Crypto currency lab manual

cryptocurrency and Blockchain technologies (Anna University)

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

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

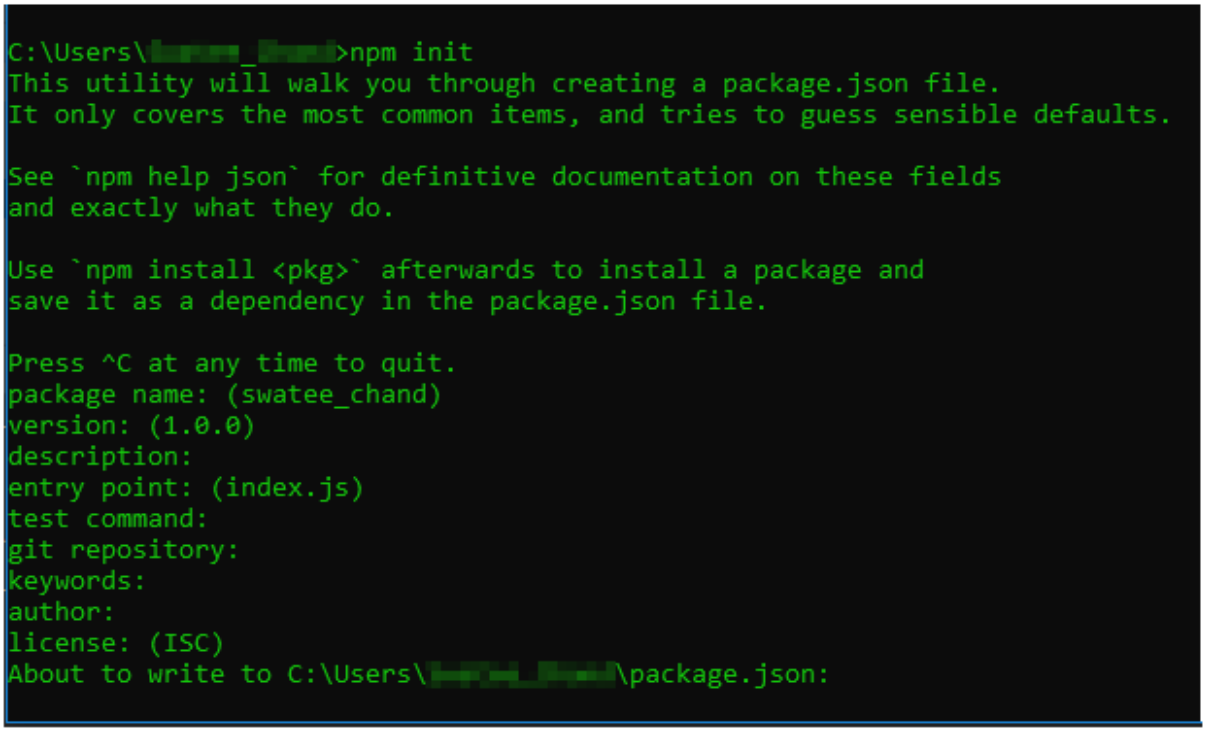
Here, we will be creating a simple CRUD REST application for Library Management using Node.js and Express.js. To build this application, you will need to install the following:

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:



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

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
    # Starting the application using npm start
    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:

```
1   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:

```
1  docker ps
```

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

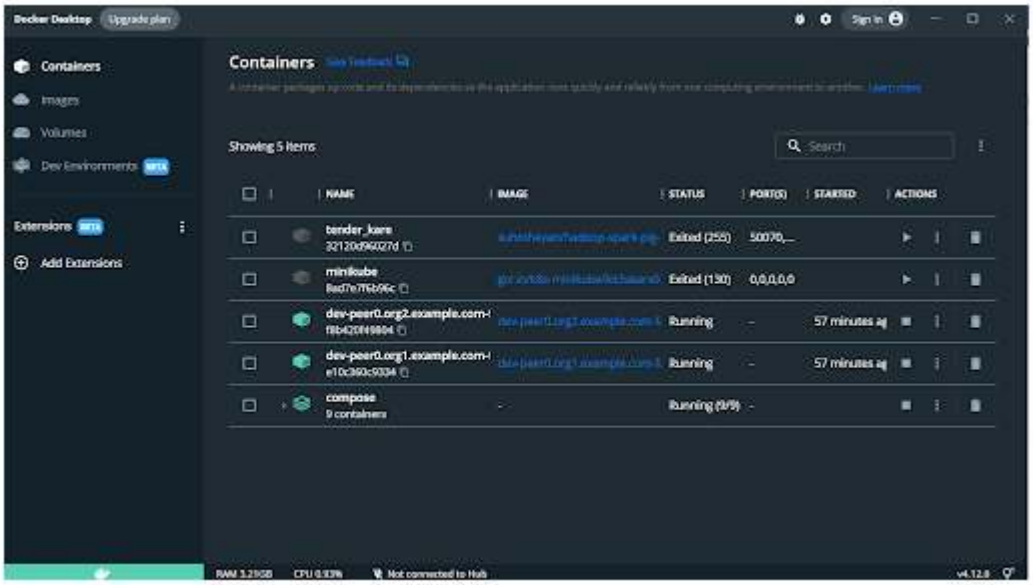
Then you can pull docker containers by running one of these commands:

./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

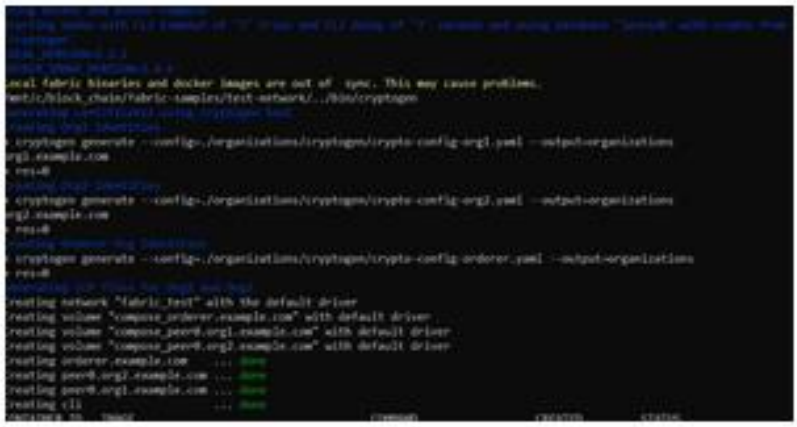
Step 2: From inside this directory, you can directly run ./network.sh script file through which we run the Test Network. By default, we would be running ./network.sh down command to remove any previous network containers or artifacts that still exist. This is to ensure that there are no conflicts when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



Etherum 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.

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
```

