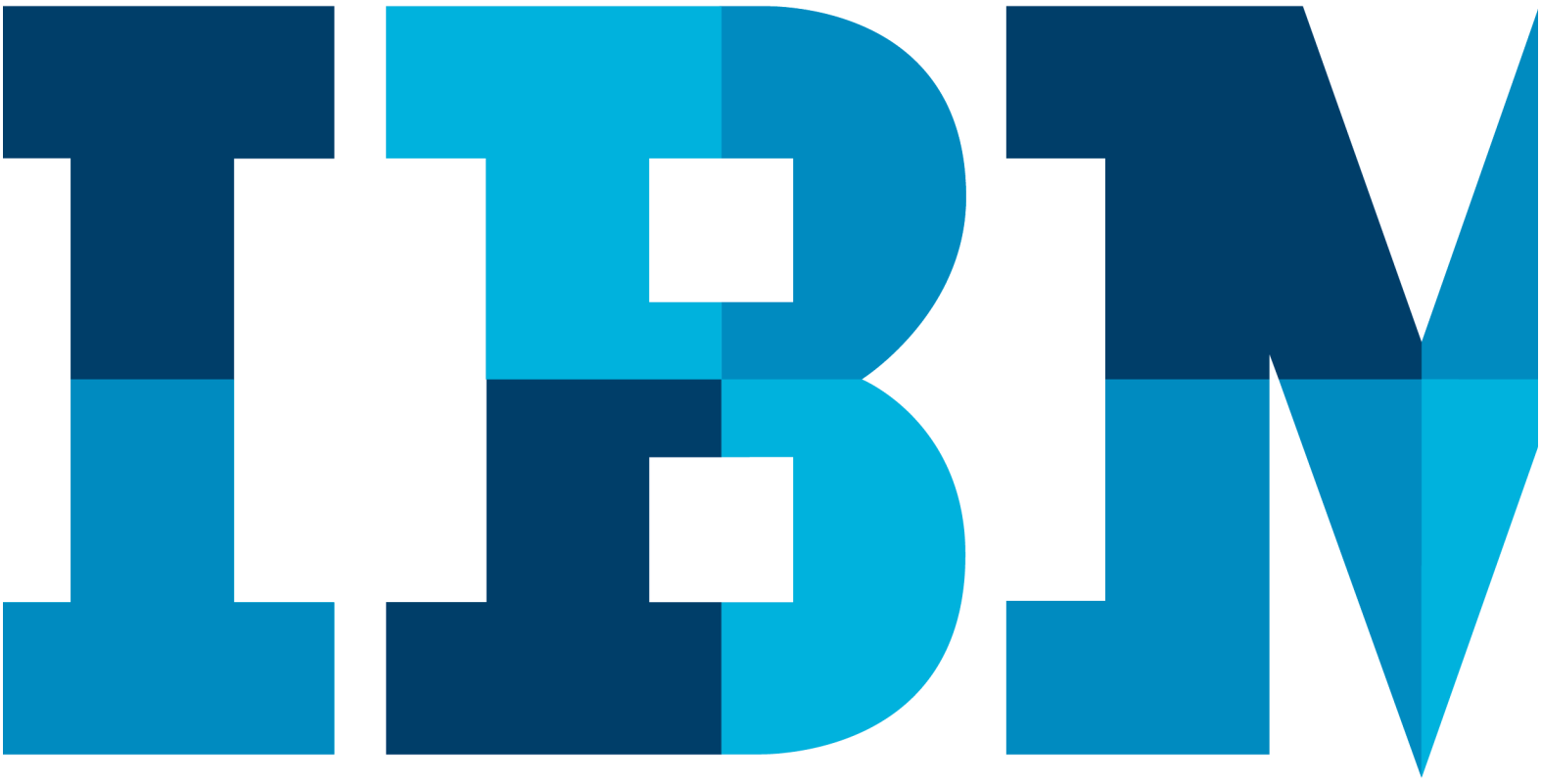
IBM Blockchain Hands-On
Fabric Node.js SDK (HFC)

Lab Two



IBM

Contents

Overview

The aim of this lab is to get you writing some basic applications with the Fabric SDK. This will be done by replacing sections of the balance-transfer demo to illustrate the key features of HFC.

More information can be found here: https://fabric-sdk-node.github.io/tutorial-index_.html

Introduction

Pre-requisites:

- 4 cores
- 4GB RAM
- VMWare V10+
- The lab virtual machine

The virtual machine is based on Linux Ubuntu 16.04 and contains Hyperledger Fabric V1.0, Golang, Git, Visual Studio Code and Firefox.

A network needs to be visible to the virtual machine (even if the network is just to the host environment). If you do not see the up/down arrows in the status bar at the top of the screen, or if you receive errors about no network being available, please tell the lab leader. The virtual machine might need to be reconfigured in NAT mode.

There are no additional files or software that is proprietary to the lab in the virtual machine. This means that the lab may be run on a machine without the without a lab virtual machine if Hyperledger Fabric and the other pre-requisites have been installed.

It is recommended that students have previously completed the Blockchain Explained and Blockchain Explored labs.

Section 1. Setup & Overview of Structure

1.1. Setting up

a. Make a new folder

Navigate to `~/workspace/hfc-getting-started/examples/balance-transfer` and create a new folder called `workshop`.

b. Copy over dependencies

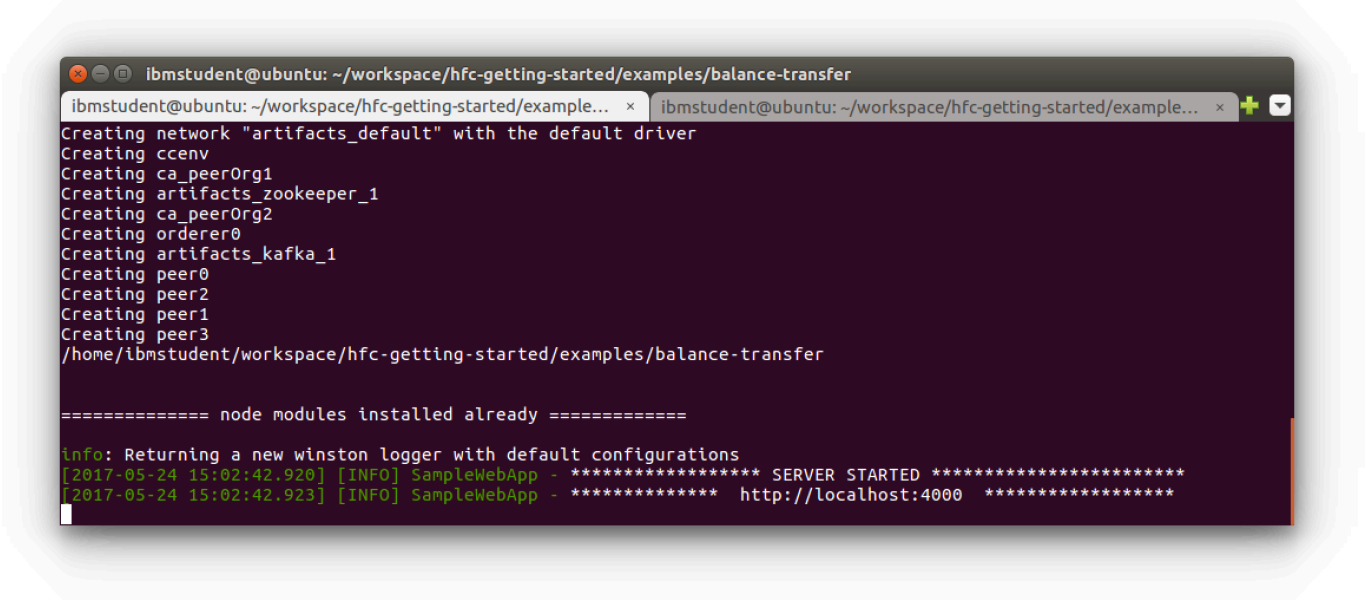
Copy `helper.js` and `network-config.json` from `balance-transfer/app` into this new directory.

