



22CY602 CRYPTOCURRENCY AND BLOCKCHAIN TECHNOLOGIES (Lab Integrated)

LAB RECORD

**VI SEMESTER R-2022
ACADEMIC YEAR 2024-2025 EVEN**



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(CYBER SECURITY)**



R.M.K. COLLEGE OF ENGINEERING AND TECHNOLOGY
(An Autonomous Institution)

Approved by AICTE, New Delhi/ Affiliated to Anna University, Chennai
Accredited by NBA (All Eligible Courses) / NAAC with "A" GRADE
An ISO 21001:2018 Certified Institution



R.S.M. Nagar, Puduvoyal - 601 206. Gummidipoondi Tk., Tiruvallur Dist. Tamilnadan, India

Department: Computer Science and Engineering (cyber security)

Laboratory: 22CY602 -Cryptocurrency and Blockchain
Technologies (Lab Integrated)

Semester : VI

Certified that this is a bonafide record done by.....

With Roll/Reg.NumberHe/She is a Student
of Computer Science and Engineering (Cyber Security) in the R.M.K College of Engineering
and Technology, Puduvoyal.

Faculty-in-charge

Head of the Department

Internal Examiner

Date:

External Examiner

INDEX

Ex. No.	Date	Name of the exercise	Page No.	Marks obtained	Signature of the faculty
1		Installation of Docker Container, Node.js and Hyperledger Fabric and Ethereum Network.	1		
2		Understand the basics of Bitcoin transactions, mining, and wallet management.	11		
3		Implement ECC key generation algorithm using a programming language of your choice	14		
4		Explore Web3j, a Java library for working with Ethereum.	19		
5		Interact with a blockchain network. Execute transactions and requests against a blockchain network by creating an app to test the network and its rules	21		
6		Create and deploy a blockchain network using Hyperledger Fabric SDK for Java	33		
7		Deploy an asset-transfer app using blockchain within Hyperledger Fabric network	41		
8		Fitness Club rewards using Hyperledger Fabric	53		
9		Create and deploy a simple smart contract on the Kadena blockchain.	56		
10		Analyze the transaction ledger to understand how Ripple processes and records transactions	58		
11		Execute transactions on the private contract and observe the privacy features of Quorum.	60		

Ex:No:1 Installation of Docker Container,Node.js and HyperledgerFabric and Ethereum Network.
Date:

Aim:

To Install and understand Docker container, Node.js, Java and Hyperledger Fabric, Ethereum and perform necessary software installation on local machine/create instance on cloud to run.

Docker:

Docker is a containerization platform which is used for packaging an application and its dependencies together within a Docker container. This ensures the effortless and smooth functioning of our application irrespective of the changes in the environment.

Talking about Docker Container it is nothing but a standardized unit that is used to deploy a particular application or environment and can be built dynamically. We can have any container such as Ubuntu, CentOS, etc. based on your requirement with respect to Operating Systems.

Dockerfile

A Dockerfile is basically a text document that contains the list of commands which can be invoked by a user using the command line in order to assemble an image. Thus, by reading instructions from this Dockerfile, Docker automatically builds images.

For executing multiple command line instructions successively, you can create an automated build using the following command:

`docker build`

Docker Image

A Docker Image can be considered something similar to a template which is typically used to build Docker Containers. In other words, these read-only templates are nothing but the building blocks of a Docker Container. In order to execute an image and build a container you need to use the following command:

`docker run`

The Docker Images that you create using this command are stored within the Docker Registry. It can be either a user's local repository or a public repository like a Docker Hub which allows multiple users to collaborate in building an application.

Docker Container

A Docker Container is the running instance of a Docker Image. These containers hold the complete package that is required to execute the application. So, these are basically ready to use applications which are created from Docker Images that is the ultimate utility of Docker.

I guess, now you are quite familiar with Docker. If you want to learn more about Docker you can refer to our other blogs on Docker.

In order to Dockerize a Node.js application, you need to go through the following steps:

Create Node.js Application

Create a Docker file

Build Docker Image

Execute

Create Node.js Application

Building REST API using Node.js

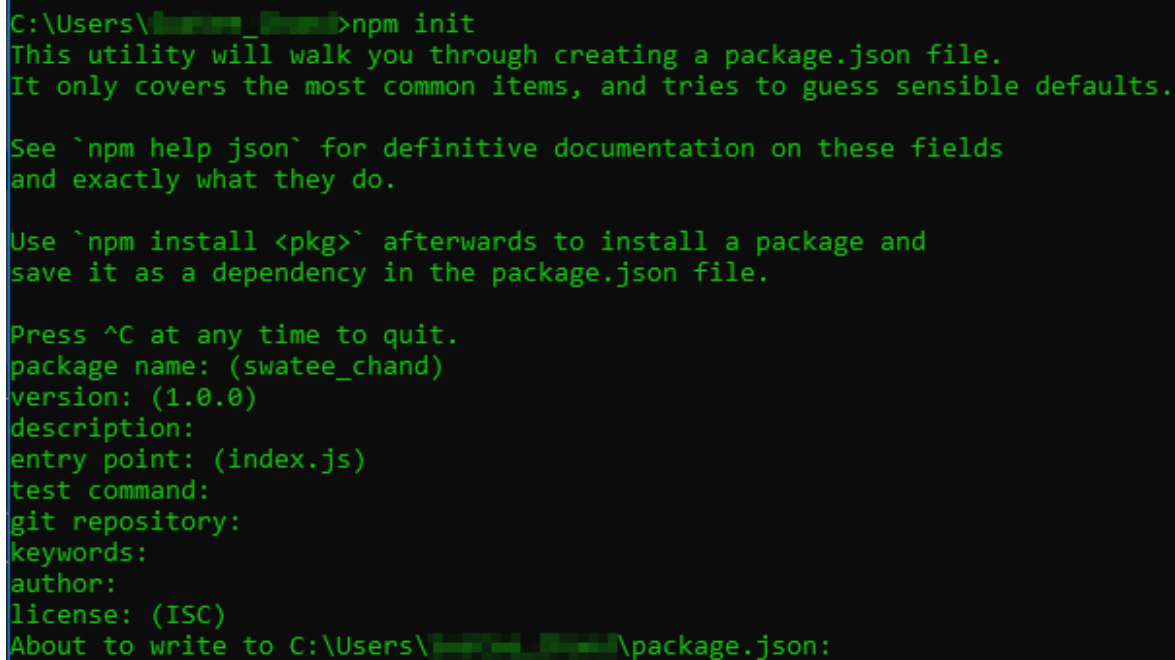
Here, we will be creating a simple CRUD REST application for Library Management using Node.js and Express.js. To build this application, you will need to install the following:

1. Node.js
2. Express.js
3. Joi
4. nodemon (Node Monitor)

First, you need to create your project directory. Next, open the command prompt and navigate to your project directory. Once there, you need to call npm using the below command:

1. npm init

When you hit enter, Node.js will ask you to enter some details to build the .json file such as:



```
C:\Users\swatee_chand>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (swatee_chand)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\swatee_chand\package.json:
```

Next, we will be installing Express.js using the below command:

```
npm i express
```

Finally, I will be installing a node monitoring package called nodemon. It keeps a watch on all the files with any type of extension present in this folder. Also, with nodemon on the watch, we don't have to restart the Node.js server each time any changes are made. nodemon will implicitly detect the changes and restart the server for us.

```
npm i -g nodemon
```

package.json

```
{
  "name":
    "samplerestapi",
  "version": "1.0.0",
  "description": "Edureka REST API with Node.js",
  "main": "script.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Edureka",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.4",
    "joi": "^13.1.0"
  }
}
```

Create a Docker file

```
1  FROM node:9-slim
2
3  # WORKDIR specifies the application directory
4
5  WORKDIR /app
6
7  # Copying package.json file to the app directory
8  COPY package.json /app
9
10 # Installing npm for DOCKER
11 RUN npm install
12
13
14 # Copying rest of the application to app directory
15 COPY . /app
16
17 # Starting the application using npm start
18 CMD ["npm", "start"]
```

Build Docker Image

Building a Docker image is rather easy and can be done using a simple command. Below I have written down the command that you need to type in your terminal and execute it:

```
docker build -t <docker-image-name> <file path>
```

```
F:\SampleRESTDocker>npm start  
  
> samplerestapi@1.0.0 start F:\SampleRESTDocker  
> node script.js  
  
Listening on port 8080..  
Terminate batch job (Y/N)?  
Terminate batch job (Y/N)? y
```

```
F:\SampleRESTDocker>docker build -t node-docker-tutorial .  
Sending build context to Docker daemon 2.351MB  
Step 1/6 : FROM node:9-slim  
----> e20bb4abe4ee  
Step 2/6 : WORKDIR /app  
----> Using cache  
----> 9e8b55075f1b
```

```
Step 3/6 : COPY package.json /app  
----> Using cache  
----> 54ea97c07839  
Step 4/6 : RUN npm install  
----> Using cache  
----> a0fd466fb335  
Step 5/6 : COPY . /app  
----> 96ab13c096d2  
Step 6/6 : CMD ["npm","start"]  
----> Running in 0711cf73910d  
Removing intermediate container 0711cf73910d  
----> 348837fc3708  
Successfully built 348837fc3708  
Successfully tagged node-docker-tutorial:latest
```

If we are getting an output something similar to the above screenshot, then it means that your application is working fine and the docker image has been successfully created. In the next section of this Node.js Docker article, I will show you how to execute this Docker Image.

Executing the Docker Image

Since you have successfully created your Docker image, now you can run one or more Docker containers on this image using the below-given command:

```
1 docker run it -d -p <HOST PORT>:<DOCKER PORT> <docker-image-name>
```

This command will start your docker container based on your Docker image and expose it on the specified port in your machine. In the above command **-d flag** indicates that you want to execute your Docker container in a detached mode. In other words, this will enable your Docker container to run in the background of the host machine. While the **-p flag** specifies which host port will be connected to the docker port.

To check whether your application has been successfully Dockerized or not, you can try launching it on the port you have specified for the host in the above command.

If We want to see the list of images currently running in your system, We can use the below command:

Installing Fabric-samples Repository

To start out with fabric samples install the Fabric-samples bash script:

```
url -sSLO https://raw.githubusercontent.com/hyperledger/fabric/main/scripts/install-fabric.sh &&
chmod +x install-fabric.sh
```

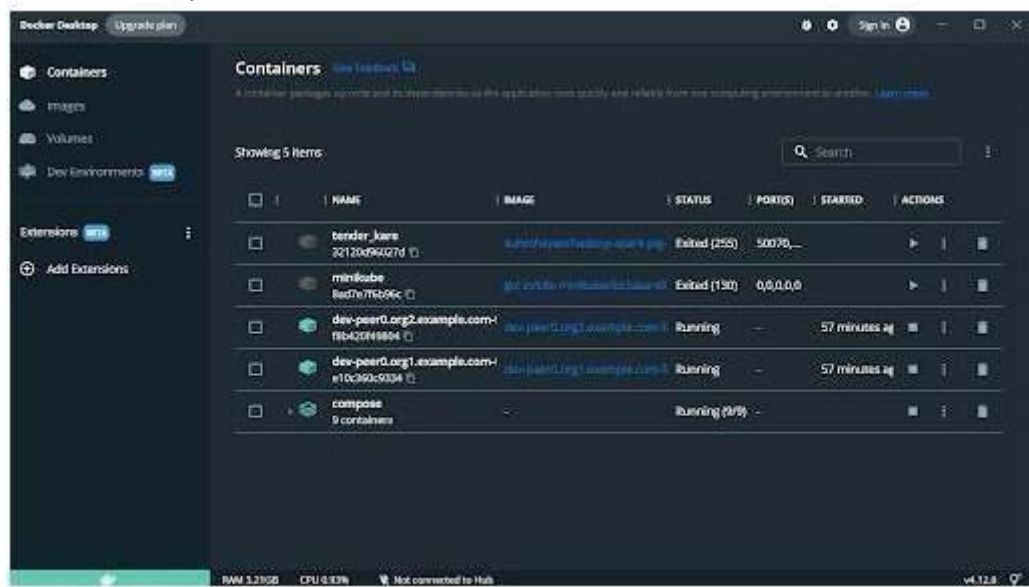
Then you can pull docker containers by running one of these commands:

```
./install-fabric.sh docker samples
```

```
./install-fabric.sh d s
```

To install binaries for Fabric samples you can use the command below:

```
./install-fabric.sh binary
```



Building First Network

Step 1: Navigate through the Fabric-samples folder and then through the Test network folder where you can find script files using these we can run our network. The test network is provided for learning about Fabric by running nodes on your local machine. Developers can use the network to test their smart contracts and applications.

```
cd fabric-samples/test-network
```

Step 2: From inside this directory, you can directly run `./network.sh` script file through which we run the Test Network. By default, we would be running `./network.sh down` command to remove any previous network containers or artifacts that still exist. This is to ensure that there are no conflicts when we run a new network.