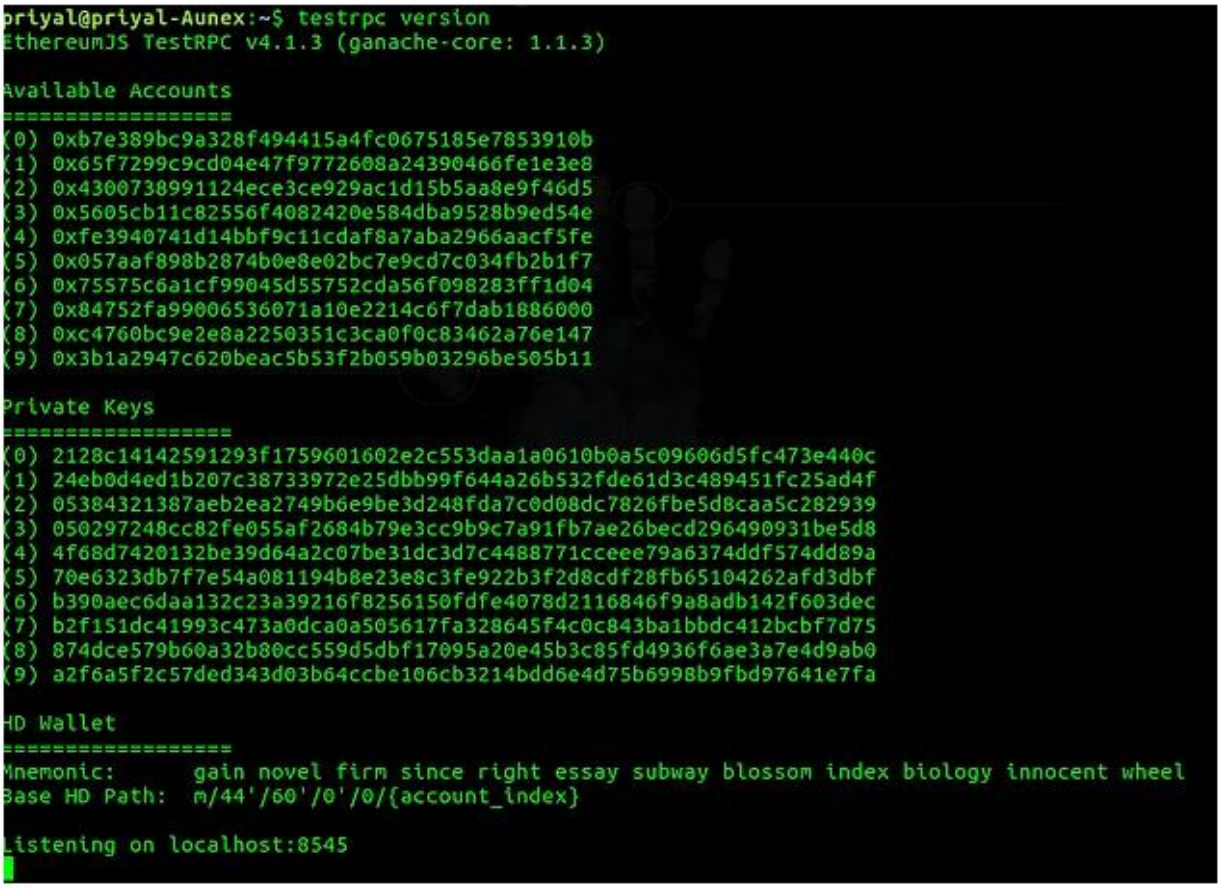


Image 3 : List all localaccounts

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": "0x0000000000000000000000000000000000000000",
    "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/c
haindata 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/l
ightchaindata cache=16 handles=16
INFO [10-26|15:06:19] Successfully wrote genesis state         database=lightchaindata
             hash=272003…b62890
```

```
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
```

Image 5 : Network folder structure

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: {64cee430d15e4a22f32ebc4247993cec540f18a0} keystore:///home/priyal/Eth/pvt/keystore/UTC--2017-10-25T
10-54-33.814607396Z--64cee430d15e4a22f32ebc4247993cec540f18a0
Account #1: {12a14eae31018046036ce82d35cf871773eebc47} keystore:///home/priyal/Eth/pvt/keystore/UTC--2017-10-25T
10-54-56.664136423Z--12a14eae31018046036ce82d35cf871773eebc47
Account #2: {36d4af267acd10d9dfffd1f958a98c732fb9b7e3} keystore:///home/priyal/Eth/pvt/keystore/UTC--2017-10-25T
10-55-02.919902286Z--36d4af267acd10d9dfffd1f958a98c732fb9b7e3
```

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", "0x12a14eae31018046036ce82d35c
f871773eebc47", "0x36d4af267acd10d9dfffd1f958a98c732fb9b7e3"]
>
```

Result:

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

EX.NO:1

VIVA QUESTIONS:

1. What is Merkel Tree?
2. What do you mean by blocks in Block chain technology?
3. What is the Ethereum block chain?
4. How to use node js with block chain?
5. What is Docker in Hyper ledger?

ASSIGNMENT QUESTIONS

S.NO	QUESTIONS	CO MAPPING	BT LEVEL	COMPLEXITY
1	Analyze and write how we can build the trust with block chain.	CO1	Analyze	Medium
2	Design a smart contract for any one block chain application.	CO1	Create	High
3	Show the future scope of block chain is good, support your justification	CO1	Apply	High
4	Analyze and write about project ideas that you can try by learning blockchain technology	CO1	Analyze	Medium

Ex:No: 2 Create and deploy a blockchain network using Hyperledger Fabric SDK for Java

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:2

VIVA QUESTIONS:

1. Can you explain the concept of channels in Hyperledger Fabric?
2. How does Hyperledger Fabric ensure privacy and confidentiality?
3. What are the advantages of using Hyperledger Fabric SDK for Java?
4. How does Fabric handle identity and access management?
5. How does chain code execution work in Fabric?

ASSIGNMENT QUESTIONS

S.NO	QUESTIONS	CO MAPPING	BT LEVEL	COMPLEXITY
1	Explain the role of the **Wallet** in Hyperledger Fabric SDK for Java and how it is used in your Java project.	**CO4**	**Understand**	**High**
2	Describe the purpose of the connection profile and crypto material in a Hyperledger Fabric network setup.	**CO4**	**Understand**	**Medium**
3	Compare and contrast the **submit transaction** and **evaluate Transaction** methods in the **Contract** object. When would you use each?	**CO4**	**Analyze**	**Medium**
4	What is the significance of the discovery(true) method in the Gateway.Builder? Explain its role in the network connectivity.	**CO4**	**Understand**	**Medium**
5	Discuss the steps involved in installing and instantiating chaincode on a Hyperledger Fabric network. What parameters are necessary for these operations, and what do they signify?	**CO4**	**Understand**	**Low**

Ex:No: 3 Interact with a blockchain network and execute transactions and requests against a blockchain network

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 is using ./stopFabric.sh if we want to. At the end of our development session, we should run ./teardownFabric.sh

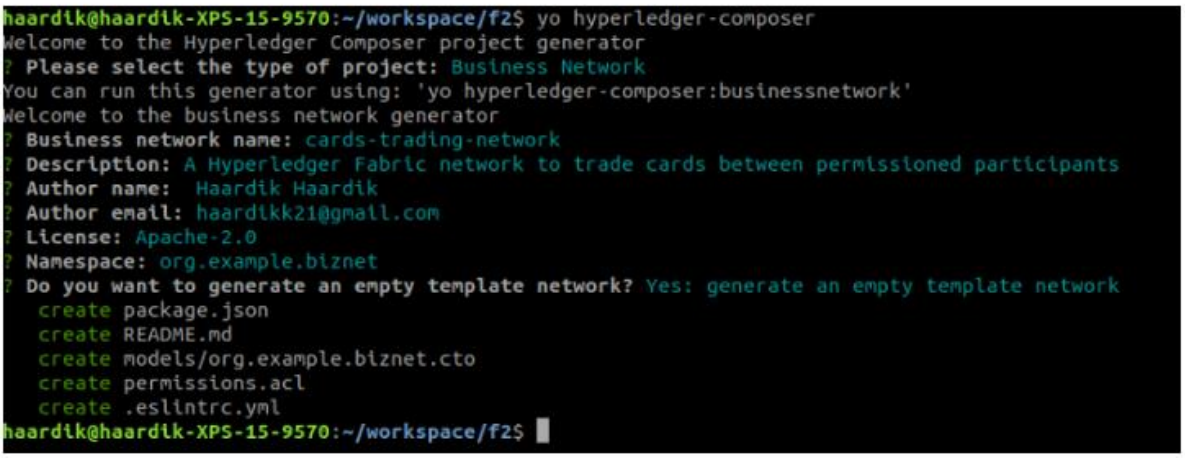
Creating and deploying our business network

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:



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 typeTradingCard , 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/ca
rds-trading-network

Found:
        Description: A Hyperledger Fabric network to trade cards between permiss
ioned 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 --networkVe
rsion 0.0.1 --networkAdmin admin --networkAdminEnrollSecret adminpw --card PeerA
dmin@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
```

