

IBM Blockchain Platform Hands-On

Lab 2:

Using an Existing Smart Contract



Table of Contents

Disclaimer	3
1 Overview of the lab environment and scenario	5
1.1 Lab Scenario	6
2 Using an Existing Contract	7
3 Next Steps	36
We Value Your Feedback!	37

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.

© 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 environment and scenario

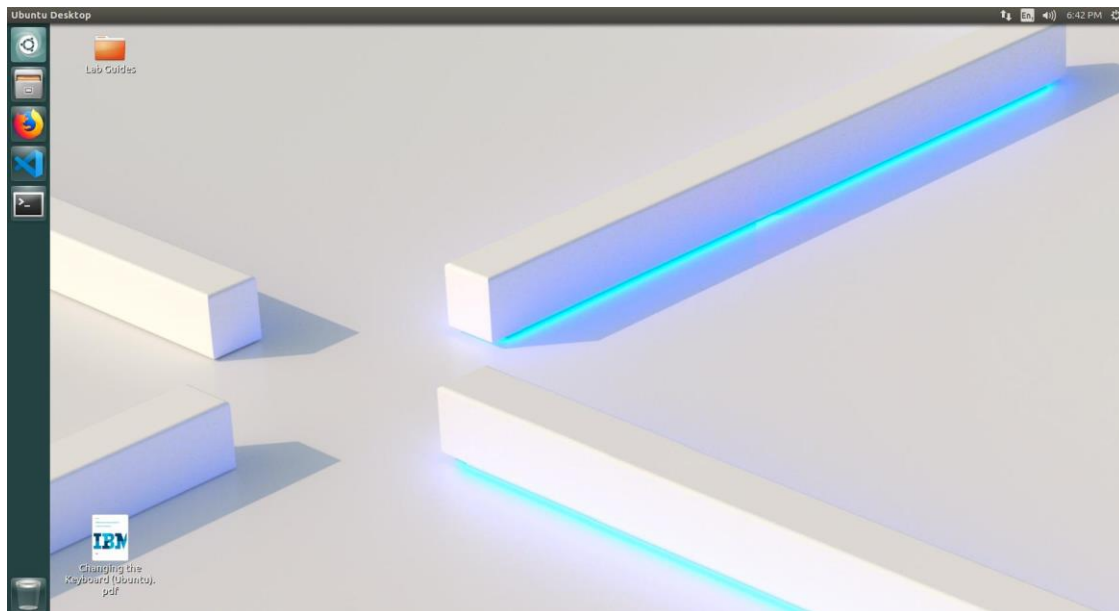
This lab is a technical introduction to blockchain, specifically smart contract development using the latest developer enhancements in the Linux Foundation's Hyperledger Fabric v1.4 and shows you how IBM's Blockchain Platform's developer experience can accelerate your pace of development.

Note: The screenshots in this lab guide were taken using version **1.37.1** of **VS Code**, and version **1.0.9** of the **IBM Blockchain Platform** extension. If you use different versions, you may see differences from 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. Your instructor will tell you how to do this if you are unsure.
- 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.

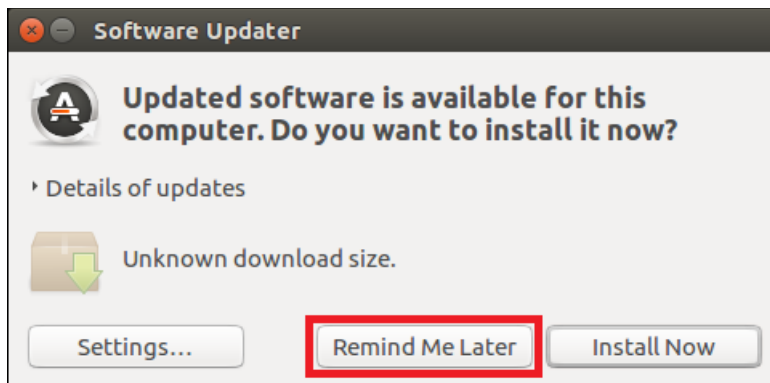
Note: If it asks you to login, the userid and password are both “**blockchain**”.



1.1 Lab Scenario

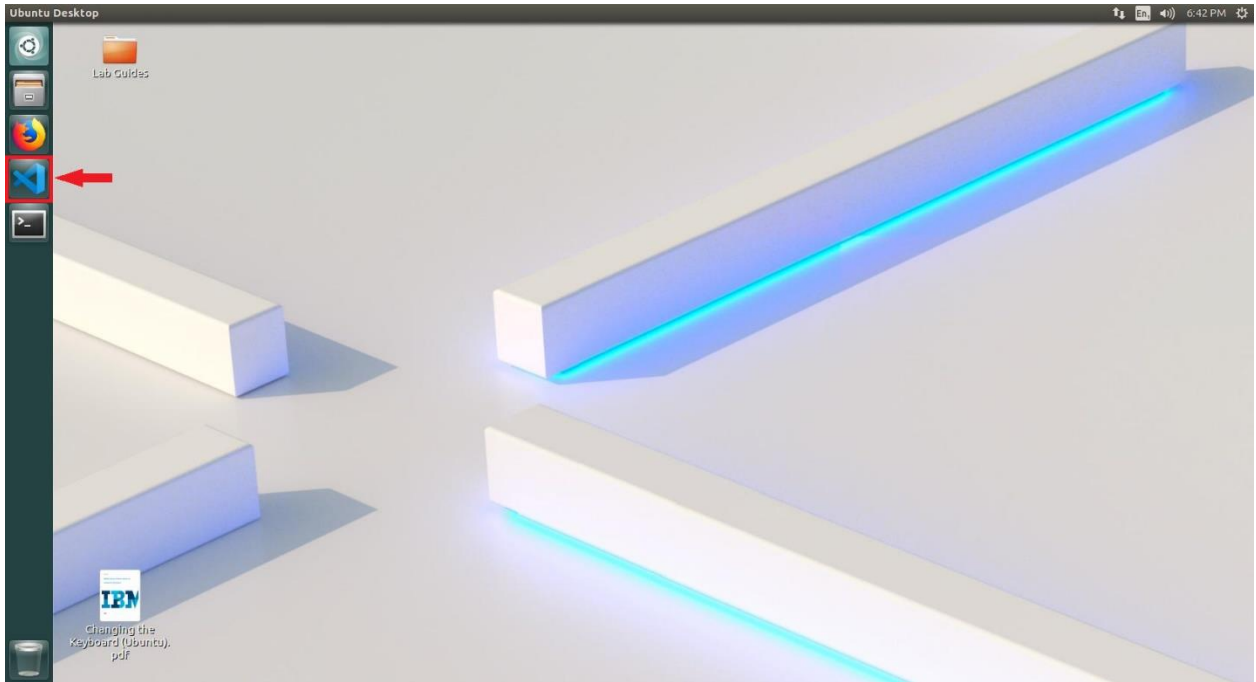
In this lab, we will take you through using a sample smart contract that comes with Hyperledger Fabric with VS Code, where you will learn how to import contracts and interact with the development environment in more detail. After importing and deploying the contract, you will then use a client application 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**”:

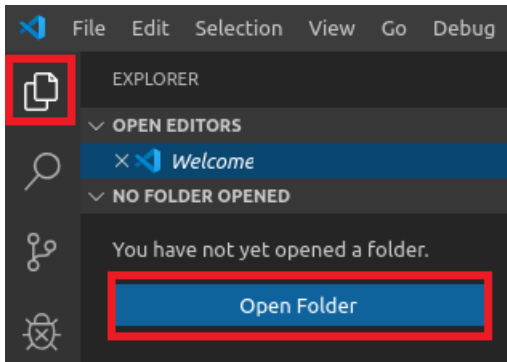



2 Using an Existing Contract

- __ 3. If you have just finished **Lab 1**, VS Code will already be open so skip this step and move straight to **Step __4**. If you are starting with **Lab 2**, launch VS Code by clicking on the VS Code Icon in the toolbar.



