HYPERLEDGER FABRIC & COMPOSER
MODULE 3
CHAINCODE DEVELOPMENT

Introduction to Chaincode

Topics covered in this module:

- understanding of what a chaincode is
- namespaces and the concept of composite key
- transaction context
- write a basic chaincode for the 'certification-network'
- deploying the chaincode on the network
- development and production mode
- automation scripts for network setup and chaincode development

What is a Chaincode?

A chaincode is a series of logic that is put inside the application. A chaincode is a collection of one or more *Smart Contracts.* Chaincode is installed on peers that acts as endorsers on the network. Applications can issue transaction proposals on these peers which will then be endorsed and put on the blockchain by the ordering service.

The Certification Network that was build had three organizations, IIT, upGrad and MHRD. The chaincode logic for this network is as follows:

1. Creating a student account
2. Issue certificate
3. Verify certificate
4. View certificates

HYPERLEDGER FABRIC & COMPOSER
MODULE 3
CHAINCODE DEVELOPMENT

Namespaces and the Transaction Context:

Namespaces as a general concept allows for grouping of data across different classes. In our use case, we store the assets in the world state on the blockchain of the peers. In all assets, the key would be *studentId*. We also will have another key for the certificate which would be the *certificateId*. Here, the range of values the keys can take for both of these would be in conflict. Hence, there is a need to differentiate between these types of assets even though the network sees them as the same. This is solved by using the concept of namespaces. This in Fabric is achieved using a *Composite Key.*

Composite Key:

Key is the unique identity of any asset parameter for the student or for the certificate. A composite key is a group of multiple keys separated by a unique separator which could be a colon (:) or anything else. This allows for a better structure to define the student key such as *<type student>:<studentID>* and certificate key such as *<type certificate>:<certificateID>*. This avoids the conflict between the keys for students and certificates. The composite key can be of any length and can have any number of parts. This also helps to build complex search queries in a way where the composite keys can have the search criteria built into them.

1. Composite Keys are formed by combining a group of words using separators like dots, hyphens or colons. For example, “org.certification-network.certnet.student.0001” can be the composite key for a student whose data is to be stored in the ledger.

2. Composite Keys can be created using namespaces. In “org.certification-network.certnet.student.0001”, “org.certification-network.certnet.student” is the namespace and “0001” is the student ID.

3. Composite Keys can be of any length.

Transaction context:

In the chaincode for this network, the function to create student will add an asset to the database. This data will be stored using a namespace. This data will be stored on the blockchain through an incoming transaction triggered by a client application. This incoming transaction will have a certain context to it. Context is the information about the issuer of the transaction, the channel, the timestamp and other metadata about the transaction itself. Access to this will be useful within the function which this transaction triggers. The data that the asset value would capture can be obtained from this transaction context. For example, if we want to capture the school which issues the certificate, it can be obtained from the sender of this transaction since it is the school which will trigger the issue certificate transaction.

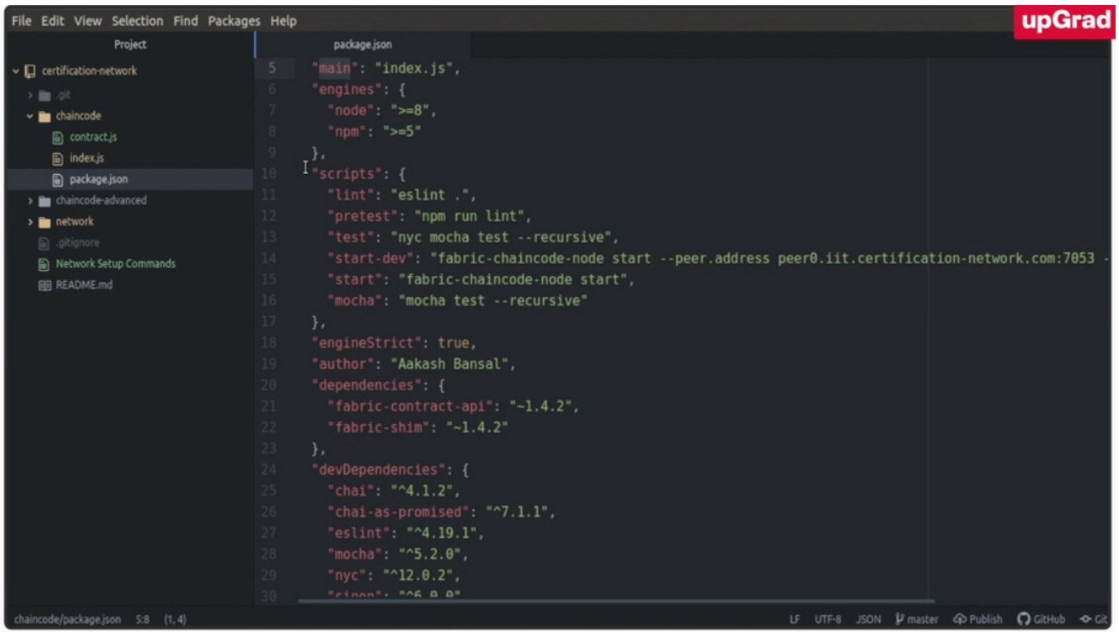
File Structure:

The logic of creating a student account and recording this as an asset on the blockchain will be implemented as a chaincode written as a node.js application. This node.js application will be deployed on the blockchain. This application has a package.json file which will have information about this application, its starting file, the dependencies and the description etc. This application also has index.js file which will be the main file. In this file, one or more smart contracts will be imported. A chaincode is a collection of one or more independent smart contracts which will be written as modules inside the node.js file.