c. Run the app

Navigate back into balance transfer and open up a terminal window with two tabs. Execute the following in one:

```
./runApp.sh
```

When you see the following output:



Run the following in the other tab:

```
./testAPIs.sh
```

This will execute balance transfer's code which will set up a network, create some channels, deploy some chaincode to it and then transact with this chaincode.

d. Structure of Balance Transfer

The code for each of the steps in balance transfer is contained within a suitably titled file (install-chaincode, invoke-transaction etc.) in `balance-transfer/app`.

Over the course of this lab we will be replacing these files with our own versions that perform the same tasks.

Section 2. Channels

3.1. Create Channel

a. Make a new file

Create a new file called `create-channel.js` in `balance-transfer/workshop` and insert the following:

```
var fs = require('fs');
var path = require('path');
var helper = require('./helper.js');
var logger = helper.getLogger('Create-Channel');

var createChannel = function(channelName, channelConfigPath, username, orgName) {
};

exports.createChannel = createChannel;
```

Each of the files in app defines a function like the one above that interacts with the chaincode in some way. We will be defining new versions of these functions.

b. Set up the network

Add the following in the body of the function:

```
logger.debug('\n\n====== Creating Channel \'' + channelName + '\' ======\n');

helper.setupOrderer();
var chain = helper.getChainForOrg(orgName);
```

The ‘helper’ module is used in balance-transfer for setting up details such as the addresses of orderers and peers. To save time we will also be making use of it. ‘setupOrderer’ scans the config file for the url of the ordering service and adds it to the client object for broadcasting to, please inspect the contents of helper.js and search for the function for the specific implementation details.

The getChainForOrg function returns the chain object for the current organisation. While channels are cross-organisation, orderers do belong to an organisation and as such we need to create the channel on an orderer that is owned by the organisation the nominated user belongs to.

c. Get the User context.

All transactions need to be signed by a user with appropriate privileges. As such the function requires a user be specified to do this signing.

The signing will be done by the Fabric Client automatically however to do this in the first place we need to check that a user context exists, again the helper function handily does this for us. Add the following below the code in **b**.

```
return helper.getRegisteredUsers(username, orgName).then((member) => {

});
```

`getRegisteredUsers` checks if there exists a User Context, if so it returns the User object representing said user. If there isn't it contacts the CA and downloads the certificate for the user, storing it in the KeyStore and setting the user as the current context. Again, please examine `helper.js` for further implementation details.

d. Send the create channel request

In the body of the `getRegisteredUsers` callback add the following:

```
var request = {
      envelope: fs.readFileSync(path.join(__dirname, channelConfigPath))
};

return chain.createChannel(request);
```

The `channelConfigPath` is passed in through the augments and is the path to `mychannel.tx`, the settings for the channel. This is packaged into a request object and passed into `createChannel` which creates a transaction, signs it with the User context and sends it to the network.

e. Respond to the outcome

Chain a `.then()` to the end of the `helper.getRegisteredUsers` promise like so:

```
return helper.getRegisteredUsers(username, orgName).then((member) => {

      var request = {
            envelope: fs.readFileSync(path.join(__dirname, channelConfigPath))
      };

      return chain.createChannel(request);

}).then((createChannelResults) => {

});
```

In the body of the new `.then()` add the following:

```
let response = {};

if (createChannelResults && createChannelResults.status === 'SUCCESS') {
    let msg = 'Channel \'' + channelName + '\' created Successfully';
    response.message = msg;
    response.success = true;
    logger.debug(msg);
} else {
```

```
    let msg = '\n!!!!!!!!! Failed to create the channel \'' + channelName +
'\' !!!!!!!!!\n\n';
    response.message = msg;
    response.success = false;
    logger.error(msg);
    throw new Error(msg);
}

return response;
```

This will output a Success or Failure message (both as a log and as a REST reply) in response to whether the channel creation has succeeded.

f. Creating a test script

To test individual scripts we are running we'll need a new test script. Create a new file in `balance-transfer` called `testWorkshop.sh`. Add the following to the new file:

```
#!/bin/bash

set -ev

jq --version > /dev/null 2>&1
if [ $? -ne 0 ]; then
     echo "Please Install 'jq' https://stedolan.github.io/jq/ to execute this
script"
     echo
     exit 1
fi
starttime=$(date +%s)


echo "POST request Enroll on Org1  ..."
echo
ORG1_TOKEN=$(curl -s -X POST \
  http://localhost:4000/users \
  -H "cache-control: no-cache" \
  -H "content-type: application/x-www-form-urlencoded" \
  -d 'username=Jim&orgName=org1')
echo $ORG1_TOKEN
ORG1_TOKEN=$(echo $ORG1_TOKEN | jq ".token" | sed "s/\"//g")
echo
echo "ORG1 token is $ORG1_TOKEN"
echo


echo "POST request Create channel  ..."
echo
curl -s -X POST \
  http://localhost:4000/channels \
  -H "authorization: Bearer $ORG1_TOKEN" \
  -H "cache-control: no-cache" \
```

```
  -H "content-type: application/json" \
  -H "x-access-token: $ORG1_TOKEN" \
  -d '{
     "channelName":"mychannel",
     "channelConfigPath":"../artifacts/channel/mychannel.tx"
}'
echo
echo
sleep 5
```

runApp.sh stands up a REST server with app.js routing requests to the files mentioned in 1.1(d). This script makes calls to the rest server, first ensuring that the credentials have been set up correctly and then calling the create channel endpoint.