- __ 4. Now VS Code is open, click the **Explorer** icon in the top left of the VS Code activity bar to show the Explorer view, and then click the **Open 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.

__ 5. Using the screen shot below as a guide, navigate to open a folder as follows:

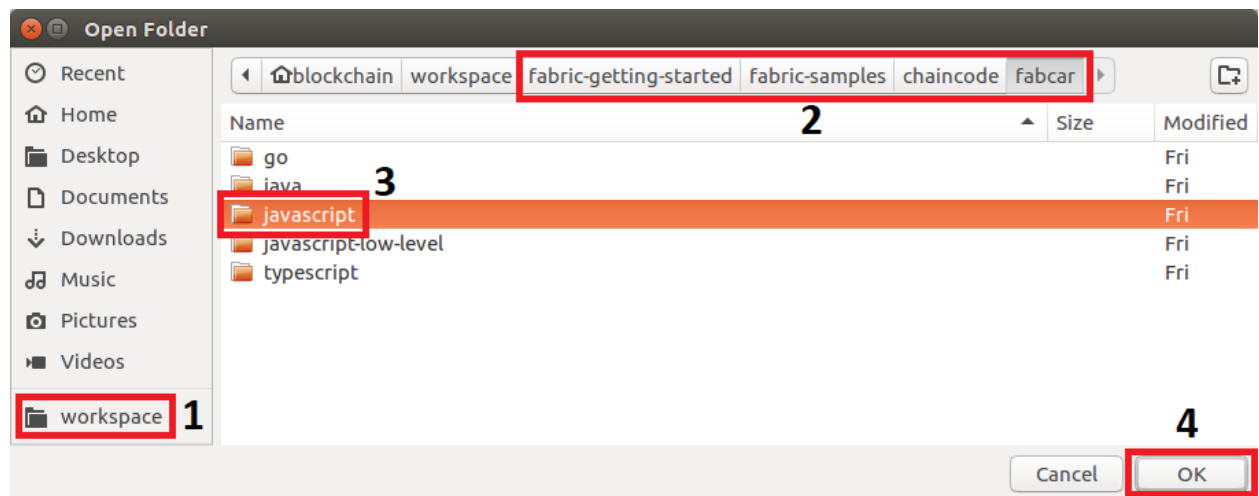
Step 1: Click in the **workspace** folder on the bottom left of the dialogue.

Step 2: Navigate to the folder:

fabric-getting-started/fabric-samples/chaincode/fabcar

Step 3: Select the folder **javascript**

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

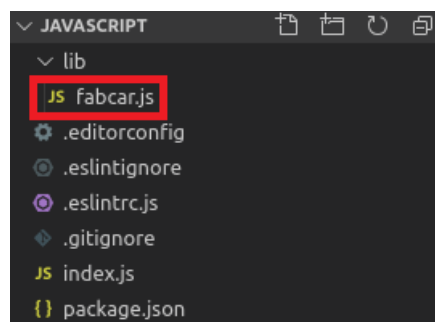


Note: the full path to the **javascript** folder you are importing for reference is:

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

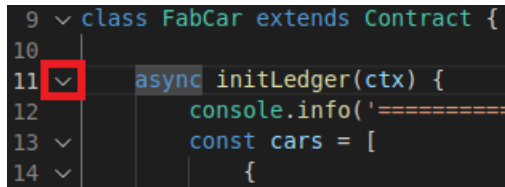
Make sure you have opened the **javascript** folder in the **chaincode** folder.

__ 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 five methods, or ‘transactions’, so we can see a simpler overview of the contract we will be using.

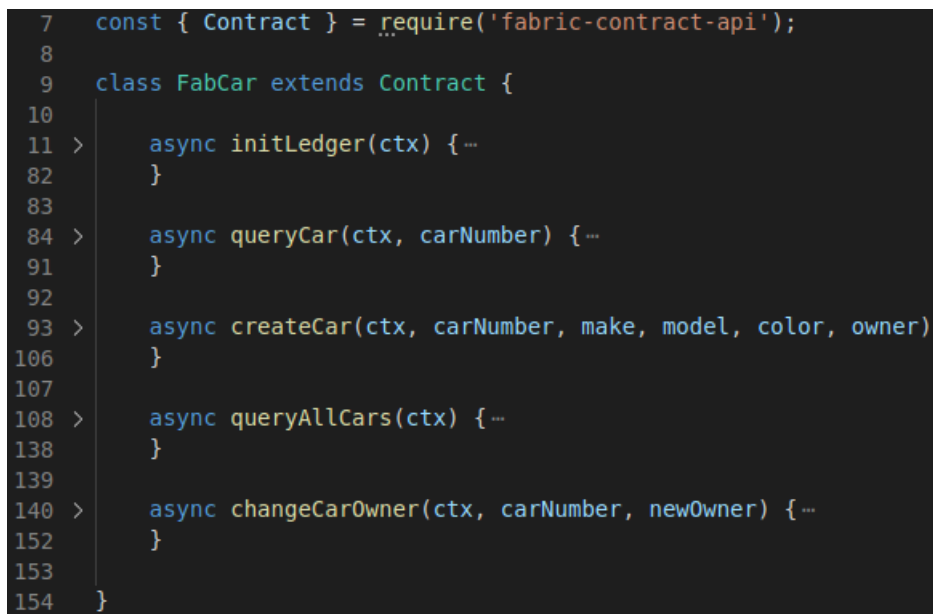
Note: The “>” buttons only appear when you move your mouse over the area next to the right of the line numbers, and left of the code:

A screenshot of a code editor with a dark background. Line 11 is highlighted, and a red square highlights a collapse button (a downward-pointing chevron) next to the line number. The code on line 11 is `async initLedger(ctx) {`.

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

The code starts with an import of a **Contract** definition from the **fabric-contract-api** node module on **line 7**. This makes the Fabric API available to the smart contract for use.

After this, there are five transactions that make up the **fabcar** sample contract and most of the transactions take parameters and will either query or update a blockchain.

A screenshot of a code editor showing the full **fabcar.js** file. The code is as follows:

```
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  }
```

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

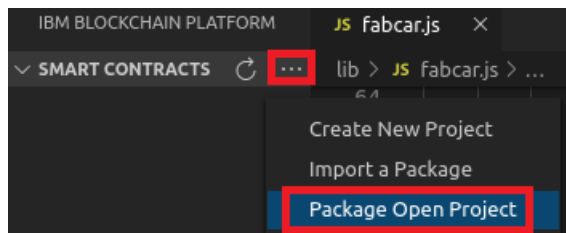
Lines 11-82 define the **initLedger** transaction. This is designed to populate the blockchain with 10 sample car definitions to work with. We can see that each car is defined by four properties; **color**, **make**, **model** and **owner**. After defining an array of 10 cars, it loops through them inserting 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 and is always the first parameter to a transaction.

We will look at some of the other transactions as we use them in this lab, but first we will package and install the **fabcar** contract into the **Local Fabric** development environment.

