



Exercise 7: Developing a client application for Hyperledger Fabric

The Hyperledger Fabric Client SDK makes it easy to use APIs to interact with a Hyperledger Fabric blockchain.

Applications can be developed to interact with the blockchain network on behalf of the users. The Hyperledger Fabric SDK for Node.js product is designed in an object-oriented programming style.

This exercise explores developing applications with the Hyperledger Fabric Client SDK. This exercise describes how Fabric applications work and interact with a Hyperledger Fabric blockchain.

7.1 Getting started

This section provides an overview of the exercise, the skills that you will gain by performing the exercise, the prerequisites for this exercise, and the results that you can expect after completing this exercise

7.1.1 What this exercise is about

The Hyperledger Fabric SDK for Node.js product provides APIs to interact with a Hyperledger Fabric blockchain. The `fabric-client` package encapsulates the APIs to interact with Peers and Orderers of the Fabric network to install and instantiate chaincodes, send transaction invocations, and perform chaincode queries.

This exercise shows you how to develop an application in Node.js that uses the Hyperledger Fabric Client SDK to interact with the Hyperledger Fabric network and interfaces with a front-end web application that is written in Angular 2. Figure 7-1 provides an overview of the components that are used in this exercise and their role.

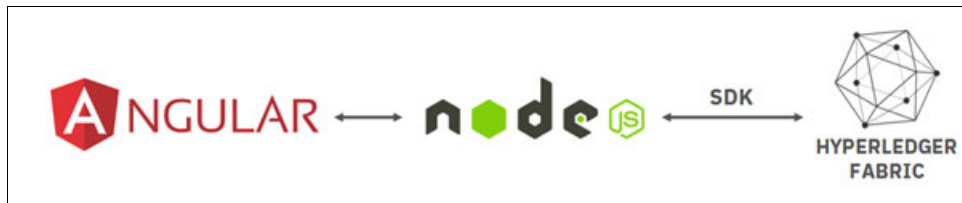


Figure 7-1 Client SDK provides the APIs to interact with Hyperledger Fabric

In this exercise, you will use some of these APIs. As shown in Figure 7-1, the Node.js application acts as a server between the front-end web application and the blockchain business network.

This exercise will cover the following topics:

- ▶ Enabling the front-end web application to interact with the chaincode through the Node.js server and the Fabric Client SDK.
- ▶ Adding a function in the Node.js server that uses Fabric SDK APIs and tests the transaction flow between the web application and the chaincode.

7.1.2 What you should be able to do

After completing this exercise, you should be able to:

- ▶ Use the Fabric Client SDK for Node.js to interact with a Fabric business network.
- ▶ Develop an application to submit transactions to the blockchain.
- ▶ Test the transaction flows that take place when a transaction is submitted.

7.1.3 Prerequisites

Enable cross-origin resource sharing (CORS) in Firefox. Ensure that the icon in your browser is green, as shown in Figure 7-2. If the icon is red, click the **Cors E** icon to enable CORS.



Figure 7-2 Cors E icon

Note: CORS is needed for Mozilla (Firefox) to enable communications between applications running on different ports. If CORS is not enabled, responses from the Node.js server to the Angular application will be blocked.

7.1.4 Expected results

By the end of this exercise, you will have a running Angular 2 web application that can add a vehicle, query existing vehicles, change the owners of vehicles, and reflect all these actions in the ledger.

7.2 Architecture

Figure 7-3 provides a high-level view of the role of the Fabric Client SDK used by the Node.js application in this exercise and the developer's tasks.

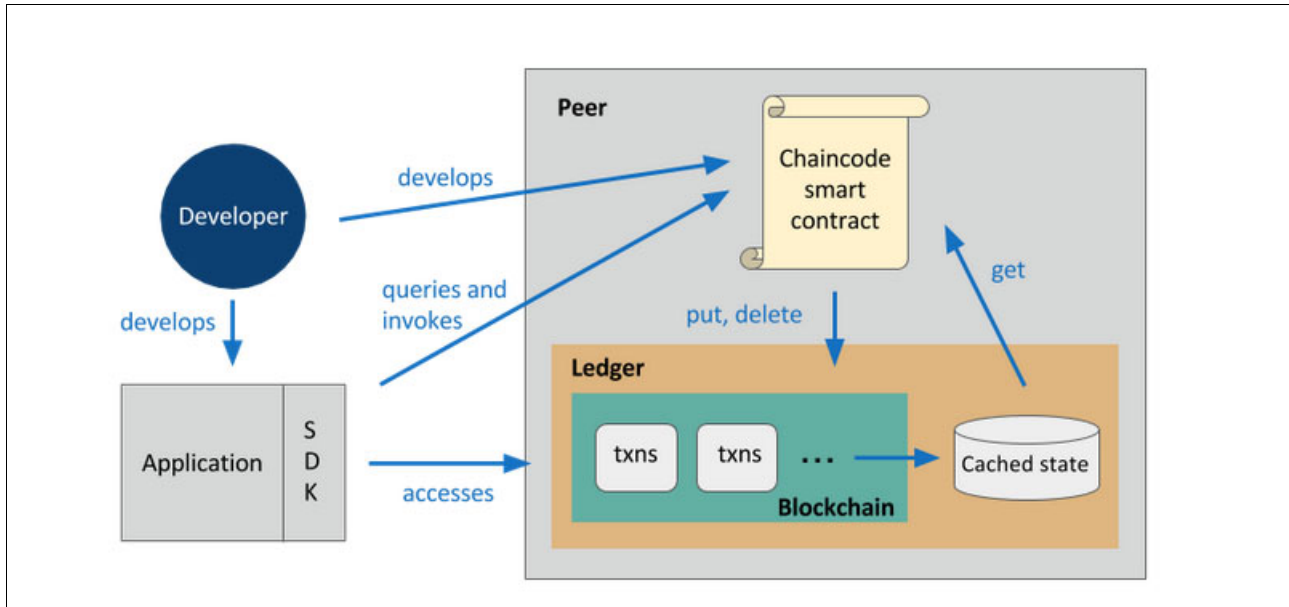


Figure 7-3 Transactional mechanics that take place during a standard asset exchange

In Figure 7-3:

- ▶ The developer develops and deploys chaincode that is written in Go or JavaScript. You learned about this process in Chapter 6, “Exercise 6: Developing chaincode for Hyperledger Fabric” on page 159.
- ▶ The developer develops an application that uses the Fabric Client SDK that is available for Node.js and Java.
- ▶ The application queries and invokes smart contract functions through the Fabric Client SDK.
- ▶ These function calls are processed by the business logic in the smart contract's chaincode.
- ▶ The application can access blockchain information through the Fabric Client APIs.

Figure 7-4 on page 207 shows an overview of the transaction steps. It highlights how the Fabric Client SDK in the intermediate Node.js server interfaces with both the front-end web UI that is represented by the Angular application, and the peers and orderer in the blockchain network.