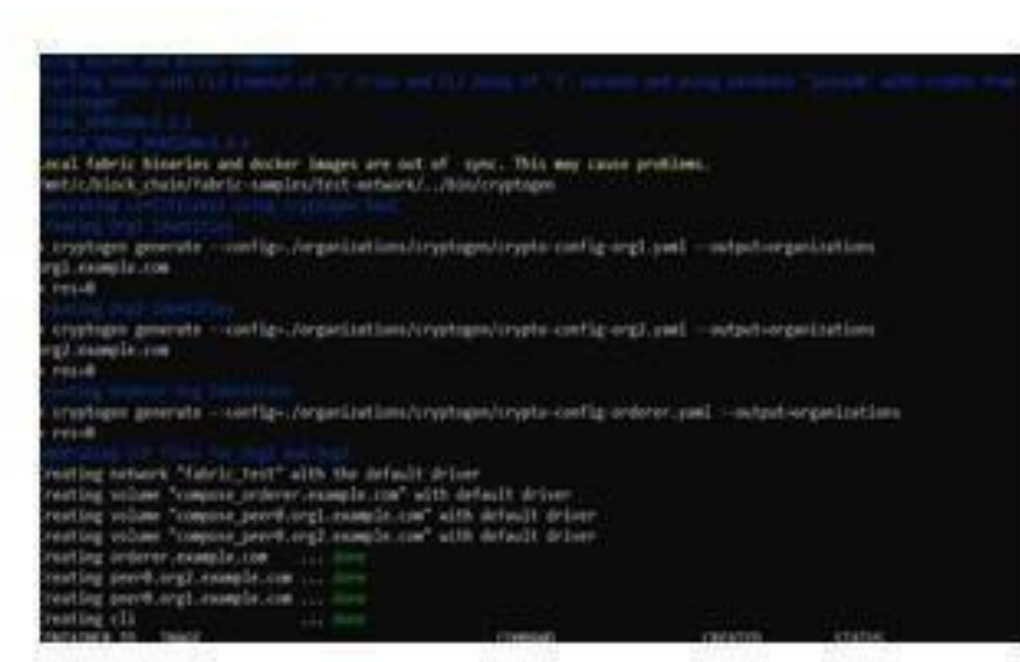
```
./network.sh down
```



Step 3: Now you can bring up a network using the following command. This command creates a fabric network that consists of two peer nodes and one ordering node. No channel is created when you run `./network.sh up`

`./network.sh up`

If the above command completes successfully, you will see the logs of the nodes being created below picture:



```
...
local fabric binaries and docker images are out of sync. This may cause problems.
fabric/bin/fabric:./network.sh up
...
$ cryptogen generate --config=./organizations/cryptogen/crypto-config-org1.yaml --output-organization
org1.example.com
$ cryptogen generate --config=./organizations/cryptogen/crypto-config-org2.yaml --output-organization
org2.example.com
$ cryptogen generate --config=./organizations/cryptogen/crypto-config-orderer.yaml --output-organization
orderer.example.com
...
$ ./network.sh up
creating network "FabricTest" with the default driver
creating volume "compose_orderer.example.com" with default driver
creating volume "compose_peer0.org1.example.com" with default driver
creating volume "compose_peer1.org2.example.com" with default driver
creating orderer.example.com ... done
creating peer0.org1.example.com ... done
creating peer1.org2.example.com ... done
creating cli ... done
...
CONTAINER ID        IMAGE               COMMAND
compose_orderer.example.com  fabric/orderer:0.4.25  fabric orderer
compose_peer0.org1.example.com  fabric/peer:0.4.25    fabric peer
compose_peer1.org2.example.com  fabric/peer:0.4.25    fabric peer
cli                            fabric/cli:0.4.25      fabric cli
```

Ethereum Network Installation

The environment we are going to set up has three main tools as follows.

1. The Go Ethereum Implementation or GETH
2. TestRPC that we are going to test put Smart Contracts
3. Truffle Build framework

Installing Geth

Geth , also known as Go Ethereum is a command line interface which allows us to run a full Ethereum node. Geth is implemented in GO and it will allow us to mine blocks , generate ether, deploy and interact with smart contracts , transfer funds , block history inspection and create accounts etc..

Geth can be used to get connected to the public Ethereum network as well apart from to create your own private network for development purposes.

Step 1:

```
$ sudo apt install software-properties-common
```

After that we need to add the Ethereum repository as follows.

```
$ sudo add-apt-repository -y ppa:ethereum/ethereum
```

Then , update the repositories.

```
$ sudo apt update
```

Then install the Ethereum.

```
$ sudo apt install ethereum
```

We can check the installation by using the \$ geth version command as follows.

A terminal window with a black background and green text. The prompt is 'priyal@priyal-Aunex:~\$'. The command 'geth version' has been executed, and the output is displayed line by line: 'Geth', 'Version: 1.7.2-stable', 'Git Commit: 1db4ecdc0b9e828ff65777fb466fc7c1d04e0de9', 'Architecture: amd64', 'Protocol Versions: [63 62]', 'Network Id: 1', 'Go Version: go1.9', 'Operating System: linux', 'GOPATH=', and 'GOROOT=/usr/lib/go-1.9'.

```
priyal@priyal-Aunex:~$ geth version
Geth
Version: 1.7.2-stable
Git Commit: 1db4ecdc0b9e828ff65777fb466fc7c1d04e0de9
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.9
Operating System: linux
GOPATH=
GOROOT=/usr/lib/go-1.9
```

Image 2 : Check geth version

Installing Test RPC

Test RPC is an Ethereum node emulator implemented in NodeJS. The purpose of this Test RPC is to easily start the Ethereum node for test and development purposes.

To install the Test RPC ;

```
sudo npm install -g ethereumjs-testrpc
```

```

priyal@priyal-Aunex:~$ testrpc version
EthereumJS TestRPC v4.1.3 (ganache-core: 1.1.3)

Available Accounts
=====
(0) 0xb7e389bc9a328f494415a4fc0675185e7853910b
(1) 0x65f7299c9cd04e47f9772608a24390466fe1e3e8
(2) 0x4300738991124ece3ce929ac1d15b5aa8e9f46d5
(3) 0x5605cb11c82556f4082420e584dba9528b9ed54e
(4) 0xfe3940741d14bbf9c11cdaf8a7aba2966aacf5fe
(5) 0x057aaf898b2874b0e8e02bc7e9cd7c034fb2b1f7
(6) 0x75575c6a1cf99045d55752cda56f098283ff1d04
(7) 0x84752fa99006536071a10e2214c6f7dab1886000
(8) 0xc4760bc9e2e8a2250351c3ca0f0c83462a76e147
(9) 0x3b1a2947c620beac5b53f2b059b03296be505b11

Private Keys
=====
(0) 2128c14142591293f1759601602e2c553daa1a0610b0a5c09606d5fc473e440c
(1) 24eb0d4ed1b207c38733972e25dbb99f644a26b532fde61d3c489451fc25ad4f
(2) 05384321387aeb2ea2749b6e9be3d248fda7c0d08dc7826fbe5d8caa5c282939
(3) 050297248cc82fe055af2684b79e3cc9b9c7a91fb7ae26becd296490931be5d8
(4) 4f68d7420132be39d64a2c07be31dc3d7c4488771ccee79a6374ddf574dd89a
(5) 70e6323db7f7e54a081194b8e23e8c3fe922b3f2d8cdf28fb65104262afd3dbf
(6) b390aec6daa132c23a39216f8256150fdfe4078d2116846f9a8adb142f603dec
(7) b2f151dc41993c473a0dca0a505617fa328645f4c0c843ba1bbdc412bcbf7d75
(8) 874dce579b60a32b80cc559d5dbf17095a20e45b3c85fd4936f6ae3a7e4d9ab0
(9) a2f6a5f2c57ded343d03b64ccbe106cb3214bdd6e4d75b6998b9fbd97641e7fa

HD Wallet
=====
Mnemonic:      gain novel firm since right essay subway blossom index biology innocent wheel
Base HD Path:  m/44'/60'/0'/0/{account_index}

Listening on localhost:8545

```

Image 3 : List all localaccounts

Installing Truffle

Truffle is a build framework and it takes care of managing your Contract artifacts so you don't have to. Includes support for custom deployments, library linking and complex Ethereum applications.

To install ;

```
$ sudo npm install -g truffle
```

check the installation as follows

```

priyal@priyal-Aunex:~$ truffle --version
Truffle v3.4.11 - a development framework for Ethereum

Usage: truffle <command> [options]

Commands:
  init           Initialize new Ethereum project with example contracts and tests
  compile        Compile contract source files
  migrate        Run migrations to deploy contracts
  deploy         (alias for migrate)
  build          Execute build pipeline (if configuration present)
  test          Run Mocha and Solidity tests
  console        Run a console with contract abstractions and commands available
  create         Helper to create new contracts, migrations and tests
  install        Install a package from the Ethereum Package Registry
  publish        Publish a package to the Ethereum Package Registry
  networks       Show addresses for deployed contracts on each network
  watch          Watch filesystem for changes and rebuild the project automatically
  serve          Serve the build directory on localhost and watch for changes
  exec           Execute a JS module within this Truffle environment
  unbox          Unbox Truffle project
  version        Show version number and exit

See more at http://truffleframework.com/docs

```

Image 4 : check truffle version

Setting up a Private Ethereum network

First of all create a folder to host the database and private accounts that we are going to create. In my case it is ~/ETH/pvt.

First of all we need to place the genesis block in our root (~/ETH/pvt). The file should be defined as genesis.json

```
{
  "nonce": "0x0000000000000042",
  "mixhash":
    "0x0000000000000000000000000000000000000000000000000000000000000000",
  "difficulty": "0x400",
  "alloc": {},
  "coinbase": "0x000000000000000000000000000000000000000000000000",
  "timestamp": "0x0",
  "parentHash":
    "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "0x",
  "gasLimit": "0xffffffff",
  "config": {
    "chainId": 4224,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
  }
}
```

To initialize the chain instance , we need to use following geth command.

```
$ geth --datadir ~/Eth/pvt init genesis.json
```

datadir param specifies where we should save our network's data. After initializing this, the root folder should contain something like following.



```
priyal@priyal-Aunex:~/Eth/pvt$ geth --datadir ~/Eth/pvt init genesis.json
INFO [10-26|15:06:19] Allocated cache and file handles   database=/home/priyal/Eth/pvt/geth/chaindata cache=16 handles=16
INFO [10-26|15:06:19] Successfully wrote genesis state   database=chaindata
                                hash=272003_b62890
INFO [10-26|15:06:19] Allocated cache and file handles   database=/home/priyal/Eth/pvt/geth/lightchaindata cache=16 handles=16
INFO [10-26|15:06:19] Successfully wrote genesis state   database=lightchaindata

priyal@priyal-Aunex:~/Eth/pvt$ tree .
.
├── genesis.json
├── geth
│   ├── chaindata
│   │   ├── 000018.ldb
│   │   ├── 000019.ldb
│   │   ├── 000020.ldb
│   │   ├── 000021.log
│   │   ├── CURRENT
│   │   ├── LOCK
│   │   ├── LOG
│   │   └── MANIFEST-000022
│   ├── lightchaindata
│   │   ├── 000002.ldb
│   │   ├── 000003.log
│   │   ├── CURRENT
│   │   ├── LOCK
│   │   ├── LOG
│   │   └── MANIFEST-000004
│   ├── LOCK
│   ├── nodekey
│   └── transactions.rlp
└── keystore
```


We can use the following geth command for this purpose.

```
$ geth --datadir ~/Eth/pvt/ account new
```

To list all created account lists you can use the account list command as



```
priyal@priyal-Aunex:~$ geth --datadir ~/Eth/pvt/ account list
Account #0: {04cee430d15e4a22f32ebc4247993cec540f18a0} keystore:///home/priyal/Eth/pvt/keystore/UTC--2017-10-25T10-54-33.814607396Z--64cee430d15e4a22f32ebc4247993cec540f18a0
Account #1: {12a14eae31018046036ce82d35cf871773eebc47} keystore:///home/priyal/Eth/pvt/keystore/UTC--2017-10-25T10-54-56.664136423Z--12a14eae31018046036ce82d35cf871773eebc47
Account #2: {36d4af267acd10d9dffd1f958a98c732fb9b7e3} keystore:///home/priyal/Eth/pvt/keystore/UTC--2017-10-25T10-55-02.919902286Z--36d4af267acd10d9dffd1f958a98c732fb9b7e3
```

Image 6 : Account list

As the next step, we need to specify following startnode.sh file. This script will start the network with given params.

startnode.sh

```
geth --networkid 4224 --mine --datadir "~/Eth/pvt" --nodiscover --rpc
--rpcport "8545"
--port "30303" --rpccorsdomain "*" --nat "any" --rpcapi
eth,web3,personal,net --unlock 0
--password ~/Eth/pvt/password.sec --ipcpath
"~/Library/Ethereum/geth.ipc"
```

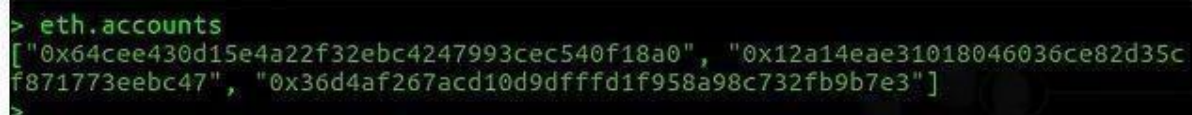
As the next step we need to log into the Geth console using the attach command as follows.

```
$ geth attach http://127.0.0.1:8545 --datadir /home/priyal/Eth/pvt
```

This will start the Geth console. To list all accounts we can use following command.

```
> eth.accounts
```

Account Creation output:



```
> eth.accounts
["0x64cee430d15e4a22f32ebc4247993cec540f18a0", "0x12a14eae31018046036ce82d35cf871773eebc47", "0x36d4af267acd10d9dffd1f958a98c732fb9b7e3"]
>
```

Result:

Thus the Installation of Docker Container, Node.js and HyperledgerFabric and Ethereum Network are executed successfully.

Ex:No:2 Understand the basics of Bitcoin transactions, mining, and wallet management.

Date:

Aim:

To understand Bitcoin transactions, mining, and wallet management by creating a wallet, performing transactions, and simulating mining.

Procedure:

Step 1: Setting Up a Bitcoin Wallet

1. Download and install **Electrum** (testnet mode) or **Trust Wallet**.
2. Create a new wallet and save the **seed phrase** securely.
3. Generate a **public Bitcoin address** to receive funds.
4. Note the **private key** (used for signing transactions).

Step 2: Receiving Bitcoin

1. Copy your **public Bitcoin testnet address**.
2. Visit a **Bitcoin Testnet Faucet** (e.g., mempool.space faucet).
3. Request free test Bitcoins.
4. Verify the balance in your wallet.

