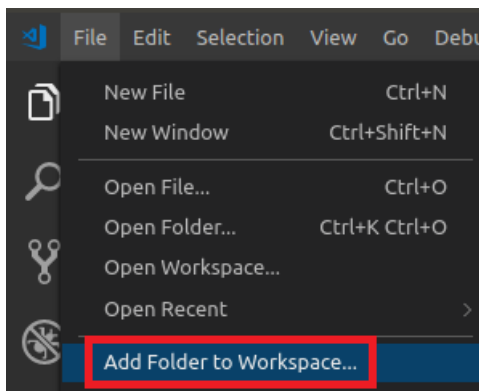
## IBM Blockchain

- \_\_\_ 4. Now VSCode is open, **if you completed the first VSCode lab**, click the **Add Folder** button from the Explorer view as shown below:

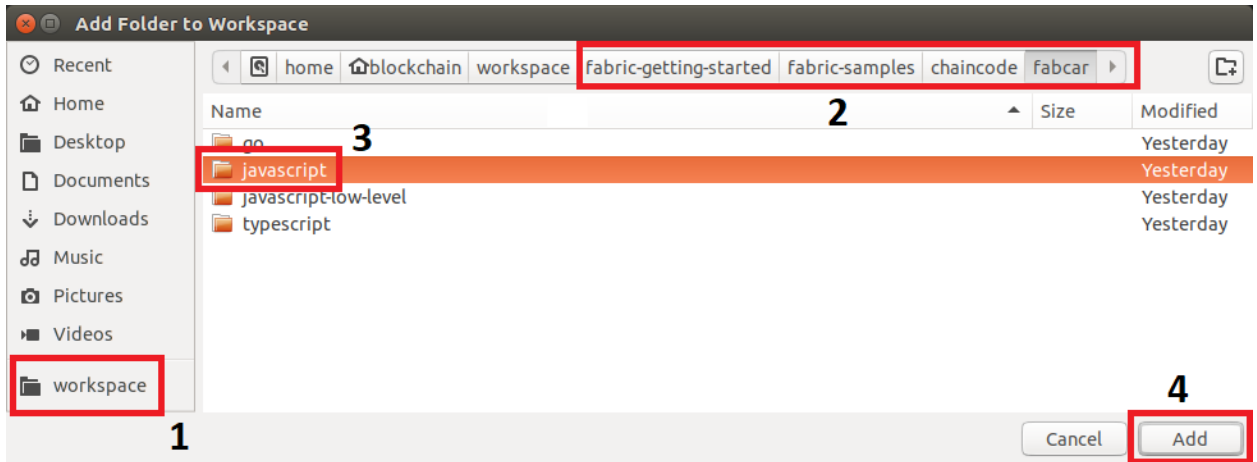


**Note:** If you cannot see the **Explorer** view for any reason, click on its icon in the activity bar (📁) or press “**ctrl + shift + e**” to show it.

However, if you did not do the first VSCode lab and **you are starting with this lab**, you will not see the “**Add Folder**” button and you need to choose the **File / Add Folder to Workspace** menu option instead:



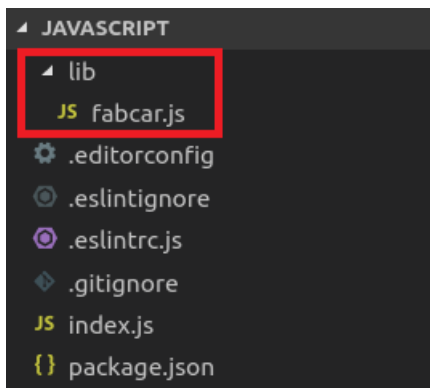
- 5. Using the screen shot below as a guide, navigate to open a folder as follows  
Step 1: Click in the **workspace** folder on the bottom left of the dialogue.  
Step 2: Navigate to the folder:  
**fabric-getting-started/fabric-samples/chaincode/fabcar**  
Step 3: Select the folder **javascript**  
Step 4: Click the **Add** button on the bottom right.



**Note:** the full path to the **javascript** folder you are importing for reference is:  
/home/blockchain/workspace/fabric-getting-started/fabric-samples/chaincode/fabcar/

**Note 2:** The “**IBM Blockchain Platform Home**” page may automatically open once the **javascript** folder has been added to the workspace. If it does, simply close it by clicking on the “**x**”.

- 6. Once the folder is added to the workspace, expand the **lib** folder and double click on the **fabcar.js** smart contract to open it in the main editing view.



- \_\_\_ 7. Use the “-” buttons in the **fabcar.js** file to collapse the transaction definitions, so we can see a simpler overview of the contract we will be using.

Like the previous example in part 1 of the lab the code starts with the same import of a **Contract** definition from the **fabric-contract-api** node module on **line 7**. Next there are five transactions that make up the **fabcar** sample contract and this time most of the transactions take parameters and will either query or update a blockchain for real.

```

7   const { Contract } = require('fabric-contract-api');
8
9   class FabCar extends Contract {
10
11     async initLedger(ctx) { ...
82   }
83
84     async queryCar(ctx, carNumber) { ...
91   }
92
93     async createCar(ctx, carNumber, make, model, color, owner) { ...
106  }
107
108    async queryAllCars(ctx) { ...
138  }
139
140    async changeCarOwner(ctx, carNumber, newOwner) { ...
152  }
153
154  }

```

**Note:** The “-” buttons only appear when you move your mouse over the area next to the right of the line numbers, and left of the code:

```

9   class FabCar extends Contract {
10
11     async initLedger(ctx) {
12       console.info('=====')
13       const cars = [
14         {

```

\_\_ 8. Expand **initLedger** and study its contents so we can understand what it will do.