- __ 9. Click on the IBM Blockchain Platform icon in the sidebar to switch to the IBM Blockchain Platform view.



- __ 10. From the **Smart Contracts** view click on the '**More Actions**' icon (...) and select "**Package a Smart Contract Project**" to package the contract into a deployment package. If you do not see the "...", first click in the **Smart Contract** view.



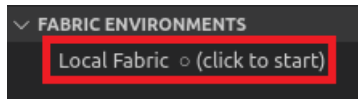
You will first see an **information message** about packaging the contract, then you will see a package appear after it is created, called **fabcar@1.0.0**.



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

Next, we will create the **Local Fabric** development environment in VS Code.

- __ 11. In the **Fabric Environments** view, click on the text “**Click to start**”.

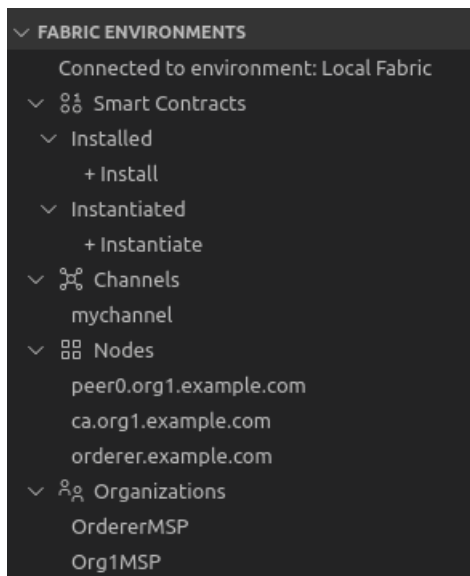


The text will change to “**Local Fabric runtime is starting**” and the circle icon next to it should appear to spin and text will appear in the **Output** window to show progress along with some **information messages**.

Note: this may take a little time to complete.

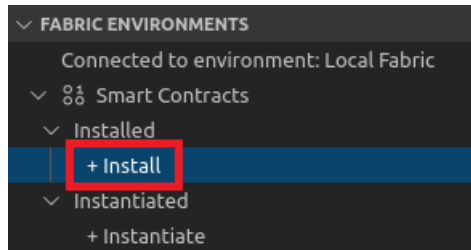
- __ 12. Once the text “[SUCCESS] Connected to Local Fabric” appears in the **Output** window, the connection is complete.

Once the connection is complete, the **Fabric Environment** view changes to show the Smart Contracts, Channels, Nodes and Organizations that make up the network:

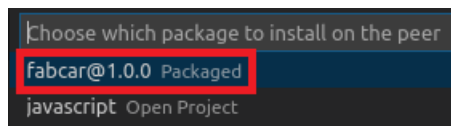


As before in Lab 1, you can see that this is the same single-channel, single-peer network that the extension creates for test and development purposes.

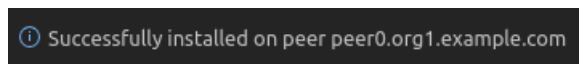
__ 13. Click on “**Install**” under “**Smart Contracts**”:



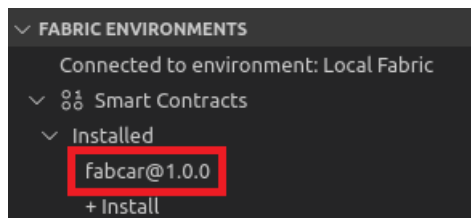
__ 14. From the “**Choose which package to install on the peer**” pop up at the top of the screen, choose “**fabcar@1.0.0 Packaged**”:



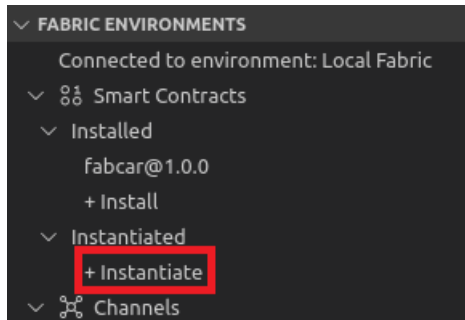
When the package is installed, a **information message** will be shown confirming the install:



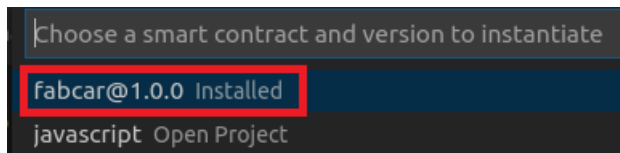
Now under “**Installed**” you can see the **fabcar@1.0.0** contract has indeed been installed:



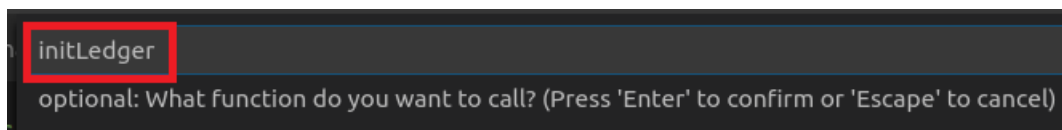
- __ 15. Next, we must instantiate the contract. Click on “**Instantiate**” under “**Smart Contracts**”:



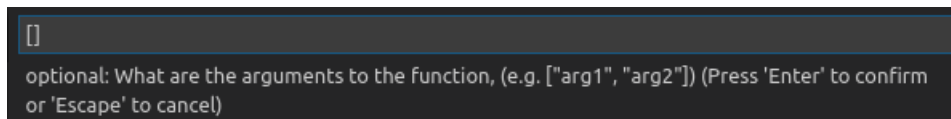
- __ 16. From the “**Choose a smart contract and version to instantiate**” pop up at the top of the screen, choose “**fabcar@1.0.0 Installed**” from the options:



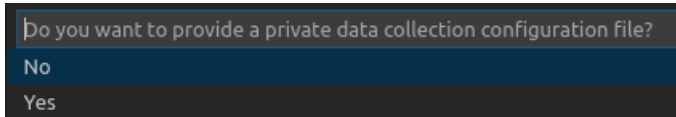
- __ 17. In the pop-up dialogue box at the top of the screen asking “**optional: What function do you want to call? ...**” make sure you enter the word **initLedger** into the entry field as shown below. Before you press enter, check your spelling and make sure it is correct and has the uppercase “**L**” on Ledger without any quotes or spaces around it. This name must 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.



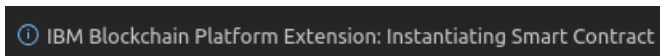
- __ 18. In the next dialogue that asks for parameters to the function, just press “**Enter**” as our **initLedger** function does not require any additional parameters apart from the context “**ctx**” which is automatically provided by the framework.