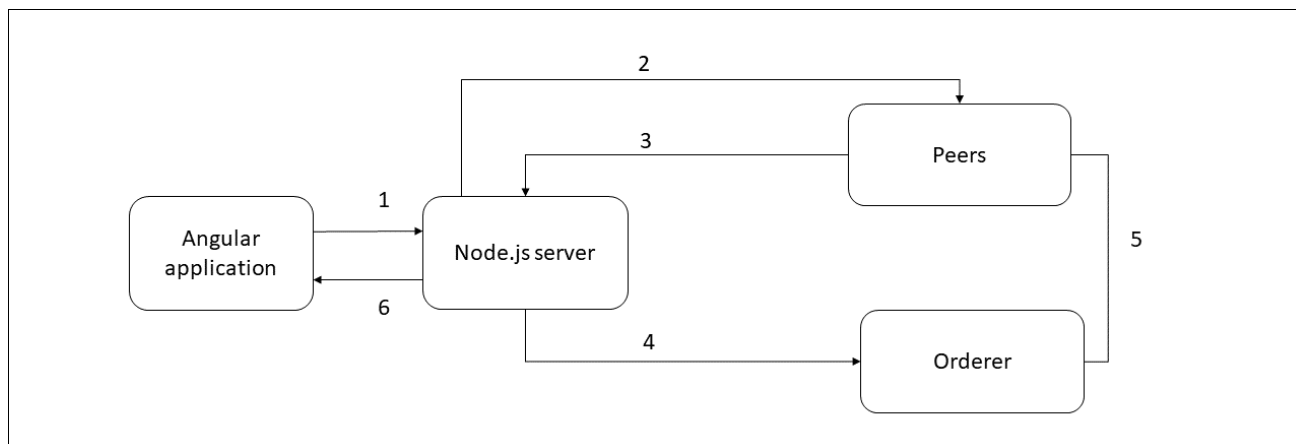


Figure 7-4 Transaction steps

Figure 7-4 shows the following flow:

1. The front-end Angular web application sends a request and corresponding data to the Node.js server.
2. The Node.js server calls the Fabric Client SDK APIs to send the request to the endorsing peers in the blockchain business network. The peers are responsible for receiving a transaction proposal for endorsement, and respond by granting or denying endorsement.
3. The endorsing peers return a response to the Node.js server. Fabric Client SDK APIs are used to receive the response, which can have its transactions endorsed or rejected.
4. If the transaction is endorsed by endorsing peers, the Node.js application uses SDK APIs to send the endorsed transaction to the orderer.
5. The orderer collects transactions into proposed blocks for distribution to committing peers that are responsible for committing transactions.
6. After the transaction is committed, the front-end application is notified of the results, that is, the transaction succeeded or failed, and whether blocks were added to the ledger.

7.3 Exercise instructions

In this exercise, you will accomplish the following tasks:

1. Register and enroll users.
2. Run the sample application.
3. Modify the Node.js sample application to add the **changeOwner** function.
4. Test the changes to the application.
5. Clean up the environment.

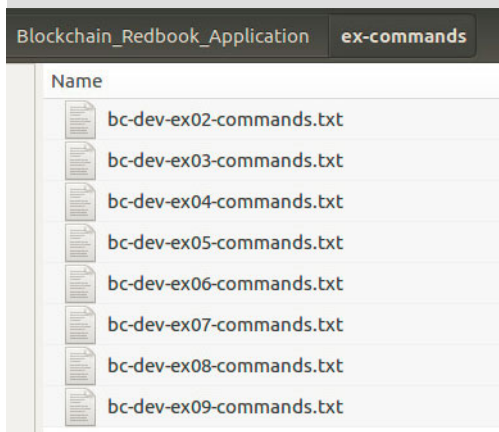
Before you start this exercise, enable cross-origin resource sharing (CORS) in Firefox. Ensure that the icon in your browser is green, as shown in Figure 7-2 on page 205. If the icon is red, click the **Cors E** icon to enable CORS.



Figure 7-5 Cors E icon

Note: CORS is needed for Mozilla (Firefox) to enable communications between applications running on different ports. If CORS is not enabled, responses from the Node.js server to the Angular application will be blocked.

Note: The commands you will run in this exercise are in a text file in your VM. Open the text file to copy and paste each command in your terminal session. To find the text file for each exercise, double-click the **File** icon and navigate to **/Blockchain_Redbook_Application/ex-commands**. Open the text file for this exercise.



7.3.1 Registering and enrolling users

In this section, you verify that the chaincode is running in the Fabric network. Then, you register the Admin user and the user that will sign the transactions to be written to the blockchain.

Perform the following steps:

1. Confirm that your chaincode is running. In a new terminal window, enter the following command:

```
docker ps --format 'table {{.Names}} \t {{.Status}}'
```

You should see an output similar to the following (the order of the containers and the version in dev-peer0 might differ):

NAMES	STATUS
dev-peer0.org1.example.com-vehicle-chaincode-go-1.0	Up 2 minutes
cli	Up 2 minutes
peer0.org1.example.com	Up 2 minutes
couchdb	Up 2 minutes
ca.example.com	Up 2 minutes
orderer.example.com	Up 2 minutes

Note: If you do not see dev-peer0.org1.example.com-vehicle-chaincode-go-1.0 (the version might differ), run the following commands to start the chaincode:

```
cd ~/fabric-tools
./stopFabric.sh
./teardownFabric.sh

cd ~/Blockchain_Redbook_Application/Fabric/Exercise7
./startFabric.sh
```

Then perform step 1 again.

2. Change to the directory of the sample application. In a new terminal window, run the following command:

```
cd ~/Blockchain_Redbook_Application/Fabric/Exercise7/Front-End/Vehicle-app
```

3. Register the Admin user and store the credentials in the .hfc-key-store folder.

The registerAdmin.js invokes a certificate signing request (CSR) and then outputs an ecert and key material into a newly created folder at the root of this project. The folder name is hfc-key-store.

Run the following command:

```
node registerAdmin.js
```

Note: If you receive an error, repeat the steps in the previous note.