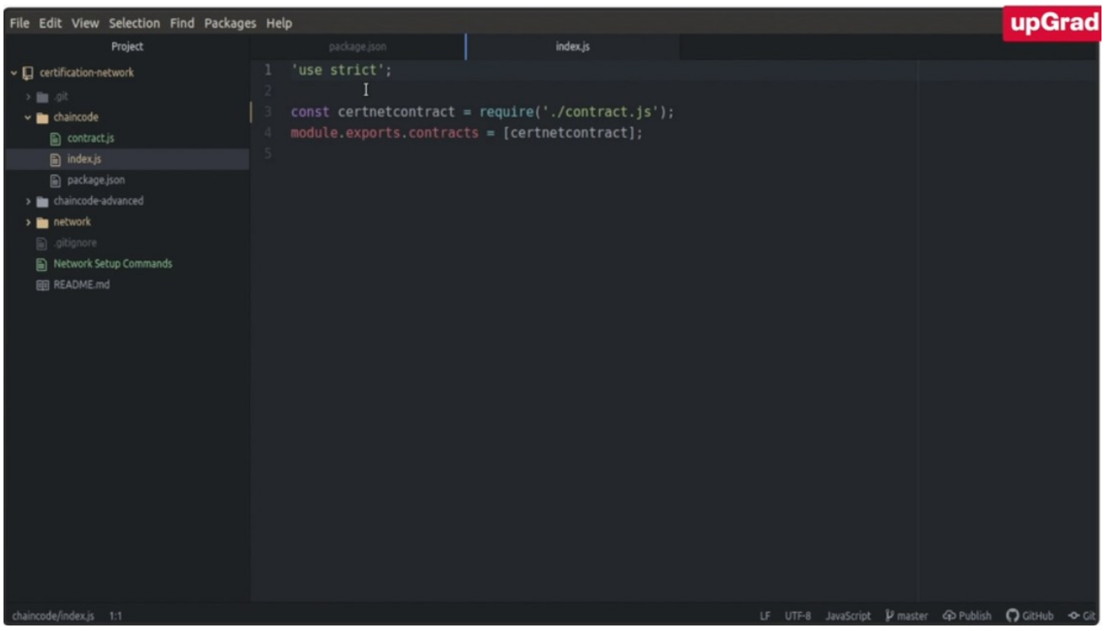
The chaincode will be written and stored in the same project directory as that of the network. A separate folder named *chaincode* will be created where the node.js application files will be stored. The package.json file will have name (certnet), version, description, main file (contract.js), other packages and other dependencies required to run the smart contract. The most important dependencies to include will be the *fabric-contract-api and fabric-shim.*

The index.js will be the main module of this application and it will import the smart contract modules and export all of them inside the *contracts* object of *module.exports*. This will make the contract available for the peer during deployment.

HYPERLEDGER FABRIC & COMPOSER
MODULE 3
CHAINCODE DEVELOPMENT



```
  "main": "index.js",
  "engines": {
    "node": ">=8",
    "npm": ">=5"
  },
  "scripts": {
    "lint": "eslint .",
    "pretest": "npm run lint",
    "test": "nyc mocha test --recursive",
    "start-dev": "fabric-chaincode-node start --peer.address peer0.iit.certification-network.com:7053 -
    "start": "fabric-chaincode-node start",
    "mocha": "mocha test --recursive"
  },
  "engineStrict": true,
  "author": "Aakash Bansal",
  "dependencies": {
    "fabric-contract-api": "~1.4.2",
    "fabric-shim": "~1.4.2"
  },
  "devDependencies": {
    "chai": "^4.1.2",
    "chai-as-promised": "^7.1.1",
    "eslint": "^4.19.1",
    "mocha": "^5.2.0",
    "nyc": "^12.0.2",
```



```
'use strict';

const certnetcontract = require('./contract.js');
module.exports.contracts = [certnetcontract];
```

Structure of contract.js file:

The smart contract logic will be written in the contract.js file. Following are the steps:

- importing:

The contract starts with importing the fabric contract API. The API is a set of functions which enable the contract application to interact with the ledgers on the peers of the Fabric network. As we know, each peer has its own ledger. The contract is started as an independent Docker container and this needs to interact with these peers to access the ledgers. This interaction is enabled using this API. There are other functions in this API which will allow to control the smart contracts on the channel. Fabric contract is a high level API which is available on top of low level fabric shim. The fabric shim runs inside the docker containers and carries out the functionalities using binary streams of data. Since the fabric shim is a low level API, it is difficult to implement. Hence the still evolving high level fabric contract API enables development of complex smart contracts.

- class:

The next step is to create base class which would extend the Contract class which is exposed by the fabric contract API.