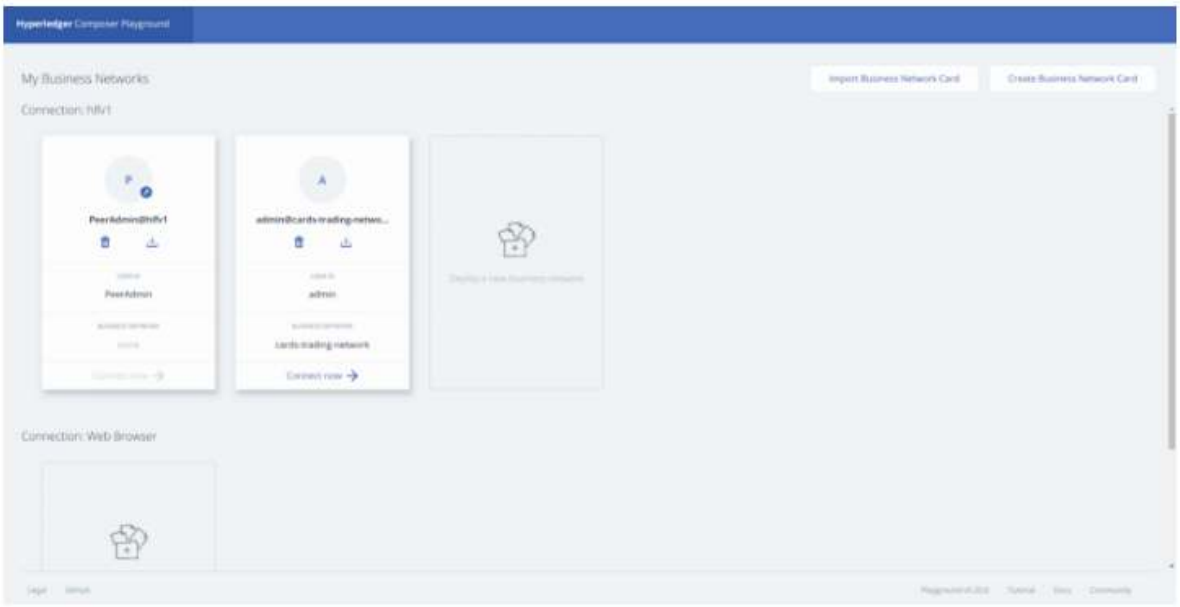
We can now confirm that our network is up and running by typing

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:

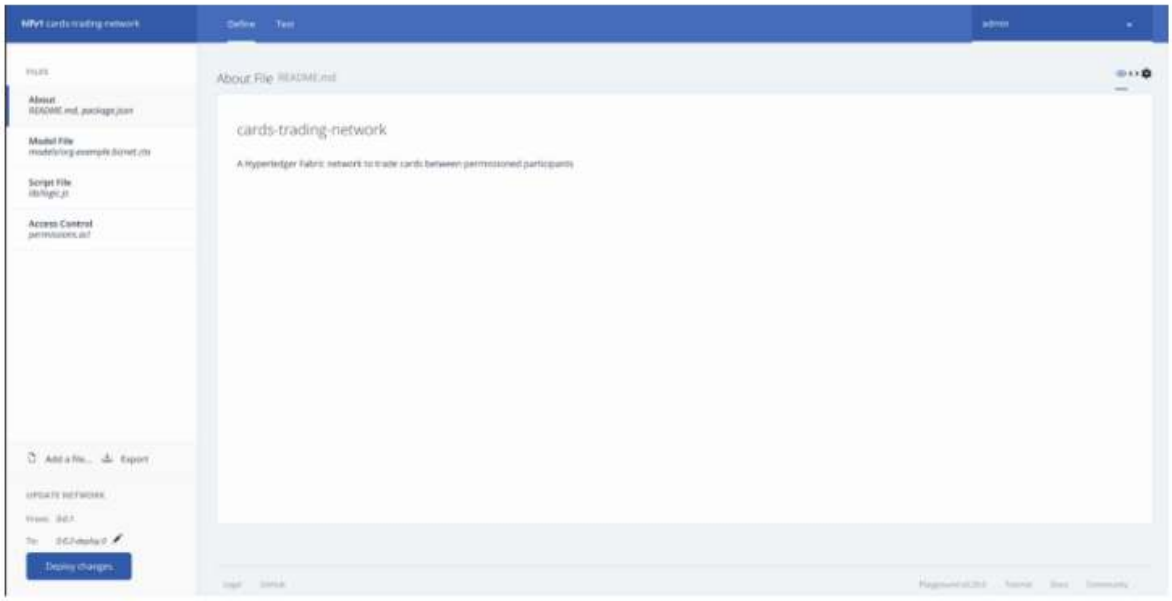
```
haardik@haardik-GT72S-6QE:~$ composer network ping --card admin@cards-trading-network
The connection to the network was successfully tested: cards-trading-network
        Business network version: 0.0.4-deploy.0
        Composer runtime version: 0.20.0
        participant: org.hyperledger.composer.system.NetworkAdmin#admin
        identity: org.hyperledger.composer.system.Identity#457abd6d405ce1dc509a16363611bae608b9984958786ae4ad5a46eb009ef10c

Command succeeded
```

