Hyperledger Fabric v1.4.6

Documentation Links

1. **Hyperledger Fabric Docs** Web PDF
    a. Official documentation for fabric v1.4.6

2. **Node**
    a. Node fabric chaincode SDK Documentation
    -> Official documentation for chaincode API's
    -> Useful for implementing chaincode

    b. Node fabric app SDK Documentation
    -> Official documentation for application API's
    -> Useful for implementing client application

++

Part 1: Fabric Installation

1A. Prerequisites:

1. A UNIX environment (Linux or MacOS)
    * Windows Users : Run a linux virtual machine with at least **20 GB** of space.

2. **Curl**: cmdline tool to transfer data with URLs
    a. Install: *sudo apt install curl*

3. **Docker and Docker Compose** : To create standard environments for building and running apps and chaincode
    (I have the old version running for this application, as newer versions give issues)
    a. Install:
        i. Linux : Perform steps in order
            a) Step 1 (To install Docker Engine Community -> Do *install using repository* section)
            b) Step 2 (To run docker w/o sudo -> Do *Manage Docker as a non-root user* section **only**)
            c) Step 3 (To install Docker Compose -> Do *Install compose* section)

        ii. MacOS

4. **Node.js**: A JavaScript runtime to run apps and chaincodes
    a. Install: Official Website

5. **Python 2** : *sudo apt install python*

6. Xcode (MacOS only)

1B. Download Fabric samples and binaries:

$ curl -sSL http://bit.ly/2ysbOFE | bash -s -- 1.4.6 1.4.6 0.4.18

The installation has few inbuilt applications like 'fabcar'. We will use one of the network configurations(1 ord, 1 org, 1 peer) available here for fabchat.

Can take a look for other material here : (https://github.com/hyperledger/fabric-samples)

1C. To check if everything works?

We can run one such application 'fabcar' to check that installation was successful. The steps are given below:

1. Launch blockchain network
    $ cd fabric-samples/fabcar // go to fabcar client app. Folder
    $ ./startFabric.sh javascript // start the network & use javascript chaincode
2. Install app dependencies defined in *package.json*
    $ cd fabric-samples/fabcar/javascript // go to javascript code folder
    $ npm install (Install app. dependencies)
3. Enroll admin (enrolls the CA admin)
    $ node enrollAdmin.js // node is javascript runtime environment
4. Register and enroll user1
    $ node registerUser.js
5. Query ledger to list all cars
    $ node query.js

If every step is executed w/o any issues, you should finally see the output of query like this:

Transaction has been evaluated, result is:
[{"Key":"CAR0","Record":{"color":"blue","docType":"car","make":"Toyota","model":"Prius","owner":"Tomoko"}},{"Key":"CAR1","Record":{"color":"red","docType":"car","make":"Ford","model":"Mustang","owner":"Brad"}},{"Key":"CAR2",.....}}]

1D. How to stop/tear the network?

Remove created wallets

$ rm -r wallet

Stop network

$ cd fabric-samples/fabcar

$ ./stopFabric.sh

Important: Run teardown.sh if you've run through this tutorial before going ahead with fabchat

$ ./teardown.sh

Remove old crypto material and config transactions

*$ rm -fr config/**

*$ rm -fr crypto-config/**

++

Part 2: Revisiting Components of FabCar

Important Components of FabCar's Network

FabCar Network: Fabric-samples come along with few different inbuilt network configurations. FabCar uses the one called "first-network". But we will look at "basic-network" which is a simpler setting i.e. 1 ord, 1 org and 1 peer and is used in fabchat.

1. Dir path for "**basic-network"**:
   fabric-samples/basic-network
2. Files of interest:
   a. *crypto-config.yaml : defines the orderer, organization, number of peers in the organization.*
   b. *docker-compose-*.yaml : details about all the containers running on docker(5 containers)*
3. Network at a glance:
   a. 1 org (*org1*): 1 peer (*peer0*) and CA
   b. 1 orderer

https://github.com/hyperledger-archives/education/tree/master/LFS171x/fabric-material/basic-network

Launching the Network

Each application has a shell script to start the network (can do each step individually but easier to have a script). To look at Fabcar's start script :

$ cd fabric-samples/fabcar
$ ./startFabric.sh javascript

We will run fabchat's start script to launch the network. This step bootstraps the entire network from scratch by doing the following steps:

1. Generates crypto material (certs, keys) and channel artifacts
2. Starts docker containers for peers, orderers etc.
3. Creates channel named *mychannel*
4. Adds all peers to my *mychannel.* In our case just the one peer.
5. Installs "fabchat" chaincode on the *peer* of *org1*
6. Instantiates "fabchat" chaincode on *mychannel*
7. Submits an ***initLedger*** txn

Note: After script finishes, check running docker containers:

$ docker ps

++

Part 3: Revisiting FabChat

Part 3.1 Setting up network

Prerequisites:

1. Download fabchat zip
2. Paste (merge/replace) zip's contents in fabric-samples dir. To make it less confusing, as fabric-samples has a lot of directories, you can simply remove everything except 'basic-network' and 'bin' directories and then add zip's content.
3. Install app. dependencies
   $ cd fabric-samples/fabchat/javascript
   $ npm install

We'll use basic-network in this section as mentioned above.
Again basic-network at a glance:

1. 1 orderer
2. 1 CA
3. 1 org (*org1*) maintaining 1 peer (*peer0*)

1. Change dir to basic-network

    $ cd fabric-samples/basic-network

(Bullet Points 2-9 are basically doing what the start script should contain, as discussed above)
2. Remove old crypto material and config transactions

    *$ rm -fr config/**
    *$ rm -fr crypto-config/**

Generate new crypto material

    $ ../bin/cryptogen generate --config=./crypto-config.yaml

Takes as input *crypto-config.yaml* and uses *cryptogen* tool to generate crypto material for orderer, peer and CA which is stored in *crypto-config* folder.

3. Generate genesis block for orderer

    $ ../bin/configtxgen -profile OneOrgOrdererGenesis -outputBlock ./config/genesis.block

Generate channel configuration txn

    $ ../bin/configtxgen -profile OneOrgChannel -outputCreateChannelTx ./config/channel.tx -channelID ***mychannel***

Takes as input *configtx.yaml* and uses *configtxgen* tool to generate channel artifacts (orderer genesis block and channel config. txn) which are stored in *config* folder as *genesis.block* and *channel.tx* respectively.

Important: Everytime you generate fresh crypto material, change FABRIC_CA_SERVER_CA_KEYFILE in *docker-compose.yml* with new key from *crypto-config/peerOrganizations/org1.example.com/ca*

4. Start the network

    $ docker-compose -f docker-compose.yml up -d ca.example.com orderer.example.com peer0.org1.example.com couchdb cli

5. Create channel named ***mychannel***

*$ docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/users/Admin@org1.example.com/msp" peer0.org1.example.com peer channel create -o orderer.example.com:7050 -c **mychannel** -f /etc/hyperledger/configtx/channel.tx*

6. Join peer0.org1.example.com to the channel

$ docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/users/Admin@org1.example.com/msp" peer0.org1.example.com peer channel join -b mychannel.block

7. Install **fabchat** chaincode on peer0

*$ docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp" cli peer chaincode install -n **fabchat** -v 1.0 -p "/opt/gopath/src/github.com/fabchat/javascript" -l "node"*

8. Instantiate chaincode on *mychannel (Endorsement Policy is shown in **bold**)*

*$ docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp" cli peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n fabchat -l "node" -v 1.0 -c '{"Args":[]}' **-P "OR ('Org1MSP.member','Org2MSP.member')"***

9. Submit ***initLedger*** txn

*docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp" cli peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n fabchat -c '{"function":"**initLedger**","Args":[]}'*

10. Start inspecting chaincode logs of peer0

$ docker logs -f dev-peer0.org1.example.com-fabchat-1.0

Note: Inspect *connection.json.* It is called a *connection profile* which describes a view of the network. Through this file, applications know the **addresses** of nodes to connect to.

Part 3.2 The Client Application Part

FabChat application code dir path:
fabric-samples/fabchat/javascript

Files of interest:

1. *enrollAdmin.js*
2. *invoke.js*
3. *query.js*
4. *registerUser.js*

Running Application:

1. Change dir
   $ cd fabric-samples/fabchat/javascript

2. Enroll admin (stores CA admin credentials in wallet dir)
   $ node enrollAdmin.js

3. Register 3 users **user1**, **user2**, **user3** (stores <userID> credentials in wallet dir)

   General usage *: $ node registerUser.js <userID>*
   Example usage *: $ node registerUser.js user1*
   $ node registerUser.js user2
   $ node registerUser.js user3

4. Post 3 messages using **user1, user2** and **user3** wallets respectively

   General usage: *$ node invoke.js createMsg <msgText> <userID> <emailID>*
   Example usage: *$ node invoke.js createMsg hello user1 u1@ashoka.edu.in*
   $ node invoke.js createMsg welcome user2 u2@ashoka.edu.in
   $ node invoke.js createMsg covid19 user3 u3@ashoka.edu.in

5. Query all messages using **user1** wallet

   General usage: *$ node query.js -1 <userID>*
   Example usage: *$ node query.js -1 user1*

6. Query message by **msgID** using **user1** wallet

   General usage: *$ node query.js <msgID> <userID>*
   Example usage: *$ node query.js 2 user1*