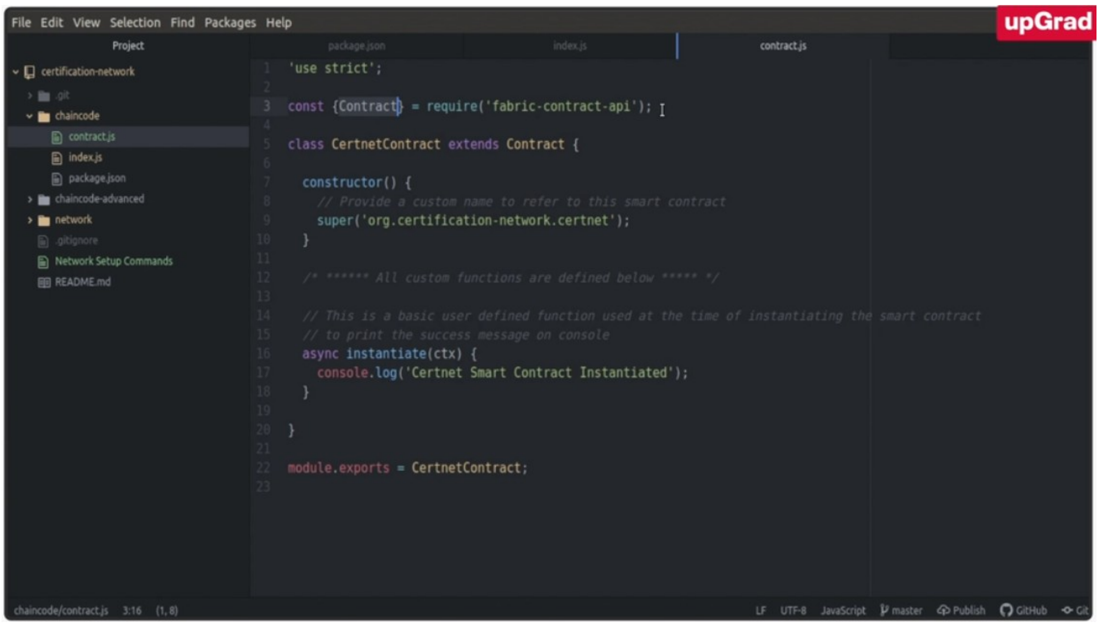
- define a constructor:

A constructor will be created inside this class which will define the base namespace for this contract. This namespace differentiates the various smart contracts within a chaincode. The constructor should call the constructor of the parent class and pass the namespace of the smart contract as the parameter.

- writing the functions:

The contract logic can be written inside any number of async functions. Every transaction triggered will have a context (ctx) which also has the fabric-shim API's inside this context. All these API's are group together in a object called the *stub*. This stub object is added to the context which is the first parameter of any function call. All other function arguments will be the user input which triggers this function call. The *ctx.stub* method will have low level *shim* API's.

async *fxName*(ctx, ---, ----, ----)



```
'use strict';

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

class CertnetContract extends Contract {

  constructor() {
    // Provide a custom name to refer to this smart contract
    super('org.certification-network.certnet');
  }

  /* ****** All custom functions are defined below ***** */

  // This is a basic user defined function used at the time of instantiating the smart contract
  // to print the success message on console
  async instantiate(ctx) {
    console.log('Certnet Smart Contract Instantiated');
  }

}

module.exports = CertnetContract;
```

<u>Writing a Basic Chaincode:</u>

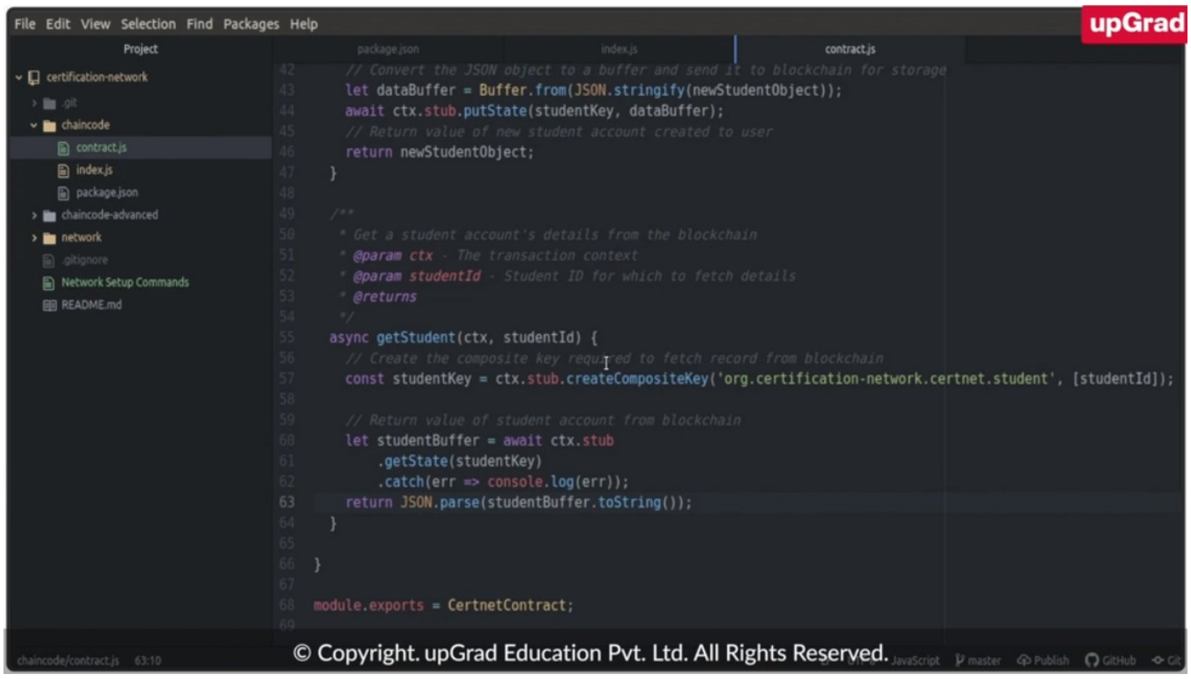
Following is the definition of the *createStudent()* function which allows to create a new student asset and store it on the blockchain.