Step 3: Sending Bitcoin

1. Enter the **recipient's address**.
2. Set the **amount** to be sent.
3. Choose a **transaction fee** (higher fees = faster confirmation).
4. Sign and broadcast the transaction.

Step 4: Tracking Transactions

1. Copy the **Transaction ID (TXID)**.
2. Go to a blockchain explorer (mempool.space/testnet).
3. Search for the TXID and check confirmations.

Step 5: Bitcoin Mining Simulation

1. Open an **SHA-256 Hash Calculator** (e.g., Xorbin SHA-256).
2. Enter a random text (e.g., block1234) and compute the hash.
3. Modify the input until you find a hash starting with 0000 (proof-of-work simulation).
4. Observe mining difficulty changes.

➤ **Creating and Managing a Bitcoin Wallet**

Procedure:

1. Download and Install Electrum (Testnet Mode)

- Open a terminal and run:

electrum --testnet

2. Generate Public and Private Keys:

Obtain the **public address** for receiving Bitcoin:

electrum getunusedaddress

View the private key (DO NOT SHARE):

electrum listaddresses --funded

3. Receive Bitcoin (Using Testnet Faucet):

Go to Testnet Faucet

Paste your wallet's testnet address and request free Bitcoin.

4. Send Bitcoin:

Use Electrum to send BTC to another address:

electrum payto <recipient_address> <amount_in_btc>

electrum broadcast

5. Backup & Restore Wallet:

Backup the wallet seed phrase.

Restore the wallet using the seed phrase on another device.

➤ **Simulating a Bitcoin Transaction on the Blockchain**

Procedure:

1. Create a Transaction:

Send BTC from your wallet to another testnet address.

Adjust the **transaction fee** and note how it affects confirmation time.

2. Monitor the Transaction:

Copy the **TXID** from your wallet.

Paste it in Mempool.space Testnet to track confirmation status.

3. Analyze Blockchain Data:

Check inputs, outputs, and miner fees.

Expected Output:

The transaction appears in the blockchain explorer with details like:

Transaction Hash: b6f1...a45d

Sender Address: mj4W...X9yf

Recipient Address: n2Uw...7BdH

Amount Sent: 0.002 BTC

Fee Paid: 0.00001 BTC

Confirmations: 3+

➤ Bitcoin Mining Simulation Procedure:

1. Simulate Hashing Process:

- Input a random string like block1234 in the SHA-256 calculator.
- Observe the generated hash.

2. Adjust Mining Difficulty:

- Try finding a hash with a specific pattern (e.g., starting with four zeros 0000).

3. Simulate Mining Rewards:

- Observe how mining rewards are distributed among miners.

Expected Output:

- Example SHA-256 hash:

```
block1234 → e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855
```

- Finding a valid hash under difficulty constraints takes multiple attempts.
- Mining simulation shows block rewards and energy consumption analysis.

Expected Output (Python Script for Generating a Bitcoin Wallet)

Private Key: 5JwD8wP1EAhVJbvUKf2wqX3Y3Xv5F6t93R7MFzP4G3G5K8jBzJ7

Public Key: 04a34df4ac3b6bc7e6f16f8e72b6e5d57b36b943a3f2...

Bitcoin Address (Testnet): muVpJrs5P1m9TqQkD9kgYa3CvxH2mZ9QbT

Expected Output (Receiving Bitcoin on Testnet Wallet)

Requesting BTC from Testnet Faucet...

Transaction Submitted! TXID: 3f7a9c...a6b5e7

Balance Updated: 0.002 BTC

Expected Output (Sending Bitcoin Transaction on Testnet)

Enter Recipient Address: mhHyzq9...JHtFZkY

Enter Amount: 0.001 BTC

Transaction Sent!

TXID: b6f1e2...a45d

Waiting for Confirmation...

Transaction Confirmed in Block: #2635847

RESULT:

The experiment successfully demonstrated Bitcoin wallet management, transactions, and mining principles using testnet tools and blockchain explorers

Ex:No:3. Implement ECC key generation algorithm using a programming language of your choice

Date:

AIM:

To implement **Elliptic Curve Cryptography (ECC)** for secure key generation, encryption, and decryption of messages using Java. The experiment demonstrates elliptic curve point operations, key generation, and encryption-decryption processes.

ALGORITHM:

- **Step 1: Initialize Elliptic Curve Parameters**
Define the elliptic curve equation: $y^2 = x^3 + ax + b \pmod{p}$.
Set the curve parameters (**a, b, p**).
- **Step 2: Select a Generator Point (G)**
Identify valid points on the curve.
Choose a generator point **G(x, y)**.
- **Step 3: Generate Public and Private Keys**
Select a **random private key (s)**.
Compute the public key: **P = s × G** using scalar multiplication.
- **Step 4: Encrypt a Message (Mx, My)**
Select a random integer **k** as the session key.
Compute **C1 = k × G** (part 1 of ciphertext).
Compute **C2 = M + k × P** (part 2 of ciphertext).
Send (**C1, C2**) as the encrypted message.
- **Step 5: Decrypt the Message**
Compute **S = s × C1** using the private key.
Retrieve the original message using **M = C2 - S**.
- **Step 6: Display the Results**
Print the generated public-private key pair.
Show the encrypted **ciphertext (C1, C2)**.
Display the decrypted message (**Mx, My**).

PROGRAM:

```
import java.util.*;
```

```
class ECC {  
    int a, b, p; // Elliptic curve parameters  
    int gx = -1, gy = -1; // Generator point  
    int privateKey = -1;  
  
    public void setCurve(int a, int b, int p) {  
        this.a = a;  
        this.b = b;  
        this.p = p;  
        System.out.println("Elliptic curve: y^2 = x^3 + " + a + "x + " + b + " mod " + p);  
    }  
  
    public void setPrivateKey(int privateKey) {  
        this.privateKey = privateKey;  
    }  
  
    public void setGeneratorPoint(int gx, int gy) {  
        this.gx = gx;  
        this.gy = gy;  
    }  
}
```

```

System.out.println("Generator point set: G(" + gx + ", " + gy + ")");
}

public int[] addPoints(int x1, int y1, int x2, int y2) {
    if (x1 == x2 && y1 == y2) {
        return doublePoint(x1, y1);
    }

    int numerator = (y2 - y1 + p) % p;
    int denominator = (x2 - x1 + p) % p;
    int lambda = (numerator * modInverse(denominator)) % p;

    int x3 = (lambda * lambda - x1 - x2) % p;
    if (x3 < 0) x3 += p;

    int y3 = (lambda * (x1 - x3) - y1) % p;
    if (y3 < 0) y3 += p;

    return new int[]{x3, y3};
}

public int[] doublePoint(int x1, int y1) {
    if (y1 == 0) return new int[]{0, 0};

    int numerator = (3 * x1 * x1 + a) % p;
    int denominator = (2 * y1) % p;
    int lambda = (numerator * modInverse(denominator)) % p;

    int x3 = (lambda * lambda - 2 * x1) % p;
    if (x3 < 0) x3 += p;

    int y3 = (lambda * (x1 - x3) - y1) % p;
    if (y3 < 0) y3 += p;

    return new int[]{x3, y3};
}

public int[] scalarMultiply(int s, int x, int y) {
    int[] result = {x, y};
    int[] temp = {x, y};

    for (int i = 1; i < s; i++) {
        result = addPoints(result[0], result[1], temp[0], temp[1]);
    }
    return result;
}

public int[][] encrypt(int[] message, int[] publicKey, int k) {
    int[] C1 = scalarMultiply(k, gx, gy);
    int[] kP = scalarMultiply(k, publicKey[0], publicKey[1]);
    int[] C2 = addPoints(message[0], message[1], kP[0], kP[1]);
    return new int[][]{C1, C2};
}

public int[] decrypt(int[][] cipherText) {
    int[] C1 = cipherText[0];

```

```

int[] C2 = cipherText[1];
int[] sC1 = scalarMultiply(privateKey, C1[0], C1[1]);
return subtractPoints(C2[0], C2[1], sC1[0], sC1[1]);
}

public int[] subtractPoints(int x1, int y1, int x2, int y2) {
    return addPoints(x1, y1, x2, (p - y2) % p);
}

public int modInverse(int a) {
    a = a % p;
    for (int x = 1; x < p; x++) {
        if ((a * x) % p == 1) return x;
    }
    return 1;
}
}

public class EllipticCurveCryptography {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ECC ecc = new ECC();

        while (true) {
            System.out.println("\n===== Elliptic Curve Cryptography Menu =====\n1. Select Elliptic
Curve\n2. Generate Generator Point\n3. Generate Public Key\n4. Encrypt Message\n5. Decrypt
Message\n6. Exit");
            System.out.print("Enter choice: ");
            int choice = sc.nextInt();

            switch (choice) {
                case 1 -> {
                    System.out.print("Enter values a, b, p: ");
                    int a = sc.nextInt(), b = sc.nextInt(), p = sc.nextInt();
                    ecc.setCurve(a, b, p);
                }
                case 2 -> {
                    System.out.println("Finding generator points...");
                    ArrayList<int[]> points = new ArrayList<>();
                    for (int x = 0; x < ecc.p; x++) {
                        int rhs = (x * x * x + ecc.a * x + ecc.b) % ecc.p;
                        for (int y = 0; y < ecc.p; y++) {
                            if ((y * y) % ecc.p == rhs) {
                                points.add(new int[]{x, y});
                                System.out.println("Point: (" + x + ", " + y + ")");
                            }
                        }
                    }
                    System.out.print("Select generator index: ");
                    int index = sc.nextInt();
                    if (index >= 0 && index < points.size()) {
                        ecc.setGeneratorPoint(points.get(index)[0], points.get(index)[1]);
                    }
                }
            }
        }
    }
}

```

```

case 3 -> {
    System.out.print("Enter private key: ");
    int s = sc.nextInt();
    ecc.setPrivateKey(s);
    int[] publicKey = ecc.scalarMultiply(s, ecc.gx, ecc.gy);
    System.out.println("Public Key: (" + publicKey[0] + ", " + publicKey[1] + ")");
}
case 4 -> {
    System.out.print("Enter message (Mx, My): ");
    int mx = sc.nextInt(), my = sc.nextInt();
    System.out.print("Enter secret key: ");
    int k = sc.nextInt();
    int[] publicKey = ecc.scalarMultiply(ecc.privateKey, ecc.gx, ecc.gy);
    int[][] encrypted = ecc.encrypt(new int[]{mx, my}, publicKey, k);
    System.out.println("Ciphertext C1: (" + encrypted[0][0] + ", " + encrypted[0][1] + ")");
    System.out.println("Ciphertext C2: (" + encrypted[1][0] + ", " + encrypted[1][1] + ")");
}
case 5 -> {
    System.out.print("Enter Ciphertext (C1x, C1y, C2x, C2y): ");
    int[][] cipherText = {{sc.nextInt(), sc.nextInt()}, {sc.nextInt(), sc.nextInt()}};
    int[] decrypted = ecc.decrypt(cipherText);
    System.out.println("Decrypted Message: (" + decrypted[0] + ", " + decrypted[1] + ")");
}
case 6 -> {
    sc.close();
    System.exit(0);
}
default -> System.out.println("Invalid choice!");
}
}
}
}
}

```

SAMPLE OUTPUT

```
[20:54] PS ECC [master]> javac .\EllipticCurveCryptography.java
• [20:55] PS ECC [master]> java EllipticCurveCryptography

===== Elliptic Curve Cryptography Menu =====
1. Select Elliptic Curve
2. Generate Generator Point
3. Generate Public Key
4. Encrypt Message
5. Decrypt Message
6. Exit
Enter choice: 1
Enter values a, b, p: 1 6 11
Elliptic curve:  $y^2 = x^3 + 1x + 6 \pmod{11}$ 

===== Elliptic Curve Cryptography Menu =====
1. Select Elliptic Curve
2. Generate Generator Point
3. Generate Public Key
4. Encrypt Message
5. Decrypt Message
6. Exit
Enter choice: 2
Finding generator points...
Point: (2, 4)
Point: (2, 7)
Point: (3, 5)
Point: (3, 6)
Point: (5, 2)
Point: (5, 9)
Point: (7, 2)
Point: (7, 9)
Point: (8, 3)
Point: (8, 8)
Point: (10, 2)
Point: (10, 9)
Select generator index: 1
Generator point set: G(2, 7)
```

```
===== Elliptic Curve Cryptography Menu =====
1. Select Elliptic Curve
2. Generate Generator Point
3. Generate Public Key
4. Encrypt Message
5. Decrypt Message
6. Exit
Enter choice: 3
Enter private key: 2
Public Key: (5, 2)

===== Elliptic Curve Cryptography Menu =====
1. Select Elliptic Curve
2. Generate Generator Point
3. Generate Public Key
4. Encrypt Message
5. Decrypt Message
6. Exit
Enter choice: 4
Enter message (Mx, My): 2 7
Enter secret key: 3
Ciphertext C1: (8, 3)
Ciphertext C2: (7, 2)

===== Elliptic Curve Cryptography Menu =====
1. Select Elliptic Curve
2. Generate Generator Point
3. Generate Public Key
4. Encrypt Message
5. Decrypt Message
6. Exit
Enter choice: 5
Enter Ciphertext (C1x, C1y, C2x, C2y): 8 3 7 2
Decrypted Message: (2, 7)
```

RESULT:

Thus The experiment demonstrated the working of ECC-based cryptographic encryption and decryption, ensuring secure communication with minimal computational overhead.

EX NO 4. Explore Web3j, a Java library for working with Ethereum.

Date:

AIM

To demonstrate the use of **Web3j** for interacting with an Ethereum blockchain, including connecting to a node, retrieving the latest block number, and checking an account balance.

ALGORITHM

Setup Web3j

- Add the Web3j dependency to the Java project.
- Connect to an Ethereum node (e.g., Infura or local Ganache).

Retrieve Block Number

- Use Web3j to get the latest block number from the blockchain.