**Lines 11-82** define the **initLedger** transaction. This is designed to populate the blockchain with 10 sample car definitions to work with. We can see that each car is defined by four properties; color, make, model and owner. After defining an array of 10 cars, it loops through them inserting their definitions into the world state in turn, by calling the **ctx.stub.putState(...)** method giving each car an incrementing index like **CAR1, CAR2** etc as it does so. The **putState** method is made available to the transaction through the context parameter, **ctx** by the framework.

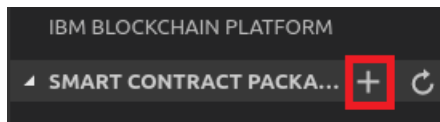
We will look at some of the others as we use them in this lab, but first we will package and install the **fabcar** contract into the **local\_fabric** dev' environment in VSCode.

\_\_ 9. Click on the IBP icon in the sidebar to switch to the IBM Blockchain Platform view.

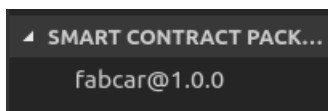


**Note:** The “**IBM Blockchain Platform Home**” page may automatically open again, and if it does, simply close it by clicking on the “**x**”.

\_\_ 10. From the **Smart Contract Packages** view click the “+” icon to package the smart contract into a deployment package. If you do not see the “+”, first click in the **Smart Contract Packages** view.



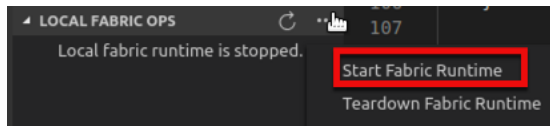
You will first see an informational message about packaging the contract, then you will see a package appear after it is created, called **fabcar@1.0.0**.



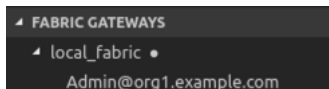
This package is now ready to be installed onto a blockchain peer.

Next, we will create the IBP **local\_fabric** development environment in VSCode.

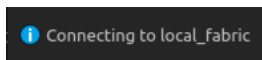
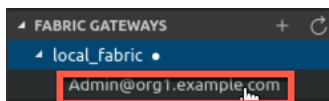
- \_\_ 11. In the **LOCAL FABRIC OPS** view, click on the ... and select **Start Fabric Runtime**.



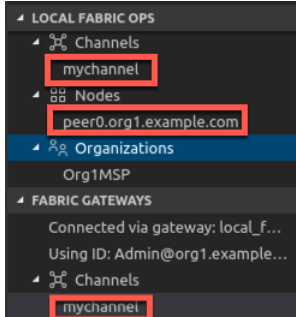
The circle icon next to **local\_fabric** under **FABRIC GATEWAYS** may appear to spin and text will appear in the **Output** window to show progress. Note that this may take a little time to complete.



- \_\_ 12. Once the text "[channelCmd] executeJoin -> INFO 002 Successfully submitted proposal to join channel" appears in the **Output** window and the **local\_fabric** circle icon is solid, click on [admin@org1.example.com](mailto:admin@org1.example.com) under **local\_fabric** in the **FABRIC GATEWAYS** view to connect. At this point there will be an information message confirming that the connection has been made:

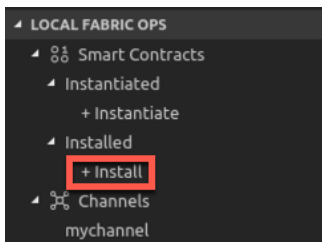


- \_\_\_ 13. When the connection is made, you should see the channel called **mychannel** appear, both under the **LOCAL FABRIC OPS** view and the **FABRIC GATEWAYS** view. Underneath Nodes in the LOCAL FABRIC OPS view, you will see the peer called **peer0.org1.example.com**.

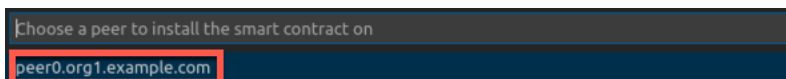


As before, this is the same single-channel, single-peer network that the plugin creates for test and development purposes.

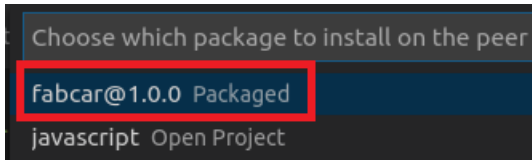
- \_\_\_ 14. Click on +Install in the **LOCAL FABRIC OPS** view. You may need to scroll on the view to see this option.



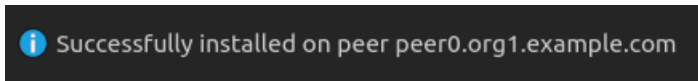
- \_\_\_ 15. From the “**Choose a peer to install the smart contract on**” pop up at the top of the screen, choose “**peer0.org1.example.com**” from the options.



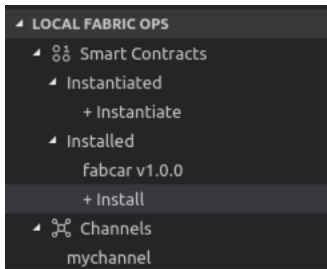
- \_\_\_ 16. From the “**Choose which package to install on the peer**” pop up at the top of the screen, choose “**fabcar@1.0.0 Packaged**” from the options.



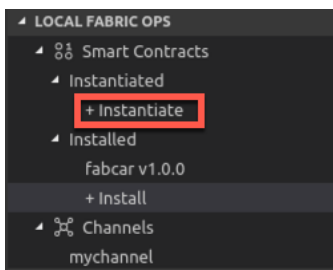
When the package is installed, an information message will be shown confirming the install:



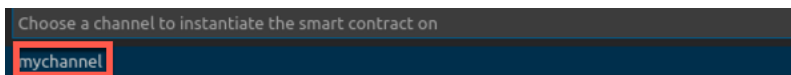
Now under **Installed** in the **LOCAL FABRIC OPS** view you can see the installed contract:



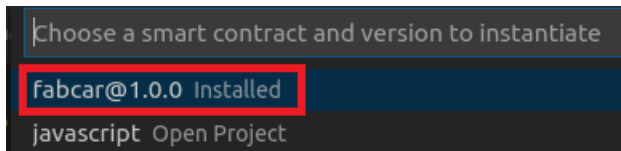
- \_\_\_ 17. Next, we have to instantiate the contract. Click on **+Instantiate** in the **LOCAL FABRIC OPS** view.