- it is an async function which will capture three parameters, *studentId, name and email*
- create a composite key using the *createCompositeKey* method available as a part of chaincode stub functions through the fabric shim API. This will take two parameters. First is the object name which is the base name of the composite key. The second is the array of different key parts which would be linked together
- create a json object to store the values. This will have various parameters such as *studentID, name, email, school etc.* School will be the identity of the initiator of this transaction. This can be obtained from the *clientIdentity* method of transaction context (ctx) available from the API
- store the json data along with the composite key on the ledgers of the peers using available API methods. Even though the chaincode works with data stored in json objects, the same data needs to be transferred to/from the ledgers as binary stream of data or a buffer. This involves conversion between json and buffer stream. This is achieved using the *stringify() and Buffer.from* methods

- use the *putState* method from the stub functions to record the data on the ledger. This method takes in two parameters which are the composite key and the buffer. This function will update the ledger and will return a promise
- return the student object as a response to this function call

Following is the definition of the getStudent*()* function which allows to fetch the stored data from the blockchain. We will use the student key to fetch the student details.

- this is also an async function with one parameter which is the *studentID*
- create a composite key using the *createCompositeKey* as done in the prior function
- use the *getState* stub method to fetch data from the blockchain. This method takes in the composite key and returns a buffer when the promise is resolved
- convert this buffer using the JSON.parse function
- return this student detail as a response to the function call



```
    // Convert the JSON object to a buffer and send it to blockchain for storage
    let dataBuffer = Buffer.from(JSON.stringify(newStudentObject));
    await ctx.stub.putState(studentKey, dataBuffer);
    // Return value of new student account created to user
    return newStudentObject;
  }

  /**
   * Get a student account's details from the blockchain
   * @param ctx - The transaction context
   * @param studentId - Student ID for which to fetch details
   * @returns
   */
  async getStudent(ctx, studentId) {
    // Create the composite key required to fetch record from blockchain
    const studentKey = ctx.stub.createCompositeKey('org.certification-network.certnet.student', [studentId]);

    // Return value of student account from blockchain
    let studentBuffer = await ctx.stub
        .getState(studentKey)
        .catch(err => console.log(err));
    return JSON.parse(studentBuffer.toString());
  }

}

module.exports = CertnetContract;
```

Chaincode Deployment – Development Mode

Commands to deploy a chaincode:

The certification network that we build has 3 organizations (IIT, upGrad, MHRD) with each of them having 2 peers each. There is a channel which all these peers have joined and also an orderer and a CLI node using which we have set up the channel and defined the anchor peers for the organizations. Following are the steps to deploy the chaincode on this network:

1. Install:

Installation is a process where the chaincode node.js application will be installed on the peers. The installation will be done on at least one of the peers in each organization. This is equivalent of copying the application to the containers of the peers. The peer on which the chaincode is installed will make them the endorsers on the network.

2. Instantiate:

Instantiation will make the chaincode available on the channel so that any other application can interact with the chaincode through the channel.

The CLI container will be used to install and instantiate the chaincode. Use the environment variables to define the peer on which the commands from the CLI needs to be run. Since the install command will need to be run on three different peers, the CLI needs to be connected to each of these peers one by one. However, instantiation needs to happen only once on the channel but it takes place through one of the peers on the network. The instantiation will run only if there is a copy of the chaincode on that peer.

Once the chaincode has been instantiated, it will create a new chaincode container for that particular peer. Any function invoked from the chaincode will run on this container. If any function is invoked from a different peer, it will automatically start another chaincode container for that peer. Since these containers are different, they can isolate the business logic from the ledger of the peers.

In the process of developing an application, this process of installation and instantiation is quite tedious. Hence, Fabric provides a Dev mode to enable easier and faster development cycles.