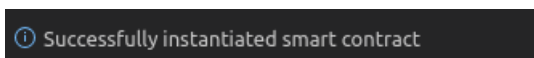
- __ **19.** In the next dialogue that asks “**Do you want to provide a private data collection configuration file**” choose the default “**No**”



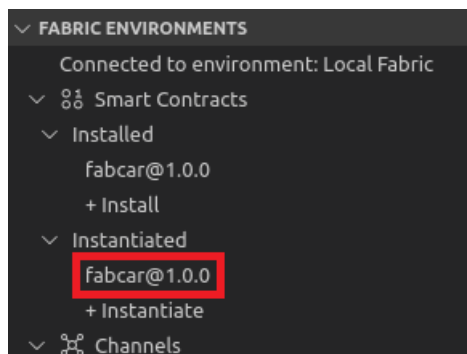
Instantiating a contract can take several minutes as a new docker container is built to contain the contract. Whilst it is happening you should see text appear in the **OUTPUT** window as well as this **information message**:



When it is complete you will see this **information message**:

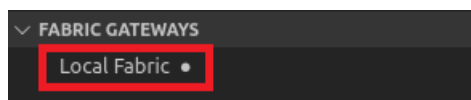


Once complete, under “**Instantiated**” you can see the **fabcar@1.0.0** contract has indeed been instantiated:

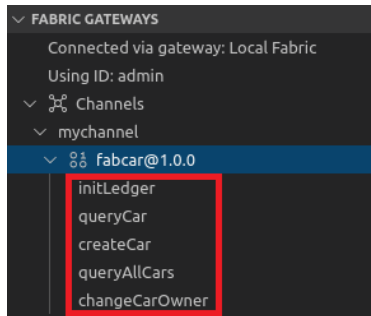


Now we need to connect to the network through a gateway so we can start issuing transactions.

- __ **20.** Under “**Fabric Gateways**” click on “**Local Fabric**”:



- __ 21. Once connected through the gateway, under “**Channels**” expand “**mychannel**” to see the **fabcar@1.0.0** contract. If you now expand the contract as well, underneath it you will see that 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 VS Code, so let's take a quick look at its implementation on lines 84-91 of **fabcar.js**:

```
84     async queryCar(ctx, carNumber) {  
85         const carAsBytes = await ctx.stub.getState(carNumber); // get the car from chaincode state  
86         if (!carAsBytes || carAsBytes.length === 0) {  
87             throw new Error(`${carNumber} does not exist`);  
88         }  
89         console.log(carAsBytes.toString());  
90         return carAsBytes.toString();  
91     }
```

As you can see, this 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**:

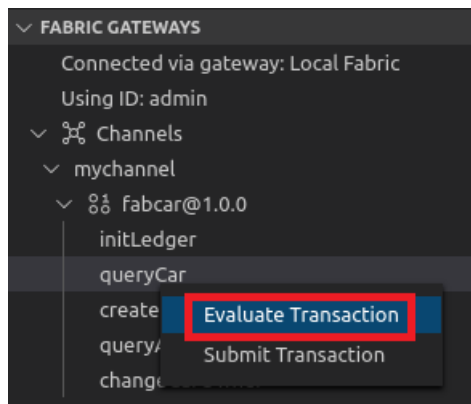
```
const cars = [  
    {  
        color: 'blue',  
        make: 'Toyota',  
        model: 'Prius',  
        owner: 'Tomoko',  
    },  
];
```

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.

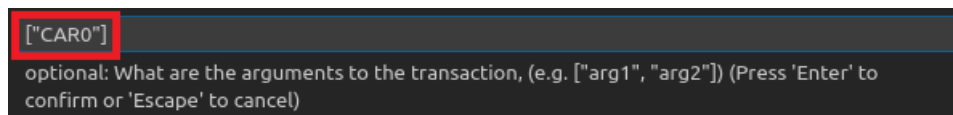
- __ **22.** Make sure the “**Fabric Gateways**” view is still expanded and showing the transactions that are part of the **fabcar@1.0.0** contract.

Now, right click on “**queryCar**” and choose “**Evaluate Transaction**” as shown below.

We use “**Evaluate Transaction**” as **queryCar** only performs a read operation so we do not need to record this on the ledger. If we did want to record the fact that **queryCar** was called, perhaps for audit purposes, we could use “**Submit Transaction**” instead and the query would then be recorded.

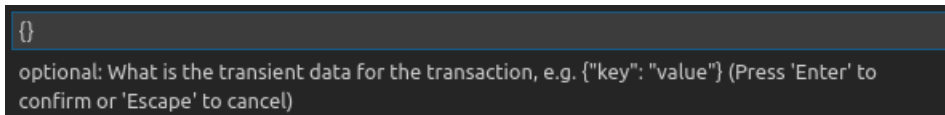


- __ **23.** In the dialogue at the top of the screen enter the text “**CAR0**” inside the square brackets as shown and press “Enter”.

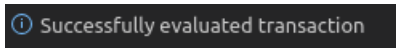


Note: The brackets and the quotes are required as you enter the parameters using JSON syntax.

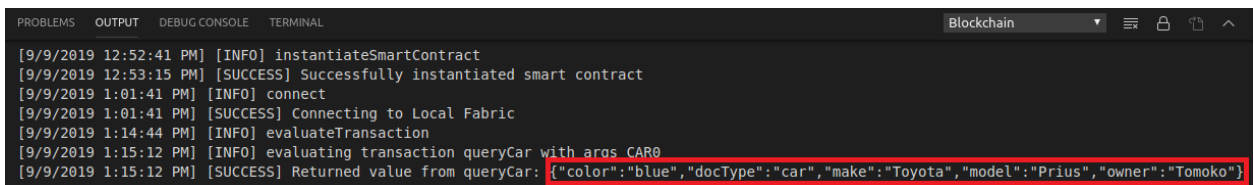
__ 24. As we have no “transient” data for this transaction, press enter at the next prompt:



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



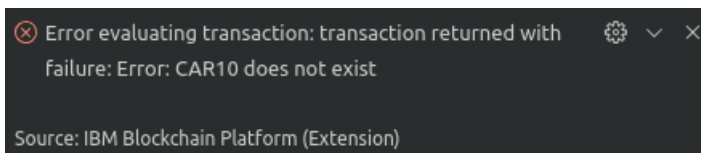
At the same time, the **OUTPUT** window will show the response from calling **queryCar**:



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

__ 25. Now you can spend a little time querying the other cars in the range **CAR0** through **CAR9** and looking at the responses in the **OUTPUT** window and comparing them to their corresponding values in **initLedger**.

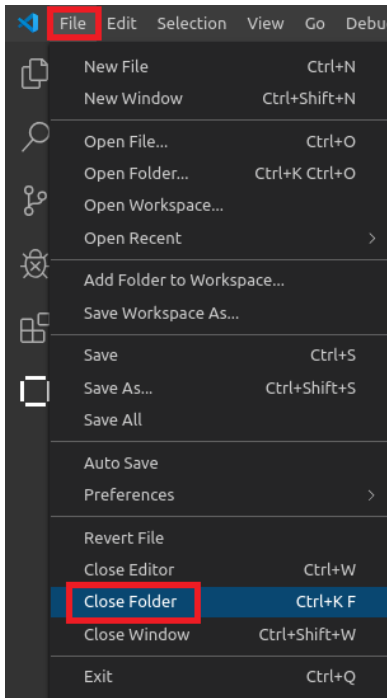
__ 26. Now let's look at how errors are handled in VS Code. To do this let's query for a car that does not exist, such as **CAR10**. Perform a query for **CAR10** and you should see this **information message** which contains the error text from the **queryCar** exception:



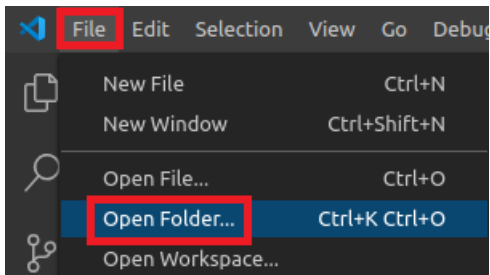
Now we are going to use an application outside of VS Code to perform a query instead of using the **OUTPUT** window. Several applications come with the **fabcar** sample for calling the different transactions, and we are going to edit the query one to use the Local Fabric blockchain created by VS Code.