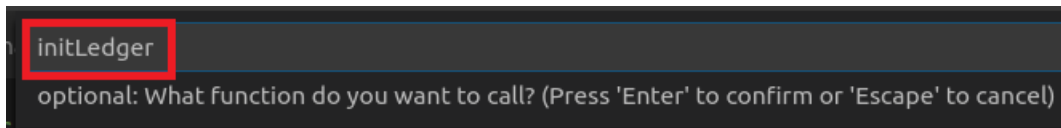
- \_\_\_ 18. From the **“Choose a channel to instantiate the smart contract on”** pop up at the top of the screen, choose **“mychannel”** from the options:



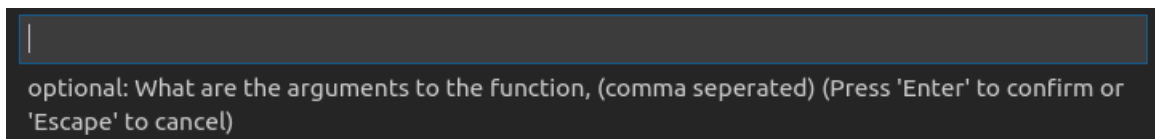
- \_\_ 19. From the “**Choose a smart contract and version to instantiate**” pop up at the top of the screen, choose “**fabcar@1.0.0 Installed**” from the options:



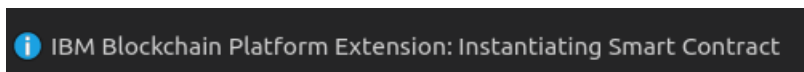
- \_\_ 20. In the pop-up dialogue box at the top of the screen asking “**optional: What function do you want to call? ...**” make sure you enter the word **initLedger** into the entry field as shown below. Before you press enter, check your spelling and make sure it is correct and has the uppercase “L” on Ledger without any quotes or spaces around it. This name has to exactly match the name of the transaction in the contract that will be called at instantiate time and in the fabcar contract as we saw above this is called **initLedger**. This is different to the previous lab, and shows that you can choose the name of a function to be called at instantiate time.



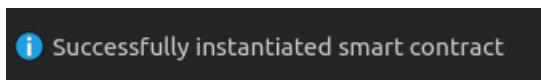
In the next dialogue that asks for parameters to the function, just press “**Enter**” as our **initLedger** function does not require any additional parameters apart from the context “**ctx**” which is automatically provided by the framework.



Instantiating a contract can take several minutes as a new docker container is built to contain the contract. Whilst it is happening you should see this information message



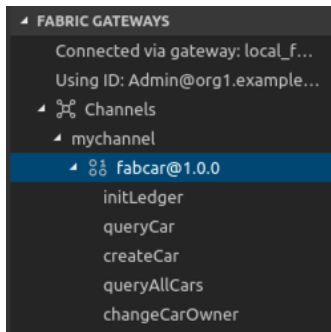
When it is complete you will see this information message



Once complete, in the “**FABRIC GATEWAYS**” view, **mychannel** can be expanded to show **fabcar@1.0.0**.



- \_\_\_ 21. Expand the instantiated contract **fabcar@1.0.0**, and you will see the five transactions that were defined in the **FabCar** contract are now available



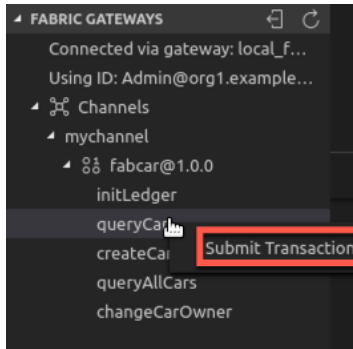
At this point **initLedger** has been called to populate the ledger and the other transactions are ready to be invoked.

In the next few steps we will check that **initLedger** has done its job correctly by querying for a car it should have created on the ledger. To do this we are going to call the **queryCar** transaction from within VSCode, so take a quick look at its implementation. The definition of **queryCar** is between lines 84-91 of **fabcar.js**. It is a very simple transaction that just looks for the car that was passed into the transaction as a parameter and either returns the requested car or an error if it does not exist. If we look at the definition of the first car in the **initLedger** transaction we can see it is defined as a **blue Toyota Prius** owned by **Tomoko**:

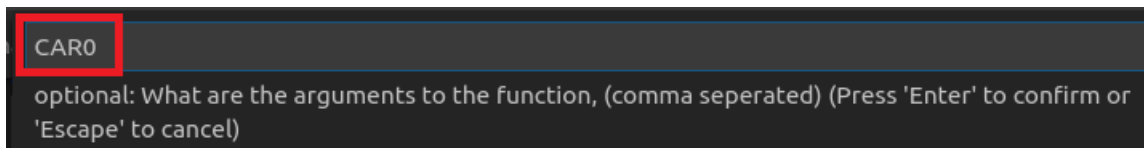
```
const cars = [
  {
    color: 'blue',
    make: 'Toyota',
    model: 'Prius',
    owner: 'Tomoko',
  },
]
```

This was inserted into the world state with the index **CARO** in the loop at the end of **initLedger** and this is the car we are going to query for.