You should receive an output similar to the following output (your certificate will be different):

```
Store path:/home/user/.hfc-key-store
Successfully enrolled admin user "admin"
Assigned the admin user to the fabric client
::{"name":"admin","mspId":"Org1MSP","roles":null,"affiliation":"","enrollmentSe
```

```
cret":"","enrollment":{"signingIdentity":"478bc0006737225e0510890a8887a4627a813
6f0fe8cc540b2f7a2d79d6ab709","identity":{"certificate":"-----BEGIN
CERTIFICATE-----\nMIICATCCAaigAwIBAgIUWsq1Jksiy1Nm1kLG0ea+Nva12WEwCgYIKoZIZjOEA
wIw\nczELMAkGA1UEBhMCVVMxEzARBgNVBAGTCkNhbg1mb3JuaWExFjAUBgNVBACjTDVNH\nbiBGcmFu
Y2l2Y28xGTAXBgNVBAoTEG9yZzEuZXhhbXBsZS5jb20xHDAaBgNVBAMT\nE2NhLm9yZzEuZXhhbXBsZ
S5jb20wHhcNMTgwNTEOMjAONzAwWhcNMTkwNTEOMjA1\nMjAwWjAhMQ8wDQYDVQLEwZjbG11bnQxDj
AMBgNVBAMTBWFkbW1uMFkwEwYHKoZI\nnzjOCAQYIKoZIZjODACQgAEn1ypM2J8+G4DqAcDt7jg0c
vFTdOM/iI2aVtw3GS\npIfcemA8C+TCj10WtaJit5Tig+b5CqqzivIyDSLdtdadhD6NsMGowDgYDVROp
AQH/\nBAQDAgeAMAwGA1UdEwEB/wQCMAAwHQYDVRO0BBYEFH4f/UakzVoKgdT0PyTZ4zpd\nnmRxtMCs
GA1UdIwQkMCKAIEI5gg3NdtruuLoM2nAYUdFFBNMarRst3dusa1c2Xk18\nnMAoGCCqGSM49BAMCA0CA
MEQCIFFjW1bRABamUT6vdDMgS43QPozYG1XxKzy/Pmmuser@
```

4. Register and enroll a new user (user1). This user will be the user that signs all the transactions for querying and updating the ledger. Run the following command:

```
node registerUser.js
```

Note: The Admin user issues the registration and enrollment calls for the new user.

You should receive the following output:

```
Store path:/home/user/.hfc-key-store
Successfully loaded admin from persistence
Successfully registered user1 - secret:vtoiKwkhPiZ
Successfully enrolled member user "user1"
User1 was successfully registered and enrolled and is ready to interact with
the fabric network
```

Similar to the admin enrollment, this program invokes a CSR and outputs the keys and cert into the hfc-key-store subdirectory. Now, you have identity material for two separate users, that is, the admin and the new user user1.

7.3.2 Running the sample application

In this section, you run the sample application and create a vehicle by using the existing code.

Perform the following steps:

1. Start the Node.js application by running the following command in a terminal window:

```
npm start
```

You will receive the following message at the end of your output:

```
Live on port:8000
```

Now, you have a running Node.js server that is listening on localhost:8000.

2. Start the Angular web application by running the following commands in a new terminal window:

```
cd ~/Blockchain_Redbook_Application/Fabric/Exercise7/Front-End/Angular2/
```

```
npm start
```

Your output should be similar to the following:

```
> Car-Manufacture-App@0.0.0 start
/home/user/Blockchain_Redbook_Application/Fabric/Exercise7/Front-End/Angular2
> ng serve
```


The --missing-translation parameter will be ignored because it is only compatible with Angular version 4.2.0 or higher. If you want to use it, please upgrade your Angular version.

** NG Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **

28% building modules 157/200 modules 43 active

...es/core-js/modules/_a-number-value.jswebpack: wait until bundle finished: /

Date: 2018-05-14T21:02:16.230Z

Hash: 289d31e4626e573f193d

Time: 53159ms

chunk {inline} inline.bundle.js (inline) 3.85 kB [entry] [rendered]

chunk {main} main.bundle.js (main) 48.7 kB [initial] [rendered]

chunk {polyfills} polyfills.bundle.js (polyfills) 864 kB [initial] [rendered]

chunk {styles} styles.bundle.js (styles) 41.5 kB [initial] [rendered]

chunk {vendor} vendor.bundle.js (vendor) 9.32 MB [initial] [rendered]

webpack: Compiled successfully.

3. Run the web application. Open the browser on localhost:4200, as shown in Figure 7-6.

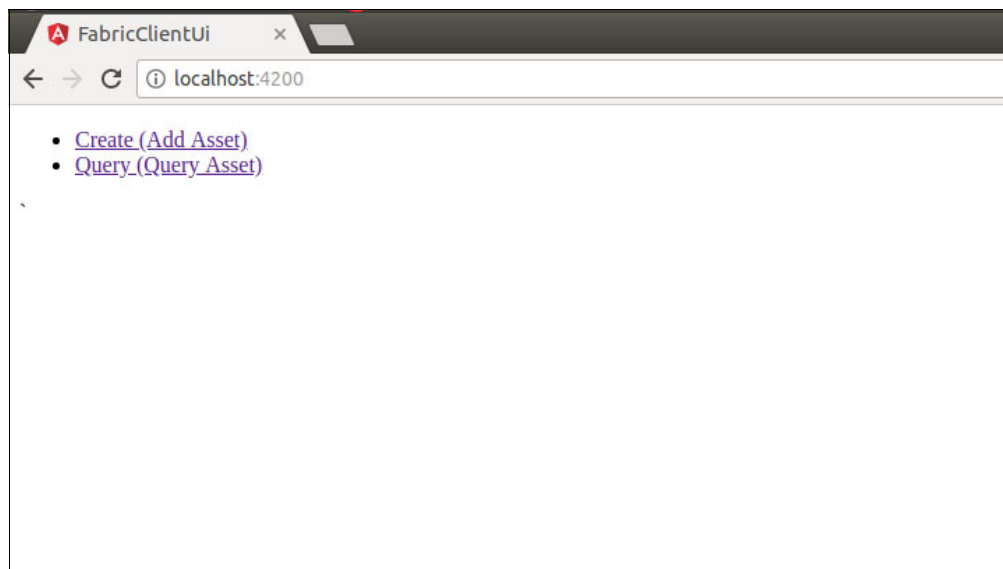


Figure 7-6 Web application landing page

4. To add a vehicle, click **Create (Add Asset)**, as shown in Figure 7-7 on page 212.

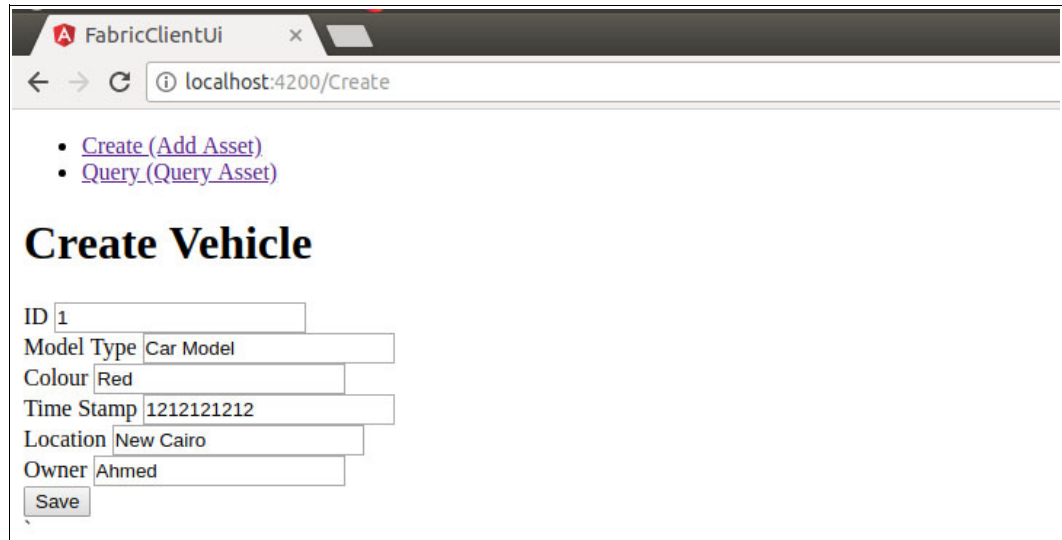


Figure 7-7 Create Vehicle web page

A Create Vehicle form with pre-loaded data is displayed.