HYPERLEDGER FABRIC & COMPOSER
MODULE 3
CHAINCODE DEVELOPMENT

<u>Dev Mode:</u>

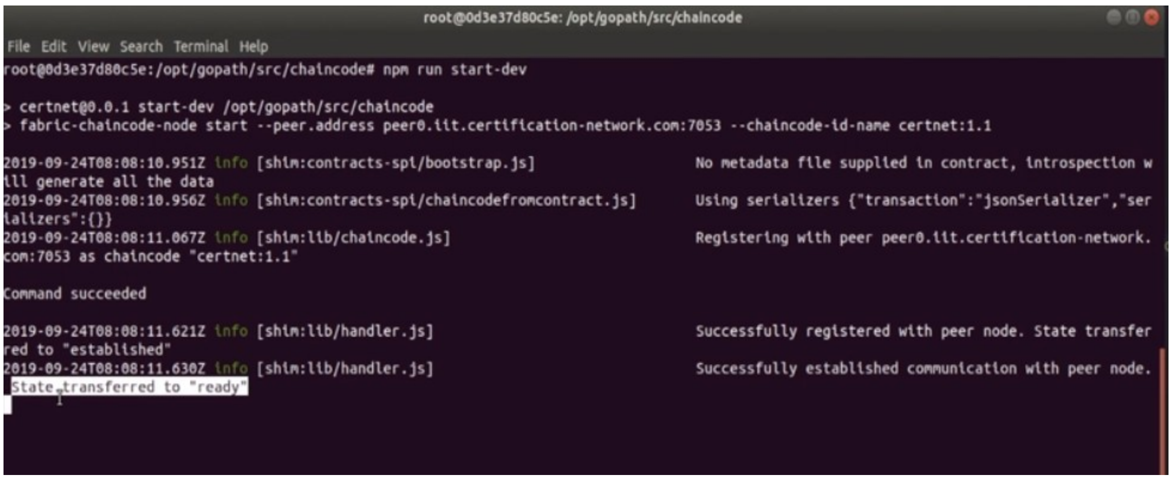
To operate in Dev mode, some changes are required on both the network and also in the chaincode. On the network, all the peers must run in dev mode. Also, the TLS must be turned off. In Dev mode, a separate standalone container called the chaincode container will be used. This container is similar to a CLI container which runs the node.js smart contract application. This chaincode container will be used by the peers and will act as the original chaincode containers that will have been started by the peers. This chaincode container will be controlled by the user and not any other peer. To update any changes on the contract, the node.js application must be restarted and automatically, all the updates will be reflected on the peers.

The network configuration settings defined in the chaincode container is already set to point to the IIT organization. There is also a startup script (start-dev) to start the node.js application inside the chaincode container and connect it to one of the peers in the network. To enable Dev mode, update the command to start the peers in the docker-compose-peer.yml file with the following command:

```
peer node start –peer-chaincodedev=true
```

<u>Chaincode Installation:</u>

SSH into the CLI container and use the steps given in the file to run the commands to install the chaincode. Also, run the start-dev script on the chaincode container. You will see a message “State transferred to ‘ready’” message.

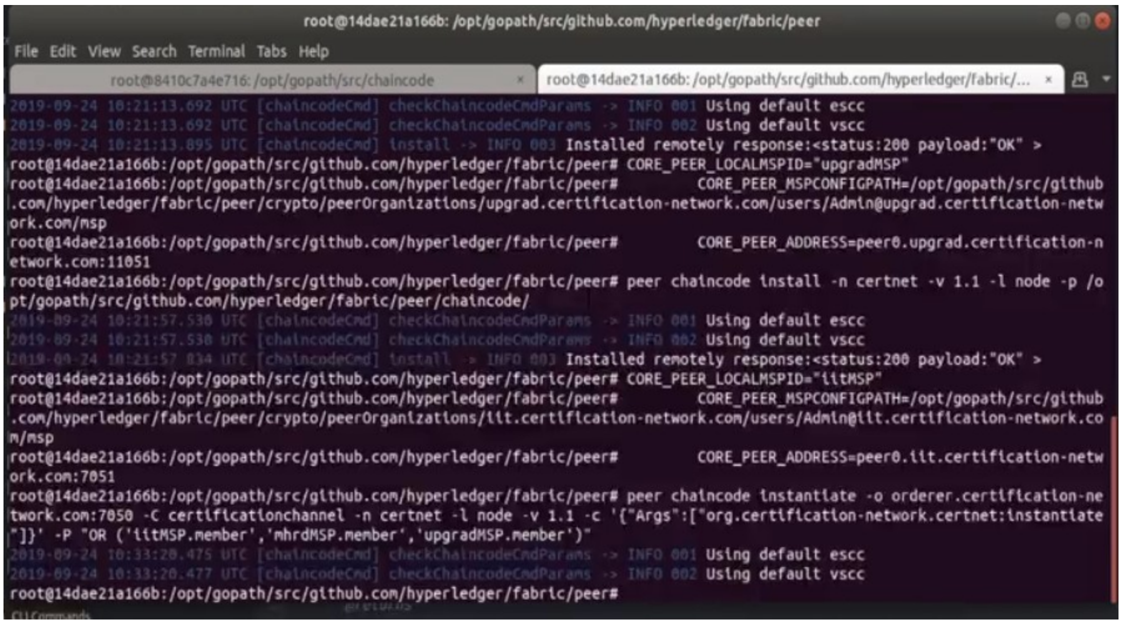

root@0d3e37d80c5e: /opt/gopath/src/chaincode
File Edit View Search Terminal Help

```
root@0d3e37d80c5e:/opt/gopath/src/chaincode# npm run start-dev

> certnet@0.0.1 start-dev /opt/gopath/src/chaincode
> fabric-chaincode-node start --peer.address peer0.iit.certification-network.com:7053 --chaincode-id-name certnet:1.1

2019-09-24T08:08:10.951Z info [shim:contracts-spi/bootstrap.js]              No metadata file supplied in contract, introspection w
ill generate all the data
2019-09-24T08:08:10.956Z info [shim:contracts-spi/chaincodefromcontract.js]  Using serializers {"transaction":"jsonSerializer","ser
ializers":{}}
2019-09-24T08:08:11.067Z info [shim:lib/chaincode.js]                        Registering with peer peer0.iit.certification-network.
com:7053 as chaincode "certnet:1.1"

Command succeeded

2019-09-24T08:08:11.621Z info [shim:lib/handler.js]                          Successfully registered with peer node. State transfer
red to "established"
2019-09-24T08:08:11.630Z info [shim:lib/handler.js]                          Successfully established communication with peer node.
State transferred to "ready"
```

Instantiation of chaincode:

To instantiate the chaincode, it must first be installed on that peer. Follow the steps in the additional documentation provided on the portal to update the environment variables and instantiate the chaincode.