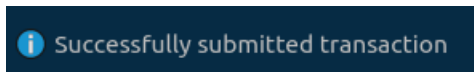
- \_\_\_ 22. From the “**FABRIC GATEWAYS**” view, expand the instantiated contract as you did before until you can see the transactions, right click on “**queryCar**” and choose “**Submit Transaction**”:



- \_\_\_ 23. In the dialogue at the top of the screen enter the text **CAR0** and press “Enter”. Note that you should not enter any quotes around the string as otherwise they will be taken as part of the string itself which will result in an error.



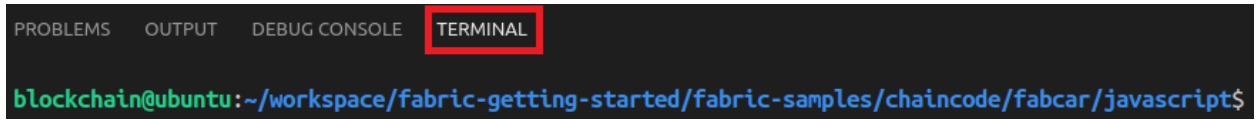
An information message will inform you when the transaction is complete:



Although the transaction is successful you do not see the data from the ledger in the response. This is because the IBP VSCode plugin itself does not currently show responses from transaction directly inside VSCode itself – this is a feature on its backlog for implementation at a later date.

Therefore, we will look at the console output from **queryCar** in the docker logs for the fabcar docker container instead. To be able to see the docker logs, we need to get the name of the docker container that is running the fabcar contract. To do this we could list all the running containers with “**docker ps**” and look for the right one. However, as the names of the containers used by Hyperledger Fabric and the IBP plugin are deterministic, we can simply issue the command in the terminal window inside VSCode.

- \_\_ 24. Switch to the terminal window at the bottom of the VSCode screen:



Note that if your window size is small, you might not be able to see the **Terminal** window and you must first click on the ellipsis (...) to allow you to view it.

- \_\_ 25. At the prompt enter this command (you can copy and paste it if you wish):

```
docker logs fabricvscodefabric-peer0.org1.example.com-fabcar-1.0.0
```

This will produce several lines of output from the initial call to **initLedger**, which you can look at if you wish and at the bottom of which will be the text:

```
{"color": "blue", "docType": "car", "make": "Toyota", "model": "Prius", "owner": "Tomoko" }
```

As you can see, this is the definition of first car, **CAR0**, from **initLedger**, showing that both **initLedger** and **queryCar** have worked as expected.

- \_\_ 26. Now you can spend a little time querying the other cars in the range **CAR0** through **CAR9** and looking at the docker logs output and comparing them to the corresponding values in **initLedger**. To see at the docker logs output, switch back to the **Terminal** view and press the “**up-arrow**” key to choose the same **docker logs** command again. If you have trouble doing this, you can just re-enter it as shown above.

- \_\_ 27. Next, try entering an invalid car index, such as **CAR99** and again looking at the **docker logs** output. This time you will see text containing the error as thrown by **queryLedger**. The error is shown twice, from where it was handled at different places in the framework code. The first error also shows the file and line number where the error was thrown from which in this case is **fabcar.js:87** and if you look at **line 87** in **fabcar.js** you can see the **throw** statement:

```
throw new Error(`${carNumber} does not exist`);
```

The two errors you see will look like this:

```
ERROR [contracts-spi/chaincodefromcontract.js]
{"message":"CAR99 does not exist","stack":"Error: CAR99 does not
exist\n    at FabCar.queryCar (/usr/local/src/lib/fabcar.js:87:19)\n
at <anonymous>\n    at process._tickCallback
(internal/process/next_tick.js:188:7)"}

```

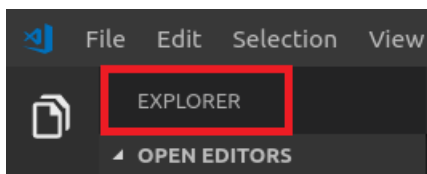
```
ERROR [lib/handler.js]
[mychannel-65455332]Calling chaincode Invoke() returned error response
[Error: CAR99 does not exist]. Sending ERROR message back to peer

```

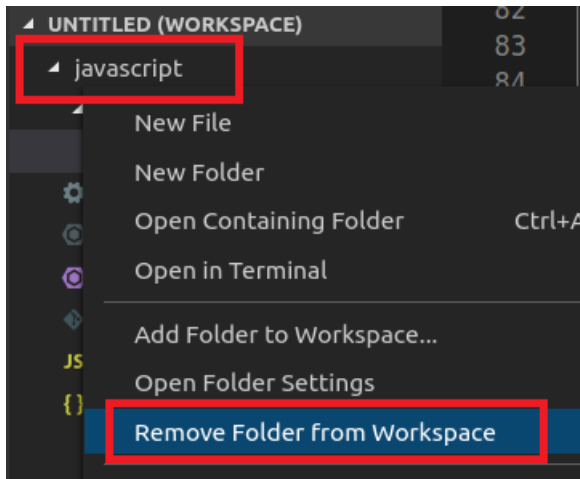
Now we are going to use an application outside on VSCode to perform a query instead of using the docker logs. Several applications come with the fabcar sample for calling the different transactions, and we are going to edit the query one to use the local\_fabric blockchain created by VSCode.

First, we will remove the **fabcar** smart contract folder from the VSCode workspace.

- \_\_ 28. Switch to the **Explorer** view in VSCode.  
**Note:** If you are having problems and cannot see the **Explorer** view for any reason, click on its icon in the activity bar (📁) or press “**ctrl + shift + e**” to show it.