5. Change the value of Colour from Red to Blue and click **Save**.

Note: Expect no results in the web UI.

6. Return to the terminal window that you opened to run the Node.js application. The transaction has been submitted.

Your output should be similar to the following:

```
Create New Vehicle
[ '1', 'Car Model', 'Blue', '1212121212', 'New Cairo', 'Ahmed' ]
Store path:/home/BlockchainUser/.hfc-key-store
Successfully loaded user1 from persistence
Assigning transaction_id:
b2f449170939213d1115aee9d4a98b55d8717279b9bc3a66a7d129b77b306355
Transaction proposal was good
Successfully sent Proposal and received ProposalResponse: Status - 200, message
- "OK"
info: [EventHub.js]: _connect - options {}
The transaction has been committed on peer localhost:7053
Send transaction promise and event listener promise have completed
Successfully sent transaction to the orderer.
Successfully committed the change to the ledger by the peer
```

7.3.3 Modifying the Node.js sample application to add the changeOwner function

In this section, you add a function that is called **changeOwner** to the Node.js sample application. You will build the function by pasting the code snippets after you understand the purpose of the code.

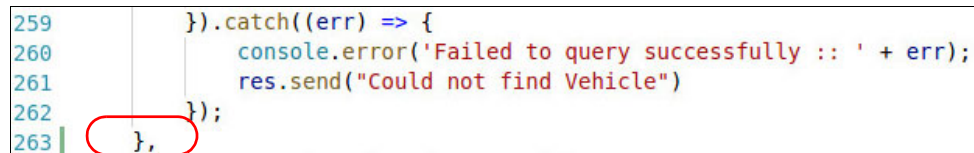
Notes:

- ▶ For the complete listing of the code created in this exercise, see “**Code solutions**” on **page 226**.
- ▶ The comments in the figures in this section are *not* included in the code snippets provided in the `bc-dev-ex07commands.txt` file and in the complete code listing.

Perform the following steps:

1. Stop the Node.js application by pressing **Ctrl + C** in the terminal window that is running the Node.js server.
2. Stop the web application by pressing **Ctrl + C** in the terminal window that is running the Angular application.
3. Edit the `controller.js` file by running the following commands:

```
cd ~/Blockchain_Redbook_Application/Fabric/Exercise7/Front-End/Vehicle-app
code controller.js
```
4. After the closing brace of the `getVehicle` function, add a comma (,) to start a function, as shown in Figure 7-8.



```
259     }).catch((err) => {
260         console.error('Failed to query successfully :: ' + err);
261         res.send("Could not find Vehicle")
262     });
263     },
```

Figure 7-8 Adding the `changeOwner` function (1 of 9)

5. After line 263 (or equivalent line number for you) add the following lines of code to start the new function (`changeOwner`):

```
changeOwner: function(req, res){
    // add the following steps here
}
```

6. Inside this function, add the following lines of code to set up a connection to Hyperledger Fabric by creating a new instance of channel, peer, and orderer:

```
        console.log("changing Owner");
        var array = req.params.holder.split("-");
var key = array[0]
var holder = array[1];
        var fabric_client = new Fabric_Client();
var channel = fabric_client.newChannel('mychannel');
var peer = fabric_client.newPeer('grpc://localhost:7051');
channel.addPeer(peer);
var order = fabric_client.newOrderer('grpc://localhost:7050')
channel.addOrderer(order);
        var member_user = null;
var store_path = path.join(os.homedir(), '.hfc-key-store');
        console.log('Store path:'+store_path);
var tx_id = null;
```

Your code should look like Figure 7-9.

A screenshot of a code editor showing the implementation of the changeOwner function. The code is as follows:

```
    },
    changeOwner: function (req, res) {
        // add the following steps here
        //Set up connection to Fabric
        console.log("changing Owner");
        var array = req.params.holder.split("-");
        var key = array[0]
        var holder = array[1];
        var fabric_client = new Fabric_Client();
        var channel = fabric_client.newChannel('mychannel');
        var peer = fabric_client.newPeer('grpc://localhost:7051');
        channel.addPeer(peer);
        var order = fabric_client.newOrderer('grpc://localhost:7050')
        channel.addOrderer(order);
        var member_user = null;
        var store_path = path.join(os.homedir(), '.hfc-key-store');
        console.log('Store path:' + store_path);
        var tx_id = null;
```

Figure 7-9 Adding the changeOwner function (2 of 9)

7. Add the following lines of code to set up the client object with state and crypto store.

This code creates a CryptoKeyStore, which is used to store sensitive information in persistent storage, such as authenticated user's private keys, certificates, and more.

The CryptoKeyStore is assigned to a CryptoSuite, which is a suite of crypto algorithms that is used by the SDK to perform digital signing, encryption/decryption, and secure hashing.

```
Fabric_Client.newDefaultKeyValueStore({ path: store_path
}).then((state_store) => {
fabric_client.setStateStore(state_store);
var crypto_suite = Fabric_Client.newCryptoSuite();
var crypto_store = Fabric_Client.newCryptoKeyStore({path: store_path});
crypto_suite.setCryptoKeyStore(crypto_store);
fabric_client.setCryptoSuite(crypto_suite);
```

Your code should look like Figure 7-10.

```
281
282 //Set up the client object with state and crypto store
283 Fabric_Client.newDefaultKeyValueStore({
284   path: store_path
285 }).then((state_store) => {
286   fabric_client.setStateStore(state_store);
287   var crypto_suite = Fabric_Client.newCryptoSuite();
288   var crypto_store = Fabric_Client.newCryptoKeyStore({ path: store_path });
289   crypto_suite.setCryptoKeyStore(crypto_store);
290   fabric_client.setCryptoSuite(crypto_suite);
291
```

Figure 7-10 Adding the changeOwner function (3 of 9)

Now, the client can use a CryptoSuite to sign and hash.