Check Account Balance

- Fetch and display the balance of a given Ethereum address.

Compile and Run

- Execute the Java program and observe the output.

PROGRAM

```
import org.web3j.protocol.Web3j;
import org.web3j.protocol.http.HttpService;
import org.web3j.protocol.core.methods.response.Web3ClientVersion;
import org.web3j.protocol.core.methods.response.EthBlockNumber;
import org.web3j.protocol.core.methods.response.EthGetBalance;
import org.web3j.utils.Convert;
import java.math.BigDecimal;
import java.math.BigInteger;

public class Web3jExample {
    public static void main(String[] args) {
try {
    // Connect to an Ethereum node via Infura
    String infuraUrl = "https://mainnet.infura.io/v3/YOUR_INFURA_PROJECT_ID";
    Web3j web3 = Web3j.build(new HttpService(infuraUrl));

    // Get client version
    Web3ClientVersion clientVersion = web3.web3ClientVersion().send();
    System.out.println("Ethereum Client Version: " + clientVersion.getWeb3ClientVersion());

    // Get latest block number
    EthBlockNumber blockNumber = web3.ethBlockNumber().send();
    System.out.println("Latest Block Number: " + blockNumber.getBlockNumber());

    // Check ETH balance of an address
    String address = "0x742d35Cc6634C0532925a3b844Bc454e4438f44e"; // Replace with any valid
    address
    EthGetBalance ethGetBalance = web3.ethGetBalance(address,
    org.web3j.protocol.core.DefaultBlockParameterName.LATEST).send();
```



```
BigInteger weiBalance = ethGetBalance.getBalance();
BigDecimal ethBalance = Convert.fromWei(new BigDecimal(weiBalance), Convert.Unit.ETHER);

System.out.println("ETH Balance of " + address + ": " + ethBalance + " ETH");

} catch (Exception e) {
System.err.println("Error: " + e.getMessage());
}
}
}
```

OUTPUT

```
Ethereum Client Version: Geth/v1.10.23-stable
Latest Block Number: 18456789
ETH Balance of 0x742d35Cc6634C0532925a3b844Bc454e4438f44e: 235.67 ETH
```

RESULT:

Successfully connected to the Ethereum blockchain using Web3j, retrieved the latest block number, and checked the balance of an Ethereum address.

EX NO:5 Interact with a blockchain network. Execute transactions and requests against a blockchain network by creating an app to test the network and its rules

Date:

Aim:

To Interact with a blockchain network and execute transactions and requests against a blockchain network by creating an app to test the network and its rules.

Procedure:

Installing the prerequisites, tools, and a Fabric runtime

1. Installing Prereqs

Now that we have a high level understanding of what is needed to build these networks, we can start developing. Before we do that, though, we need to make sure we have the prerequisites installed on our system. An updated list can be found here.

- Docker Engine and Docker Compose
- Nodejs and NPM
- Git
- Python 2.7.x

yperledger has a bash script available to make this process extremely easy. Run the following commands in your terminal:

2. Installing tools to ease development

Run the following commands in your Terminal, and make sure you're NOT using sudo when running npm commands.

3. Installing a local Hyperledger Fabric runtime

Let's go through the commands and see what they mean. First, we make and enter a new directory. Then, we download and extract the tools required to install Hyperledger Fabric.

We then specify the version of Fabric we want, at the time of writing we need 1.2, hence hlfv12. Then, we download the fabric runtime and start it up.

The following Business Network Cards are available:

Connection Profile: hlfv1

Card Name	UserId	Business Network
PeerAdmin@hlfv1	PeerAdmin	

Issue `composer card list --card <Card Name>` to get details a specific card

Command succeeded

Also, if you type ls you'll see this:

```
createComposerProfile.sh  fabric-scripts  stopFabric.sh
createPeerAdminCard.sh  _loader.sh     teardownAllDocker.sh
downloadFabric.sh        package.json    teardownFabric.sh
fabric-dev-servers.tar.gz startFabric.sh
```

Basically what we did here was just download and start a local Fabric network. We can stop it using ./stopFabric.sh if we want to. At the end of our development session, we should run ./teardownFabric.sh

Creating and deploying our business network

1. Generating a business network

Open terminal in a directory of choice and type yo hyperledger-composer

```
haardik@haardik-XPS-15-9570:~/workspace/f2$ yo hyperledger-composer
Welcome to the Hyperledger Composer project generator
? Please select the type of project: (Use arrow keys)
> Angular
  Business Network
  LoopBack
  Model
```

you'll be greeted with something similar to the above. Select Business Network and name it cards-trading-network as shown below:

```
haardik@haardik-XPS-15-9570:~/workspace/f2$ yo hyperledger-composer
Welcome to the Hyperledger Composer project generator
? Please select the type of project: Business Network
You can run this generator using: 'yo hyperledger-composer:businessnetwork'
Welcome to the business network generator
? Business network name: cards-trading-network
? Description: A Hyperledger Fabric network to trade cards between permissioned participants
? Author name: Haardik Haardik
? Author email: haardikk21@gmail.com
? License: Apache-2.0
? Namespace: org.example.biznet
? Do you want to generate an empty template network? Yes: generate an empty template network
  create package.json
  create README.md
  create models/org.example.biznet.cto
  create permissions.acl
  create .eslintrc.yml
haardik@haardik-XPS-15-9570:~/workspace/f2$
```

2. Modeling our business network

The first and most important step towards making a business network is identifying the resources present. We have four resource types in the modeling language:

Assets

Participants

Transactions

Events

For our cards-trading-network, we will define an asset type TradingCard, a participant type Trader, a transaction TradeCard and an event TradeNotification.

Go ahead and open the generated files in a code editor of choice. Open up `org.example.biznet.cto` which is the modeling file. Delete all the code present in it as we're gonna rewrite it (except for the namespace declaration).

This contains the specification for our asset `TradingCard`. All assets and participants need to have a unique identifier for them which we specify in the code, and in our case, it's `cardId`

Also, our asset has a `GameType cardType` property which is based off the enumerator defined below. Enums are used to specify a type which can have up to N possible values, but nothing else. In our example, no `TradingCard` can have a `cardType` other than `Baseball`, `Football`, or `Cricket`.

Now, to specify our `Trader` participant resource type, add the following code in the modeling file

This is relatively simpler and quite easy to understand. We have a participant type `Trader` and they're uniquely identified by their `traderIds`.

Now, we need to add a reference to our `TradingCards` to have a reference pointing to their owner so we know who the card belongs to. To do this, add the following line inside your `TradingCard` asset:

```
--> Trader owner
```

so that the code looks like this:

This is the first time we've used `-->` and you must be wondering what this is. This is a relationship pointer. `o` and `-->` are how we differentiate between a resource's own properties vs a relationship to another resource type. Since the owner is a `Trader` which is a participant in the network, we want a reference to that `Trader` directly, and that's exactly what `-->` does.

Finally, go ahead and add this code in the modeling file which specifies what parameters will be required to make a transaction and emitting an event.

3. Adding logic for our transactions

To add logic behind the `TradeCard` function, we need a Javascript logic file. Create a new directory named `lib` in your project's folder and create a new file named `logic.js` with the following code:

4. Defining permissions and access rules

Add a new rule in `permissions.acl` to give participants access to their resources. In production, you would want to be more strict with these access rules. You can read more about them [here](#).

5. Generating a Business Network Archive (BNA)

Now that all the coding is done, it's time to make an archive file for our business network so we can deploy it on our local Fabric runtime. To do this, open Terminal in your project directory and type this:

```
composer archive create --sourceType dir --sourceName
```

This command tells Hyperledger Composer we want to build a BNA from a directory which is our current root folder.

```

haardik@haardik-GT72S-6QE:~/Desktop/workspace/hyperledger-tutorial/cards-trading-network$ composer archive create --sourceType dir --sourceName .
Creating Business Network Archive

Looking for package.json of Business Network Definition
Input directory: /home/haardik/Desktop/workspace/hyperledger-tutorial/cards-trading-network

Found:
  Description: A Hyperledger Fabric network to trade cards between permissioned participants
  Name: cards-trading-network
  Identifier: cards-trading-network@0.0.1

Written Business Network Definition Archive file to
Output file: cards-trading-network@0.0.1.bna

Command succeeded

```

6. Install and Deploy the BNA file

We can install and deploy the network to our local Fabric runtime using the PeerAdmin user. To install the business network, type

```
composer network install --archiveFile cards-trading-network@0.0.1.bna --card PeerAdmin@hlfv1
```

```

haardik@haardik-GT72S-6QE:~/Desktop/workspace/hyperledger-tutorial/cards-trading-network$ composer network install --archiveFile cards-trading-network@0.0.1.bna --card PeerAdmin@hlfv1
✓ Installing business network. This may take a minute...
Successfully installed business network cards-trading-network, version 0.0.1

Command succeeded

```

To deploy the business network, type

```
composer network start --networkName cards-trading-network --networkVersion 0.0.1 --networkAdmin admin --networkAdminEnrollSecret adminpw --card PeerAdmin@hlfv1 --file cards-trading-admin.card
```

```

haardik@haardik-GT72S-6QE:~/Desktop/workspace/hyperledger-tutorial/cards-trading-network$ composer network start --networkName cards-trading-network --networkVersion 0.0.1 --networkAdmin admin --networkAdminEnrollSecret adminpw --card PeerAdmin@hlfv1 --file cards-trading-admin.card
Starting business network cards-trading-network at version 0.0.1

Processing these Network Admins:
  userName: admin

✓ Starting business network definition. This may take a minute...
Successfully created business network card:
  Filename: cards-trading-admin.card

Command succeeded

```

The networkName and networkVersion must be the same as specified in your package.json otherwise it won't work.

--file takes the name of the file to be created for THIS network's business card. This card then needs to be imported to be usable by typing

```
composer card import --file cards-trading-admin.card
```

```

haardik@haardik-GT72S-6QE:~/Desktop/workspace/hyperledger-tutorial/cards-trading-network$ composer card import --file cards-trading-admin.card

Successfully imported business network card
    Card file: cards-trading-admin.card
    Card name: admin@cards-trading-network

Command succeeded

```

We can now confirm that our network is up and running by typing

composer network ping --card admin@cards-trading-network
 --card this time takes the admin card of the network we want to ping.
 If everything went well, you should see something similar to this:

```

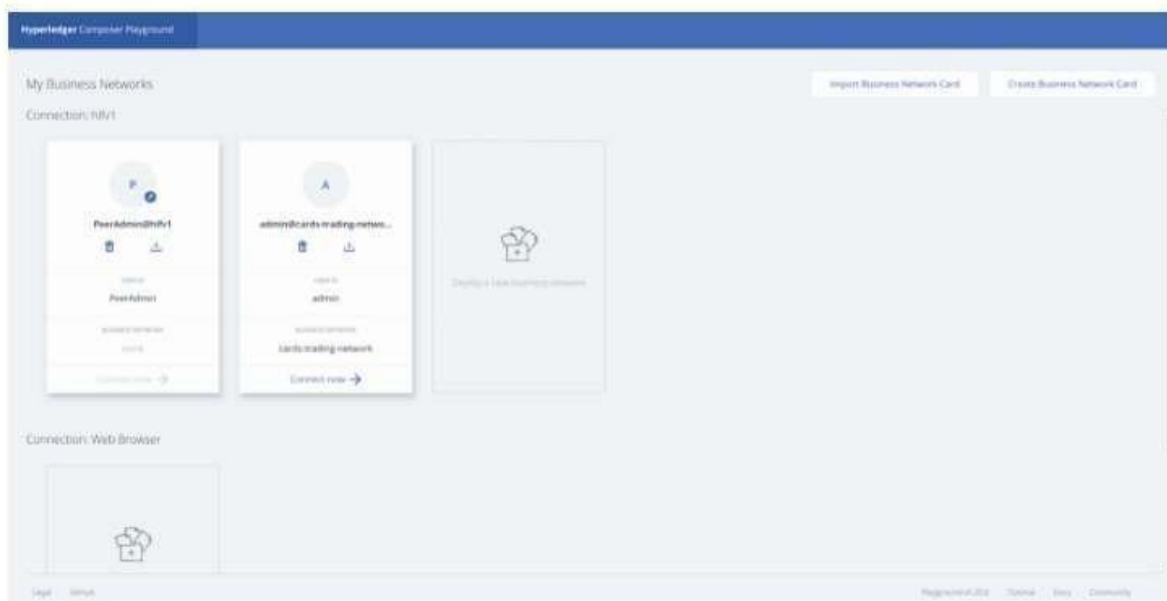
haardik@haardik-GT72S-6QE:~$ composer network ping --card admin@cards-trading-network
The connection to the network was successfully tested: cards-trading-network
    Business network version: 0.0.4-deploy.0
    Composer runtime version: 0.20.0
    participant: org.hyperledger.composer.system.NetworkAdmin#admin
    identity: org.hyperledger.composer.system.Identity#457abd6d405ce1dc509a16363611bae608b9904958786ae4ad5a46eb009ef10c

Command succeeded

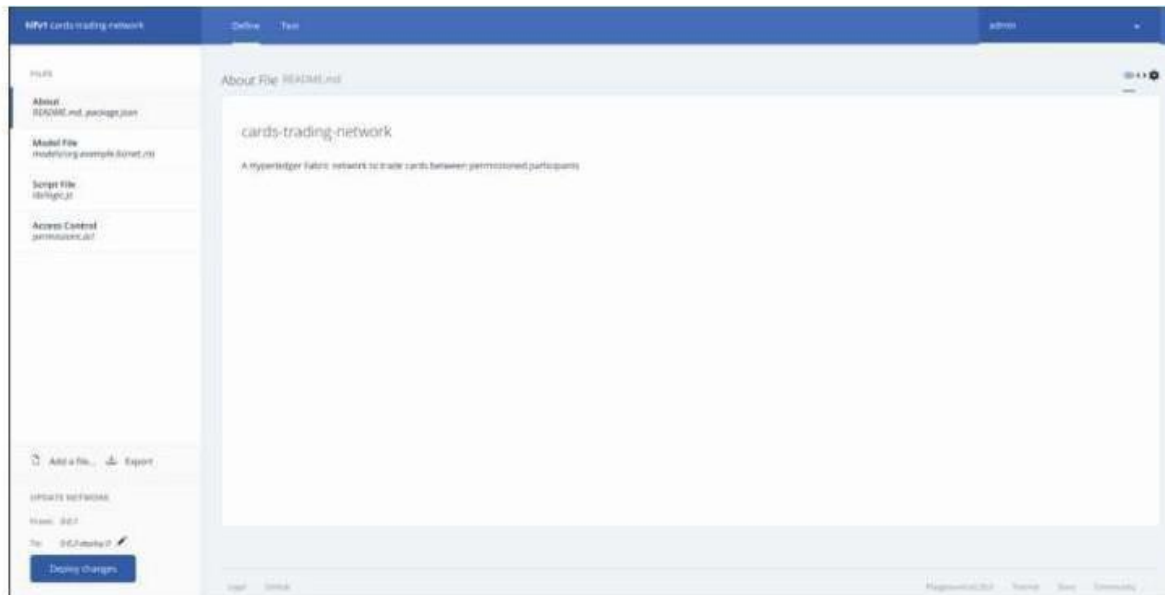
```