To do this, we need to load the application source code into VS Code, so first, we will remove the **fabcar** smart contract folder from VS Code.

__ 27. From the **File** menu choose “**Close Folder**” to close the smart contract source folder:



__ 28. From the **File** menu, choose “**Open Folder**” to open the application source folder:



__ **29.** Using the screen shot below as a guide, navigate to open the application source folder as follows:

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

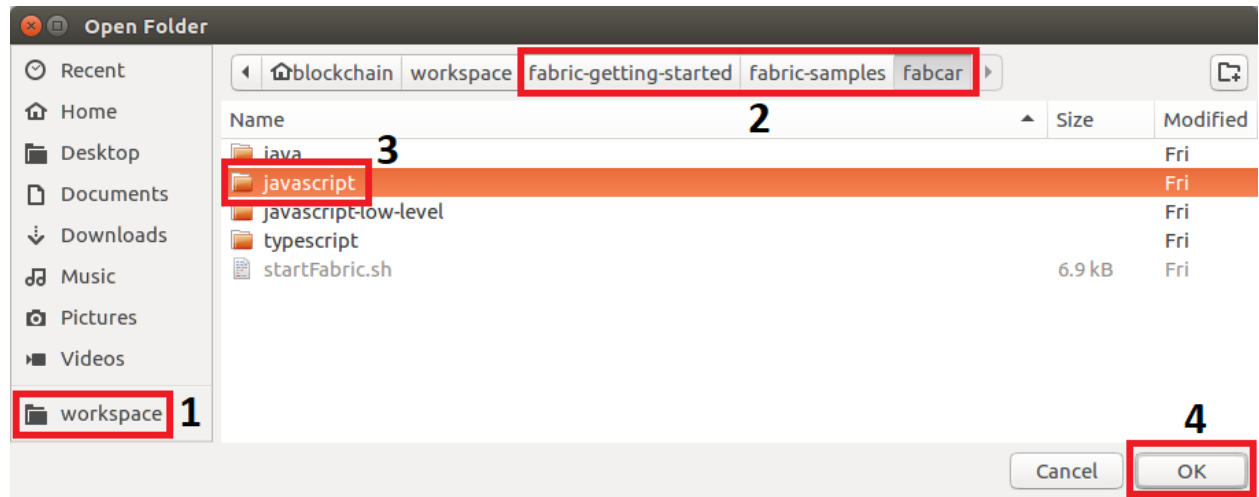
Step 2: Navigate to the folder:

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.

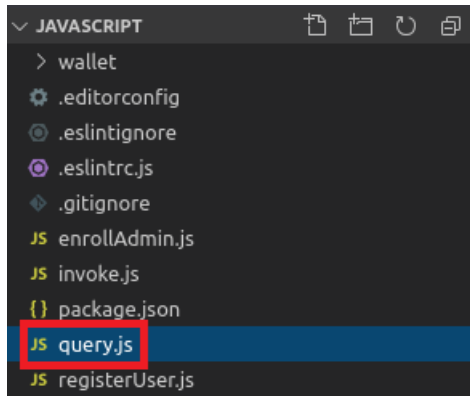
Step 4: Click the **OK** button on the bottom right.



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

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

- __ 30. Once the folder is added to the workspace, double click on the **query.js** application to open it in the main editing view. A double click in VS Code keeps a file open so we can refer to it later when we open other files.



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**, as we saw earlier, is a transaction that does not get sent for ordering into a block and so is normally used for query transactions.

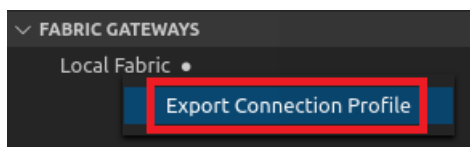
This sample application is designed to connect to the sample network that comes with the **fabric-samples**. However, we need to change it to connect to the **Local Fabric** network that VS Code has created so we can query the ledger we have been using earlier in this lab.

But before we can edit the application, we first need to export the connection details and the wallet information from the VS Code **Local Fabric** network so we can use them in our **query.js** application.

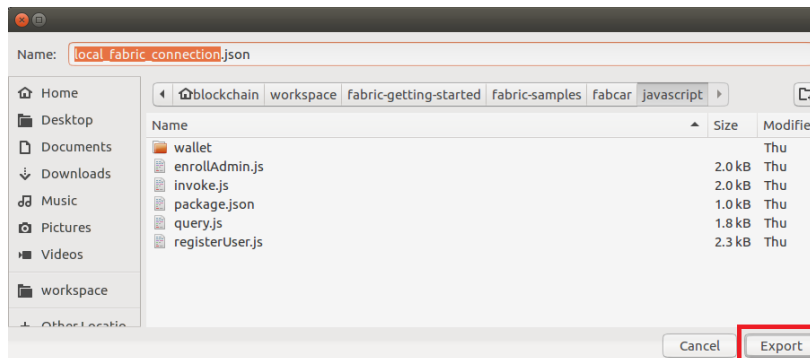
- __ 31. Click on the IBM Blockchain Platform icon in the sidebar to switch to the IBM Blockchain Platform view.



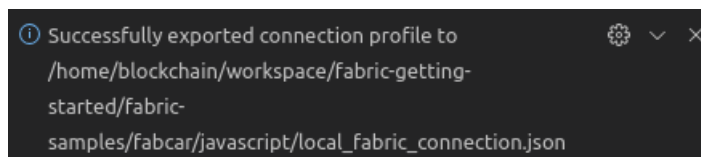
- __ 32. In the “**Fabric Gateways**” view, right click on “**Local Fabric**” and choose the “**Export Connection Profile**” option:



- __ 33. In the dialogue that opens, leave the file name as “**local_fabric_connection.json**” and the location at the default and click “**Export**”:



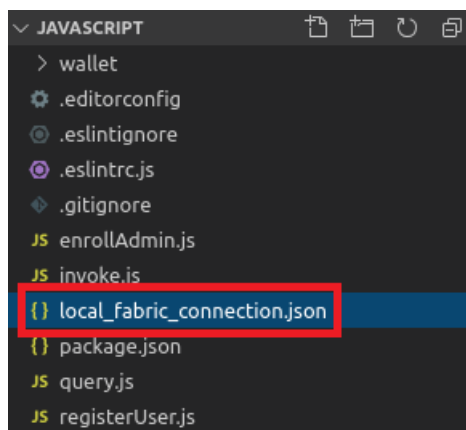
You should see an **information message** telling you the export was successful:



The full path of the exported file for reference is:

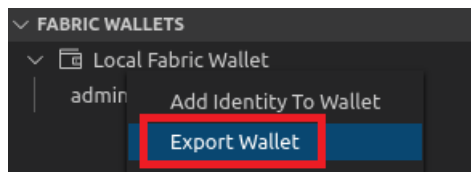
```
/home/blockchain/workspace/fabric-getting-started/fabric-samples/fabcar/javascript/local_fabric_connection.json
```

This shows that the connection details have been exported into the **javascript** folder we opened earlier to load the **fabcar** sample applications. The path is also shown in the **OUTPUT** view and if you switch to the **Explorer** view, the new **local_fabric_connection.json** file is shown in the explorer view as well.