8. Add the following lines of code to add the user that you enrolled and registered in 7.3.1, “Registering and enrolling users” on page 209. This user will sign all the transactions. If the user is not found, you probably forgot to run registerUser.js.

```
return fabric_client.getUserContext('user1', true);
}).then((user_from_store) => {
  if (user_from_store && user_from_store.isEnrolled()) {
    console.log('Successfully loaded user1 from persistence');
    member_user = user_from_store;
  } else {
    throw new Error('Failed to get user1.... run registerUser.js');
  }
}
```

9. Add the following line of code to generate a transaction ID. The transaction ID is added to the requests. The identity of the user submitting the request (user1) is also added to the transaction.

```
tx_id = fabric_client.newTransactionID();
```

Your code should look like Figure 7-11.

```
292 //Get user1
293 return fabric_client.getUserContext('user1', true);
294 }).then((user_from_store) => {
295   if (user_from_store && user_from_store.isEnrolled()) {
296     console.log('Successfully loaded user1 from persistence');
297     member_user = user_from_store;
298   } else {
299     throw new Error('Failed to get user1.... run registerUser.js');
300   }
301 //Generate transaction ID
302 tx_id = fabric_client.newTransactionID();
303
```

Figure 7-11 Adding the changeOwner function (4 of 9)

10. Add the following lines of code to create the request to send a proposal to the endorsers:

```
var request = {
  chaincodeId: chainCodeName,
  fcn: 'changeVehicleOwner',
  args: [key, holder],
  chainId: channelName,
  txId: tx_id
};
```

Your code should look like Figure 7-12.

```
304 //Create request for proposal
305 var request = {
306     chaincodeId: chainCodeName,
307     fcn: 'changeVehicleOwner',
308     args: [key, holder],
309     chainId: channelName,
310     txId: tx_id
```

Figure 7-12 Adding the changeOwner function (5 of 9)

11. The Fabric client will be looking for peers that are defined in the role of endorsing peer. The Fabric client will then send the proposal to the located peers and return all the endorsements in the results object.

Add the following lines of code to send the transaction proposal to the peers to validate:

```
return channel.sendTransactionProposal(request);
}).then((results) => {
var proposalResponses = results[0];
var proposal = results[1];
let isProposalGood = false;
if (proposalResponses && proposalResponses[0].response &&
proposalResponses[0].response.status === 200) {
isProposalGood = true;
console.log('Transaction proposal was good');
} else {
console.error('Transaction proposal was bad');
}
if (isProposalGood) {
console.log(util.format(
'Successfully sent Proposal and received ProposalResponse: Status - %s, message - "%s"',
proposalResponses[0].response.status, proposalResponses[0].response.message));
```

Your code should look like Figure 7-13.

```
312 //Send transaction proposal to peers for validation
313 return channel.sendTransactionProposal(request);
314 }).then((results) => {
315     var proposalResponses = results[0];
316     var proposal = results[1];
317     let isProposalGood = false;
318     if (proposalResponses && proposalResponses[0].response &&
319         proposalResponses[0].response.status === 200) {
320         isProposalGood = true;
321         console.log('Transaction proposal was good');
322     } else {
323         console.error('Transaction proposal was bad');
324     }
325     if (isProposalGood) {
326         console.log(util.format(
327             'Successfully sent Proposal and received ProposalResponse: Status - %s, message - "%s"',
328             proposalResponses[0].response.status, proposalResponses[0].response.message));
```

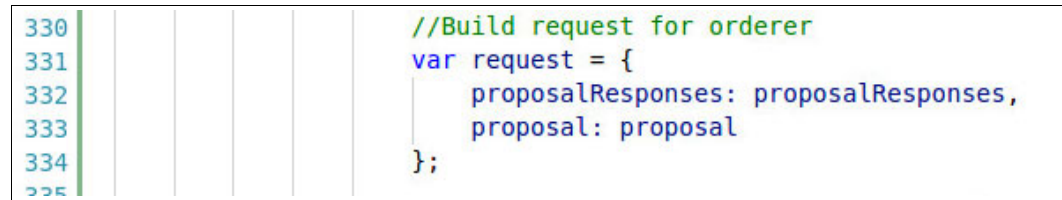
Figure 7-13 Adding the changeOwner function (6 of 9)

12. After receiving endorsements from the peers for a transaction proposal, the transactions are sent to an orderer along with the proposal for the transaction to be committed to the ledger.

Add the following lines of code to build the request for the orderer:

```
var request = {  
  proposalResponses: proposalResponses,  
  proposal: proposal  
};
```

Your code should look like Figure 7-14.

A screenshot of a code editor with a light blue background and a vertical green line on the left side. The code is as follows:

```
330 //Build request for orderer  
331 var request = {  
332   proposalResponses: proposalResponses,  
333   proposal: proposal  
334 };  
335
```

Figure 7-14 Adding the changeOwner function (7 of 9)

13. Add the following lines of code to set a transaction listener. If the transaction is not committed within this period, report a timeout.

```
var transaction_id_string = tx_id.getTransactionID(); //Get the transaction ID  
string to be used by the event processing  
var promises = [];  
var sendPromise = channel.sendTransaction(request);  
promises.push(sendPromise); //we want the send transaction first, so that we  
know where to check status
```

```
// get an eventhub once the fabric client has a user assigned. The user is  
required because the event registration must be signed  
let event_hub = fabric_client.newEventHub();  
event_hub.setPeerAddr('grpc://localhost:7053');  
// using resolve the promise so that result status may be processed under the  
then clause rather than having the catch clause process the status  
let txPromise = new Promise((resolve, reject) => {  
  let handle = setTimeout(() => {  
    event_hub.disconnect();  
    resolve({event_status : 'TIMEOUT'}); //we could use reject(new  
    Error('Trnasaction did not complete within 30 seconds'));  
  }, 3000);  
  event_hub.connect();  
  event_hub.registerTxEvent(transaction_id_string, (tx, code) => {  
    clearTimeout(handle);  
    event_hub.unregisterTxEvent(transaction_id_string);  
    event_hub.disconnect();
```

Your code should look like Figure 7-15.

```
336 //Set a transaction listener
337 var transaction_id_string = tx_id.getTransactionID(); //Get the transaction ID string to be used by t
338 var promises = [];
339 var sendPromise = channel.sendTransaction(request);
340 promises.push(sendPromise); //we want the send transaction first, so that we know where to check stat
341
342 // get an eventhub once the fabric client has a user assigned. The useris required because the event
343 let event_hub = fabric_client.newEventHub();
344 event_hub.setPeerAddr('grpc://localhost:7053');
345 // using resolve the promise so that result status may be processed under the then clause rather than
346 let txPromise = new Promise((resolve, reject) => {
347     let handle = setTimeout(() => {
348         event_hub.disconnect();
349         resolve({ event_status: 'TIMEOUT' }); //we could use reject(new Error('Trnasaction did not co
350     }, 3000);
351     event_hub.connect();
352     event_hub.registerTxEvent(transaction_id_string, (tx, code) => {
353         clearTimeout(handle);
354         event_hub.unregisterTxEvent(transaction_id_string);
355         event_hub.disconnect();
```

Figure 7-15 Adding the changeOwner function (8 of 9)