Testing our Business Network

Now that our network is up and running on Fabric, we can start Composer Playground to interact with it. To do this, type composer-playground in Terminal and open up <http://localhost:8080/> in your browser and you should see something similar to this:

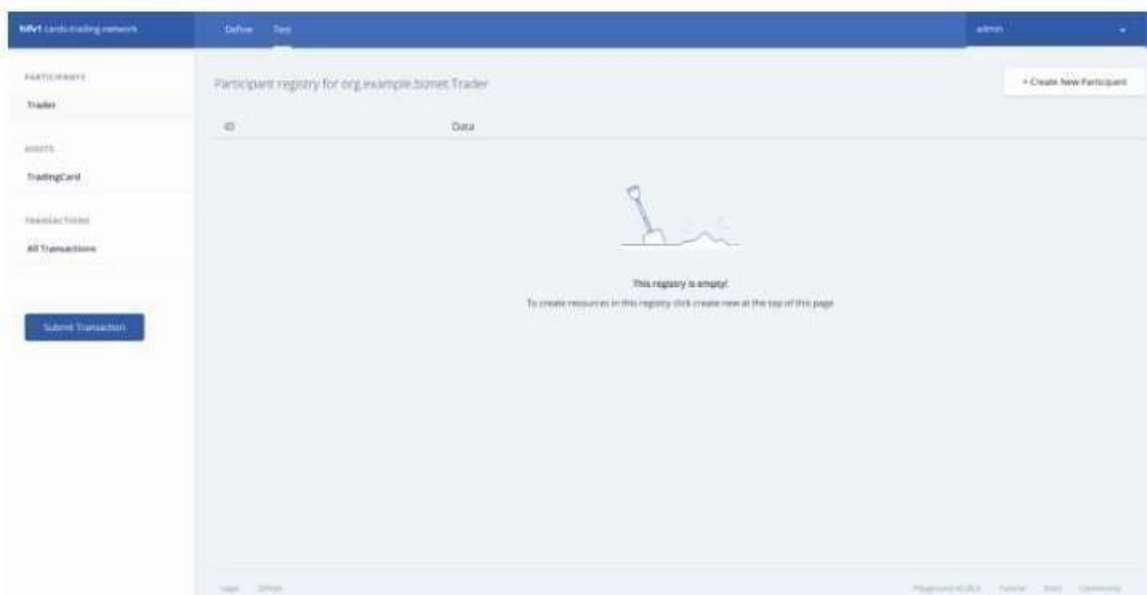


Press Connect Now for admin@cards-trading-network and you'll be greeted with this screen:



The **Define** page is where we can make changes to our code, deploy those changes to upgrade our network, and export business network archives.

Head over to the **Test** page from the top menu, and you'll see this:



Select **Trader** from **Participants**, click on **Create New Participant** near the top right, and make a new **Trader** similar to this:

Create New Participant ✕

In registry: **org.example.biznet.Trader**

JSON Data Preview

```
1 {
2   "$class": "org.example.biznet.Trader",
3   "traderId": "1",
4   "traderName": "Haardik"
5 }
```

☐ Optional Properties

Go ahead and make a couple more Traders. Here are what my three traders look like with the names Haardik, John, and Tyrone.

Participant registry for org.example.biznet.Trader		+ Create New Participant
ID	Data	
1	<pre>{ "\$class": "org.example.biznet.Trader", "traderId": "1", "traderName": "Haardik" }</pre>	 
2	<pre>{ "\$class": "org.example.biznet.Trader", "traderId": "2", "traderName": "John" }</pre>	 
3	<pre>{ "\$class": "org.example.biznet.Trader", "traderId": "3", "traderName": "Tyrone" }</pre>	 

Click on TradingCard from the left menu and press **Create New Asset**. Notice how the owner field is

```
"owner": "resource:org.example.biznet.Trader#3649"
```

particularly interesting here, looking something like this:

Go ahead and finish making a TradingCard something similar to this:

```
1 {
2   "$class": "org.example.biznet.TradingCard",
3   "cardId": "1",
4   "cardName": "Babe Ruth",
5   "cardDescription": "George Herman 'Babe' Ruth Jr. was an American
   professional baseball player whose career in Major League Baseball
   spanned 22 seasons, from 1914 through 1935.",
6   "cardType": "Baseball",
7   "forTrade": false,
8   "owner": "resource:org.example.biznet.Trader#1"
9 }
```

Notice how the owner field points to Trader#1 aka Haardik for me. Go ahead and make a couple more cards, and enable a couple to have forTrade set to true.

Asset registry for org.example.biznet.TradingCard

ID	Data
1	<pre>{ "\$class": "org.example.biznet.TradingCard", "cardId": "1", "cardName": "Babe Ruth", "cardDescription": "George Herman 'Babe' Ruth Jr. was an Americ "cardType": "Baseball", "forTrade": false, "owner": "resource:org.example.biznet.Trader#1" }</pre>
2	<pre>{ "\$class": "org.example.biznet.TradingCard", "cardId": "2", "cardName": "Cy Young", "cardDescription": "Denton True 'Cy' Young was an American Majc "cardType": "Baseball", "forTrade": true, "owner": "resource:org.example.biznet.Trader#2" }</pre>
3	<pre>{ "\$class": "org.example.biznet.TradingCard", "cardId": "3", "cardName": "Virat Kohli", "cardDescription": "Virat Kohli is an Indian international cric "cardType": "Cricket", "forTrade": false, "owner": "resource:org.example.biznet.Trader#3" }</pre>

Notice how my Card#2 has forTrade == true?
Now for the fun stuff, let's try trading cards :D

Click on **Submit Transaction** in the left and
make card point to TradingCard#2 and
newOwner point to Trader#3 like this

Submit Transaction

Transaction Type TradeCard

JSON Data Preview

```

1 {
2   "$class": "org.example.biznet.TradeCard",
3   "card": "resource:org.example.biznet.TradingCard#2",
4   "newOwner": "resource:org.example.biznet.Trader#3"
5 }

```

Generating a REST API Server

Doing transactions with Playground is nice, but not optimal. We have to make client-side software for users to provide them a seamless experience, they don't even have to necessarily know about the underlying blockchain technology. To do so, we need a better way of interacting with our business network. Thankfully, we have the `composer-rest-server` module to help us with just that.

Type `composer-rest-server` in your terminal, specify `admin@cards-trading-network`, select **never use namespaces**, and continue with the default options for the rest as follows:

```

haardik@haardik-GT725-6QE:~/Desktop/workspace/hyperledger-tutorial/cards-trading-network$ composer-rest-server
? Enter the name of the business network card to use: admin@cards-trading-network
? Specify if you want namespaces in the generated REST API: never use namespaces
? Specify if you want to use an API key to secure the REST API: No
? Specify if you want to enable authentication for the REST API using Passport: No
? Specify if you want to enable event publication over WebSockets: Yes
? Specify if you want to enable TLS security for the REST API: No

To restart the REST server using the same options, issue the following command:
  composer-rest-server -c admin@cards-trading-network -n never -w true

Discovering types from business network definition ...
Discovering the Returning Transactions..
Discovered types from business network definition
Generating schemas for all types in business network definition ...
Generated schemas for all types in business network definition
Adding schemas for all types to Loopback ...
Added schemas for all types to Loopback
Web server listening at: http://localhost:3000
Browse your REST API at http://localhost:3000/explorer

```

This will go on to run `npm install`, give it a minute, and once it's all done you'll be able to load up <http://localhost:4200/> and be greeted with a page similar to this: **Edit:** Newer versions of the software may require you to run `npm install` yourself and then run `npm start`



You can now play with your network from this application directly, which communicates with the network through the REST server running on port 3000.

Congratulations! You just set up your first blockchain business network using Hyperledger Fabric and Hyperledger Composer :D

You can add more features to the cards trading network, setting prices on the cards and giving a balance to all Trader. You can also have more transactions which allow the Traders to toggle the value of forTrade . You can integrate this with non blockchain applications and allow users to buy new cards which get added to their account, which they can then further trade on the network.

The possibilities are endless, what will you make of them? Let me know in the comments :D

1. Get a modal to open when you press the button

The first change we need to make is have the button open the modal window. The code already contains the required modal window, the button is just missing the (click) and data-target attributes.

To resolve this, open up /cards-trading-angular-app/src/app/TradeCard/TradeCard.component.html

The file name can vary based on your transaction name. If you have multiple transactions in your business network, you'll have to do this change across all the transaction resource type HTML files.

Scroll down till the very end and you shall see a <button> tag. Go ahead and add these two attributes to that tag:

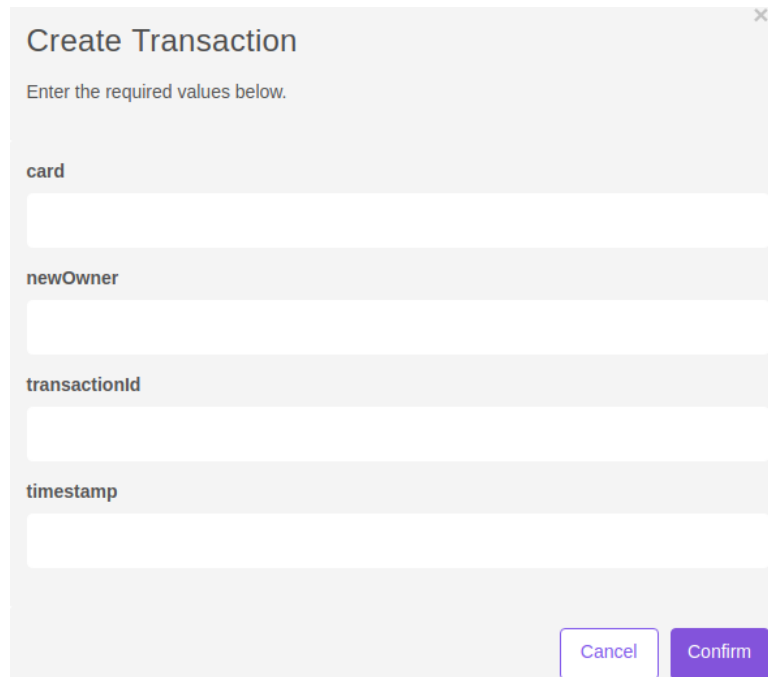
(click)="resetForm();" data-target="#addTransactionModal"

so the line looks like this:

```
<button type="button" class="btn btn-primary invokeTransactionBtn" data-toggle="modal"
(click)="resetForm();" data-target="#addTransactionModal">Invoke</button>
```

The (click) attribute calls resetForm(); which sets all the input fields to empty, and data-target specifies the modal window to be opened upon click.

Save the file, open your browser, and try pressing the invoke button. It should open this modal:

A modal window titled "Create Transaction" with a close button (X) in the top right corner. Below the title is a subtitle "Enter the required values below." The form contains four text input fields, each with a label to its left: "card", "newOwner", "transactionId", and "timestamp". At the bottom right of the modal are two buttons: "Cancel" and "Confirm".

2. Removing unnecessary fields

Just getting the modal to open isn't enough. We can see it requests `transactionId` and `timestamp` from us even though we didn't add those fields in our modeling file. Our network stores these values which are intrinsic to all transactions. So, it should be able to figure out these values on its own. And as it turns out, it actually does. These are spare fields and we can just comment them out, the REST API will handle the rest for us.

In the same file, scroll up to find the input fields and comment out the divs responsible for those input fields inside `addTransactionModal`

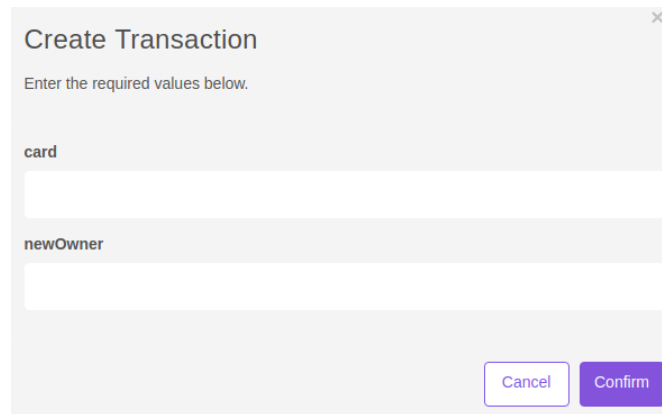
```
<!-- <div class="form-group text-left">
  <label for="transactionId">transactionId</label>

  <input formControlName="transactionId" type="text" class="form-control">
</div>

<div class="form-group text-left">
  <label for="timestamp">timestamp</label>

  <input formControlName="timestamp" type="text" class="form-control">
</div> -->
```

Save your file, open your browser, and press Invoke. You should see this:

A dialog box titled "Create Transaction" with a close button (X) in the top right corner. Below the title is the instruction "Enter the required values below." There are two input fields: the first is labeled "card" and the second is labeled "newOwner". At the bottom right, there are two buttons: "Cancel" and "Confirm".