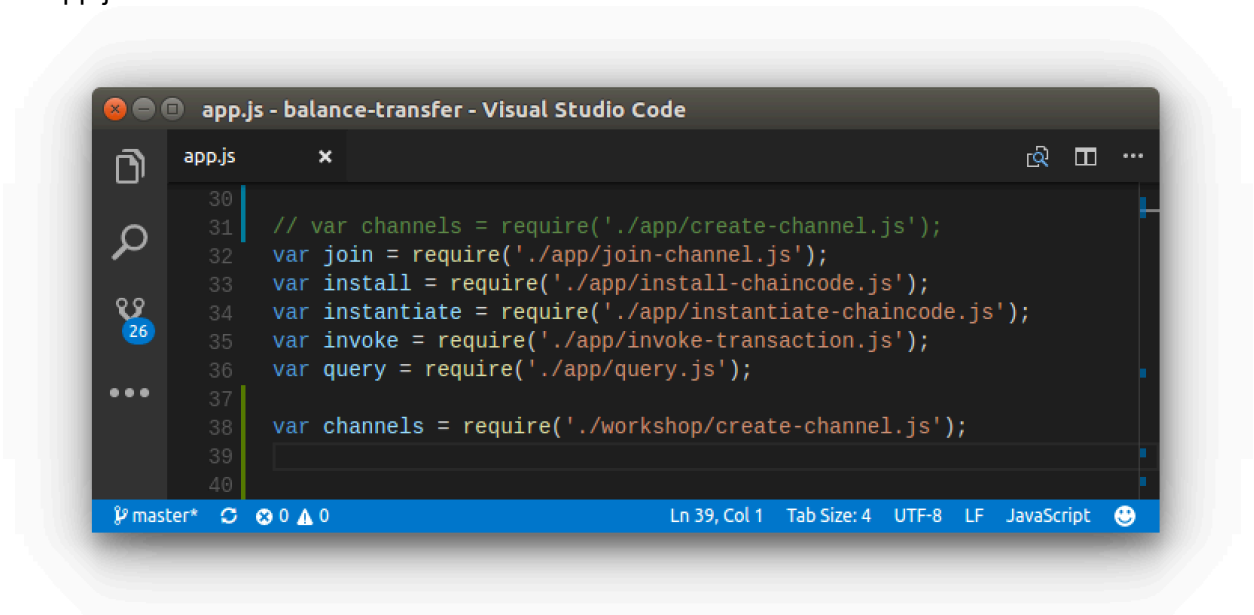
Issue the following to make testWorkshop.sh executable:

```
chmod +x testWorkshop.sh
```

Currently, this will be routed to the default file. We shall now change this.

g. Edit app.js

Open app.js in Visual Studio Code:



Comment out the line importing app/create-channel and add a new one below importing the file we have just created:

```
var channels = require('./workshop/create-channel.js');
```

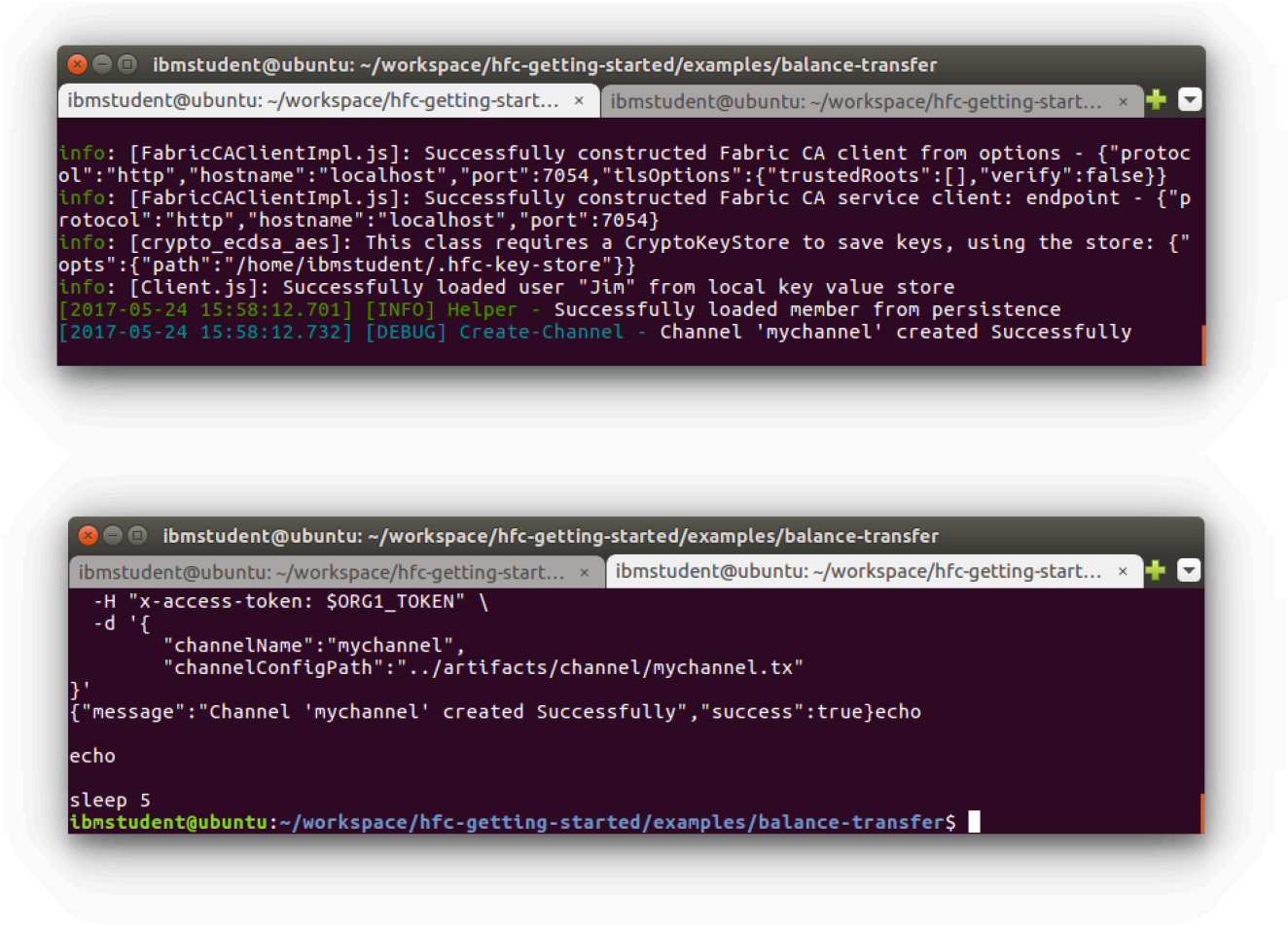
h. Run the test script

Restart the network by using `./runApp.sh`, wait until the same text shown in 1.1 appears.