14. Add the following lines of code to report the results to the application and to receive a notification reporting whether the transaction has been committed or rejected:

```
var return_status = {event_status : code, tx_id : transaction_id_string};
if (code !== 'VALID') {
    console.error('The transaction was invalid, code = ' + code);
    resolve(return_status); // we could use reject(new Error('Problem with the
transaction, event status ::'+code));
} else {
    console.log('The transaction has been committed on peer ' +
event_hub._ep_endpoint.addr);
    resolve(return_status);
}
}, (err) => {
//this is the callback if something goes wrong with the event registration or
processing
reject(new Error('There was a problem with the eventhub ::'+err));
});
});
promises.push(txPromise);
return Promise.all(promises);
} else {
    console.error('Failed to send Proposal or receive valid response. Response null
or status is not 200. exiting...');
    res.send("Could not find Vehicle");
    // throw new Error('Failed to send Proposal or receive valid response. Response
null or status is not 200. exiting...');
}
}).then((results) => {
    console.log('Send transaction promise and event listener promise have
completed');
    // check the results in the order the promises were added to the promise all
list
    if (results && results[0] && results[0].status === 'SUCCESS') {
        console.log('Successfully sent transaction to the orderer.');
```



```

    } else {
      console.error('Failed to order the transaction. Error code: ' +
        response.status);
      res.send("Could not find vehicle");
    }
    if(results && results[1] && results[1].event_status === 'VALID') {
      console.log('Successfully committed the change to the ledger by the peer');
      res.json(tx_id.getTransactionID())
    } else {
      console.log('Transaction failed to be committed to the ledger due to
        ::'+results[1].event_status);
    }
  }).catch((err) => {
  });
}

```

Your code should look like Figure 7-16.

```

    //Report results and receive notification reporting committed or rejected tr
    var return_status = { event_status: code, tx_id: transaction_id_string };
    if (code !== 'VALID') {
      console.error('The transaction was invalid, code = ' + code);
      resolve(return_status); // we could use reject(new Error('Problem with
    } else {
      console.log('The transaction has been committed on peer ' + event_hub.
      resolve(return_status);
    }
  }, (err) => {
    //this is the callback if something goes wrong with the event registration
    reject(new Error('There was a problem with the eventhub ::' + err));
  });
});
promises.push(txPromise);
return Promise.all(promises);
} else {
  console.error('Failed to send Proposal or receive valid response. Response null or
  res.send("Could not find Vehicle");
  // throw new Error('Failed to send Proposal or receive valid response. Response nul
}
}).then((results) => {
  console.log('Send transaction promise and event listener promise have completed');
  // check the results in the order the promises were added to the promise all list
  if (results && results[0] && results[0].status === 'SUCCESS') {
    console.log('Successfully sent transaction to the orderer. ');
  } else {
    console.error('Failed to order the transaction. Error code: ' + response.status);
    res.send("Could not find vehicle");
  }
  if (results && results[1] && results[1].event_status === 'VALID') {
    console.log('Successfully committed the change to the ledger by the peer');
    res.json(tx_id.getTransactionID())
  } else {
    console.log('Transaction failed to be committed to the ledger due to ::' + results[
  }
}).catch((err) => {
});

```

Figure 7-16 Adding the changeOwner function (9 of 9)

Figure 7-17 on page 220 through Figure 7-20 on page 222 list the complete code for the changeOwner function.

Notes:

- ▶ For the complete listing of the code created in this exercise, see “Code solutions” on page 226.
- ▶ The comments in the figures in this section are *not* included in the code snippets provided in the `bc-dev-ex07commands.txt` file and in the complete code listing.

```
    },
    changeOwner: function(req, res){
        console.log("changing Owner");

        var array = req.params.holder.split("-");
        var key = array[0];
        var holder = array[1];

        var fabric_client = new Fabric_Client();

        // setup the fabric network
        var channel = fabric_client.newChannel('mychannel');
        var peer = fabric_client.newPeer('grpc://localhost:7051');
        channel.addPeer(peer);
        var order = fabric_client.newOrderer('grpc://localhost:7050');
        channel.addOrderer(order);

        var member_user = null;
        var store_path = path.join(os.homedir(), '.hfc-key-store');
        console.log('Store path: '+store_path);
        var tx_id = null;

        // create the key value store as defined in the fabric-client/config/default.json 'key-value-store' setting
        Fabric_Client.newDefaultKeyValueStore({ path: store_path
    }).then((state_store) => {
        // assign the store to the fabric client
        fabric_client.setStateStore(state_store);
        var crypto_suite = Fabric_Client.newCryptoSuite();
        // use the same location for the state store (where the users' certificate are kept)
        // and the crypto store (where the users' keys are kept)
        var crypto_store = Fabric_Client.newCryptoKeyStore({path: store_path});
        crypto_suite.setCryptoKeyStore(crypto_store);
        fabric_client.setCryptoSuite(crypto_suite);

        // get the enrolled user from persistence, this user will sign all requests
        return fabric_client.getUserContext('user1', true);
    }).then((user_from_store) => {
        if (user_from_store && user_from_store.isEnrolled()) {
            console.log('Successfully loaded user1 from persistence');
            member_user = user_from_store;
        } else {
            throw new Error('Failed to get user1.... run registerUser.js');
        }

        // get a transaction id object based on the current user assigned to fabric client
        tx_id = fabric_client.newTransactionID();
        console.log("Assigning transaction_id: ", tx_id.transaction_id);

        // Change Owner of vehicle - ID - new Owner
    });
    }
};
```

Figure 7-17 `changeOwner` function - Complete code (1 of 4)

```

300
309 // Change Owner of vehicle - ID , new Owner
310 // send proposal to endorser
311 var request = {
312 //targets : --- letting this default to the peers assigned to the channel
313 chaincodeId: chainCodeName,
314 fcn: 'changeVehicleOwner',
315 args: [key, holder],
316 chainId: channelName,
317 txId: tx_id
318 };
319
320 // send the transaction proposal to the peers
321 return channel.sendTransactionProposal(request);
322 }).then((results) => {
323 var proposalResponses = results[0];
324 var proposal = results[1];
325 let isProposalGood = false;
326 if (proposalResponses && proposalResponses[0].response &&
327 proposalResponses[0].response.status === 200) {
328 isProposalGood = true;
329 console.log('Transaction proposal was good');
330 } else {
331 console.error('Transaction proposal was bad');
332 }
333 if (isProposalGood) {
334 console.log(util.format(
335 'Successfully sent Proposal and received ProposalResponse: Status - %s, message - "%s"',
336 proposalResponses[0].response.status, proposalResponses[0].response.message));
337
338 // build up the request for the orderer to have the transaction committed
339 var request = {
340 proposalResponses: proposalResponses,
341 proposal: proposal
342 };
343
344 // set the transaction listener and set a timeout of 30 sec
345 // if the transaction did not get committed within the timeout period,
346 // report a TIMEOUT status
347 var transaction_id_string = tx_id.getTransactionID(); //Get the transaction ID string to be used by the event pro
348 var promises = [];
349
350 var sendPromise = channel.sendTransaction(request);
351 promises.push(sendPromise); //we want the send transaction first, so that we know where to check status
352
353 // get an eventhub once the fabric client has a user assigned. The user
354 // is required because the event registration must be signed
355 let event_hub = fabric_client.newEventHub();
356 event_hub.setPeerAddr('grpc://localhost:7053');
357