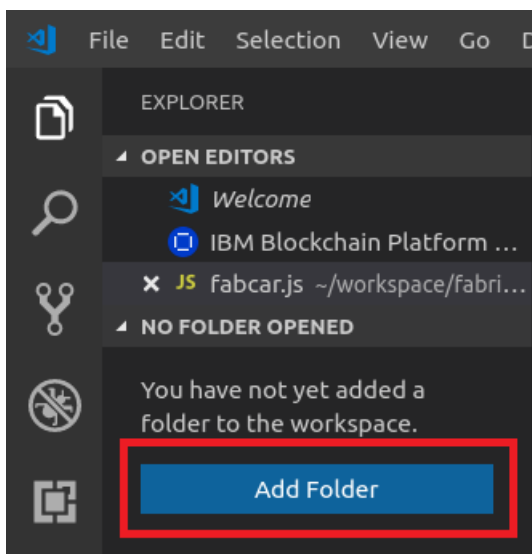
- \_\_\_ 29. Right click on the **javascript** folder in the Explorer view and chose the “**Remove Folder from Workspace**” context menu option.



**Note:** The “IBM Blockchain Platform Home” page may automatically open once the **fabcar** folder has been removed from the workspace. If it does, simply close it by clicking on the “x”.

Note 2: Do not close the fabcar.js file which should still be open as you may want to refer to it later in this lab

- \_\_\_ 30. In the empty workspace, click the **Add Folder** button from the Explorer view as shown below:



\_\_ 31. Using the screen shot below as a guide, navigate to open a folder as follows:

**Step 1:** Click in the **workspace** folder on the bottom left of the dialogue.

**Step 2:** Navigate to the folder:

**fabric-getting-started/fabric-samples/fabcar**

**Step 3:** Select the folder **javascript**

Note that this is a **different javascript folder** to the one used by the contract earlier.

**Step 4:** Click the **Add** button on the bottom right.

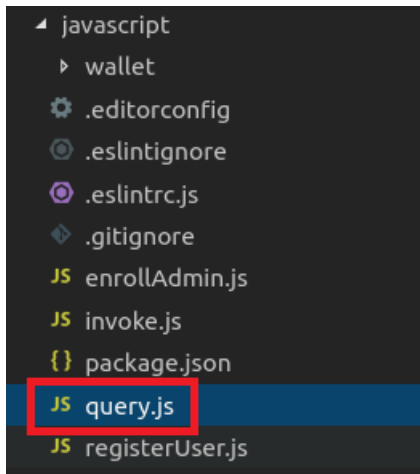


**Note:** the full path to the **javascript** folder you are importing for reference is:

`/home/blockchain/workspace/fabric-getting-started/fabric-samples/fabcar/`

**Note 2:** The “**IBM Blockchain Platform Home**” page may automatically open once the **javascript** folder has been added to the workspace. If it does, simply close it by clicking on the “**x**”.

- \_\_ 32. Once the folder is added to the workspace, double click on the **query.js** application to open it in the main editing view.



Have a look at the source code for **query.js**. This is a simple application designed to connect to a Hyperledger Fabric network and issue an **evaluateTransaction**. An **evaluateTransaction** is a transaction that does not get sent for ordering into a block and so is normally used for query transactions.

This sample application is designed to connect to a sample network that comes with the fabric-samples. However, we need to change it to connect to the network that VSCode has created so we can query the ledger we have been using earlier in this lab.

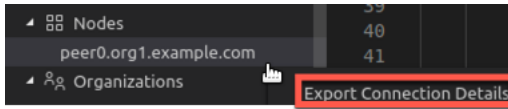
But before we can edit the application, we first need to export the connection details from the VSCode network so we can use them in our query application.

- \_\_ 33. Click on the IBP icon in the sidebar to switch to the IBM Blockchain Platform view.

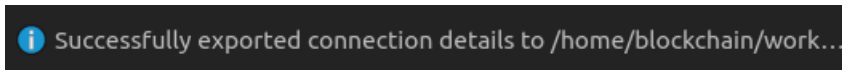


**Note:** The “**IBM Blockchain Platform Home**” page may automatically open again, and if it does, simply close it by clicking on the “**x**”.

- \_\_\_ 34. In the **LOCAL FABRIC OPS** view, Right click on the **peer0.org1.example.com** and choose the **Export Connection Details** option



You should see an informational message telling you the export was successful:

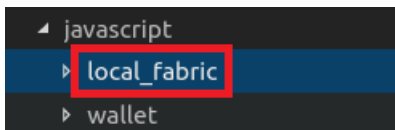


The full path of the export is:

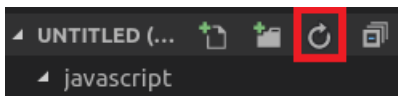
```
/home/blockchain/workspace/fabric-getting-started/fabric-samples/fabcar/javascript/local_fabric
```

This shows that the connection details have been exported into a folder called **local\_fabric** inside the **javascript** folder we opened earlier to load the fabcar sample applications. The path is also shown in the **output** view and the new **local\_fabric** folder is shown in the explorer view as well.

- \_\_\_ 35. Switch back to the explorer view and you should see the newly created **local\_fabric** folder under the main **javascript** one:

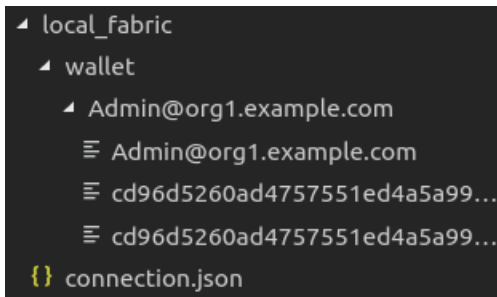


**Note:** If you do not see the **local\_fabric** folder, click the refresh icon on the workspace bar just above the javascript folder:





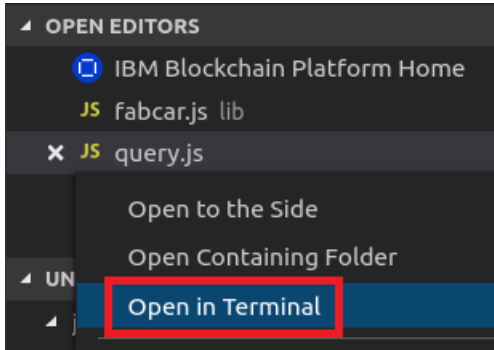
\_\_ 36. Expand the **local\_fabric** folder and have a look at what was exported:



You can see it has exported two main things. Firstly, there is a **connection.json** file that contains the details about where the **local\_fabric** blockchain is running. Secondly, there is a **wallet** folder inside of which is an identity for the administrator of this blockchain peer, called **Admin@org1.example.com** along with their keys required for connection.