When this is the case, switch to the other tab and issue the following:

```
./testWorkshop.sh
```

You should get output similar to the following:



3.2. Join Channel

a. Make a new file

Create a new file in workshop called join-channel.js, insert the following code:

```
var util = require('util');
var helper = require('./helper.js');
var logger = helper.getLogger('Join-Channel');

var joinChannel = function(channelName, peers, username, org) {
});

exports.joinChannel = joinChannel;
```

b. Set up the network

Add the following code inside the `joinChannel` body:

```
helper.setupOrderer();
var chain = helper.getChainForOrg(org);
var targets = helper.getTargets(peers, org);
```

In addition to the same code as last time we also have `getTargets.` The join channel request will go out to multiple peers concurrently. Unlike where there is a single ordering service we need a list of the 'targets' to issue the transaction request. The `getTargets` function gathers this for us while it is registering the peers with the client. See helper.js for implementation details.

c. Get the User Context

Use the same code as last time for this, added below the network setup code:

```
return helper.getRegisteredUsers(username, org).then((user) => {
});
```

d. Send the join channel command

Add the following code to the body of the `getRegisteredUsers` callback to issue the `joinNetwork` request:

```
        nonce = helper.getNonce();
        tx_id = chain.buildTransactionID(nonce, user);

        var request = {
              targets: targets,
              txId: tx_id,
              nonce: nonce
        };

    return chain.joinChannel(request);
```

The new parts of this code are the `nonce` and the `tx_id`. Transaction IDs must be generated by you – fortunately HFC makes ready some tools to easily do this (namely an inbuilt function in chain).]

The `nonce` is a random integer of predetermined length that is combined with details about the `user` in `buildTransactionID` forms a pseudorandom transaction ID that can be used by the transaction.

e. Respond to the outcome

Attach a new then to the end of the `getRegisteredUsers` promise like in 2.1(e). Add the following code in the body of the new promise callback:

```
}).then((joinChannelResults) => {
```

```
// Check they were ok
let ok = joinChannelResults.length && true;
for(let i = 0; i < joinChannelResults.length; i++) {
    if(joinChannelResults[i].response.status !== 200) {
        ok = false;
    }
}

// Respond accordingly
let response = {};

if(ok) {
    let msg = util.format('Successfully joined peers in organization %s to the
channel \'%s\'', org, channelName);
    response.message = msg;
    response.success = true;
    logger.debug(msg);
} else {
    let msg = util.format('Failed to join peers in organization %s to the channel
\'%s\'', org, channelName);
    response.message = msg;
    response.success = false;
    logger.error(msg);
    throw new Error(msg);
}

return response;
```

This code iterates through the `joinChannelResults` looking for any bad replies. If it finds them it sets the `ok` flag to false which in turn triggers the output of error messages.

f. Add a new section to testWorkshop.sh

Add a new section to `testWorkshop.sh` after the existing code:

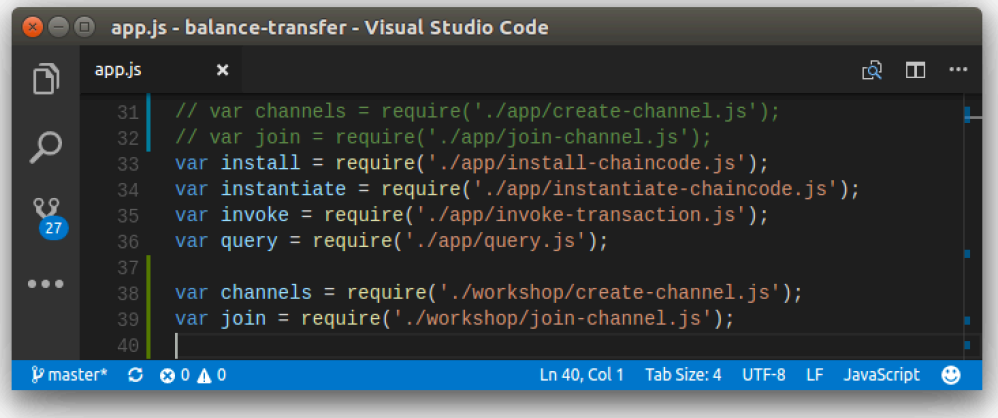
```
echo "POST request Join channel on Org1"
echo
curl -s -X POST \
  http://localhost:4000/channels/mychannel/peers \
  -H "authorization: Bearer $ORG1_TOKEN" \
  -H "cache-control: no-cache" \
  -H "content-type: application/json" \
  -H "x-access-token: $ORG1_TOKEN" \
  -d '{
      "peers": ["localhost:7051","localhost:7056"]
}'
echo
echo

sleep 5
```

g. Edit app.js

Comment out the import for the `join` library and add an entry for `join-channel.js`:

```
var join = require('./workshop/join-channel.js');
```

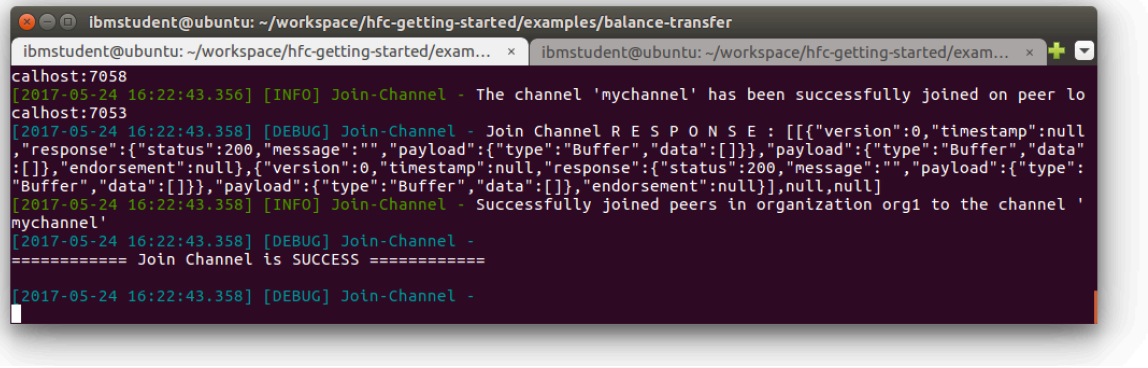


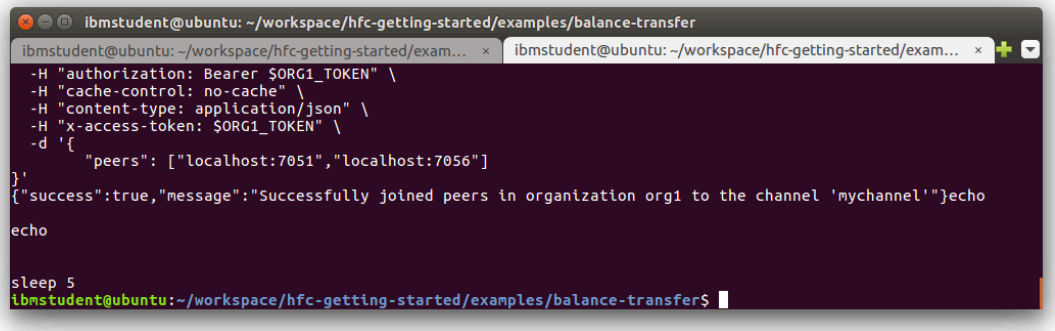
h. Run the Test script

Restart the network by using `./runApp.sh`, wait until the same text shown in 1.1 appears.