```

Figure 7-18 changeOwner function - Complete code (2 of 4)

```

357 // using resolve the promise so that result status may be processed
358 // under the then clause rather than having the catch clause process
359 // the status
360 let txPromise = new Promise((resolve, reject) => {
361   let handle = setTimeout(() => {
362     event_hub.disconnect();
363     resolve({event_status : 'TIMEOUT'}); //we could use reject(new Error('Trnsaction did not complete within 30 sec
364   }, 3000);
365   event_hub.connect();
366   event_hub.registerTxEvent(transaction_id_string, (tx, code) => {
367     // this is the callback for transaction event status
368     // first some clean up of event listener
369     clearTimeout(handle);
370     event_hub.unregisterTxEvent(transaction_id_string);
371     event_hub.disconnect();
372
373     // now let the application know what happened
374     var return_status = {event_status : code, tx_id : transaction_id_string};
375     if (code !== 'VALID') {
376       console.error('The transaction was invalid, code = ' + code);
377       resolve(return_status); // we could use reject(new Error('Problem with the tranaction, event status ::'+code
378     } else {
379       console.log('The transaction has been committed on peer ' + event_hub.ep_endpoint.addr);
380       resolve(return_status);
381     }
382   }, (err) => {
383     //this is the callback if something goes wrong with the event registration or processing
384     reject(new Error('There was a problem with the eventhub ::'+err));
385   });
386 });
387 promises.push(txPromise);
388
389 return Promise.all(promises);
390 } else {
391   console.error('Failed to send Proposal or receive valid response. Response null or status is not 200. exiting...');
392   res.send("Could not find Vehicle");
393   // throw new Error('Failed to send Proposal or receive valid response. Response null or status is not 200. exiting...');
394 }
395 }).then((results) => {
396   console.log('Send transaction promise and event listener promise have completed');
397   // check the results in the order the promises were added to the promise all list
398   if (results && results[0] && results[0].status === 'SUCCESS') {
399     console.log('Successfully sent transaction to the orderer. ');
400     res.json(tx_id.getTransactionID());
401   } else {
402     console.error('Failed to order the transaction. Error code: ' + response.status);
403     res.send("Could not find Vehicle");
404   }
405 }
406
407 if(results && results[1] && results[1].event_status === 'VALID') {

```

Figure 7-19 changeOwner function - Complete code (3 of 4)

```

407   if(results && results[1] && results[1].event_status === 'VALID') {
408     console.log('Successfully committed the change to the ledger by the peer');
409     res.json(tx_id.getTransactionID())
410   } else {
411     console.log('Transaction failed to be committed to the ledger due to ::'+results[1].event_status);
412   }
413 }).catch((err) => {
414
415 });
416
417 }
418
419 }

```

Figure 7-20 changeOwner function - Complete code (4 of 4)

15. Save the file.

7.3.4 Testing the changes to the application

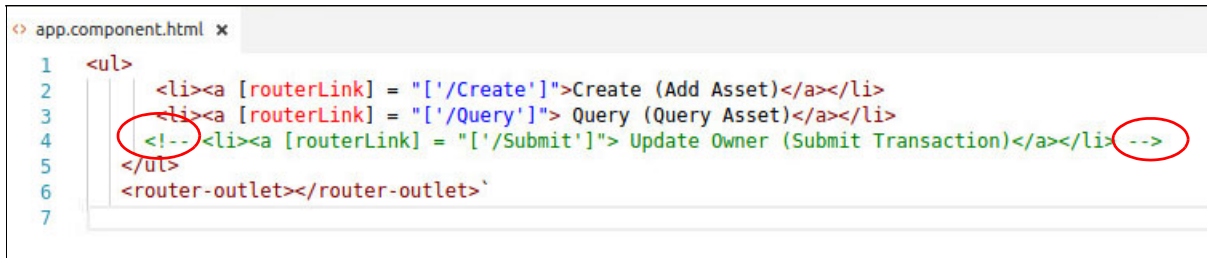
In this section, you use the Angular web application to test the changes that you made to the Node.js sample code.

Perform the following steps:

1. Open the file `app.component.html` by running the following commands:

```
cd
~/Blockchain_Redbook_Application/Fabric/Exercise7/Front-End/Angular2/src/app/
code app.component.html
```

2. Uncomment line 4 by removing `<!--` from the beginning and `-->` from the end of the line to test the results of the `changeOwner` function that was added to the Node.js server (Figure 7-21).
3. Press **Ctrl + S** to save the file after making the change.



```
<> app.component.html x
1 <ul>
2   <li><a [routerLink] = "['/Create']">Create (Add Asset)</a></li>
3   <li><a [routerLink] = "['/Query']"> Query (Query Asset)</a></li>
4   <!--<li><a [routerLink] = "['/Submit']"> Update Owner (Submit Transaction)</a></li>-->
5 </ul>
6 <router-outlet></router-outlet>`
7
```

Figure 7-21 `App.component.html`

4. Repeat all the steps in 7.3.1, “Registering and enrolling users ” on page 209 and steps 1 on page 210 and 2 on page 210 in 7.3.2, “Running the sample application” on page 210.

Note: If you get the following response, ignore it; it means `user1` is already registered.

```
Failed to register: Error: fabric-ca request register failed with errors
[[{"code":0,"message":"Registration of 'user1' failed: Identity 'user1' is
already registered"}]]
```

You should have two terminal windows:

- Terminal window 1: Running the Node.js server on `localhost:8000`.
- Terminal window 2: Running the Angular web application on `localhost:4200`.

5. Open your browser on `localhost:4200`, as shown in Figure 7-22.

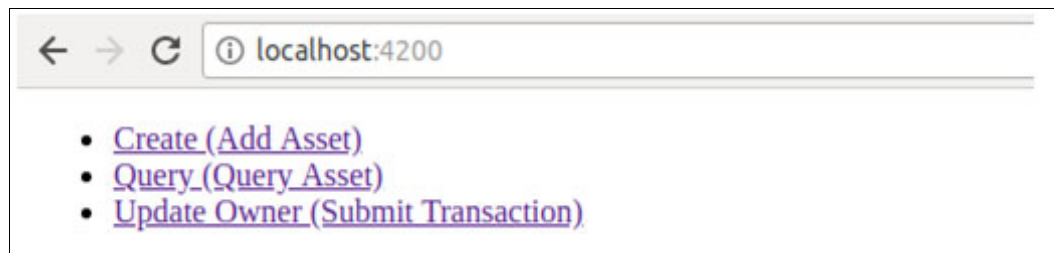


Figure 7-22 Web application page

There is a new option, `Update Owner (Submit Transaction)`, which is provided by the function `changeOwner` in the Node.js server.