- Instantiation will first execute the constructor function in the contract. Hence it should contain the function to initialize the chaincode
- This command must also know the Endorsement Policy. Hence this command must define the policy using conditional statement
- The command has an arguments object which will call the constructor function. Here we will define the *instantiate* function defined in the contract. This will be the name space of the contract: function name
- Use the peer logs using the command *$docker logs* to ensure the chaincode has been instantiated properly. The chaincode container will also show the output that has been printed by the instantiate function.


root@14dae21a166b: /opt/gopath/src/github.com/hyperledger/fabric/peer
File Edit View Search Terminal Tabs Help
root@8410c7a4e716: /opt/gopath/src/chaincode
root@14dae21a166b: /opt/gopath/src/github.com/hyperledger/fabric/...

```
2019-09-24 10:21:13.692 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using default escc
2019-09-24 10:21:13.692 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using default vscc
2019-09-24 10:21:13.895 UTC [chaincodeCmd] install -> INFO 003 Installed remotely response:<status:200 payload:"OK" >
root@14dae21a166b:/opt/gopath/src/github.com/hyperledger/fabric/peer# CORE_PEER_LOCALMSPID="upgradMSP"
root@14dae21a166b:/opt/gopath/src/github.com/hyperledger/fabric/peer#        CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github
.com/hyperledger/fabric/peer/crypto/peerOrganizations/upgrad.certification-network.com/users/Admin@upgrad.certification-netw
ork.com/msp
root@14dae21a166b:/opt/gopath/src/github.com/hyperledger/fabric/peer#        CORE_PEER_ADDRESS=peer0.upgrad.certification-n
etwork.com:11051
root@14dae21a166b:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode install -n certnet -v 1.1 -l node -p /o
pt/gopath/src/github.com/hyperledger/fabric/peer/chaincode/
2019-09-24 10:21:57.530 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using default escc
2019-09-24 10:21:57.530 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using default vscc
2019-09-24 10:21:57.834 UTC [chaincodeCmd] install -> INFO 003 Installed remotely response:<status:200 payload:"OK" >
root@14dae21a166b:/opt/gopath/src/github.com/hyperledger/fabric/peer# CORE_PEER_LOCALMSPID="iitMSP"
root@14dae21a166b:/opt/gopath/src/github.com/hyperledger/fabric/peer#        CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github
.com/hyperledger/fabric/peer/crypto/peerOrganizations/iit.certification-network.com/users/Admin@iit.certification-network.co
m/msp
root@14dae21a166b:/opt/gopath/src/github.com/hyperledger/fabric/peer#        CORE_PEER_ADDRESS=peer0.iit.certification-netw
ork.com:7051
root@14dae21a166b:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode instantiate -o orderer.certification-ne
twork.com:7050 -C certificationchannel -n certnet -l node -v 1.1 -c '{"Args":["org.certification-network.certnet:instantiate
"]}' -P "OR ('iitMSP.member','mhrdMSP.member','upgradMSP.member')"
2019-09-24 10:33:20.475 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using default escc
2019-09-24 10:33:20.477 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using default vscc
root@14dae21a166b:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

Invoking the functions:

Instead of connecting to the chaincode from a client application we will test out the chaincode, by directly connecting to a peer and invoking the functions. The chaincode container which is already running the node.js application is now registered to the peer0 of the IIT organization.

SSH into the peer0 of the IIT organization and follow the invoke commands given as a part of the separate documentation on the portal.

The peer logs will show that a new block has been created since the ledger was updated with a new transaction initiated by the functions.

Summary:

Following are the summary of the steps to deploy and invoke the chaincode.

- Ensured that the network is configured for Dev mode
- Restarted the network