When this is the case, switch to the other tab and issue the following: `./testWorkshop.sh`

You should get output similar to the following:





Section 3. Chaincode

3.1. Installing Chaincode

a. Make a new file

Create a new file in `workshop` called `install-chaincode.js`, insert the following code into it:

```
const path = require('path');
const fs = require('fs');
const util = require('util');
const config = require('../config.json');
const helper = require('./helper.js');
const logger = helper.getLogger('install-chaincode');

var installChaincode = function(peers, chaincodeName, chaincodePath,
chaincodeVersion, username, org) {
});

exports.installChaincode = installChaincode;
```

b. Set up the network

Add the following code to the body:

```
helper.setupChaincodeDeploy();
let chain = helper.getChainForOrg(org);
helper.setupOrderer();
let targets = helper.getTargets(peers, org);
helper.setupPeers(chain, peers, targets);
```

This builds slightly on the `join-channel` setup code. The block begins with `setupChaincodeDeploy` which sets the GOPATH environment variable (for this process) to the location of the chaincode in, this being `hfc-getting-started/test/fixtures/src/github.com/example_cc`. The final line wires the peers together, registering the endpoint of each peer with the others.

c. Get the user context

Add the following under the setup code:

```
return helper.getRegisteredUsers(username, org).then((user) => {
});
```

d. Put together and send the install proposal

Add the following to the body of the `getRegisteredUsers` function:

```
let nonce = helper.getNonce();
let tx_id = chain.buildTransactionID(nonce, user);

let request = {
    targets: targets,
    chaincodePath: chaincodePath,
    chaincodeId: chaincodeName,
    chaincodeVersion: chaincodeVersion,
    txId: tx_id,
    nonce: nonce
};

// Send the install proposal (although it's really more of an order...)
logger.info('Sending Chaincode Install Proposal for ' + chaincodeName + '...');
return chain.sendInstallProposal(request);
```

As you can see, the same pattern has existed in the last 3 sections. A request object is built up containing a `nonce` and a `tx_id` accompanied by various data depending on what the function requires. If you inspect Lab 1 and look at the install/instantiate/channel functions you will notice that the request objects being built here map onto the arguments taken by those commands.

e. Respond to the outcome

Attach a new then to the end of the `getRegisteredUsers` promise like in 2.1(e). Add the following code in the body of the new promise callback:

```
}).then((results) => {

    // Decompose the responses
    let installChaincodeResults = results[0];

    // Check they were ok
    let ok = installChaincodeResults.length && true;
    for(let i = 0; i < installChaincodeResults.length; i++) {
        if(installChaincodeResults[i].response.status !== 200) {
            ok = false;
        }
    }

    // Respond accordingly
    let response = {};

    if(ok) {
        let msg = util.format('Successfully installed chaincode on peers in
organization %s', org);
        response.message = msg;
```

```
        response.success = true;
        logger.info(msg);
    } else {
        let msg = util.format('Failed to install chaincode on peers in
organization %s', org);
        response.message = msg;
        response.success = false;
        logger.info(msg);
        throw new Error(msg);
    }

    return response;
});
```

This code functions in an almost identical manner to that of the code which examines the join channel responses, save for the names of the variables it is inspecting and that the results obtained in a different manner. Responses to chaincode install requests contain a variety of data, one piece of which is the node responses.

f. Add a new section to testWorkshop.sh

Add a new section to `testWorkshop.sh` after the existing code:

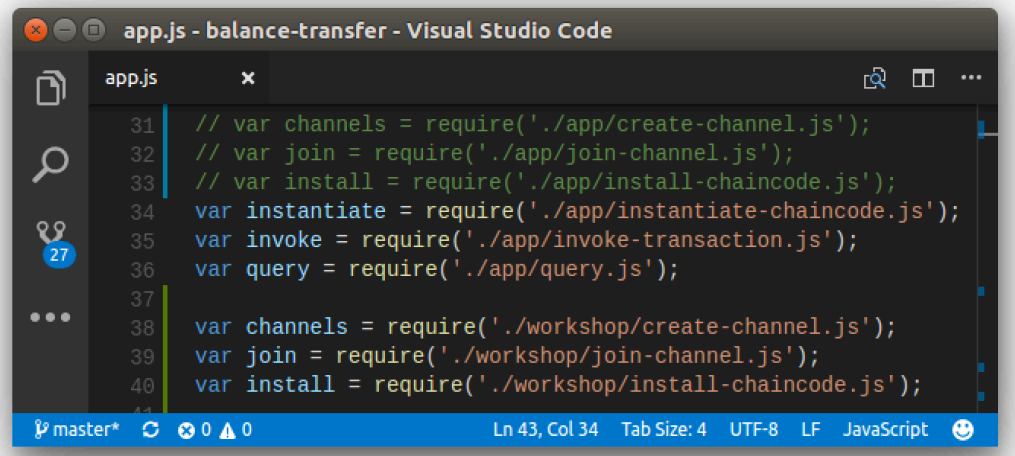
```
echo "POST Install chaincode on Org1"
echo
curl -s -X POST \
  http://localhost:4000/chaincodes \
  -H "authorization: Bearer $ORG1_TOKEN" \
  -H "cache-control: no-cache" \
  -H "content-type: application/json" \
  -H "x-access-token: $ORG1_TOKEN" \
  -d '{
      "peers": ["localhost:7051","localhost:7056"],
      "chaincodeName":"mycc",
      "chaincodePath":"github.com/example_cc",
      "chaincodeVersion":"v0"
}'
echo
echo

sleep 10
```

g. Edit app.js

Comment out the import for the `install` library and add an entry for `install-chaincode.js`:

```
var install = require('./workshop/install-chaincode.js');
```