Create Transaction

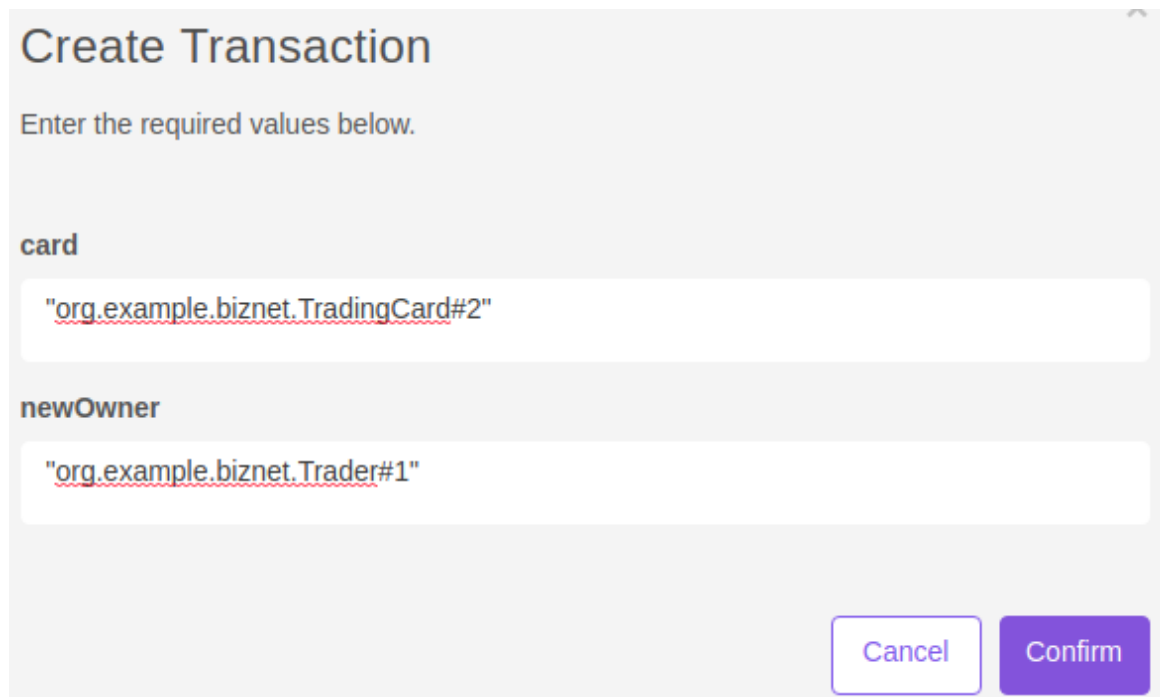
Enter the required values below.

card

newOwner

Cancel Confirm

You can now create transactions here by passing data in these fields. Since card and newOwner are relationships to other resources, we can do a transaction like this:

A dialog box titled "Create Transaction" with a close button (X) in the top right corner. Below the title is the instruction "Enter the required values below." There are two input fields: the first is labeled "card" and contains the value "org.example.biznet.TradingCard#2"; the second is labeled "newOwner" and contains the value "org.example.biznet.Trader#1". At the bottom right, there are two buttons: "Cancel" and "Confirm".

Create Transaction

Enter the required values below.

card

org.example.biznet.TradingCard#2

newOwner

org.example.biznet.Trader#1

Cancel Confirm

Press **Confirm**, go back to the **Assets** page, and you will see that TradingCard#2 now belongs to Trader#1:

Result:

Thus the Interact with a blockchain network and execute transactions by creating app are executed successfully.

EX.NO:6 Create and deploy a blockchain network using Hyperledger Fabric SDK for Java

Date:

Aim:

To create and deploy a blockchain network using Hyperledger Fabric SDK for Java

Procedure:

Set up and initialize the channel, install and instantiate chaincode, and perform invoke and query on your blockchain network

Blockchain is a shared, immutable ledger for recording the history of transactions. The Linux Foundation's Hyperledger Fabric, the software implementation of blockchain IBM is committed to, is a permissioned network. Hyperledger Fabric is a platform for distributed ledger solutions underpinned by a modular architecture delivering high degrees of confidentiality, resiliency, flexibility and scalability.

In a Blockchain solution, the Blockchain network works as a back-end with an application front-end to communicate with the network using a SDK. To set up the communication between front-end and back-end, Hyperledger Fabric community offers a number of SDKs for a wide variety of programming languages like the NodeJS SDK and Java SDK. This code pattern explains the methodology to create, deploy and test the blockchain network using Hyperledger Fabric SDK Java.

It would be helpful for the Java developers, who started to look into Hyperledger Fabric platform and would like to use Fabric SDK Java for their projects. The SDK helps facilitate Java applications to manage the lifecycle of Hyperledger channels and user chaincode. The SDK also provides a means to execute user chaincode, query blocks and transactions on the channel, and monitor events on the channel. This code pattern will help to get the process started to build a Hyperledger Fabric v1.4.1 Java application.

When the reader has completed this pattern, they will understand how to create, deploy and test a blockchain network using Hyperledger Fabric SDK Java. This pattern will provision a Hyperledger Fabric 1.4.1 network consisting of two organizations, each maintaining two peer node, two certificate authorities (ca) for each organization and a solo ordering service. The following aspects will be demonstrated in this code pattern:

- Create and initialize channel
- Install and instantiate chain code
- Register and enroll the users
- Perform invoke and query on the blockchain network.

Steps

1. Setup the Blockchain Network

Clone this repo using the following command.

\$ git clone <https://github.com/IBM/blockchain-application-using-fabric-java-sdk>

To build the blockchain network, the first step is to generate artifacts for peers and channels using cryptogen and configtx. The utilities used and steps to generate artifacts are explained here. In this pattern all required artifacts for the peers and channel of the network are already generated and provided to use as-is. Artifacts can be located at:

network_resources/crypto-config

network_resources/config

The automated scripts to build the network are provided under network directory. The network/docker-compose.yaml file defines the blockchain network topology.

cd network

chmod +x build.sh

./build.sh

To stop the running network, run the following script.

cd network

chmod +x stop.sh

./stop.sh

To delete the network completely, following script need to execute.

cd network

chmod +x teardown.sh

./teardown.sh

2. Build the client based on Fabric Java SDK

The previous step creates all required docker images with the appropriate configuration.

Java Client

The java client sources are present in the folder java of the repo.

Check your environment before executing the next step. Make sure, you are able to run mvn commands properly.

If mvn commands fails, please refer to Pre-requisites to install maven.

To work with the deployed network using Hyperledger Fabric SDK java 1.4.1, perform the following steps.

Open a command terminal and navigate to the java directory in the repo. Run the command mvn install.

cd ../java

mvn install

A jar file blockchain-java-sdk-0.0.1-SNAPSHOT-jar-with-dependencies.jar is built and can be found under the target folder. This jar can be renamed to blockchain-client.jar to keep the name short.

cd target

cp blockchain-java-sdk-0.0.1-SNAPSHOT-jar-with-dependencies.jar blockchain-client.jar

Copy this built jar into network_resources directory. This is required as the java code can access required artifacts during execution.

cp blockchain-client.jar ../../network_resources

3. Create and Initialize the channel

In this code pattern, we create one channel mychannel which is joined by all four peers. The java source code can be seen at src/main/java/org/example/network/CreateChannel.java. To create and initialize the channel, run the following command.

cd ../../network_resources

java -cp blockchain-client.jar org.example.network.CreateChannel

Output:

INFO: Deleting - users

Apr 20, 2018 5:11:45 PM org.example.network.CreateChannel main

INFO: Channel created mychannel

Apr 20, 2018 5:11:45 PM org.example.network.CreateChannel main

INFO: peer0.org1.example.com at grpc://localhost:7051

Apr 20, 2018 5:11:45 PM org.example.network.CreateChannel main

INFO: peer1.org1.example.com at grpc://localhost:7056

Apr 20, 2018 5:11:45 PM org.example.network.CreateChannel main

INFO: peer0.org2.example.com at grpc://localhost:8051

Apr 20, 2018 5:11:45 PM org.example.network.CreateChannel main

INFO: peer1.org2.example.com at grpc://localhost:8056

4. Deploy and Instantiate the chaincode

This code pattern uses a sample chaincode fabcar to demo the usage of Hyperledger Fabric SDK Java APIs. To deploy and instantiate the chaincode, execute the following command.

java -cp blockchain-client.jar org.example.network.DeployInstantiateChaincode

Output:

INFO: Deploying chaincode fabcar using Fabric client Org1MSP admin

Apr 23, 2018 10:25:22 AM org.example.network.DeployInstantiateChaincode main

INFO: fabcar- Chain code deployment SUCCESS

Apr 23, 2018 10:25:22 AM org.example.network.DeployInstantiateChaincode main

INFO: fabcar- Chain code deployment SUCCESS

Apr 23, 2018 10:25:22 AM org.example.client.FabricClient deployChainCode

INFO: Deploying chaincode fabcar using Fabric client Org2MSP admin

Apr 23, 2018 10:25:22 AM org.example.network.DeployInstantiateChaincode main

INFO: fabcar- Chain code deployment SUCCESS

Apr 23, 2018 10:25:22 AM org.example.network.DeployInstantiateChaincode main

INFO: fabcar- Chain code deployment SUCCESS

Apr 23, 2018 10:25:22 AM org.example.client.ChannelClient instantiateChainCode

INFO: Instantiate proposal request fabcar on channel mychannel with Fabric client Org2MSP admin

Apr 23, 2018 10:25:22 AM org.example.client.ChannelClient instantiateChainCode

INFO: Instantiating Chaincode ID fabcar on channel mychannel

Apr 23, 2018 10:25:25 AM org.example.client.ChannelClient instantiateChainCode

INFO: Chaincode fabcar on channel mychannel instantiation

java.util.concurrent.CompletableFuture@723ca036[Not completed]

Apr 23, 2018 10:25:25 AM org.example.network.DeployInstantiateChaincode main

INFO: fabcar- Chain code instantiation SUCCESS

Apr 23, 2018 10:25:25 AM org.example.network.DeployInstantiateChaincode main

INFO: fabcar- Chain code instantiation SUCCESS

Apr 23, 2018 10:25:25 AM org.example.network.DeployInstantiateChaincode main

INFO: fabcar- Chain code instantiation SUCCESS

Apr 23, 2018 10:25:25 AM org.example.network.DeployInstantiateChaincode main

INFO: fabcar- Chain code instantiation SUCCESS

Note: The chaincode fabcar.go was taken from the fabric samples available at -
<https://github.com/hyperledger/fabric-samples/tree/release-1.4/chaincode/fabcar/go>.

5. Register and enroll users

A new user can be registered and enrolled to an MSP. Execute the below command to register a new user and enroll to Org1MSP.

java -cp blockchain-client.jar org.example.user.RegisterEnrollUser

Output:

INFO: Deleting - users

log4j:WARN No appenders could be found for logger (org.hyperledger.fabric.sdk.helper.Config).

log4j:WARN Please initialize the log4j system properly.

log4j:WARN See <https://logging.apache.org/log4j/1.2/faq.html#noconfig> for more info.

Apr 23, 2018 10:26:35 AM org.example.client.CAClient enrollAdminUser

INFO: CA -http://localhost:7054 Enrolled Admin.

Apr 23, 2018 10:26:35 AM org.example.client.CAClient registerUser

INFO: CA -http://localhost:7054 Registered User - user1524459395783

Apr 23, 2018 10:26:36 AM org.example.client.CAClient enrollUser

INFO: CA -http://localhost:7054 Enrolled User - user1524459395783

6. Perform Invoke and Query on network

Blockchain network has been setup completely and is ready to use. Now we can test the network by performing invoke and query on the network. The fabcar chaincode allows us to create a new asset which is a car. For test purpose, invoke operation is performed to create a new asset in the network and query operation is performed to list the asset of the network. Perform the following steps to check the same.

java -cp blockchain-client.jar org.example.chaincode.invocation.InvokeChaincode

Output:

INFO: CA -http://localhost:7054 Enrolled Admin.

Apr 20, 2018 5:13:04 PM org.example.client.ChannelClient sendTransactionProposal

INFO: Sending transaction proposal on channel mychannel

Apr 20, 2018 5:13:04 PM org.example.client.ChannelClient sendTransactionProposal

INFO: Transaction proposal on channel mychannel OK SUCCESS with transaction

id:a298b9e27bdb0b6ca18b19f9c78a5371fb4d9b8dd199927baf37379537ca0d0f

Apr 20, 2018 5:13:04 PM org.example.client.ChannelClient sendTransactionProposal

INFO:

Apr 20, 2018 5:13:04 PM org.example.client.ChannelClient sendTransactionProposal

INFO: java.util.concurrent.CompletableFuture@22f31dec[Not completed]

Apr 20, 2018 5:13:04 PM org.example.chaincode.invocation.InvokeChaincode main

INFO: Invoked createCar on fabcar. Status - SUCCESS

```
java -cp blockchain-client.jar org.example.chaincode.invocation.QueryChaincode
```

Output:

Apr 20, 2018 5:13:28 PM org.example.client.CAClient enrollAdminUser

INFO: CA -http://localhost:7054 Enrolled Admin.

Apr 20, 2018 5:13:29 PM org.example.chaincode.invocation.QueryChaincode main

INFO: Querying for all cars ...