- Log in to the chaincode container and start the chaincode
- Install the chaincode on peer0 of each organization
- Instantiate the chaincode from peer0 of iit organization
- Invoke transactions from peer0 of iit organization to test createStudent() and getStudent() functions in the 'certnet' smart contract

‘Certnet’ Smart Contract: Complete Solution

Function - ‘Issue Certificate’:

The first two functions in this smart contract were *createStudent()* and *getStudent()*. These two served the purpose of enrolling students and viewing their details. The next functionality required is for the institute to be able to issue certificates to the students who are able to complete the courses. For this, a function named *issueCertificate()* will be written which will create an asset to store the certificate details. The students will then be able to present this certificate to the prospective employers who will then need the ability to verify the authenticity of these certificates. This verification is enabled by using hash. Hashing process will take in the original certificate details and outputs a string of unique hash data. This is the data that will be saved on the ledgers as a part of the certificate issuance process. The verification process will check if the hash data presented matches with that which is stored on the blockchain. If there is a match, then the certificate is valid.

The *issueCertificate()* function is also an async function which will take in parameters such as *studentId, courseId, gradeReceived, originalHash.*

- capture the issuer of the certificate
  The issuer of this certificate will be the organization which triggers this transactions. Hence, the ID of the caller is obtained using the *clientIdentity* method which is available as a part of the stub API.
- create composite keys
  Create the *certificateKey* and the *studentKey.* We use appropriate namespace to store the key for the certificate to differentiate this from the other keys. The student key will be used to fetch the details of the student from the blockchain.
- check if the student details are already available
  Fetch the details of the student from the blockchain using the *getState* function and check whether it contains any data. This will ensure that the details of the student that is provided as an input this function actually exists on records.
- check if the certificate for this student already exists
  Compare the student and certificate details to ensure that the student exists but the certificate does not exists.
- create data buffer and record this on the ledger
  Use the *stringify* method to covert the json object into data buffer and use the *putState()* method to record this data on the ledger.

HYPERLEDGER FABRIC & COMPOSER
MODULE 3
CHAINCODE DEVELOPMENT

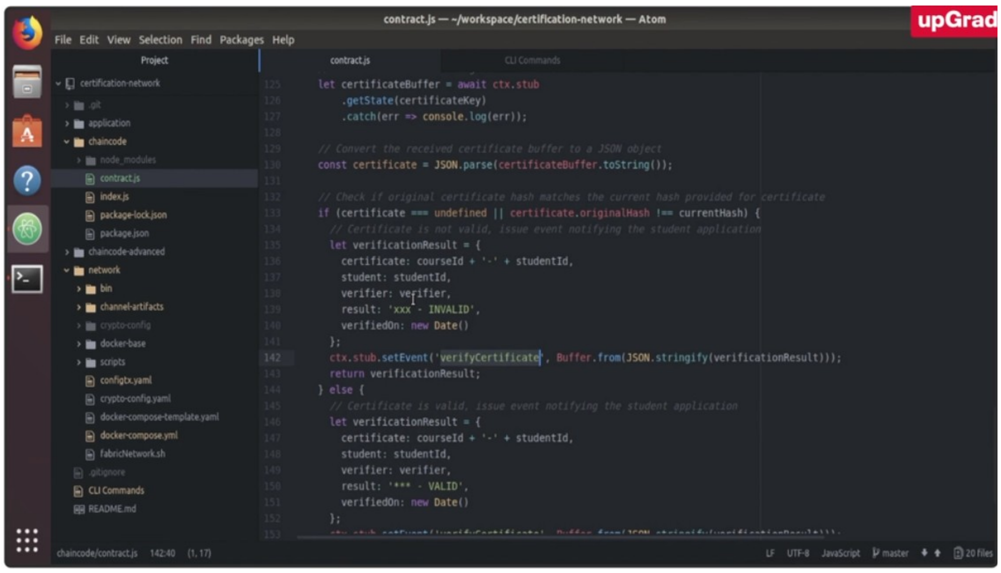
<u>Function - ‘Verify Certificate’:</u>

Verify certificate compares the current hash that is provided as a part of the function to the original hash of the certificate that is stored on the blockchain. If these two hashes match, the certificate is valid.

- capture the identity
  Capture the ID of the caller of this transaction who will be the verifier of the certificate.
- create keys
  Create the certificate key using the same parameters used before while storing the certificate.
- confirm if the certificate exists
  Fetch the certificate asset from the blockchain using the *getState()* method using the stub API. Covert the stream of data into json object using the JSON.parse method.
- compare the hash of the certificates
  Compare the hash data from the ledger and the input to this function to see if they match.

Events

Events are unidirectional updates from server to the client. Clients will have listeners which will process the even triggers as they get issued. In this case, the success of the verification process will be sent to the client application using events. The *setEvent* method available in the stub API will be used to trigger this event. Here, a unique name to identify this event must be used.