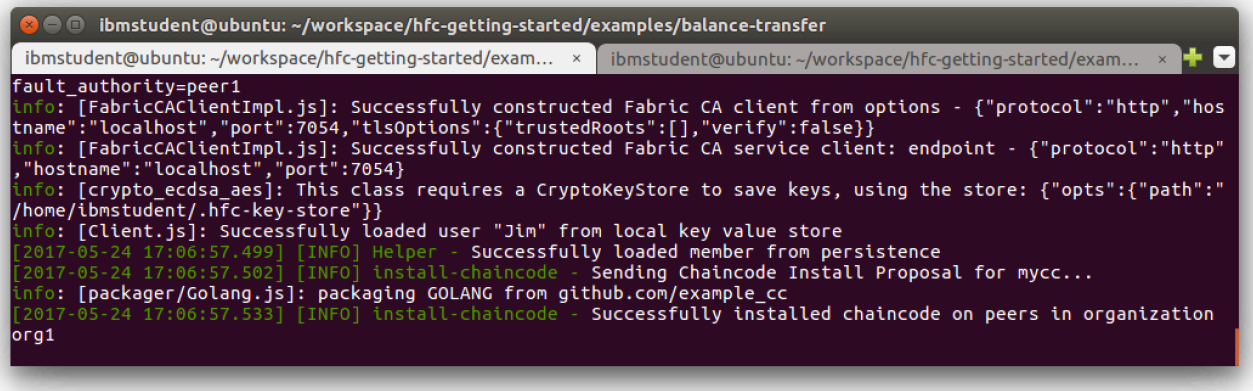


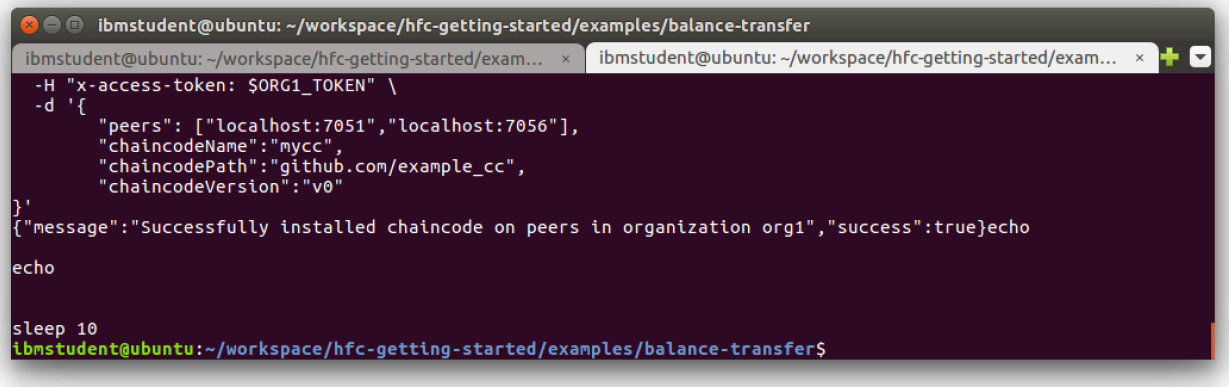
h. Run the Test script

Restart the network by using `./runApp.sh`, wait until the same text shown in 1.1 appears.

When this is the case, switch to the other tab and issue the following: `./testWorkshop.sh`

You should get output similar to the following:





3.1. Instantiating Chaincode

a. Make a new file

Create a new file in `workshop` called `instantiate-chaincode.js`, insert the following code into it:

```
var helper = require('./helper.js');
var logger = helper.getLogger('instantiate-chaincode');

var instantiateChaincode = function(peers, channelName, chaincodeName,
chaincodePath, chaincodeVersion, functionName, args, username, org) {
});

exports.instantiateChaincode = instantiateChaincode;
```

b. Set up the network

Add the following code to the body:

```
helper.setupChaincodeDeploy();
let chain = helper.getChainForOrg(org);
helper.setupOrderer();
let targets = helper.getTargets(peers, org);
helper.setupPeers(chain, peers, targets);
```

The code used for network setup when instantiating is identical to that used for installation.

c. Get the user context

Add the following under the setup code:

```
return helper.getRegisteredUsers(username, org).then((user) => {
});
```

d. Initialise the MSPs

Add the following code in the body of `getRegisteredUsers`:

```
member = user;

return chain.initialize();
```

The first line simply saves the user context object, we need to do this because of the second line – i.e. a new promise is started and as such we can't just use it locally.

The `chain.initialize()` function initialises the connections to the MSPs (CAs) that allow it to verify the signatures on any incoming transaction proposals related to this chaincode.

e. Send the instantiate proposal

Attach another `then()` on the end of the `getRegisteredUsers` promise chain:

```
}).then((success) => {
    nonce = helper.getNonce();
    tx_id = chain.buildTransactionID(nonce, member);

    var request = {
        targets: targets,
        chaincodePath: chaincodePath,
        chaincodeId: chaincodeName,
        chaincodeVersion: chaincodeVersion,
        fcn: functionName,
        args: args,
        chainId: channelName,
        txId: tx_id,
        nonce: nonce
    };

    return chain.sendInstantiateProposal(request);
});
```

The instantiate begins first with a proposal much like invoke transactions do, this contains the bulk of the information needed to instantiate the chaincode missing only the approval of the network to do so.

Remember that adding chaincode to a ledger is an equivalent action to adding data to a ledger, as such chaincode must adhere to the same submission protocol and as such can be rejected before it is formally made part of a channel.

f. Send the instantiate transaction

Attach another `then()` on the end of the `getRegisteredUsers` promise chain:

```
}).then((results) => {

    var proposalResponses = results[0];
    var proposal = results[1];
    var header = results[2];

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

    return chain.sendTransaction(request);

})
```

If the responses are acceptable (this is not checked here for code simplicity) then issue the instantiate transaction.

g. Respond to outcome

Add a final then() to the promise chain:

```
}).then((instantiateChaincodeResults) => {
    let response = {};

    if (instantiateChaincodeResults && instantiateChaincodeResults.status ===
'SUCCESS') {
        let msg = chaincodeName + ' instantiated successfully';
        response.message = msg;
        response.success = true;
        logger.debug(msg);
    } else {
        let msg = 'Failed to instantiate ' + chaincodeName;
        response.message = msg;
        response.success = false;
        logger.error(msg);
        throw new Error(msg);
    }

    return response;
});
```

Much the same as previous response callbacks this again inspects the response for markers that the operation was successful and if this is so, outputs appropriately.

h. Add a new section to testWorkshop.sh

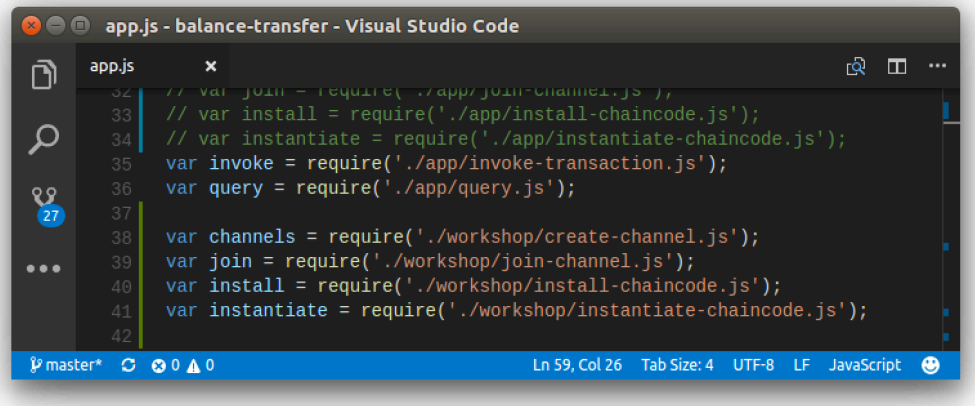
Add a new section to `testWorkshop.sh` after the existing code:

```
echo "POST instantiate chaincode on peer1 of Org1"
echo
curl -s -X POST \
  http://localhost:4000/channels/mychannel/chaincodes \
  -H "authorization: Bearer $ORG1_TOKEN" \
  -H "cache-control: no-cache" \
  -H "content-type: application/json" \
  -H "x-access-token: $ORG1_TOKEN" \
  -d '{
      "peers": ["localhost:7051"],
      "chaincodeName":"mycc",
      "chaincodePath":"github.com/example_cc",
      "chaincodeVersion":"v0",
      "functionName":"init",
      "args":["a","100","b","200"]
}'
echo
sleep 10
```

i. Edit app.js

Comment out the import for the `instantiate` library and add an entry for `instantiate-chaincode.js`:

```
var instantiate = require('./workshop/instantiate-chaincode.js');
```