Apr 20, 2018 5:13:29 PM org.example.client.ChannelClient queryByChainCode

INFO: Querying queryAllCars on channel mychannel

Apr 20, 2018 5:13:29 PM org.example.chaincode.invocation.QueryChaincode main

INFO: [{"Key": "CAR1",
"Record": {"make": "Chevy", "model": "Volt", "colour": "Red", "owner": "Nick"}}]

Apr 20, 2018 5:13:39 PM org.example.chaincode.invocation.QueryChaincode main

INFO: Querying for a car - CAR1

Apr 20, 2018 5:13:39 PM org.example.client.ChannelClient queryByChainCode

INFO: Querying queryCar on channel mychannel

Apr 20, 2018 5:13:39 PM org.example.chaincode.invocation.QueryChaincode main

INFO: {"make": "Chevy", "model": "Volt", "colour": "Red", "owner": "Nick"}

Program:

```
import java.io.IOException;
```

```
import java.nio.charset.StandardCharsets;
```

```
import java.nio.file.Path;
```

```
import java.nio.file.Paths;
```

```
import java.util.concurrent.TimeoutException;
```

```
import org.hyperledger.fabric.gateway.Contract;
```

```
import org.hyperledger.fabric.gateway.ContractException;
```

```
import org.hyperledger.fabric.gateway.Gateway;
```

```

import org.hyperledger.fabric.gateway.Network;

import org.hyperledger.fabric.gateway.Wallet;

import org.hyperledger.fabric.gateway.Wallets;


class Sample {

    public static void main(String[] args) throws IOException {

        // Load an existing wallet holding identities used to access the network.

        Path walletDirectory = Paths.get("wallet");

        Wallet wallet = Wallets.newFileSystemWallet(walletDirectory);


        // Path to a common connection profile describing the network.

        Path networkConfigFile = Paths.get("connection.json");


        // Configure the gateway connection used to access the network.

        Gateway.Builder builder = Gateway.createBuilder()

            .identity(wallet, "user1")

            .networkConfig(networkConfigFile);


        // Create a gateway connection

        try (Gateway gateway = builder.connect()) {

            // Obtain a smart contract deployed on the network.

            Network network = gateway.getNetwork("mychannel");

            Contract contract = network.getContract("fabcar");


            // Submit transactions that store state to the ledger.

            byte[] createCarResult = contract.createTransaction("createCar")

                .submit("CAR10", "VW", "Polo", "Grey", "Mary");

```

```
System.out.println(new String(createCarResult, StandardCharsets.UTF_8));
```

```
// Evaluate transactions that query state from the ledger.
```

```
byte[] queryAllCarsResult = contract.evaluateTransaction("queryAllCars");
```

```
System.out.println(new String(queryAllCarsResult, StandardCharsets.UTF_8));
```

```
} catch (ContractException | TimeoutException | InterruptedException e) {
```

```
    e.printStackTrace();
```

```
}
```

```
}
```

```
}
```

Result:

Thus the creation and deploy a blockchain network using Hyperledger Fabric SDK for Java are executed successfully.

Ex:No: 7

Deploy an asset-transfer app using blockchain within Hyperledger Fabric network

Date:

Aim: To Deploy an asset-transfer app using blockchain. Learn app development within a Hyperledger Fabric network.

Procedure:

The private asset transfer smart contract is deployed with an endorsement policy that requires an endorsement from any channel member. This allows any organization to create an asset that they own without requiring an endorsement from other channel members. The creation of the asset is the only transaction that uses the chaincode level endorsement policy. Transactions that update or transfer existing assets will be governed by state based endorsement policies or the endorsement policies of private data collections.

About Asset Transfer

This Asset Transfer (basic) sample demonstrates how to initialize a ledger with assets, query those assets, create a new asset, query a single asset based on an asset ID, update an existing asset, and transfer an asset to a new owner. It involves the following two components:

1. Sample application: which makes calls to the blockchain network, invoking transactions implemented in the chaincode (smart contract). The application is located in the following fabric- samples directory:

```
asset-transfer-basic/application-javascript
```

2. Smart contract itself, implementing the transactions that involve interactions with the ledger. The smart contract (chaincode) is located in the following fabric-samples directory

```
asset-transfer-basic/chaincode-(javascript, java, go, typescript)
```

3. Explore a sample smart contract. We'll inspect the sample assetTransfer (javascript) smart contract to learn about the transactions within it, and how they are used by an application to query and update the ledger.

3. Interact with the smart contract with a sample application. Our application will use the asset Transfer smart contract to create, query, and update assets on the ledger. We'll get into the code of the app and the transactions they create, including initializing the ledger with assets, querying an asset, querying a range of assets, creating a new asset, and transferring an asset to a new owner.

Install the Hyperledger Fabric SDK for Node.js.

If you are on Windows, you can install the windows-build-tools with npm which installs all required compilers and tooling by running the following command

```
npm install --global windows-build-tools
```

Set up the blockchain network

Navigate to the `test-network` subdirectory within your local clone of the `fabric-samples` repository.

```
cd fabric-samples/test-network
```

If you already have a test network running, bring it down to ensure the environment is clean.

```
./network.sh down
```

Launch the Fabric test network using the `network.sh` shell script.

```
./network.sh up createChannel -c mychannel -ca
```

Next, let's deploy the chaincode by calling the `./network.sh` script with the chaincode name and language options.

```
./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-javascript/ -ccl javascript
```

If the chaincode is successfully deployed, the end of the output in your terminal should look similar to below:

```
Committed chaincode definition for chaincode 'basic' on channel 'mychannel':
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc, Approvals: [Org1MSP:
===== Query chaincode definition successful on peer0.org2 on channel 'mychannel' =====
===== Chaincode initialization is not required =====
```

Sample application

Next, let's prepare the sample Asset Transfer Javascript application that will be used to interact with the deployed chaincode.

JavaScript Application

Open a new terminal, and navigate to the `application-javascript` folder.

```
cd asset-transfer-basic/application-javascript
```

This directory contains sample programs that were developed using the Fabric SDK for Node.js. Run the following command to install the application dependencies. It may take up to a minute to complete:

```
npm install
```

This process is installing the key application dependencies defined in the application's `package.json`. The most important of which is the `fabric-network` Node.js module; it enables an application to use identities, wallets, and gateways to connect to channels, submit transactions, and wait for notifications. This tutorial also uses the `fabric-ca-client` module to enroll users with their respective certificate authorities, generating a valid identity which is then used by the `fabric-network` module to interact with the blockchain network.

Once `npm install` completes, everything is in place to run the application. Let's take a look at the sample JavaScript application files we will be using in this tutorial. Run the following command to list the files in this directory:

```
ls
```

```
app.js          node_modules    package.json    package-lock.json
```

Let's run the application and then step through each of the interactions with the smart contract functions. From the `asset-transfer-basic/application-javascript` directory, run the following command:

```
node app.js
```

In the sample application code below, you will see that after getting reference to the common connection profile path, making sure the connection profile exists, and specifying where to create

the wallet, `enrollAdmin()` is executed and the admin credentials are generated from the Certificate Authority.

```

async function main() {
  try {

    // build an in memory object with the network configuration (also known as a connection profile)
    const ccp = buildCCP();

    // build an instance of the fabric ca services client based on
    // the information in the network configuration
    const caClient = buildCAClient(FabricCAServices, ccp);

    // setup the wallet to hold the credentials of the application user
    const wallet = await buildWallet(Wallets, walletPath);

    // in a real application this would be done on an administrative flow, and only once
    await enrollAdmin(caClient, wallet);
  }
}

```

This command stores the CA administrator's credentials in the **wallet** directory. You can find administrator's certificate and private key in the **wallet/admin.id** file

```

Wallet path: /Users/<your_username>/fabric-samples/asset-transfer-basic/application-javascript/wallet
Successfully enrolled admin user and imported it into the wallet

```

Because the admin registration step is bootstrapped when the Certificate Authority is started, we only need to enroll the admin.

Second, the application registers and enrolls an application user

Now that we have the administrator's credentials in a wallet, the application uses the **admin** user to register and enroll an app user which will be used to interact with the blockchain network. The section of the application code is shown below.

```

// in a real application this would be done only when a new user was required to be added
// and would be part of an administrative flow
await registerAndEnrollUser(caClient, wallet, mspOrg1, org1UserId, 'org1.department1');

```

Scrolling further down in your terminal output, you should see confirmation of the app user registration similar to this:

```

Successfully registered and enrolled user appUser and imported it into the wallet

```

You will notice that in the following lines of application code, the application is getting reference to the Contract using the contract name and channel name via Gateway:

```

// Create a new gateway instance for interacting with the fabric network.
// In a real application this would be done as the backend server session is setup for
// a user that has been verified.

```

```

const gateway = new Gateway();

try {
  // setup the gateway instance
  // The user will now be able to create connections to the fabric network and be able to
  // submit transactions and query. All transactions submitted by this gateway will be
  // signed by this user using the credentials stored in the wallet.
  await gateway.connect(ccp, {
    wallet,
    identity: userId,
    discovery: {enabled: true, asLocalhost: true} // using asLocalhost as this gateway is using a
    fabric network deployed locally
  });

  // Build a network instance based on the channel where the smart contract is deployed
  const network = await gateway.getNetwork(channelName);

  // Get the contract from the network.
  const contract = network.getContract(chaincodeName);

```

When a chaincode package includes multiple smart contracts, on the `getContract()` API you can specify both the name of the chaincode package and a specific smart contract to target. For example:

```
const contract = await network.getContract('chaincodeName', 'smartContractName');
```

Fourth, the application initializes the ledger with some sample data

The `submitTransaction()` function is used to invoke the chaincode `InitLedger` function to populate the ledger with some sample data.

Sample application `InitLedger` call

```

// Initialize a set of asset data on the channel using the chaincode 'InitLedger' function.
// This type of transaction would only be run once by an application the first time it was started after
it
// deployed the first time. Any updates to the chaincode deployed later would likely not need to run
// an "init" type function.
console.log("\n--> Submit Transaction: InitLedger function creates the initial set of assets on the
ledger");
await contract.submitTransaction('InitLedger');

```

```
console.log('*** Result: committed');
```

Chaincode `'InitLedger'` function

```
async InitLedger(ctx) {
  const assets = [
    {
      ID: 'asset1',
      Color: 'blue',
      Size: 5,
      Owner: 'Tomoko',
      AppraisedValue: 300,
    },
    {
      ID: 'asset2',
      Color: 'red',
      Size: 5,
      Owner: 'Brad',
      AppraisedValue: 400,
    },
    {
      ID: 'asset3',
      Color: 'green',
      Size: 10,
      Owner: 'Jin Soo',
      AppraisedValue: 500,
    },
    {
      ID: 'asset4',
      Color: 'yellow',
      Size: 10,
      Owner: 'Max',
      AppraisedValue: 600,
    },
    {
      ID: 'asset5',
      Color: 'black',
      Size: 15,
      Owner: 'Adriana',
      AppraisedValue: 700,
    },
    {
      ID: 'asset6',
      Color: 'white',
      Size: 15,
      Owner: 'Michel',
      AppraisedValue: 800,
    },
  ];

  for (const asset of assets) {
    asset.docType = 'asset';
    await ctx.stub.putState(asset.ID, Buffer.from(JSON.stringify(asset)));

    console.info(`Asset ${asset.ID} initialized`);
  }
}
```

The terminal output entry should look similar to below:

```
Submit Transaction: InitLedger, function creates the initial set of assets on the ledger
```

Fifth, the application invokes each of the chaincode functions

Sample application `'GetAllAssets'` call

```
// Let's try a query type operation (function).  
// This will be sent to just one peer and the results will be shown.  
console.log('\n--> Evaluate Transaction: GetAllAssets, function returns all the current assets on t  
let result = await contract.evaluateTransaction('GetAllAssets');  
console.log(`*** Result: ${prettyJSONString(result.toString())}`);
```

`'GetAllAssets'` function

```
// GetAllAssets returns all assets found in the world state.  
async GetAllAssets(ctx) {  
  const allResults = [];  
  // range query with empty string for startKey and endKey does an open-ended query of all asset  
  const iterator = await ctx.stub.getStateByRange('', '');  
  let result = await iterator.next();  
  while (!result.done) {  
    const strValue = Buffer.from(result.value.value.toString()).toString('utf8');  
    let record;  
    try {  
      record = JSON.parse(strValue);  
    } catch (err) {  
      console.log(err);  
      record = strValue;  
    }  
    allResults.push({ Key: result.value.key, Record: record });  
    result = await iterator.next();  
  }  
  return JSON.stringify(allResults);  
}
```

Evaluate Transaction:GetAllAssets,function returns all the current assets on the ledger Result:[
{


```

    "Key": "asset1",
    "Record": {
      "ID": "asset1",
      "Color": "blue",
      "Size": 5,
      "Owner": "Tomoko",
      "AppraisedValue": 300,
      "docType": "asset"
    }
  },
  {
    "Key": "asset2",
    "Record": {
      "ID": "asset2",
      "Color": "red",
      "Size": 5,
      "Owner": "Brad",
      "AppraisedValue": 400,
      "docType": "asset"
    }
  },
  {
    "Key": "asset3",
    "Record": {
      "ID": "asset3",
      "Color": "green",
      "Size": 10,
      "Owner": "Jin Soo",
      "AppraisedValue": 500,
      "docType": "asset"
    }
  },
  {
    "Key": "asset4",
    "Record": {
      "ID": "asset4",
      "Color": "yellow",
      "Size": 10,
      "Owner": "Max",
      "AppraisedValue": 600,
      "docType": "asset"
    }
  },
  {
    "Key": "asset5",
    "Record": {
      "ID": "asset5",
      "Color": "black",
      "Size": 15,
      "Owner": "Adriana",
      "AppraisedValue": 700,
      "docType": "asset"
    }
  }
}

```

```

    }
  },
  {
    "Key": "asset6",
    "Record": { "ID":
"asset6",

    "Color": "white",
    "Size": 15, "Owner":
"Michel",

    "AppraisedValue": 800,

    "docType": "asset"
  }
}
]

```

Next, the sample application submits a transaction to create ‘asset13’.