Testing our Business Network

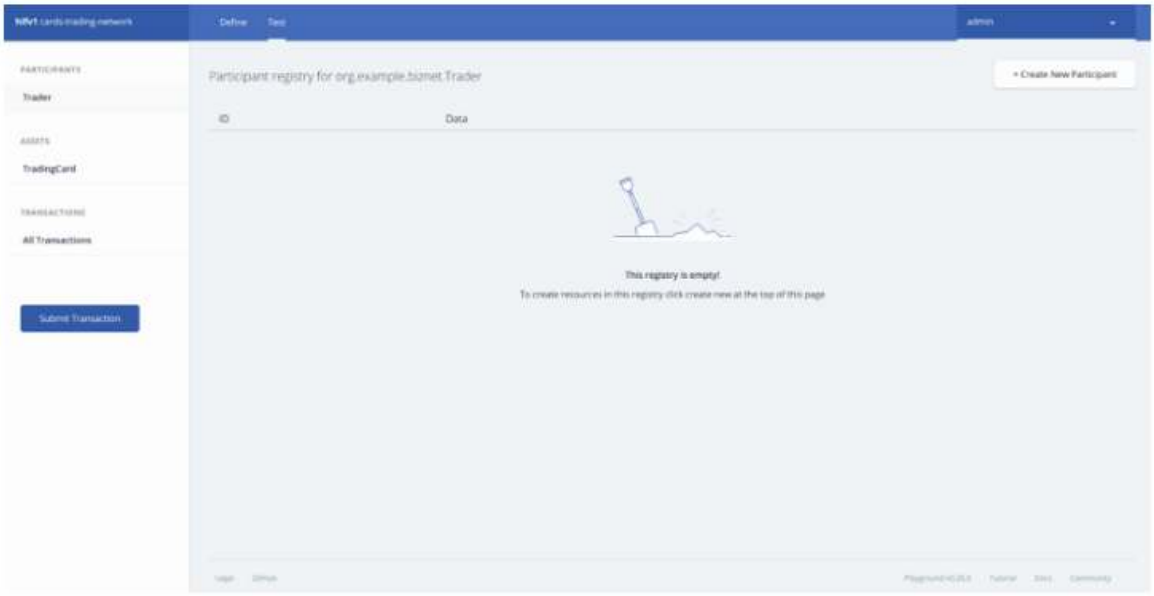
Now that our network is up and running on Fabric, we can start Composer Playground to interact with it. To do this, type composer-playground in Terminal and open up http://localhost:8080/ in your browser and you should see something similar to this:



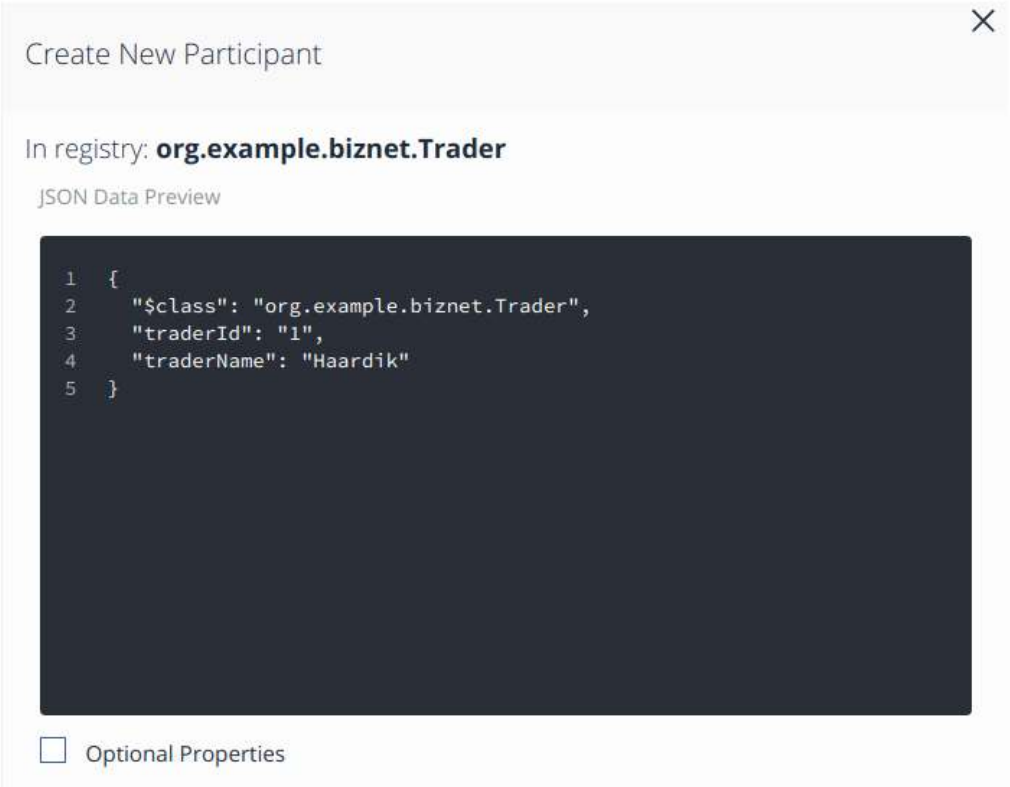
Press Connect Now for admin@cards-trading-network and you'll be greeted with this screen:



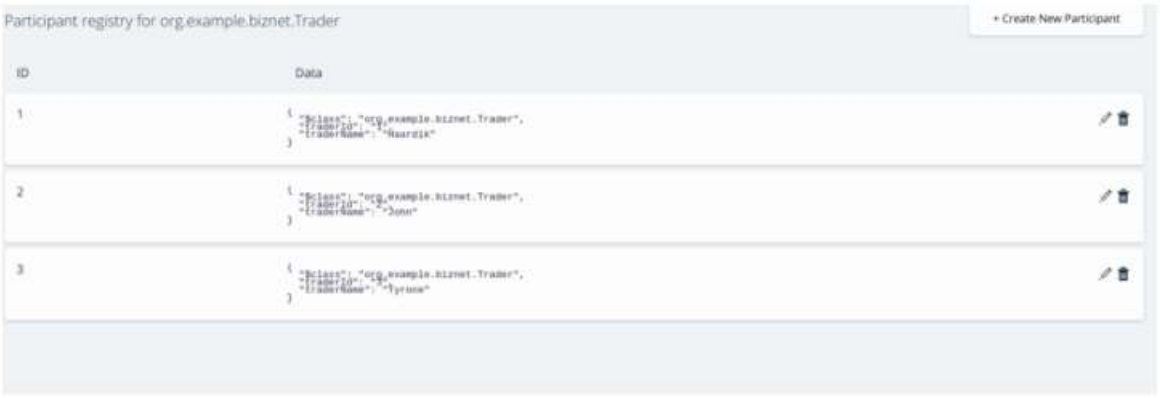
The **Define** page is where we can make changes to our code, deploy those changes to upgrade our network, and export business network archives.
Head over to the **Test** page from the top menu, and you'll see this:



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



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



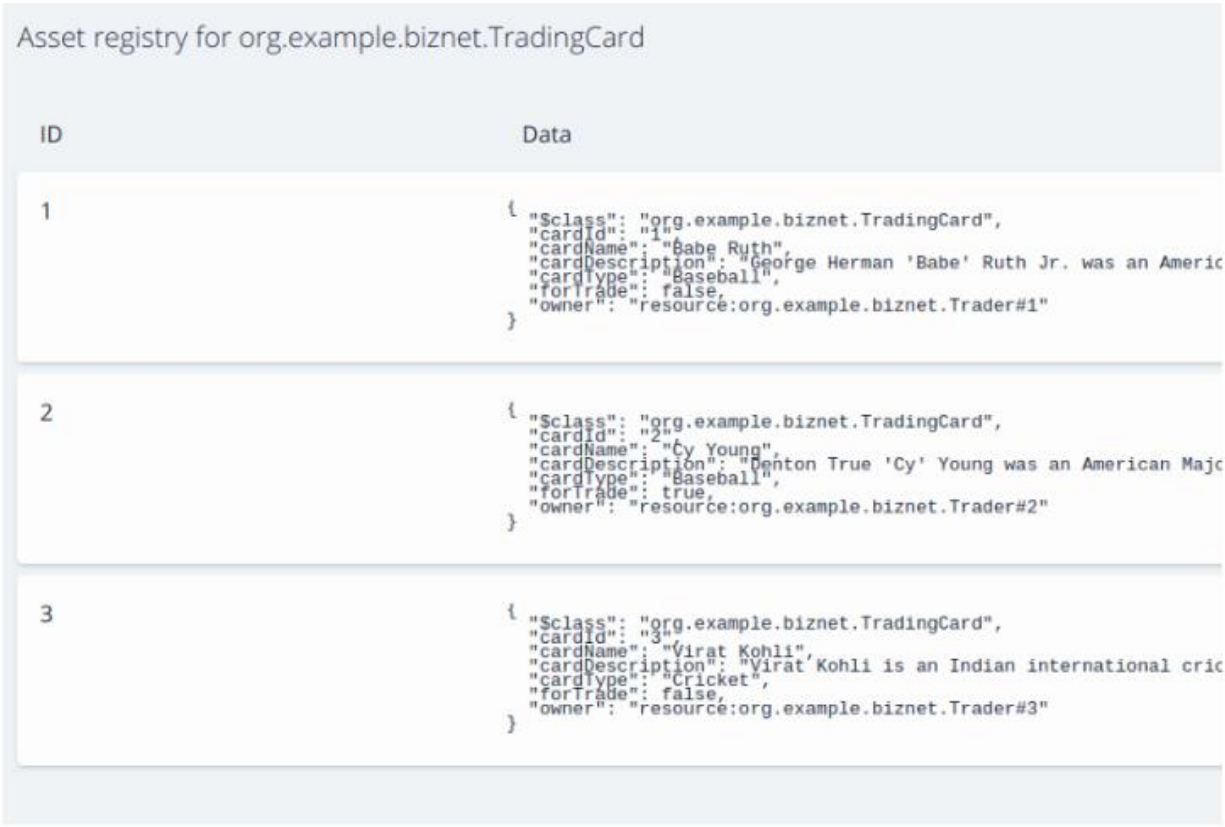
Click on TradingCard from the left menu and press **Create New Asset**. Notice how the owner field is particularly interesting here, looking something like this:

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

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

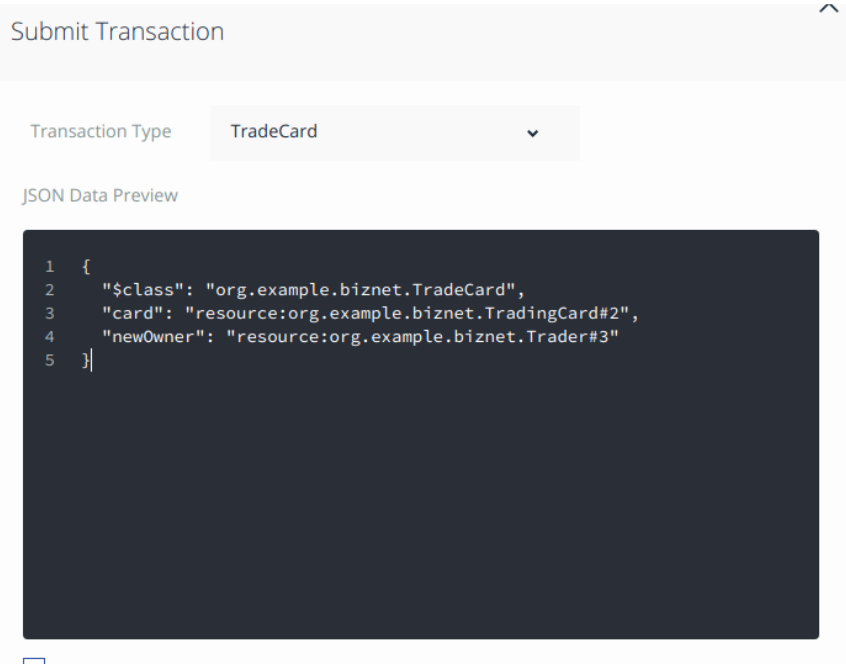
```
{
  "$class": "org.example.biznet.TradingCard",
  "cardId": "1",
  "cardName": "Babe Ruth",
  "cardDescription": "George Herman 'Babe' Ruth Jr. was an American professional baseball player whose career in Major League Baseball spanned 22 seasons, from 1914 through 1935.",
  "cardType": "Baseball",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

Notice how the owner fields 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.



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:



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-GT72S-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
```

Open http://localhost:3000/explorer/ and you'll be greeted with a documented version of an automatically generated REST API :D

Generating an Angular application which uses the REST API

To create the Angular web application, type yo hyperledger-composer in your Terminal, select Angular, choose to connect to an existing business network with the card admin@cards-trading-network, and connect to an existing REST API as well.

```
haardik@haardik-GT72S-6QE:~/Desktop/workspace/hyperledger-tutorial/cards-trading-network$ yo hyperledger-composer
Welcome to the Hyperledger Composer project generator
? Please select the type of project: Angular
You can run this generator using: 'yo hyperledger-composer:angular'
Welcome to the Hyperledger Composer Angular project generator
? Do you want to connect to a running Business Network? Yes
? Project name: cards-trading-angular-app
? Description: Hyperledger Composer Angular project
? Author name: Haardik
? Author email: hhaardik@edu.uwaterloo.ca
? License: Apache-2.0
? Name of the Business Network card: admin@cards-trading-network
? Do you want to generate a new REST API or connect to an existing REST API? Connect to an existing REST API
? REST server address: http://localhost
? REST server port: 3000
? Should namespaces be used in the generated REST API? Namespaces are not used
Created application!
Completed generation process
```

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

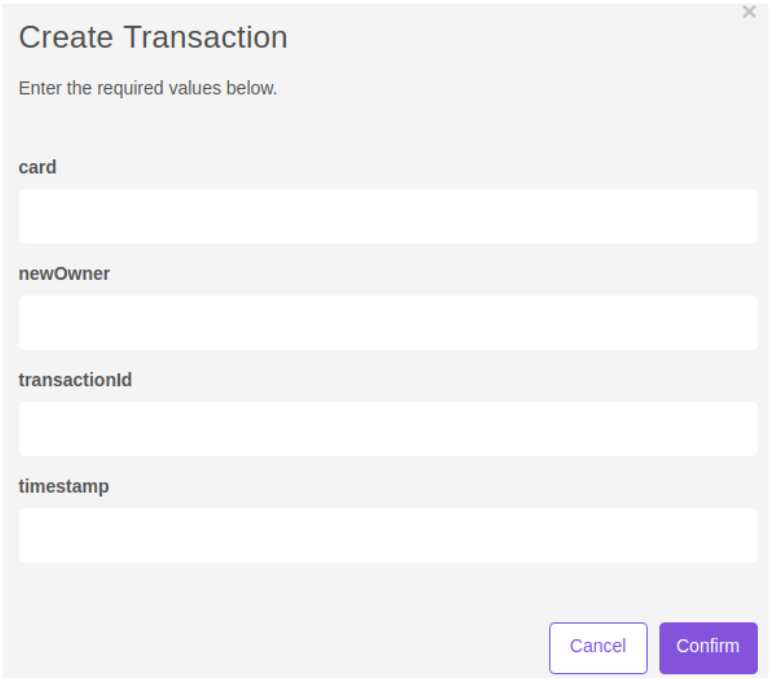
You can add more features to the cards trading network, setting prices on the cards and giving a balance to all Trader. You can also have more transactions which allow the Traders to toggle the value of forTrade . You can integrate this with non blockchain applications and allow users to buy new cards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