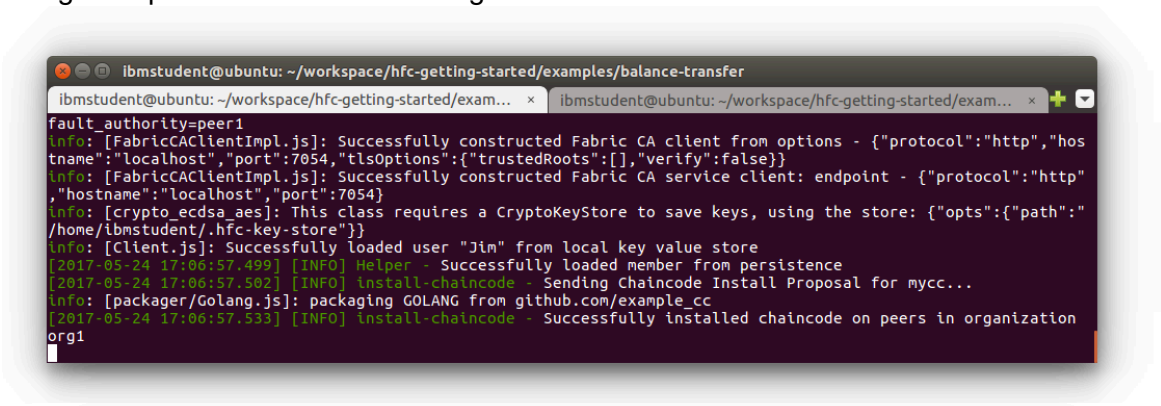


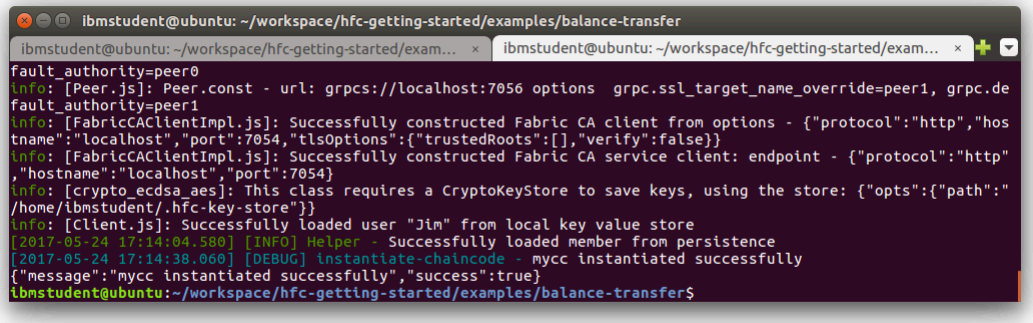
j. Run the Test script

Restart the network by using `./runApp.sh`, wait until the same text shown in 1.1 appears.

When this is the case, switch to the other tab and issue the following: `./testWorkshop.sh`

You should get output similar to the following:





Section 4. Transactions

4.1. Invocations

a. Make a new file

Create a new file in `workshop` called `invoke.js`, insert the following code into it:

```
var helper = require('./helper.js');
var hfc = require('fabric-client');
var logger = helper.getLogger('invoke-chaincode');

const invokeChaincode = function(peers, channelName, chaincodeName,
chaincodeVersion, args, username, org) {
});

exports.invokeChaincode = invokeChaincode;
```

b. Set up the network

Add the following code to the body:

```
    let chain = helper.getChainForOrg(org);
    helper.setupOrderer();
    let targets = helper.getTargets(peers, org);
    helper.setupPeers(chain, peers, targets);
```

c. Get the user context

Add the following under the setup code:

```
return helper.getRegisteredUsers(username, org).then((user) => {
});
```

d. Build the transaction proposal

Add the following under the user context code:

```
nonce = helper.getNonce();
tx_id = chain.buildTransactionID(nonce, user);

let request = {
    targets: targets,
    chaincodeId: chaincodeName,
    fcn: 'invoke',
    args: args,
```

```
    chainId: channelName,
    txId: tx_id,
    nonce: nonce
};

return chain.sendTransactionProposal(request);
```

As discussed in Blockchain Explored, the consensus system in Hyperledger involves a two-stage process whereby transaction proposals are first submitted to the network. On the basis of the response to them they may then be formally issued as transactions. In Fabric SDK you must write code to handle both stages – here you can implement systems to examine the responses yourself to see if they are valid (avoiding proceeding and this being discovered at a later stage).

e. Build the transaction proposal

Add a new then() to the promise chain:

```
}).then((results) => {

// We'll assume it went ok
let proposalResponses = results[0];
let proposal = results[1];
let header = results[2];

let request = {
    proposalResponses: proposalResponses,
    proposal: proposal,
    header: header
};

return chain.sendTransaction(request);
});
```

Assuming the proposal responses are acceptable, send off the transaction request. It should be noted that in instances like the above where no checks are made at this stage the code will still be rejected if it is found that it does not meet the endorsement policy during the checks performed right before committal.

f. Add a new section to testWorkshop.sh

Add a new section to `testWorkshop.sh` after the existing code:

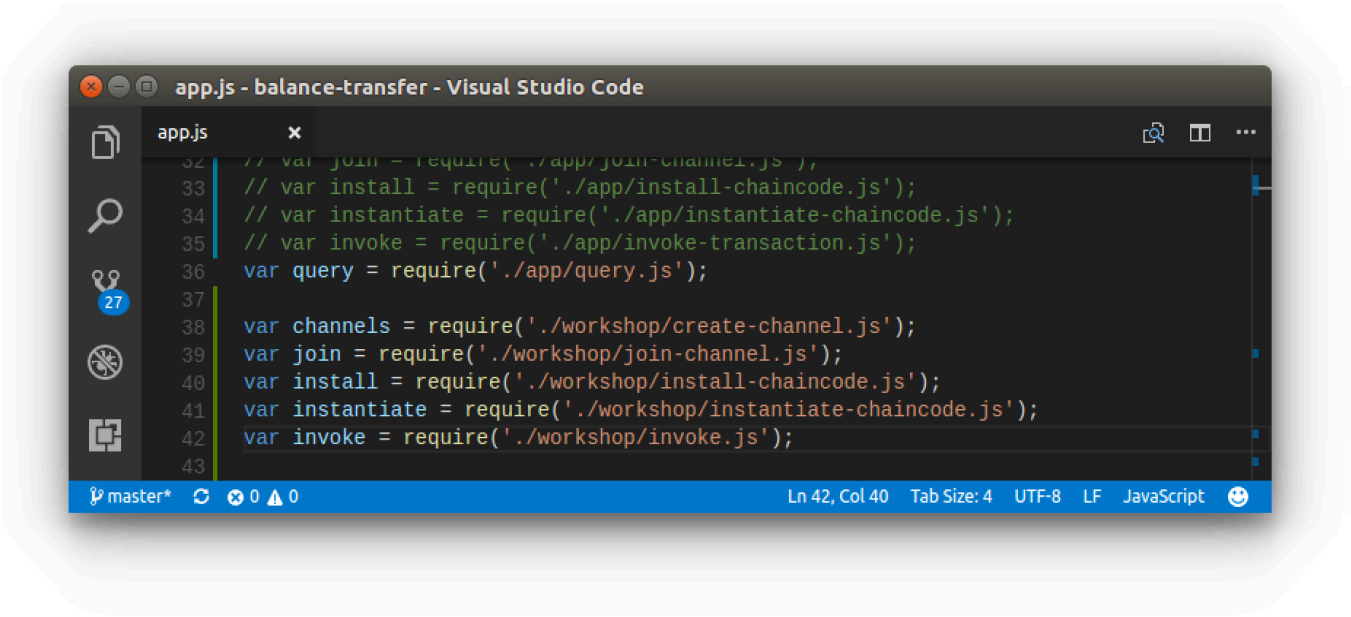
```
echo "POST invoke chaincode on peers of Org1"
echo
TRX_ID=$(curl -s -X POST \
  http://localhost:4000/channels/mychannel/chaincodes/mycc \
  -H "authorization: Bearer $ORG1_TOKEN" \
  -H "cache-control: no-cache" \
  -H "content-type: application/json" \
  -H "x-access-token: $ORG1_TOKEN" \
```

```
  -d '{
      "peers": ["localhost:7051", "localhost:7056"],
      "chaincodeVersion":"v0",
      "functionName":"invoke",
      "args":["move","a","b","10"]
}')
echo "Transacton ID is $TRX_ID"
echo
echo
sleep 10
```

g. Edit app.js

Comment out the import for the `invoke` library and add an entry for `invoke.js`:

```
var invoke = require('./workshop/invoke.js');
```

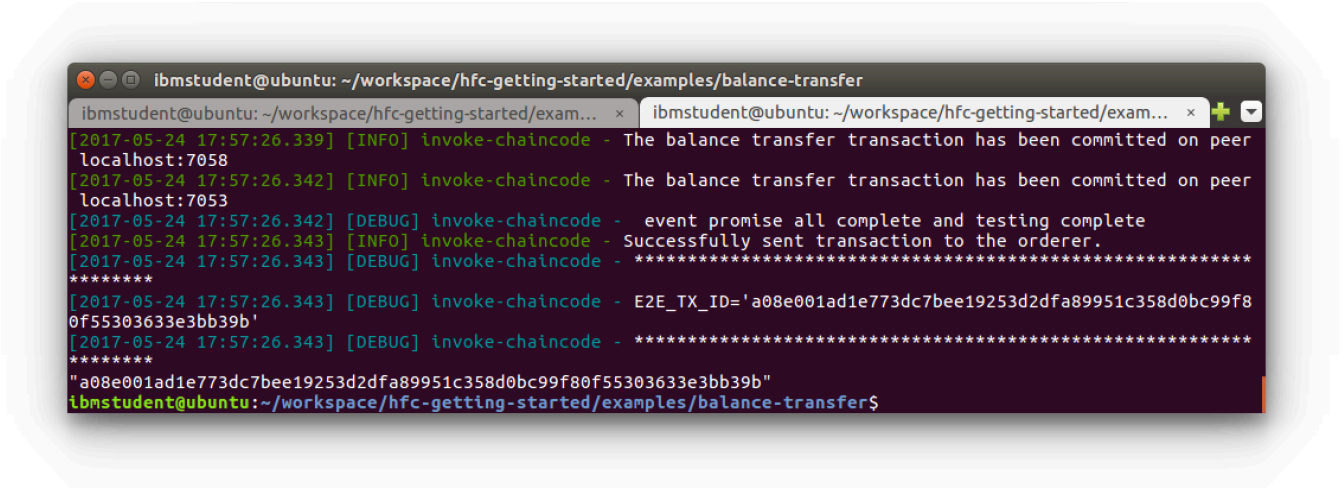


h. Run the Test script

Restart the network by using `./runApp.sh`, wait until the same text shown in 1.1 appears.

When this is the case, switch to the other tab and issue the following: `./testWorkshop.sh`

You should get output similar to the following:



There is no second screenshot as the invoke code does not output or return anything to the main window, it's output is contained to the hash string you see in the above:

4.2. Queries

a. Make a new file

Create a new file in `workshop` called `query.js`, insert the following code into it:

```
var helper = require('./helper.js');

var queryChaincode = function(peer, channelName, chaincodeName, chaincodeVersion,
args, username, org) {
});

exports.queryChaincode = queryChaincode;
```

b. Set up the network

Add the following code to the body:

```
var peers = [];
peers.push(helper.getPeerAddressByName(org, peer));
var chain = helper.getChainForOrg(org);
var targets = helper.getTargets(peers, org);
helper.setupPeers(chain, peers, targets);
```

c. Get the user context

Add the following under the setup code:

```
return helper.getRegisteredUsers(username, org).then((user) => {
});
```

d. Build the transaction proposal

Add the following under the user context code:

```
nonce = helper.getNonce();
tx_id = chain.buildTransactionID(nonce, user);

var request = {
    targets: targets,
    chaincodeId: chaincodeName,
    chaincodeVersion: chaincodeVersion,
    chainId: channelName,
    txId: tx_id,
    nonce: nonce,
    fcn: 'query',
    args: args
};

return chain.queryByChaincode(request);
```

e. Return the output

Add a final `then()` to the promise chain:

```
}).then((response_payloads) => {
      return 'User b now has ' + response_payloads[i].toString('utf8') + ' after the
move';
});
```

As you can see, query functions are incredibly simple compared to the bulk of other transactions as their purpose is simply to read data. Therefore there is little need for any kind of two-stage process or consensus about the activity.

Queries can also be directed at individual peers – as the activity is only reading it only needs to come from a single replica.

Notices

This information was developed for products and services offered in the U.S.A.
IBM may not offer the products, services, or features discussed in this document in other countries.

Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.
IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.
For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan
The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.
This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.
Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.
IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.
Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM

products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.
All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.
This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental. All references to fictitious companies or individuals are used for illustration purposes only.
COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Appendix A. Trademarks and copyrights

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

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

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.
IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.
Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
ITIL is a registered trademark, and a registered community trademark of The Minister for the Cabinet Office, and is registered in the U.S. Patent and Trademark Office.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.
Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

NOTES

NOTES

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. This information is based on current IBM product plans and strategy, which are subject to change by IBM without notice. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

Please Recycle

IBM Software