```
let certificateBuffer = await ctx.stub
    .getState(certificateKey)
    .catch(err => console.log(err));

// Convert the received certificate buffer to a JSON object
const certificate = JSON.parse(certificateBuffer.toString());

// Check if original certificate hash matches the current hash provided for certificate
if (certificate === undefined || certificate.originalHash !== currentHash) {
  // Certificate is not valid, issue event notifying the student application
  let verificationResult = {
    certificate: courseId + '-' + studentId,
    student: studentId,
    verifier: verifier,
    result: 'xxx - INVALID',
    verifiedOn: new Date()
  };
  ctx.stub.setEvent('verifyCertificate', Buffer.from(JSON.stringify(verificationResult)));
  return verificationResult;
} else {
  // Certificate is valid, issue event notifying the student application
  let verificationResult = {
    certificate: courseId + '-' + studentId,
    student: studentId,
    verifier: verifier,
    result: '*** - VALID',
    verifiedOn: new Date()
  };
  ctx.stub.setEvent('verifyCertificate', Buffer.from(JSON.stringify(verificationResult)));
```

contract.js — ~/workspace/certification-network — Atom
upGrad
File Edit View Selection Find Packages Help
Project
contract.js
CLI Commands
chaincode/contract.js 142:40 (1, 17)
LF UTF-8 JavaScript master 20 files

Testing the chaincode in Dev Mode:

In order to update the chaincode when running in Dev mode, use ctrl+c to stop the node.js application in the chaincode container. Restart the application using *npm run start-dev* command. Invoke the functions from the peer. Follow the steps in the additional documentation on the portal to invoke the functions.

1. Command to invoke an issueCertificate transaction:

peer chaincode invoke -o orderer.certification-network.com:7050 -C certificationchannel -n certnet -c '{"Args":["org.certification-network.certnet:issueCertificate","0001", "PGDBC", "A", "asdfgh"]}'

2. Command to invoke a verifyCertificate transaction:

peer chaincode invoke -o orderer.certification-network.com:7050 -C certificationchannel -n certnet -c '{"Args":["org.certification-network.certnet:verifyCertificate","0001", "PGDBC", "asdfgh"]}'

Production Mode:

In the production mode, the chaincode is started and maintained by the peer. This will allow to invoke transactions from any peer that has the chaincode installed on them.

1. It is compulsory to comment the '--peer-chaincode-dev=true' flag from the command to start the peer. This command is specified inside the 'peer-base' service in the docker-compose-peer.yaml file.
2. Removing the chaincode service from the docker-compose.yml file is not compulsory. You may allow the 'chaincode' container to continue existing in your docker network.
3. It is not compulsory to enable TLS in production mode.

Start the network again after cleanly shutting it down and removing the orphan volumes and containers. Now, the network is ready to install and instantiate the chaincode in production mode.

HYPERLEDGER FABRIC & COMPOSER
MODULE 3
CHAINCODE DEVELOPMENT

Network Automation

This session covers:

The basics of Bash scripting
The automation of network setup
The automation of chaincode installation and instantiation

Need for automation:

During any new network and chaincode development, we have to redo several steps such as generating the crypto materials, creating the channel artifacts, start all the Docker services, create the channel, make the peers join the channel and update the anchor peers for each organization. We also have to install and instantiate the chaincode on the peers and the channel. These steps are complex and very time consuming involving at least 20 commands for the certification network. Hence the solution to automating these steps using shell scripts. This will bring down the number of commands to just one for starting the network and one more to install and instantiate the chaincode.

Network Setup Automation – I:

The shell scripts are already a part of the Fabric Samples directory available upon installation of the Hyperledger Fabric. It is only required to understand the different parameters inside these script files so that it can be customized to suit the network being built.

The script is run using the command *./fabricNetwork.sh* followed by the command or the function which needs to be run from within the script file. This also has various command options.

printHelp() - helper function to list down the commands and options
askProceed() - confirm with the user whether it needs to be run or not
clearContainers() - helper function to clear the docker network of any containers when it is brought down
removeUnwantedImages() - removes any unwanted Docker images
checkPrereqs() - checks whether all available prerequisites and binaries are available on the local computer to run all the commands required

networkUp() - this is the main function to start the network. This performs three main tasks: generate the crypto materials, replace the private keys and generate the channel artifacts
generateCerts() - running the crytogen generate command to generate the required certificates
replacePrivateKey() - this function copies the docker compose template to a new file named docker-compose.yaml and replace the keywords with the actual file path of the cryto material that have been generated
generateChannelArtifacts() - runs the configtxgen tool to create the genesis block, channel creation artifacts and create the anchor peers creation files

After the Docker containers are started, the script will wait for 10 seconds to allow all the containers to finish started and be ready for the next set of commands. The next set of commands are run from *bootstrap.sh* inside the *scripts* folder. These commands are run within the CLI container.

createChannel() - executes command to create the channel. The *setGlobals* function within the *utils* script sets the environment variables correctly This script is imported into the bootstrap script
joinChannel() - executes command to make each of the six peers to join the channel. It iterates through all of the peers in each organization and executes the join command
updateAnchorPeers() - define the anchor peers for each organization

<u>Network Setup Automation – II:</u>

The script files use the input from the user to define the mode in which the network has to be run. The ‘up’ mode will run the *networkUp* function while the ‘down’ mode will run the *networkDown* function. The ‘generate’ mode will just create the crypto material and the channel artifacts which completes the pre-setup phase. There are also other modes such as ‘restart’, ‘retry’.

1. up: This command will do the following:

a. Generate crypto materials,
b. Generate channel artifacts, and
c. Start the Docker network

./fabricNetwork.sh up

2. down: This command will do the following:

a. Delete crypto materials,
b. Delete channel artifacts,
c. Stop the Docker network, and
d. Kill all the Docker containers

./fabricNetwork.sh down

3. generate: This command will only generate the crypto materials and channel artifact files.

./fabricNetwork.sh generate

Automation of Chaincode Installation:

The ./fabricNetwork.sh script has a *install* command which will run the *installChaincode()* function. This will first run the *checkPrereqs* function to check if all the binaries and all the prerequisites are available. Next, this function will run the *installChaincode* script available within the scripts folder. This script will be run within the CLI container using the 'docker exec' command. The following functions are defined inside the *installChaincode.sh* script file:

- calls the installChaincode() which is available within the *utils* script file
- calls the instantiateChaincode() which is also available within the *utils* script file

The number of the peer, the organization and the version of the chaincode is passed as parameters to these function calls. This will use the *setGlobals* function to correctly use the environment variables to point to the correct peer from within the CLI container.

The 'update' function allows the changes made on the chaincode to be updated on the network. This will run the *updateChaicode.sh* script file available within the scripts folder. This will install the chaincode with a new version of the chaincode. It also runs the *upgradeChaincode* function to upgrade the chaincode to the next new version.

./fabricNetwork.sh install