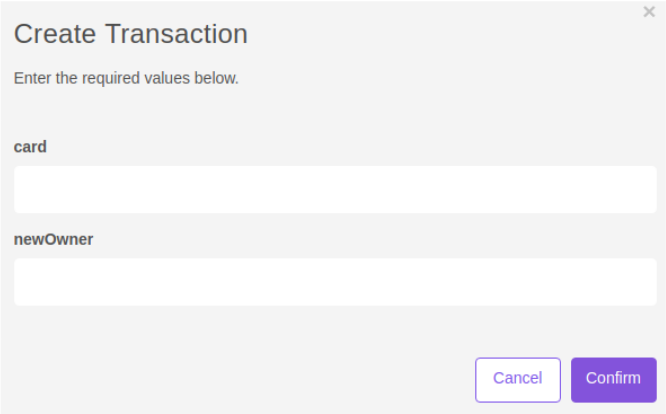


2. Removing unnecessary fields

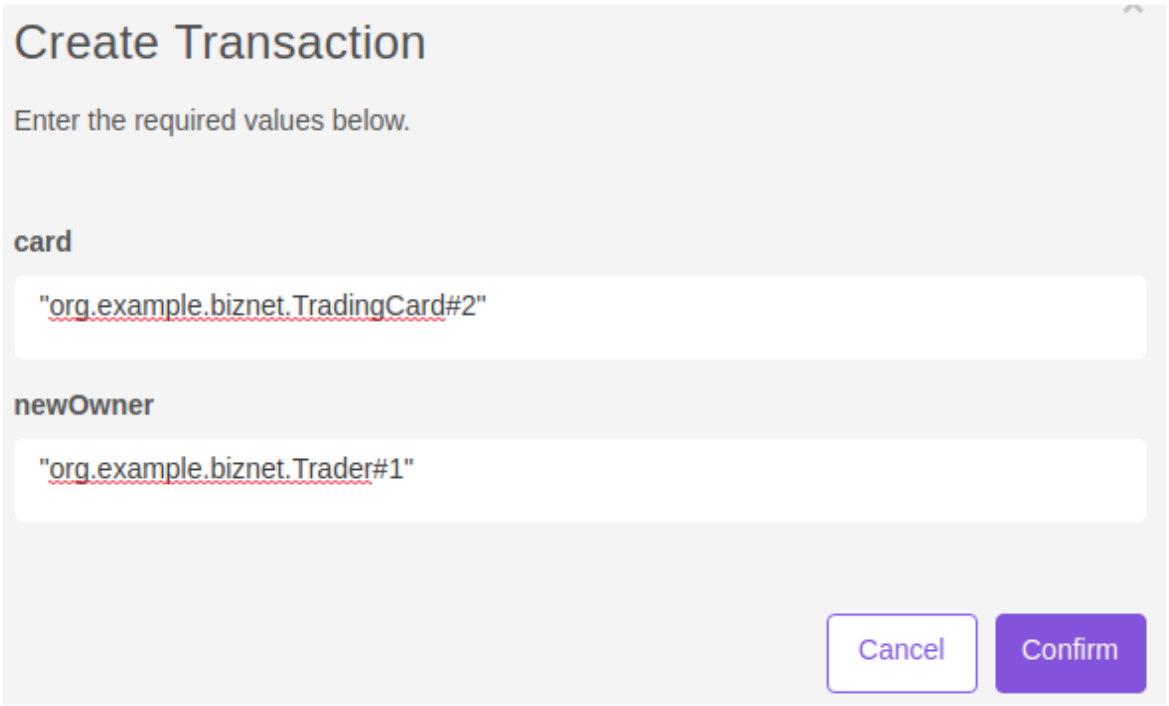
Just getting the modal to open isn't enough. We can see it requests transactionId and timestamp from us even though we didn't add those fields in our modeling file. Our network stores these values which are intrinsic to all transactions. So, it should be able to figure out these values on it's own. And 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:



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:



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: 3

VIVA QUESTIONS:

1. What is the purpose of interacting with a blockchain network through an application?
2. What are the common methods of interacting with a blockchain network?
3. How does a blockchain network authenticate requests from applications?
4. What are some common challenges faced when developing applications to interact with blockchain networks?
5. Explain the difference between transactions and requests in the context of a blockchain network.

ASSIGNMENT QUESTIONS

S.NO	QUESTIONS	CO MAPPING	BT LEVEL	COMPLEXITY
1	Develop a Java application that interacts with a Hyperledger Fabric blockchain network	CO4	Create	High
2	Implement authentication mechanisms in your application to ensure that only authorized users can interact with the blockchain network.	CO1	Create	High
3	Discuss security considerations when developing applications to interact with blockchain networks, including data privacy, encryption, and secure communication protocols.	CO1	Understand	Medium
4	Write test cases to verify the functionality of your application, including submitting valid transactions, handling errors, and enforcing access control.	CO1	Create	High

Ex:No: 4 **Deploy an asset-transfer app using blockchain within Hyperledger Fabric network.**

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/wall
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())}`);
```

Chaincode '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);
 }
```

The terminal output should look like this:

```
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

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

Chaincode '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"
}
```

Sample application '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 chainc
  console.log('\n--> Submit Transaction: UpdateAsset asset70, asset70 does not exist and should ret
  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: 4

VIVA QUESTIONS:

1. What is the purpose of deploying an asset-transfer app on a Hyperledger Fabric network?
2. What are the key components of a Hyperledger Fabric application stack?
3. What are some potential challenges or limitations of developing asset-transfer apps on Hyperledger Fabric?
4. What programming languages can be used to develop applications for Hyperledger Fabric? Why are they preferred?
5. How does Hyperledger Fabric ensure data integrity and security in asset transfer transactions?

ASSIGNMENT QUESTIONS

S.NO	QUESTIONS	CO MAPPING	BT LEVEL	COMPLEXITY
1	Design and implement smart contracts to define the rules and logic for asset transfer operations within the application.	CO5	Create	Medium
2	Create a user interface (UI) for the asset-transfer app, allowing users to initiate and track asset transfer transactions.	CO5	Create	High
3	Document the development process, including design decisions, code implementation, and testing procedures.	CO4	Analyze	Medium
4	Explore additional features or enhancements that could be added to the asset-transfer application to improve usability, efficiency, or functionality.	CO4	Apply	Medium
5	Develop an asset-transfer application using Hyperledger Fabric and a programming language of your choice (e.g., Java, Go, JavaScript).	CO5	Create	High

Ex: No:5 **Fitness Club rewards using Hyperledger Fabric**

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 requirements.

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: 5

VIVA QUESTIONS:

1. What is the purpose of using blockchain technology to track fitness club rewards?
2. What types of data can be stored on the blockchain to track fitness club rewards, and how can this data be accessed and updated by authorized users?
3. What are the advantages of using a distributed ledger like Hyperledger Fabric?
4. Which blockchain has the highest transactions?
5. Explain the features of blockchain?