6. Click **Update Owner (Submit Transaction)**. Change the owner of the vehicle to Maria and click **Submit** (Figure 7-23).

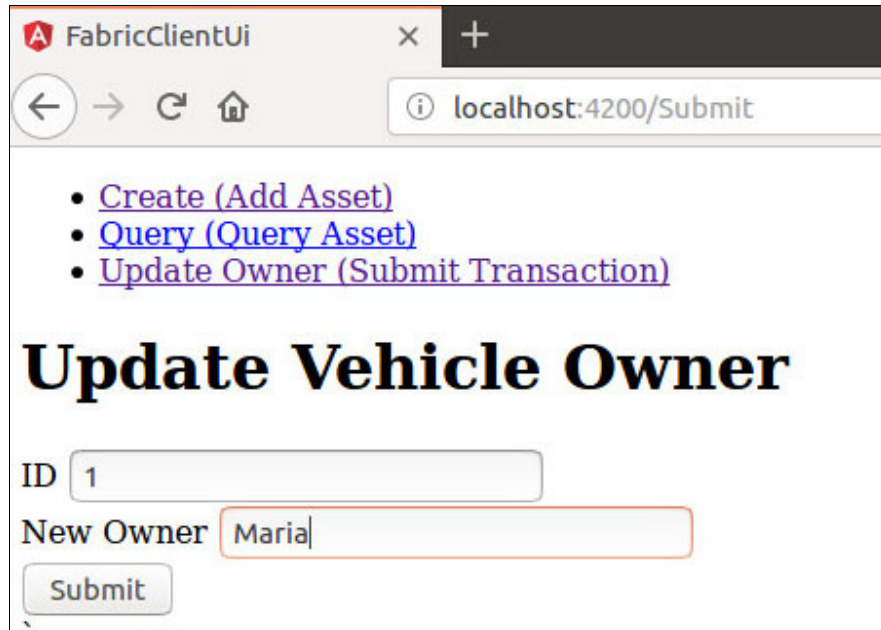


Figure 7-23 Update Vehicle owner page

7. Return to the terminal window where the Node.js server is running. Verify that the transaction was committed successfully by reading the following output:
changing Owner
Store path:/home/BlockchainUser/.hfc-key-store
Successfully loaded user1 from persistence
Transaction proposal was good
Successfully sent Proposal and received ProposalResponse: Status - 200, message - "OK"

8. Click **Query (Query Asset)** to search for the vehicle. Enter ID = 1 and click **Query**, as shown in Figure 7-24. Maria should be the new owner.

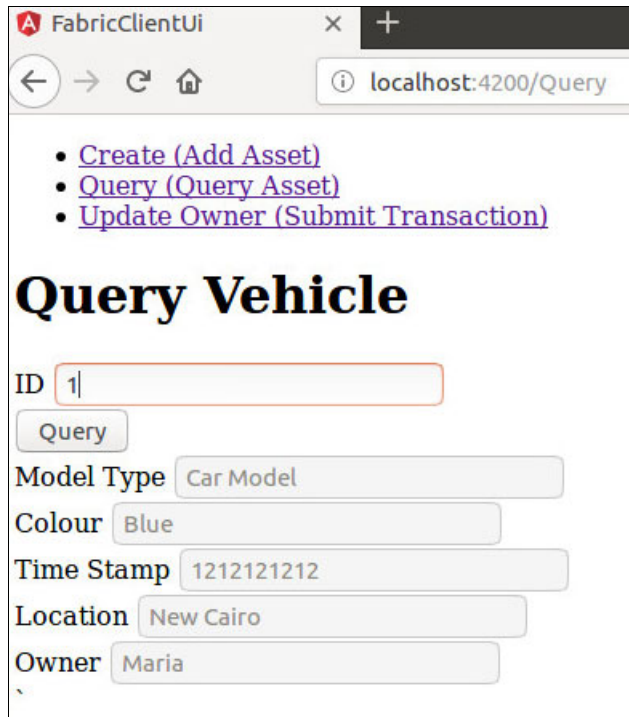


Figure 7-24 Query Vehicle

Note: If your query returns no results, ensure that Cors is enabled in your Firefox browser as described in 7.1.3, “Prerequisites” on page 205. If you forgot to enable Cors, enable it now, restart your browser and open your browser on localhost:4200, as shown in Figure 7-22. Repeat step 8 on page 225 (Query).

7.3.5 Cleaning up the environment

In this section, you will clean up the environment to have it ready for the next exercises.

Perform the following steps:

1. Stop the Node.js server by pressing **Ctrl + C** in the terminal window where it is running.
2. Stop the Angular web application by pressing **Ctrl + C** in the terminal window where it is running.
3. Run the following commands to remove the running docker containers:

```
docker rm -f $(docker ps -aq)
docker system prune -f
docker volume rm $(docker volume ls -q)
docker rm $(docker ps -aq)
```
4. Run the following commands to remove the stored credentials for the Admin and user1:

```
cd ~
rm -rf .hfc-key-store/
```

7.4 Exercise review and wrap-up

In this exercise, you enhanced the code in the sample Node.js application to add a function that is called **changeOwner**. You use the Fabric Client SDK to build the function.

You learned the flow of a submitted transaction that interacts with the chaincode.

You tested the **changeOwner** function and observed the effects that the submitted transaction had on the web application and the chaincode.

Code solutions

If you run into problems copying and pasting the code in this exercise, you can find the complete code in the code-solutions folder. Double-click the **File** icon, navigate to **/Blockchain_Redbook_Application/code-solutions** and locate the folder for this exercise (Figure 7-25).

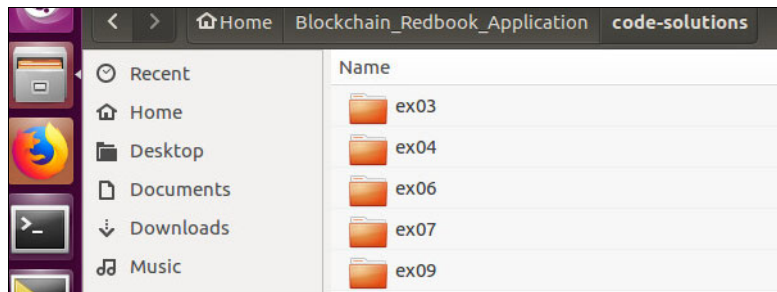


Figure 7-25 Code-solutions folder

Recommended reading

Here are some resources for more information about the topics that are described in this exercise:

- ▶ Hyperledger Fabric SDK for node.js overview
<https://fabric-sdk-node.github.io/>
- ▶ Hyperledger Fabric SDK for Node.js code
<https://github.com/hyperledger/fabric-sdk-node>
- ▶ Hyperledger Fabric: Writing Your First Application
https://hyperledger-fabric.readthedocs.io/en/v1.0.6/write_first_app.html