Sample application **'CreateAsset'** Call

Chaincode **'CreateAsset'** function

```

// CreateAsset issues a new asset to the world state with given details.
async CreateAsset(ctx, id, color, size, owner, appraisedValue) {
  const asset = {
    ID: id,
    Color: color,
    Size: size,
    Owner: owner,
    AppraisedValue: appraisedValue,
  };
  return ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
}

```

Terminal output:

```

Submit Transaction: CreateAsset, creates new asset with ID, color, owner, size, and appraisedValue

```

Sample application **'ReadAsset'** call

Chaincode

```
console.log('\n--> Evaluate Transaction: ReadAsset, function returns an asset with a given assetID');
result = await contract.evaluateTransaction('ReadAsset', 'asset13');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);
```

'ReadAsset' function

```
// ReadAsset returns the asset stored in the world state with given id.
async ReadAsset(ctx, id) {
  const assetJSON = await ctx.stub.getState(id); // get the asset from chaincode state
  if (!assetJSON || assetJSON.length === 0) {
    throw new Error(`The asset ${id} does not exist`);
  }
  return assetJSON.toString();
}
```

Terminal output:

```
Evaluate Transaction: ReadAsset, function returns an asset with a given assetID
Result: {
  "ID": "asset13",
  "Color": "yellow",
  "Size": "5",
  "Owner": "Tom",
  "AppraisedValue": "1300"
}
```

'UpdateAsset' call

```
try {
  // How about we try a transactions where the executing chaincode throws an error:
  // Notice how the submitTransaction will throw an error containing the error thrown by the chaincode
  console.log('\n--> Submit Transaction: UpdateAsset asset70, asset70 does not exist and should return an error');
  await contract.submitTransaction('UpdateAsset', 'asset70', 'blue', '5', 'Tomoko', '300');
  console.log('***** FAILED to return an error');
} catch (error) {
  console.log(`*** Successfully caught the error: \n  ${error}`);
}
```

Chaincode 'UpdateAsset' function

```
// UpdateAsset updates an existing asset in the world state with provided parameters.
async UpdateAsset(ctx, id, color, size, owner, appraisedValue) {
  const exists = await this.AssetExists(ctx, id);
  if (!exists) {
    throw new Error(`The asset ${id} does not exist`);
  }

  // overwriting original asset with new asset
  const updatedAsset = {
    ID: id,
    Color: color,
    Size: size,
    Owner: owner,
    AppraisedValue: appraisedValue,
  };
  return ctx.stub.putState(id, Buffer.from(JSON.stringify(updatedAsset)));
}
```

Terminal output:

```
Submit Transaction: UpdateAsset asset70
2020-08-02T11:12:12.322Z - error: [Transaction]: Error: No valid responses from any peers. Errors:
  peer=peer0.org1.example.com:7051, status=500, message=error in simulation: transaction returned w
  peer=peer0.org2.example.com:9051, status=500, message=error in simulation: transaction returned w
Expected an error on UpdateAsset of non-existing Asset: Error: No valid responses from any peers. E
  peer=peer0.org1.example.com:7051, status=500, message=error in simulation: transaction returned w
  peer=peer0.org2.example.com:9051, status=500, message=error in simulation: transaction returned w
```

When you are finished using the asset-transfer sample, you can bring down the test network using `network.sh` script.

```
./network.sh down
```

Result:

Thus the deployment of an asset-transfer app using blockchain within a Hyperledger Fabric network are executed successfully.

Ex: No:8

Fitness Club rewards using Hyperledger Fabric

Date:

Aim:

To build a web app that tracks fitness club rewards using Hyperledger Fabric
several steps:

Procedure:

1. Set up Hyperledger Fabric network.
2. Define smart contracts for handling rewards.
3. Develop a web application frontend.
4. Connect the frontend to the Hyperledger Fabric network.

1. Set up Hyperledger Fabric network:

You need to set up a Hyperledger Fabric network with a few nodes. Refer to the official documentation for detailed instructions.

2. Define smart contracts:

Define smart contracts to handle fitness club rewards. Here's a simple example in Go:

```
package main

import (
    "encoding/json"
    "fmt"
    "github.com/hyperledger/fabric-contract-api-go/contractapi"
)

type RewardsContract struct {
    contractapi.Contract
}

type Reward struct {
    MemberID string `json:"memberID"`
    Points int `json:"points"`
}

func (rc *RewardsContract) IssueReward(ctx contractapi.TransactionContextInterface, memberID string,
points int) error {
```

```

reward := Reward{
    MemberID: memberID,
    Points: points,
}

rewardJSON, err := json.Marshal(reward)

if err != nil {
    return err
}

return ctx.GetStub().PutState(memberID, rewardJSON)
}

func (rc *RewardsContract) GetReward(ctx contractapi.TransactionContextInterface, memberID string)
(*Reward, error) {
    rewardJSON, err := ctx.GetStub().GetState(memberID)

    if err != nil {
        return nil, err
    }

    if rewardJSON == nil {
        return nil, fmt.Errorf("reward for member %s not found", memberID)
    }

    var reward Reward

    err = json.Unmarshal(rewardJSON, &reward)

    if err != nil {
        return nil, err
    }

    return &reward, nil
}

```

3. Develop a web application frontend:

You can use any frontend framework like React, Vue.js, etc. Here's a simple React component to interact with the smart contract:

RewardsComponent.js

```

import React, { useState } from 'react';
import { useContract } from './useContract'; // Assume this hook connects to the contract
const RewardsComponent = () => {

```



```

const [memberID, setMemberID] = useState("");
const [points, setPoints] = useState("");
const { issueReward, getReward } = useContract();
const handleIssueReward = async () => {
  await issueReward(memberID, points);
};
const handleGetReward = async () => {
  const reward = await getReward(memberID);
  console.log(reward);
};
return (
  <div>
    <input type="text" placeholder="Member ID" value={memberID} onChange={(e) =>
setMemberID(e.target.value)} />
    <input type="number" placeholder="Points" value={points} onChange={(e) =>
setPoints(e.target.value)} />
    <button onClick={handleIssueReward}>Issue Reward</button>
    <button onClick={handleGetReward}>Get Reward</button>
  </div>
);
};
export default RewardsComponent;

```

4. Connect the frontend to the Hyperledger Fabric network:

Use a library like fabric-network to interact with the Hyperledger Fabric network. Implement functions like issueReward and getReward to interact with the smart contract.

Now, integrate this frontend component into your web application. Here's a screenshot of what the UI might look like:

In this example, users can input a member ID and points to issue rewards, and they can retrieve rewards by providing the member ID. Remember, this is a basic example. In a real-world application, you would need to consider security, scalability, and other factors. Additionally, you'll need to handle user authentication, authorization, and other functionalities as per your requirement.

Result:

Thus the blockchain to track fitness club rewards and build a web app that uses Hyperledger Fabric to track and trace member rewards are executed successfully.

Ex no: 9 Create and deploy a simple smart contract on the Kadena blockchain.

Date :

Aim:

To create and deploy a simple smart contract on the Kadena blockchain that manages a basic asset ledger.

Algorithm:

Setup Development Environment:

- Install Pact (Kadena's smart contract language).
- Use the Kadena testnet for deployment.

Write a Smart Contract:

- Define a module.
- Implement a function to store and retrieve assets.

Deploy the Contract:

- Use `pact -l` to load the contract locally or deploy on testnet using the Kadena API.

Interact with the Contract:

- Call functions to add and retrieve assets.

Program (Smart Contract in Pact)

```
(module simple-ledger G
  (defcap G ()
    true)

  (defschema asset-schema
    asset-name: string
    asset-value: integer)

  (deftable asset-table:{asset-schema})

  (defun create-asset (name:string value:integer)
    "Creates a new asset in the ledger"
    (insert asset-table name { "asset-name": name, "asset-value": value })
  )

  (defun get-asset (name:string)
    "Retrieves asset details"
    (read asset-table name)
  )
)
```

Sample Output

After deploying the contract and calling functions:

Sample Output

After deploying the contract and calling functions:

1. Adding an Asset

```
json  
  
(create-asset "Gold" 100)
```

Response:

```
json  
  
{ "status": "success", "message": "Asset created" }
```

2. Retrieving an Asset

```
json  
  
(get-asset "Gold")
```

Response:

```
json  
  
{ "asset-name": "Gold", "asset-value": 100 }
```

Result:

- Successfully deployed a **basic asset ledger** smart contract on Kadena.
- Users can **store and retrieve assets** using the Pact language.
- The contract is **secure and scalable** on the Kadena blockchain.

Ex.No.10 Analyze the transaction ledger to understand how Ripple processes and records transactions.

Date:

Analyzing the Ripple Transaction Ledger

Introduction

Ripple (XRP Ledger) is a decentralized blockchain-based system designed for fast and efficient cross-border transactions. Understanding how Ripple processes and records transactions involves analyzing its **ledger structure, consensus mechanism, and transaction flow**.

1. XRP Ledger Structure

Ripple uses a unique ledger system that consists of:

- **Accounts:** Each user has an account with an XRP balance and transaction history.
- **Transactions:** Actions recorded on the ledger, including payments, trust line adjustments, and offers.
- **Objects:** Data structures that store information about accounts, balances, and transactions.

The **XRP Ledger (XRPL)** is a continuously updated record of all transactions, structured as a **sequence of validated ledgers**.

2. Transaction Processing in Ripple

Step 1: Transaction Submission

A user submits a transaction using the **XRP Ledger API** or **RippleNet**. Each transaction contains:

- **Account details** (sender, receiver)
- **Transaction type** (payment, escrow, trustline)
- **Amount and fees**
- **Signature for authorization**

Example transaction in JSON format:

```
{
  "TransactionType": "Payment",
  "Account": "rExampleSenderAddress",
  "Destination": "rExampleReceiverAddress",
  "Amount": "1000000",
  "Fee": "10",
  "Sequence": 100
}
```

3. Consensus Mechanism

Ripple does **not** use Proof-of-Work (PoW) or Proof-of-Stake (PoS). Instead, it relies on the **Ripple Consensus Algorithm (RCA)**:

1. **Transaction Submission:** Users send transactions to the network.
2. **Validation by Validators:** Unique Node List (UNL) validators verify transactions.
3. **Consensus Agreement:** 80% of validators must agree on transaction validity.
4. **Ledger Finalization:** Transactions are added to the next **validated ledger**.

The **average transaction confirmation time** is **3-5 seconds**, making XRP faster than Bitcoin or Ethereum.

4. Transaction Recording in the Ledger

Each transaction in the Ripple ledger is recorded with the following attributes:

- **Ledger Index:** Position in the ledger sequence.
- **Transaction Hash:** Unique identifier for the transaction.
- **Timestamp:** Unix time of transaction processing.
- **Account Changes:** Updates in sender/receiver balances.

Example transaction entry:

```
{
  "ledger_index": 123456,
  "tx": {
    "hash": "A1B2C3D4E5...",
    "Account": "rSenderAddress",
    "Destination": "rReceiverAddress",
    "Amount": "1000000",
    "Fee": "10",
    "Validated": true
  }
}
```

5. Result and Key Takeaways

- **Ripple's ledger updates every few seconds**, allowing real-time transaction finality.
- Transactions are validated using the **Ripple Consensus Algorithm**, ensuring security without mining.
- The ledger maintains a **transparent, immutable record** of all transactions.
- Ripple supports advanced financial features like **trust lines, escrow, and decentralized exchange (DEX) transactions**.

Ex.No.11 Execute transactions on the private contract and observe the privacy features of Quorum.

Date:

Aim

To deploy and execute transactions on a private smart contract using Quorum, and observe its privacy features.

Algorithm

1. **Setup Quorum Network**
 - Install Quorum and Tessera (private transaction manager).
 - Set up a multi-node Quorum network.
2. **Write a Private Smart Contract**
 - Define a simple Solidity smart contract.
 - Enable private transactions between specific participants.
3. **Deploy the Contract Privately**
 - Deploy the contract using one Quorum node while keeping it private from others.
4. **Execute Private Transactions**
 - Perform transactions visible only to selected participants.
 - Verify that other nodes cannot access private transaction details.
5. **Observe Privacy Features**
 - Use quorum-explorer or logs to verify privacy enforcement.

Program (Private Smart Contract in Solidity)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract PrivateStorage {
    string private secretData;
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function setSecret(string memory _data) public {
        require(msg.sender == owner, "Not authorized");
        secretData = _data;
    }

    function getSecret() public view returns (string memory) {
        require(msg.sender == owner, "Not authorized");
        return secretData;
    }
}
```

Deployment and Execution in Quorum

1. Deploy Contract Privately

Using GoQuorum RPC API, deploy the contract from **Node 1** but keep it private to **Node 2**.

```
bash Copy Edit  
  
PRIVATE_FOR='["BULKE203F..."]' # Public key of Node 2  
geth --exec "loadScript('deploy.js')" attach http://127.0.0.1:22000
```

2. Interact with Private Contract

Only **Node 1** and **Node 2** can execute transactions.

```
bash Copy Edit  
  
# Storing private data (executed from Node 1)  
PRIVATE_FOR='["BULKE203F..."]'  
geth --exec "privateContract.setSecret('Confidential Info')" attach http://127.0.0.1:22000
```

3. Attempt Access from Unauthorized Node

A third node (**Node 3**) tries to access the data:

```
bash Copy Edit  
  
geth --exec "privateContract.getSecret()" attach http://127.0.0.1:23000
```

Expected Output:

```
javascript Copy Edit  
  
Error: Not authorized
```

Sample Output

- Transaction on Private Network:

json

Copy

Edit

```
{
  "txHash": "0xabcdef123456...",
  "status": "success",
  "privateFor": ["BULKE203F..."]
}
```

- Unauthorized Access Attempt:

json

Copy

Edit

```
{
  "error": "Not authorized"
}
```

Result:

- Private transactions were successfully executed in Quorum.
- Data was restricted to selected participants (Node 1 and Node 2).
- Unauthorized nodes were unable to access or view private transaction details.