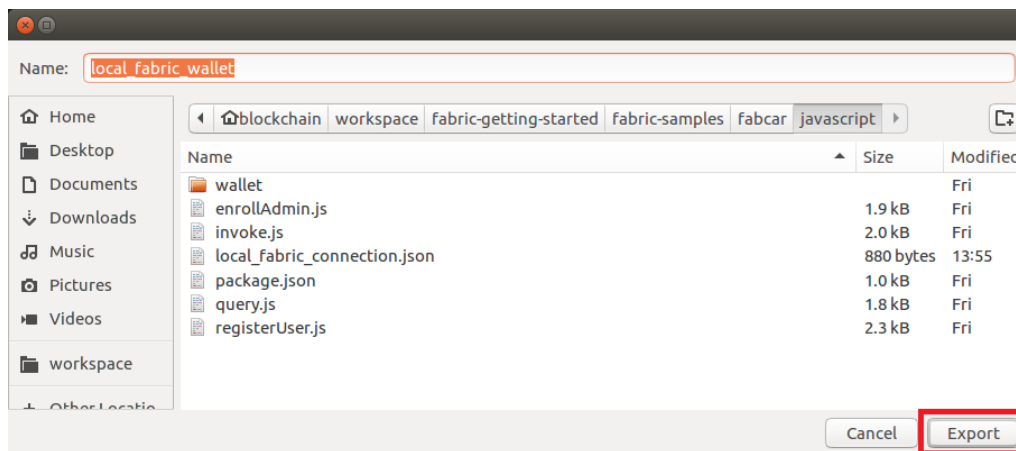


Next we need to perform a similar set of actions to export the **wallet** that contains the ID used to connect to the **Local Fabric** network. It is important to note that the exported wallet is actually a folder which contains a sub-folder for each ID in the wallet, with each ID folder containing multiple files that represent the individual ID.

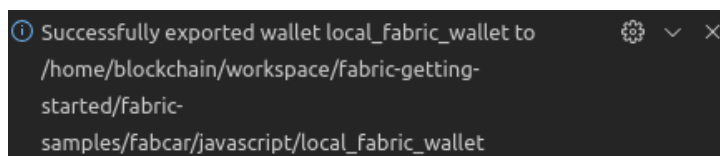
- __ 34. From the IBM Blockchain Platform view, under the “**Fabric Wallets**” view, right click on the “**Local Fabric Wallet**” and choose the **Export Wallet** option:



In the dialogue that opens, leave the folder name as “**local_fabric_wallet**” and the location at the default and click “**Export**”:



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

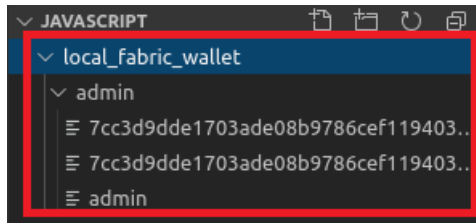


The full path of the exported wallet folder for reference is:

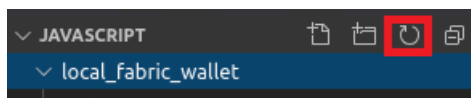
```
/home/blockchain/workspace/fabric-getting-started/fabric-samples/fabcar/javascript/local_fabric_wallet
```

This shows that the wallet has been exported into the **javascript** folder we opened earlier to load the fabcar sample applications. The path is also shown in the **OUTPUT** view.

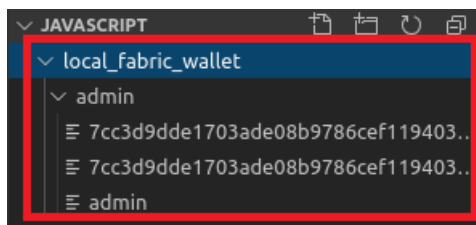
- __ 35. Switch back to the **Explorer** view to see the new **local_fabric_wallet** folder is shown in the their as well:



Note: If you do not see the **local_fabric_wallet** folder, click the “**Refresh Explorer**” icon on the bar on the **JAVASCRIPT** folder:



- __ 36. Expand the **local_fabric_wallet** folder and have a look at what was exported:

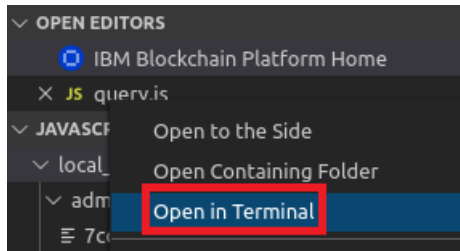


You can see it has exported an identity for the administrator of this blockchain peer, called **admin** along with the keys required to make a connection to the network.

Also, take a look at the **local_fabric_connection.json** file we exported earlier. You will see it contains the connection details for the **Local Fabric** network

Feel free to open up these files and to see 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$
```

Note: If you get a message from npm regarding npm vulnerabilities or audits this can be ignored for this lab.

__ 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 10: 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 16: 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 21: This line loads the user from the wallet and as we are using a different user called **admin** from **Local Fabric**, we need to change the line to be:

```
const userExists = await wallet.exists('admin');
```

Line 23: This line simply logs an error if the user does not exist, so for completeness change the text "**user1**" to be "**admin**".

Line 30: This line connects to the network gateway using a specific user, so again change the text "**user1**" to be "**admin**".

Line 41: 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 VS Code, passing in **CAR0** as a parameter:

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

__ 40. When completed, the code should look like this:

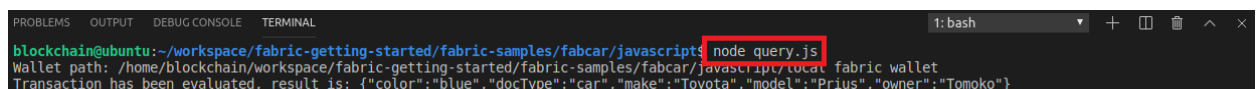
```

10 const ccpPath = path.resolve(__dirname, 'local_fabric_connection.json');
11
12 async function main() {
13   try {
14
15     // Create a new file system based wallet for managing identities.
16     const walletPath = path.join(process.cwd(), 'local_fabric_wallet');
17     const wallet = new FileSystemWallet(walletPath);
18     console.log(`Wallet path: ${walletPath}`);
19
20     // Check to see if we've already enrolled the user.
21     const userExists = await wallet.exists('admin');
22     if (!userExists) {
23       console.log('An identity for the user "admin" does not exist in the wallet');
24       console.log('Run the registerUser.js application before retrying');
25       return;
26     }
27
28     // Create a new gateway for connecting to our peer node.
29     const gateway = new Gateway();
30     await gateway.connect(ccpPath, { wallet, identity: 'admin', discovery: { enabled: true, asLocalhost: true } });
31
32     // Get the network (channel) our contract is deployed to.
33     const network = await gateway.getNetwork('mychannel');
34
35     // Get the contract from the network.
36     const contract = network.getContract('fabcar');
37
38     // Evaluate the specified transaction.
39     // queryCar transaction - requires 1 argument, ex: ('queryCar', 'CAR4')
40     // queryAllCars transaction - requires no arguments, ex: ('queryAllCars')
41     const result = await contract.evaluateTransaction('queryCar', 'CAR0');
42     console.log(`Transaction has been evaluated, result is: ${result.toString()}`);
43
44   } catch (error) {

```

__ 41. Save the file by pressing **ctrl + s** or use the **File / Save** menu option.

__ 42. 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"}

```

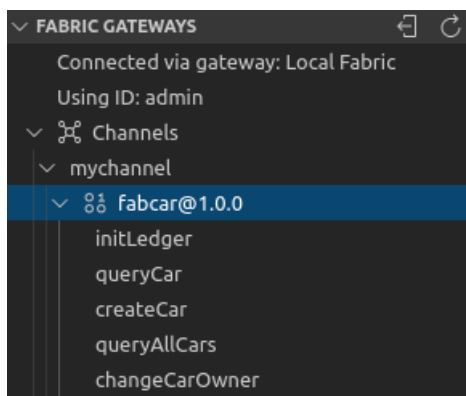
As you can see, the output is the same output for **CAR0** that we got earlier which is a **blue Toyota Prius**, owner by **Tomoko**.