Feel free to open up these files and take a look at the details, but be careful not to make any changes to them. Make sure you close them without saving once you are finished.

\_\_ 37. In the **Open Editors** part of the **Explorer** view, right click on the **query.js** file and choose **Open in Terminal**:



- \_\_\_ 38. At the Terminal prompt type **npm install** and press enter. This installs the node modules required by the applications like **query.js** in the fabcar javascript folder. This will take a few minutes as it downloads the modules and when it is complete, your **Terminal** should look similar to this:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
node-pre-gyp WARN Using request for node-pre-gyp https download
[grpc] Success: "/home/blockchain/workspace/fabric-getting-started/fabric-samples/fabcar/
extension_binary/node-v57-linux-x64-glibc/grpc_node.node" is installed via remote
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN fabcar@1.0.0 No repository field.

added 517 packages from 1049 contributors and audited 1694 packages in 47.416s
found 0 vulnerabilities

blockchain@ubuntu:~/workspace/fabric-getting-started/fabric-samples/fabcar/javascript$
```

\_\_ 39. We will now make a few simple changes to **query.js** to make it work with the **local\_fabric** network. You can copy and paste or type these changes manually into **query.js**:

**Line 11:** This line resolves the path to the connection profile to use to connect to the network. As we need to use the one from **local\_fabric**, change the line to be:

```
const ccpPath = path.resolve(__dirname, 'local_fabric', 'connection.json');
```

**Line 19:** This line gets the path to the wallet to use. As we again need to use the one from **local\_fabric**, change the line to be:

```
const walletPath = path.join(process.cwd(), 'local_fabric/wallet');
```

**Line 24:** This line loads the user from the wallet and as we are using a different user from **local\_fabric**, we need to change the line to be:

```
const userExists = await wallet.exists('Admin@org1.example.com');
```

**Line 26:** This line simply logs an error if the user does not exist, so for completeness change the text "**user1**" to be "**Admin@org1.example.com**".

**Line 33:** This line connects to the network gateway using a specific user, so again change the text "**user1**" to be "**Admin@org1.example.com**".

**Line 44:** This line actually calls **evaluateTransaction** to issue the specified transaction. As we want to query a single car, **CAR0**, rather than all cars, change the line to call the **queryCar** transaction that we called earlier from within VSCode, passing in **CAR0** as a parameter:

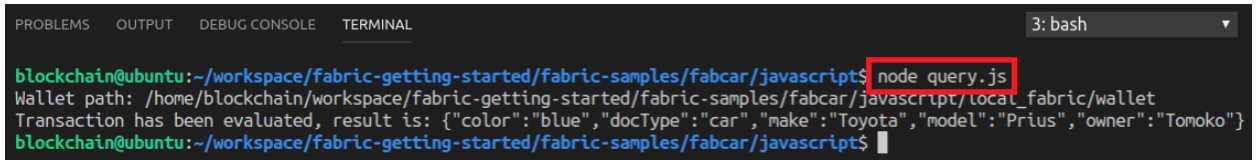
```
const result = await contract.evaluateTransaction('queryCar', 'CAR0');
```

**Line 45:** This line logs the result from calling the transaction to the console. As we want the output to format properly at the console, change this line to be:

```
console.log(`Transaction has been evaluated, result: ${JSON.parse(result)}`);
```

\_\_ 40. Save the file by pressing **ctrl + s** or use the **File / save** menu option.

- \_\_ 41. From the Terminal type **node query.js** and press enter to run the command. If you have made the above edits correctly, you should see output like this below:



```
blockchain@ubuntu:~/workspace/fabric-getting-started/fabric-samples/fabcar/javascript$ node query.js
Wallet path: /home/blockchain/workspace/fabric-getting-started/fabric-samples/fabcar/javascript/local_fabric/wallet
Transaction has been evaluated, result is: {"color":"blue","docType":"car","make":"Toyota","model":"Prius","owner":"Tomoko"}
blockchain@ubuntu:~/workspace/fabric-getting-started/fabric-samples/fabcar/javascript$
```

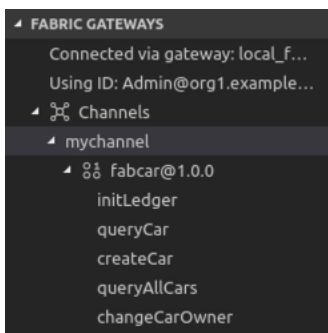
As you can see, the output is the same output for **CARO** that we got earlier which is a **blue Toyota Prius**, owner by **Tomoko**.

Next we are going to call the **changeCarOwner** transaction to update the owner of **CARO**, and then we will query the ledger again to see the result.

- \_\_ 42. Switch back to the IBM Blockchain Platform view by clicking on the IBP icon in the sidebar:



- \_\_ 43. Expand myChannel and the FabCar contract so you can see the transactions.

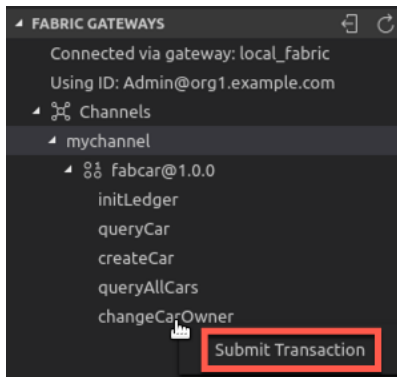


- 44. Before we call it, let's take a look at the **changeCarOwner** transaction implementation. The definition is between lines 140-150 of **fabcar.js** which should still be open in your workspace from earlier. It is a relatively simple transaction that takes two parameters, the car and a new owner and first looks for the car that was passed into the transaction as a parameter. If it does not find it, it will throw an error back to the user. However, if it exists, it will de-serialise it, update the owner and then store it back in the world state with the **putState** method.

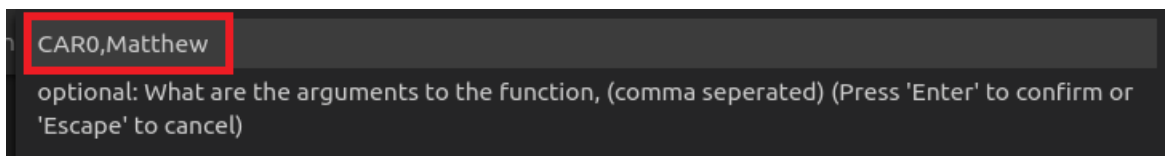
**Note:** If you have closed **fabcar.js** you can reopen the it from the **File / Open** menu, from this path:

```
workspace/fabric-getting-started/fabric-samples/chaincode/fabcar/javascript/lib/fabcar.js
```

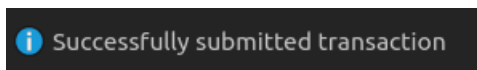
- 45. Right click on the **changeCarOwner** transaction under the **FabCar** contract in the **FABRIC GATEWAYS** view, and choose **Submit Transaction**.



- 46. In the dialogue at the top of the screen enter the text **CAR0,Matthew** and press "Enter". Note that you should not enter any quotes or spaces around this string as otherwise they will be taken as part of the string itself which will result in an error.



When the transaction has finished, you should see the information message:



**Note:** If instead you get an error, you may have entered the string incorrectly. You can check the **docker logs** for an error as you did earlier in this lab and try this step again.

- \_\_\_ 47. Once the transaction has been successfully submitted, back in the **Terminal** window, enter **node query.js** again to see the resulting changes.

```
blockchain@ubuntu:~/workspace/fabric-getting-started/fabric-samples/fabcar/javascript$ node query.js
Wallet path: /home/blockchain/workspace/fabric-getting-started/fabric-samples/fabcar/javascript/local_fabric/wallet
Transaction has been evaluated, result is: {"color":"blue","docType":"car","make":"Toyota","model":"Prius","owner":"Matthew"}
blockchain@ubuntu:~/workspace/fabric-getting-started/fabric-samples/fabcar/javascript$
```

You should be able to see that the owner for **CARO** has been updated to **Matthew**.

We will now make one final change to the ledger, by creating a new car using the **createCar** transaction. As we did with **queryCar**, look at the **createCar** transaction in **fabcar.js**. It is defined between lines 93-106 and takes 5 caller-passed-in parameters: **carNumber**, **make**, **model**, **color** and **owner**. The transaction uses these parameters to create an object with the parameters and passes a serialised version of the object to the **putState** method.

- \_\_\_ 48. As you did with **queryCar** above, right click on the **createCar** transaction under the **FabCar** contract in the **FABRIC GATEWAYS** view, and choose **Submit Transaction**.
- \_\_\_ 49. In the dialogue at the top of the screen enter the text for your new car, using **CAR10** as the first parameter as that is the next index available. Enter some value of your choice like **CAR10,black,Tesla,Model X,Matthew** and press "Enter". Note that you should not enter any quotes or extra spaces around this string as otherwise they will be taken as part of the string itself which will result in an error.

```
CAR10,black,Tesla,Model X,Matthew|
optional: What are the arguments to the function, (comma separated) (Press 'Enter' to confirm or 'Escape' to cancel)
```

- \_\_\_ 50. When the transaction has completed successfully, update **query.js** to query for the car with the index you created, in this case **CAR10**.

```
41 // Evaluate the specified transaction.
42 // queryCar transaction - requires 1 argument, ex: ('queryCar', 'CAR4')
43 // queryAllCars transaction - requires no arguments, ex: ('queryAllCars')
44 const result = await contract.evaluateTransaction('queryCar', 'CAR10');
45 console.log(`Transaction has been evaluated, result is: ${result.toString()}`);
```

**Note:** Remember to save the file before you run it.

- \_\_ 51. Run **node query.js** again from the **Terminal** as you did before.

You should see the details of the new car in the output window:

```
blockchain@ubuntu:~/workspace/fabric-getting-started/fabric-samples/fabcar/javascript$ node query.js
Wallet path: /home/blockchain/workspace/fabric-getting-started/fabric-samples/fabcar/javascript/local_fabric/wallet
Transaction has been evaluated, result is: {"color":"Model X","docType":"car","make":"black","model":"Tesla","owner":"Matthew"}
blockchain@ubuntu:~/workspace/fabric-getting-started/fabric-samples/fabcar/javascript$
```

- \_\_ 52. We are now coming towards the end of part two of this lab. In case you have some spare time, here are a few slightly more advanced (but optional things) you could try if you want to branch out a little on your own.

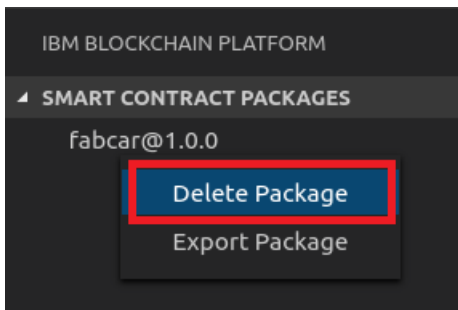
First you could update **query.js** to run **queryAllCars** to return the details of all cars instead on just one.

Secondly, you could look at another application that comes with the **fabcar** sample – **invoke.js** which you can find in the same folder as **query.js**. This is a very similar application to **query.js** except that it uses **submitTransaction** instead of **evaluateTransaction** to make changes to the ledger.

Following the steps above for updating **query.js** to use the **local\_fabric** blockchain network, you could make the same changes to **invoke.js** so it can create another new car in the **local\_fabric** network from the **Terminal** instead of using VSCode to do it. Remember to run **query.js** afterwards to make sure your new car made it to the ledger.

We have now almost completed this lab – using an existing contract. All that remains is to clear up the environment ready for the next lab.

- \_\_ 53. From the IBP **Smart Contract Packages** view, right-click on the **fabcar@1.0.0** package and choose the **Delete Package** option as shown below to remove it:



## IBM Blockchain

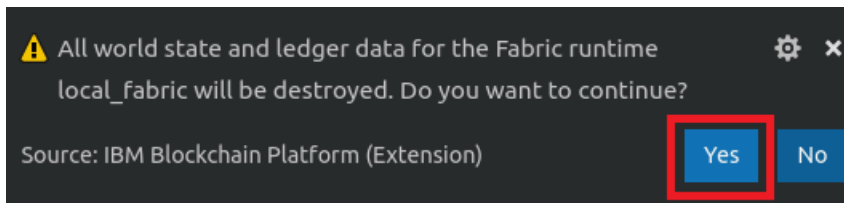
- \_\_\_ 54. From the **FABRIC GATEWAYS** view, select the **Disconnect from Gateway** icon as shown below:



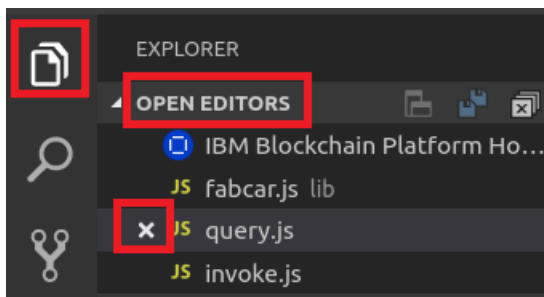
- \_\_\_ 55. From the **LOCAL FABRIC OPS** view, click on the ... and select the “**Teardown Fabric Runtime**” option from the context menu:



- \_\_\_ 56. From the dialogue that appears in the bottom right, choose the “Yes” button:



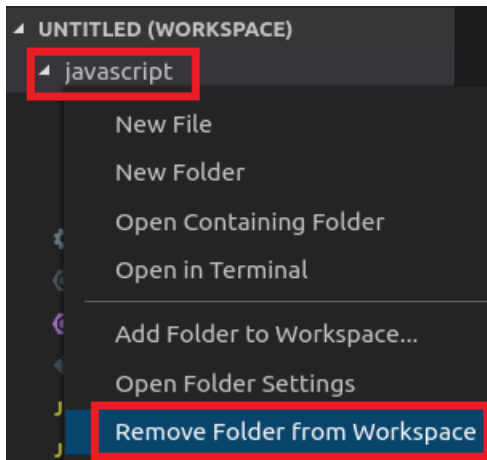
- \_\_\_ 57. Switch back to the **Explorer** view and close all open editors in the **Open Editors** view, including the “**IBM Blockchain Platform Home**” by clicking on the “x” button on each one in turn:





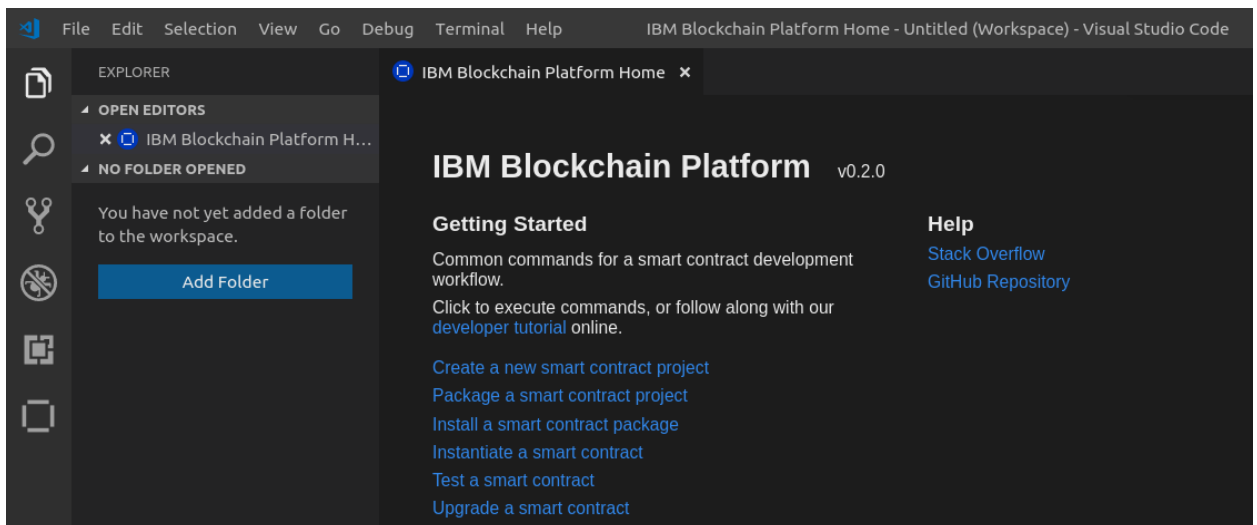
## IBM Blockchain

- \_\_\_ 58. Right click on the “**javascript**” folder and chose the “**Remove Folder from Workspace**” context menu option.



**Note:** if you cannot see the **Explorer** view, click on the **Explorer** icon again to make it re-appear.

- \_\_\_ 59. This may re-open the “**IBM Blockchain Platform Home**” page and leaves your workspace ready for the next lab as shown below:



- \_\_\_ 60. We have now completed this lab – **Using an Existing Contract** and we hope you enjoyed it.