ASSIGNMENT QUESTIONS

S.NO	QUESTIONS	CO MAPPING	BT LEVEL	COMPLEXITY
1	Design the architecture and data model for a web app that tracks fitness club rewards using Hyperledger Fabric as the underlying block chain platform.	CO5	Create	High
2	Implement a user interface for the web app using HTML, CSS, and JavaScript, allowing members to view their reward balances, redeem rewards, and track transaction history.	CO5	Create	High
3	Document the development process, including system architecture, smart contract implementation, web app design, and integration with Hyperledger Fabric.	CO4	Analyze	Medium
4	Integrate the web app with the Hyperledger Fabric network using the Fabric SDK for JavaScript	CO4	Analyze	Medium

Ex: No:6 Car auction network using the Hyperledger fabric node SDK

Aim:

To create a "Hello World" example of a car auction network using the Hyperledger Fabric Node SDK and IBM Blockchain Starter Plan

Procedure:

1. Set up a Hyperledger Fabric network using the IBM Blockchain Starter Plan.

2. Define and deploy chaincode (smart contract) for the car auction.

3. Develop a Node.js application using the Hyperledger Fabric Node SDK to interact with the chaincode.

4. Implement basic functionalities such as creating an auction, bidding, and retrieving auction results.

1. Set up a Hyperledger Fabric network using IBM Blockchain Starter Plan:

Follow the instructions provided by IBM to set up a Hyperledger Fabric network using the IBM Blockchain Starter Plan.

2. Define and deploy chaincode:

Define chaincode for the car auction network. This chaincode will contain functions to create auctions, place bids, and retrieve auction results.

Program:

```
package main

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

type CarAuctionContract struct {
    contractapi.Contract
}


 type Auction struct {
    ID         string `json:"id"`
    Car        string `json:"car"`
    HighestBid  int    `json:"highestBid"`
```

```
    HighestBidder string `json:"highestBidder"`
}

func (c *CarAuctionContract) CreateAuction(ctx contractapi.TransactionContextInterface, auctionID
string, car string) error {
    auction := Auction{
        ID:    auctionID,
        Car:   car,
    }
    auctionJSON, err := json.Marshal(auction)
    if err != nil {
        return err
    }
    return ctx.GetStub().PutState(auctionID, auctionJSON)
}

func (c *CarAuctionContract) PlaceBid(ctx contractapi.TransactionContextInterface, auctionID string,
bidder string, bidAmount int) error {
    auctionJSON, err := ctx.GetStub().GetState(auctionID)
    if err != nil {
        return err
    }
    if auctionJSON == nil {
        return fmt.Errorf("auction %s does not exist", auctionID)
    }

    var auction Auction
    err = json.Unmarshal(auctionJSON, &auction)
    if err != nil {
        return err
    }
    if bidAmount > auction.HighestBid {
        auction.HighestBid = bidAmount
```

```
        auction.HighestBidder = bidder
        updatedAuctionJSON, _ := json.Marshal(auction)
        return ctx.GetStub().PutState(auctionID, updatedAuctionJSON)
    }
    return fmt.Errorf("bid amount should be higher than the current highest bid")
}
func (c *CarAuctionContract) GetAuction(ctx contractapi.TransactionContextInterface, auctionID string)
(*Auction, error) {
    auctionJSON, err := ctx.GetStub().GetState(auctionID)
    if err != nil {
        return nil, err
    }
    if auctionJSON == nil {
        return nil, fmt.Errorf("auction %s does not exist", auctionID)
    }
    var auction Auction
    err = json.Unmarshal(auctionJSON, &auction)
    if err != nil {
        return nil, err
    }
    return &auction, nil
}
```

3. Develop a Node.js application:

Use the Hyperledger Fabric Node SDK to interact with the chaincode. This application will have functions to create auctions, place bids, and retrieve auction results.

```
// app.js
const { Gateway, Wallets } = require('fabric-network');
const path = require('path');
const fs = require('fs');
const ccpPath = path.resolve(__dirname, '..', 'connection.json');
const ccpJSON = fs.readFileSync(ccpPath, 'utf8');
```

```
const ccp = JSON.parse(ccpJSON);
async function main() {
    try {
        const walletPath = path.join(process.cwd(), 'wallet');
        const wallet = await Wallets.newFileSystemWallet(walletPath);
        const gateway = new Gateway();
        await gateway.connect(ccp, {
            wallet,
            identity: 'user1',
            discovery: { enabled: true, asLocalhost: true }
        });
        const network = await gateway.getNetwork('mychannel');
        const contract = network.getContract('carAuction');
        // Invoke chaincode functions here
        await gateway.disconnect();
    } catch (error) {
        console.error(`Failed to submit transaction: ${error}`);
        process.exit(1);
    }
}
main();
```

4. Implement basic functionalities:

Implement functions in `app.js` to interact with the chaincode.

```
async function createAuction(auctionID, car) {
    await contract.submitTransaction('CreateAuction', auctionID, car);
    console.log(`Auction ${auctionID} created for car ${car}`);
}
async function placeBid(auctionID, bidder, bidAmount) {
```

```
    await contract.submitTransaction('PlaceBid', auctionID, bidder, bidAmount);
    console.log(`Bid placed for auction ${auctionID} by ${bidder} with amount ${bidAmount}`);
}
async function getAuction(auctionID) {
    const auctionJSON = await contract.evaluateTransaction('GetAuction', auctionID);
    const auction = JSON.parse(auctionJSON.toString());
    console.log(`Auction ${auctionID}: ${JSON.stringify(auction, null, 2)}`);
}
```

Output:

```
Auction CAR123 created for car BMW
Bid placed for auction CAR123 by bidder1 with amount 5000
Auction CAR123: {
  "id": "CAR123",
  "car": "BMW",
  "highestBid": 5000,
  "highestBidder": "bidder1"
}
```

Result:

Thus the creation of "Hello World" example of a car auction network using the Hyperledger Fabric Node SDK and IBM Blockchain Starter Plan are executed successfully.

EX.NO: 6

VIVA QUESTIONS:

1. How can analytics and reporting features be integrated into the web app to provide insights into member behaviour and reward redemption patterns?
2. Explain the role of the IBM Blockchain Starter Plan
3. Describe the data model for a car auction network
4. How is data stored and retrieved from the IBM Blockchain Starter Plan, and what mechanisms are in place to ensure data privacy and security?
5. What are the two parts of Hyperledger fabric ledger?

ASSIGNMENT QUESTIONS

S.NO	QUESTIONS	CO MAPPING	BT LEVEL	COMPLEXITY
1	Document the development process, including smart contract implementation, application design, integration with IBM Blockchain Starter Plan, and testing procedures.	CO5	Analyze	Medium
2	Install the Hyperledger Fabric Node SDK and other necessary dependencies for developing and deploying the car auction network application.	CO4	Create	Medium
3	Implement security features such as authentication, authorization, and encryption to protect sensitive auction data and transactions within the application	CO1	Create	Medium
4	Integrate the Node.js application with the IBM Blockchain Starter Plan to store and retrieve auction data from the cloud-based blockchain ledger.	CO4	Analyze	High

EX.NO: 7

Creating Crypto-currency Wallet

AIM :

To creating a Crypto-currency Wallet

STEPS:

1. Choose the type of wallet.
2. Sign up for an account, buy the device or download the software needed.
3. Set up security features, including a recovery phrase.
4. Purchase crypto currency or transfer coins from another wallet or exchange.