Next we are going to call the **changeCarOwner** transaction to update the owner of **CAR0**, and then we will query the ledger again to see the result.

- __ 43. Switch back to the IBM Blockchain Platform view by clicking on the IBM Blockchain Platform icon in the sidebar:



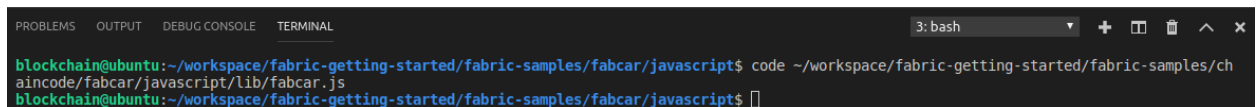
- __ 44. Made sure you are connected to **Local Fabric** network by clicking in the **Fabric Gateways** view and expand **mychannel** and the **fabcar@1.0.0** contract so you can see the transactions.



Before we call it, let's take a look at the **changeCarOwner** transaction implementation and to do that we need to open the smart contract file **fabcar.js** file again.

- __ 45. In the terminal window you used earlier, copy and paste the following command to open the **fabcar.js** file that we looked at earlier in this lab:

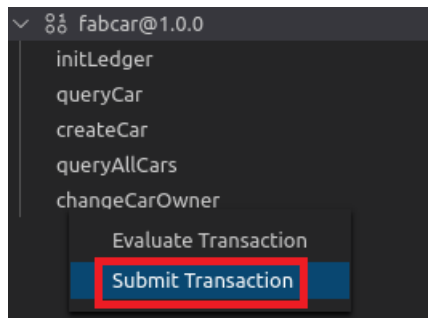
```
code ~/workspace/fabric-getting-started/fabric-samples/chaincode/fabcar/javascript/lib/fabcar.js
```



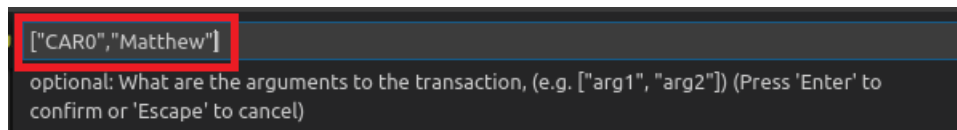
Note: you could use **File / Open File** menu and navigate to the file manually from the above path if you prefer.

The definition of **changeCarOwner** is between lines 140-152 of **fabcar.js**. 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.

- __ 46. Right click on the **changeCarOwner** transaction under the **FabCar** contract in the **FABRIC GATEWAYS** view and choose **Submit Transaction**. We use **Submit Transaction** this time as the **changeCarOwner** transaction makes an update to the ledger.

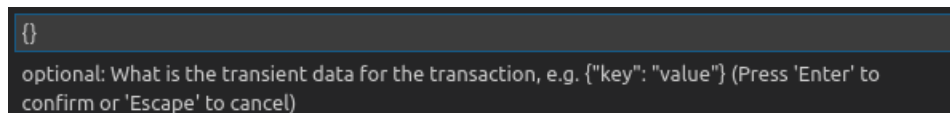


- __ 47. In the dialogue at the top of the screen enter the text **"CAR0","Matthew"** inside the square brackets as shown below and press "Enter".

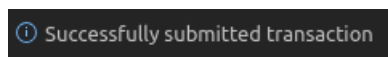


Note: As we saw earlier in this lab, the brackets and the quotes are required as you enter the parameters using JSON syntax.

As we have no “transient” data for this transaction, press enter at the next prompt:

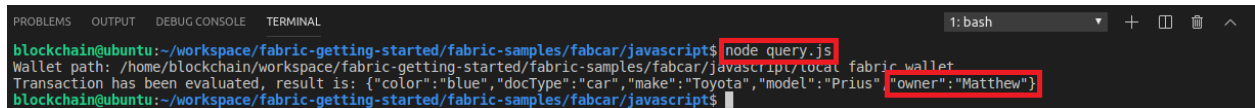


When the transaction has finished, you should see the **information message**:



Note: If instead you get an error, you may have entered the string incorrectly. You can check the **OUTPUT** window for an error and try this step again.

- __ 48. Once the transaction has been successfully submitted, click back on the **Terminal** window, enter **node query.js** again to see the resulting changes.



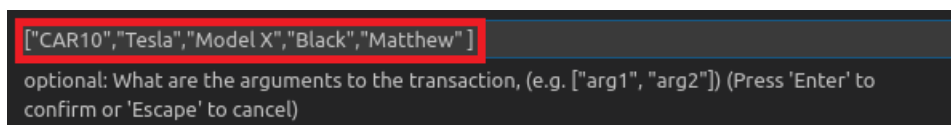
```
blockchain@ubuntu:~/workspace/fabric-getting-started/fabric-samples/fabcar/javascript$ node query.js
Wallet path: /home/blockchain/workspace/fabric-getting-started/fabric-samples/fabcar/javascript/tocal fabric.wallet
Transaction has been evaluated, result is: {"color":"blue","docType":"car","make":"Toyota","model":"Prius","owner":"Matthew"}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

You should be able to see that the owner for **CAR0** 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 **changeCarOwner**, 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.

- __ 49. As you did with **changeCarOwner** above, right click on the **createCar** transaction under the **FabCar** contract in the IBM Blockchain Platform Blockchain Connections view and choose **Submit Transaction**.
- __ 50. 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 values of your choice like "**CAR10**", "**Tesla**", "**Model X**", "**Black**", "**Matthew**" inside the square brackets and press "Enter". Press enter at the next prompt as well, as there is no transient data for this transaction.

Note: Remember, you should not enter any quotes or extra spaces around this string as otherwise they may be taken as part of the string itself which will result in an error.



```
["CAR10","Tesla","Model X","Black","Matthew"]
optional: What are the arguments to the transaction, (e.g. ["arg1", "arg2"]) (Press 'Enter' to confirm or 'Escape' to cancel)
```

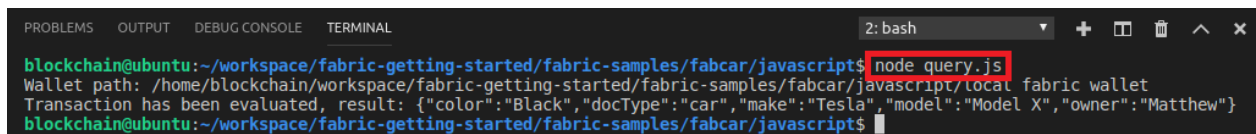
- __ 51. When the transaction has completed successfully, update **query.js** to query for the car with the index you created, in this case **CAR10**.

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

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

- __ 52. 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: {"color":"Black","docType":"car","make":"Tesla","model":"Model X","owner":"Matthew"}
```

- __ 53. We are now coming towards the end of **Lab 2**. 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.

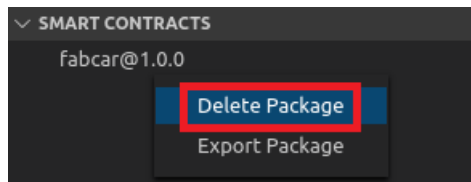
First you could update **query.js** to run **queryAllCars** to return the details of all cars instead of just one.

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.

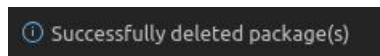
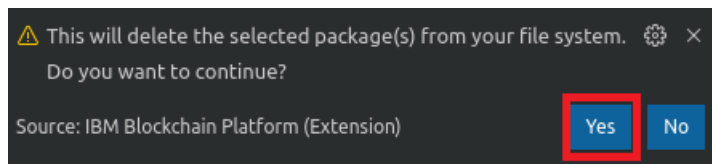
Following the steps above for updating **query.js** to use the **Local Fabric** blockchain network's wallet and connection details, you could make the same changes to **invoke.js** so it can create another new car in the **Local Fabric** network from the **Terminal** instead of using VS Code to do it. Remember to run **query.js** afterwards to make sure your new car made it to the ledger.

We have now almost completed **Lab 2 – using an existing contract**. All that remains is to clear up the environment ready for part three of this lab.

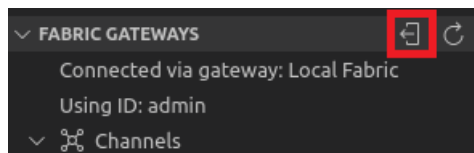
- __ 54. From the IBM Blockchain Platform **Smart Contracts** view, right-click on the **fabcar@1.0.0** package and choose the **Delete Package** option as shown below to remove it:



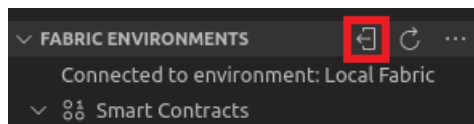
Confirm this with the informational message in the bottom right:



- __ 55. From the IBM Blockchain Platform **Fabric Gateways** view, select the **Disconnect from Gateway** icon as shown below:



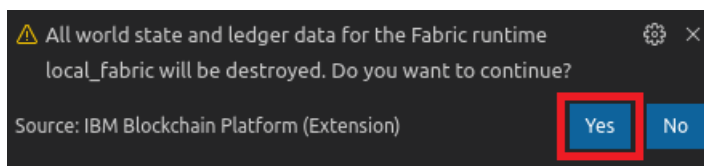
- __ 56. In the IBM Blockchain Platform **Fabric Environments** view, if you are still connected to the **Local Fabric environment**, click the “Disconnect from a Fabric environment” icon:



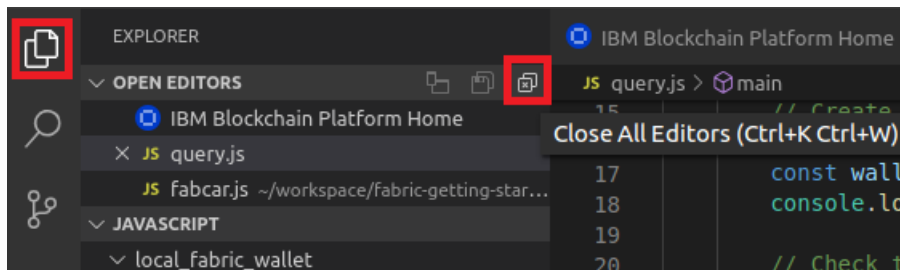
- __ 57. In the IBM Blockchain Platform **Fabric Environments** view, right click on **Local Fabric** and choose **Teardown Fabric Runtime**:



- __ 58. From the **information message** that appears in the bottom right, choose the **“Yes”** button:



- __ 59. Switch to the **Explorer** view and click the **“Close All Editors”** button on the **“Open Editors”** toolbar. This will close all the editors that are open:

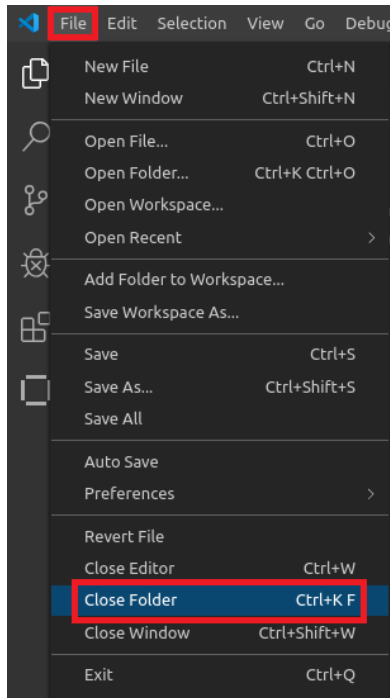


Note: To see the button you will need to move your mouse over the **“Open Editors”** toolbar

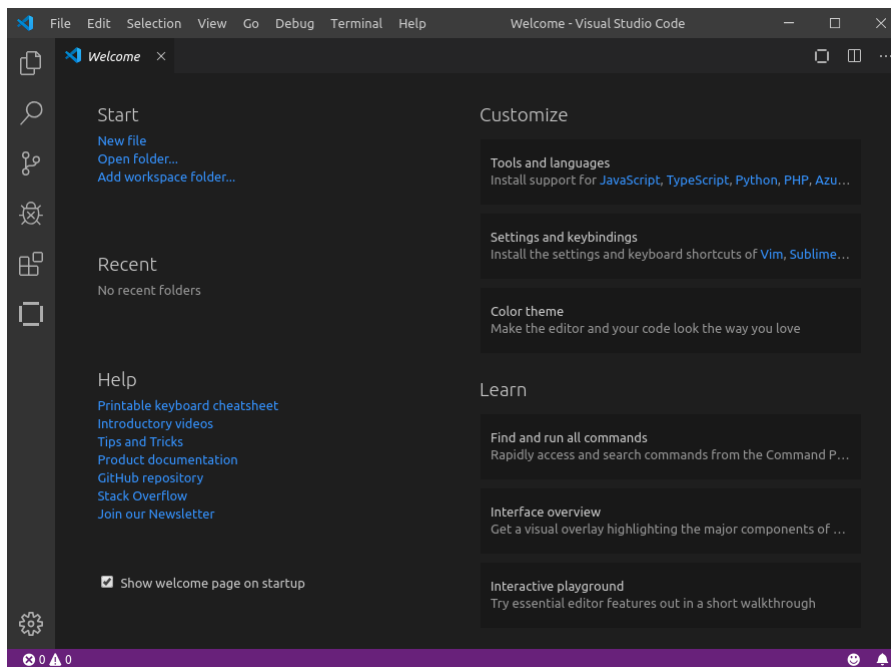
Note 2: if you cannot see the **Explorer** view, click on the **Explorer** icon again to make it re-appear.

When you have finished, the **“Open Editors”** view should be empty.

__ 60. From the “File” menu, choose “**Close Folder**”:



__ 61. This will reopen the VS Code “**Welcome**” editor. This now leaves your workspace empty and ready for the next lab as shown below:



3 Next Steps

In this lab you have experienced an overview of using the IBM Blockchain Platform development environment with an existing contract. In the next lab, we will take this further and show you how to use a more complex sample that comes with Hyperledger Fabric and use VS Code to connect to a network other than **Local Fabric**.

We Value Your Feedback!

- Please ask your instructor for an evaluation form. Your feedback is very important to us as we use it to continually improve the lab material.
- If no forms are available, or you want to give us extra information after the lab has finished, please send your comments and feedback to “**blockchain@uk.ibm.com**”