Types of crypto wallets:

There are three basic types of wallets for virtual currency.

One option is a software wallet or hot wallet that stores your crypto on an internet-connected device that you own.

Another option to consider with added security is a cold wallet, a specialized piece of hardware that keeps your crypto offline.

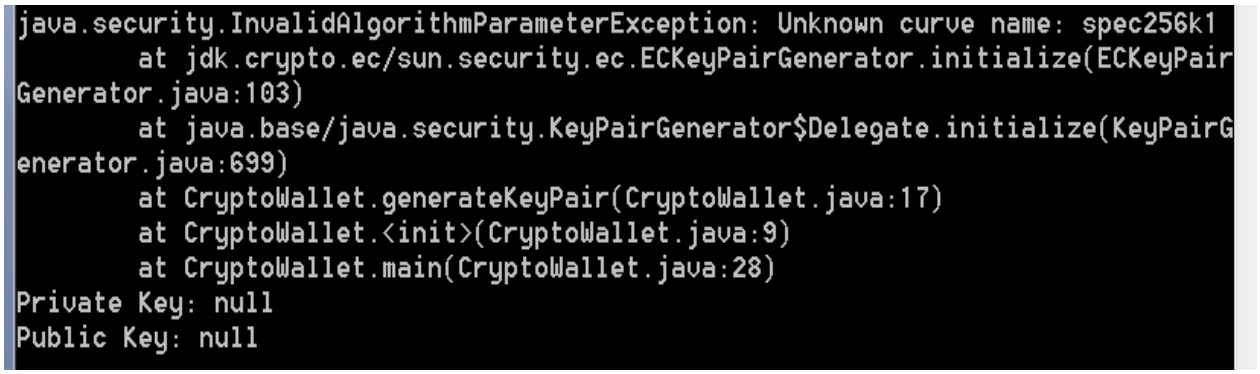
Custodial wallets, which leave your crypto in the control of a company you trust, such as a crypto exchange, are another storage method to consider.

SOURCE CODE:

```
import java.security.*;
import java.security.spec.ECGenParameterSpec;
public class CryptoWallet {
private PrivateKey privateKey;
private PublicKey publicKey;
public CryptoWallet() {
generateKeyPair();
}
public void generateKeyPair() {
try {
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("EC");
SecureRandom random = SecureRandom.getInstanceStrong();
```

```
ECGenParameterSpec ecSpec = new ECGenParameterSpec("spec256k1");
keyGen.initialize(ecSpec, random);
KeyPair keyPair = keyGen.generateKeyPair();
privateKey = keyPair.getPrivate();
publicKey = keyPair.getPublic();
} catch (Exception e) {
e.printStackTrace();
}}
public static void main(String[] args) {
CryptoWallet wallet = new CryptoWallet();
System.out.println("Private Key: " + wallet.privateKey);
System.out.println("Public Key: " + wallet.publicKey);
}}
```

EXPECTED OUTPUT:



RESULT:

Thus the program creating a Crypto currency wallet has been executed successfully.

EX.NO:8

CREATING MERKLE TREE

AIM:

To Create a Merkle tree

STEPS:

Merkle tree is a tree data structure with leaf nodes and non leaf nodes. It also known as Hash tree.The reason behind it is it only stores the hashes in its nodes instead of data. In its leaf nodes, it willstore the hash of the data. Non leaf nodes contain the hash of its children.

Bit coin's merkle-tree implementation works the following way:

1.split the transactions in the blockup into pairs

2.byte-swap the txids

3.concatenate the txids

4.double hash the concatenated pairs

SOURCE CODE:

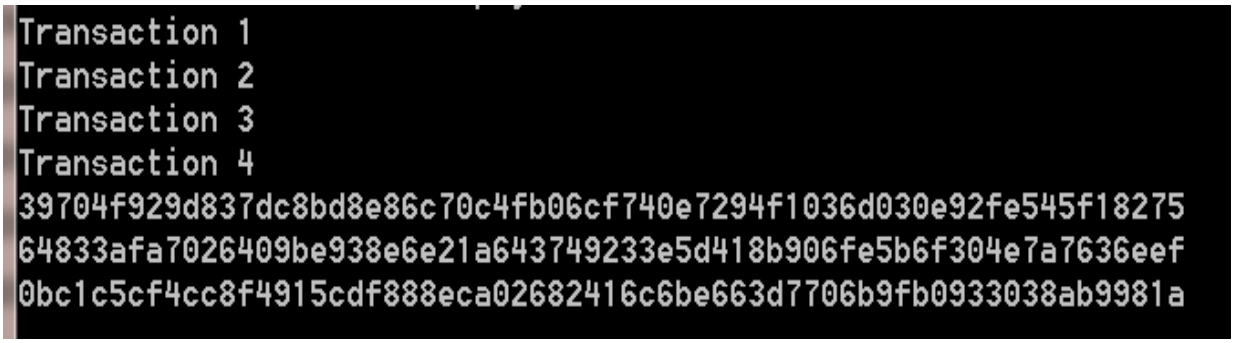
```
Import java.nio.charset.StandardCharsets;
Import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;
import java.util.List;
 public class MerkleTree {
privateList<String>transactions;
private List<String> merkleTree;
public MerkleTree(List<String> transactions)
{
this.transactions = transactions;
this.merkleTree = buildMerkleTree(transactions);
```

```
}
private String calculateHash(String data)
 {
try
{
MessageDigest digest = MessageDigest.getInstance("SHA-256");
byte[]  hashBytes = digest.digest(data.getBytes(StandardCharsets.UTF_8));
StringBuilder hexString = new StringBuilder();
for (byte hashByte : hashBytes) {
String hex = Integer.toHexString(0xff &hashByte);if (hex.length() == 1)
{

                hexString.append('0');
}
hexString.append(hex);
}
return hexString.toString();
}
catch
(NoSuchAlgorithmException e) {
e.printStackTrace();
}
return null;
}
private List<String> buildMerkleTree(List<String> transactions)
 {
List<String> merkleTree = new ArrayList<>(transactions);
int levelOffset = 0;
for (int levelSize = transactions.size(); levelSize> 1;
```

```
levelSize = (levelSize + 1) / 2)
{
for (int left = 0; left <levelSize; left += 2)
{
int right = Math.min(left + 1, levelSize - 1);
String leftHash = merkleTree.get(levelOffset +left);
String rightHash = merkleTree.get(levelOffset +right);
String parentHash = calculateHash(leftHash + rightHash);
merkleTree.add(parentHash);
}
levelOffset += levelSize;
}
return merkleTree;
}
public
List<String>getMerkleTree()
 {
return merkleTree;
}
public static void main(String[] args)
{
List<String> transactions = new
ArrayList<>();
transactions.add("Transaction 1");
transactions.add("Transaction 2");
transactions.add("Transaction 3");
transactions.add("Transaction 4");
MerkleTree merkleTree = new MerkleTree(transactions);
```

```
List<String> tree = merkleTree.getMerkleTree();
for (String hash : tree)
{
System.out.println(hash;
}
}
}
```

EXPECTED OUTPUT:



RESULT:

Thus the program creating a Merkle tree has been executed successfully.

Fake Product Detection System Using Blockchain