7. Flag “covid19” message with **msgID** “2” using **user1** and **user2** wallet

General usage: *$ node invoke.js flagMsg <msgID> <userID>*
Example usage: *$ node invoke.js flagMsg 2 user1*
$ node invoke.js flagMsg 2 user2

8. Perform a query. The email ID of the msg poster will be revealed.
   $ node query.js 2 user1

9. Try to flag "covid19" message with **msgID** "2" again using **user2** wallet. It will fail. Thus, a user cannot flag the same msg twice.
   $ node invoke.js flagMsg 2 user2

10. Try to flag **user2**'s "welcome" msg with **msgID** "1" using **user2** wallet. It will fail. Thus, a user cannot flag its own msg.
    $ node invoke.js flagMsg 1 user2

Part 3.2.1 Inspecting *query.js/invoke.js*:

1. Import *FileSystemWallet* and *Gateway* classes from *fabric-network* module. Note that the *fabric-network* module was downloaded in the *node_modules* folder with *npm install.*

   const { FileSystemWallet, Gateway } = require('fabric-network');

2. Use ***FileSystemWallet*** class to create a wallet object

   const walletPath = path.join(process.cwd(), 'wallet');
   *const wallet = new **FileSystemWallet**(walletPath);*

3. Use ***Gateway*** class to create a new gateway and use it to connect to the network using <*userID*> identity from wallet.

   *const gateway = new **Gateway**();*
   *await gateway.connect(**ccp**, { wallet, identity: <userID>});*

4. Note that ***ccp*** is a connection profile which describes a *view* of the network. It is loaded from:

   fabric-samples/basic-network/connection.json

5. Get the network (channel) our contract is deployed to

   *const network = await gateway.getNetwork(**'mychannel'**);*

6. Get contract from the network

   const contract = network.getContract(**'fabchat'**);

7. Connect to a peer address defined in **ccp** and send txn proposal

   *await contract.evaluateTransaction(**'queryAllMsgs'**);*
   OR
   *await contract.evaluateTransaction(**'queryMsg'**, <msgID>);*
   OR
   *await contract.submitTransaction(**'createMsg''**, <msg>, <emailID>);*
   OR
   *await contract.submitTransaction(**'flagMsg''**, <msgID>);*

Part 3.3 The Chaincode Part

Prerequisites:

1. Download new fabchat zip and paste (merge/replace) its contents in fabric-samples
2. The zip updates fabchat chaincode and adds 2 main scripts in *fabric-samples/fabchat* :
   a. *startFabChat.sh* : bootstraps the entire FabChat network
   b. *teardownFabChat.sh* : teardowns the entire FabChat network

Perform the following steps:

1. *cd fabric-samples/fabchat*
2. *$ ./teardownFabChat.sh*
3. *$ ./startFabChat.sh*
4. *$ docker logs -f dev-peer0.org1.example.com-fabchat-1.0 // chaincode log*
5. View world state through Fauxton web interface :
   http://localhost:5984/_utils/#database/mychannel_fabchat/_all_docs

FabChat chaincode path:

*fabric-samples/chaincode/fabchat/javascript/lib/**fabchat.js***

Key Logic: A message remains anonymous unless **at least 50%** of the users flag it.

Functions (txns) of interest:

1. ***createMsg***
2. ***flagMsg***
3. ***queryMsg***
4. ***queryAllMsgs***
5. ***initLedger****

Chaincode uses the following imports:
1. *Contract* class from *fabric-contract-api* module. Note: *FabChat* class **extends** *Contract* class.
2. *ClientIdentity* class from *fabric-shim* module. This class will be used to get the ID of the invoking identity.

*** Refer Part 4 Miscellaneous section**

Chaincode uses the following global variables:

1. **msgID** : stores the **msgID** of the last **msg** that was posted (Initially -1)
2. *users* : an array that stores <*userID*> of all registered users (Initially empty)

An asset (key, value) is a **(msgID, msg)** pair. For eg. the **msg** object with **msgID** "2" after getting flagged by **user1** looks like this:

```
{       msgText : covid19,                      // text of the msg
        userID : user3*.                        // <userID> of the poster
        flag :  1,                              // no. of flags msg has received
        flaggers : [ user1* ],                  // users who've flagged the msg
        emailID : u3@ashoka.edu.in      }       // email ID of the poster
```

* Shortened for clarity. For eg. actual **userID** for **user3** looks like:
x509::/OU=client+OU=org1+OU=department1/CN=**user3**::/C=US/ST=California/L=San Francisco/O=org1.example.com/CN=ca.org1.example.com

Before moving to **createMsg** section, perform the following steps:

1. Change dir
   $ cd fabric-samples/fabchat/javascript
2. Enroll admin (stores CA admin credentials in wallet dir)
   $ node enrollAdmin.js
3. Register 3 users **user1**, **user2**, **user3** (stores <userID> credentials in wallet dir)
   $ node registerUser.js user1
   $ node registerUser.js user2
   $ node registerUser.js user3
4. Post a message using **user1** wallet
   $ node invoke.js createMsg hello user1 u1@ashoka.edu.in
5. Inspect fauxton web interface and chaincode logs

createMsg txn: Takes as input a **msgTxt** (eg. "*hello*") and an **emailID** (eg. "*u1@ashoka.edu.in")* and proceeds as follows:

1. Get ID associated with the invoking identity using **getID** function

   *let cid = new ClientIdentity(**ctx.stub**);*
   *let **userID** = cid.**getID**();*

   Note: **ctx.stub** defines a txn context. It gives access to APIs which enable the chaincode to access the ledger, retrieve txid, retrieve user's identity, etc.

2. Create a javascript **msg** object

   { **msgText** : hello,
   userID : user1*.
   flag : 0, // set to 0
   flaggers : [ ], // empty array
   emailID : u1@ashoka.edu.in }

3. IF **userID** is not in *users* array
   THEN
   push **userID** in *users* array

4. Increment **msgID** by 1

5. Add **(msgID, msg)** to the world state using **putstate** function. Note: *JSON.stringify()* converts javascript **msg** object to JSON string. *Buffer.from()* converts this JSON string to a sequence of bytes.

   *await ctx.stub.**putState**(**msgID**.toString(), Buffer.from(JSON.stringify(**msg**)));*

Before moving to **flagMsg** section, perform the following steps:

1. Post messages using **user2** and **user3** wallets respectively
   $ node invoke.js createMsg welcome user2 u2@ashoka.edu.in
   $ node invoke.js createMsg covid19 user3 u3@ashoka.edu.in
2. Flag "covid19" message with **msgID** "2" using **user1** wallet
   $ node invoke.js flagMsg 2 user1
3. Inspect fauxton web interface and chaincode logs

flagMsg txn :Takes as input a **msgID** (eg. "2") and proceeds as follows:

1. Get ID associated with the invoking identity using **getID** function

   let cid = new ClientIdentity(ctx.stub);

*let **flagger** = cid.**getID**();*

2. Calculate current *threshold*

*let threshold = Math.ceil(0.5 * users.length);*

3. Load from the world state the **msg** referenced by the **msgID** (passed as function argument) using **getState** function. Note: *msgAsBytes.toString()* converts **msg** bytes to a JSON string. *JSON.parse()* converts this JSON string to a javascript object.

*const msgAsBytes = await ctx.stub.**getState**(**msgID**);*
*const **msg** = JSON.parse(msgAsBytes.toString());*

4. IF **msg.userID** ≠ **flagger** AND **flagger** is not in **msg.flaggers** AND **msg.flag** ≠ -1
   THEN
   Push **flagger** in **msg.flaggers** array
   Increment **msg.flag** by 1
   IF **msg.flag** >= *threshold*
   THEN
   Set **msg.flag =** -1
   ELSE
   FAIL

   The IF statements take care of four things:
   1. A flagger cannot flag its own msg.
   2. A flagger cannot flag the same msg twice.
   3. A flagger cannot flag a msg with **msg.flag** = -1
   4. Any time the number of flags a msg receives exceeds the current threshold, **msg.flag** is set to -1.

   *** We'll see later that the email ID of a msg with flag = -1 is disclosed. ***

4. Update the world state using **putState** function

*await ctx.stub.**putState**(**msgID**, Buffer.from(JSON.stringify(**msg**)));*

Before moving to **queryMsg** section, perform the following steps:

1. Flag "covid19" message with **msgID** "2" using **user2** wallet
   $ node invoke.js flagMsg 2 user2

Important: After the above step, the flag field of "convid19" msg with msgID "2" will be set to **-1**. This is because this msg has received the threshold no. of flags.

2. Inspect fauxton web interface and chaincode logs

3. Query “covid19” by msgID “2”
   $ node query.js 2 user1

queryMsg txn : Takes as input a **msgID** (eg. “2”) and proceeds as follows:

1. Load from the world state the **msg** referenced by the **msgID** (passed as function argument) using **getState** function

   *const msgAsBytes = await ctx.stub.**getState**(**msgID**);*
   *const **msg** = JSON.parse(msgAsBytes.toString());*

2. Modify **msg** object subject to the following condition

   IF **msg.flag ≠** -1
   THEN
   delete **msg.emailID** in **msg**

Important: Any changes to **msg** do not reflect in the world state unless **putState()** is used.

3. No need to show these fields anyway

   *delete **msg.flag**;*
   *delete **msg.flaggers**;*
   *delete **msg.userID**;*

4. Return **msg** as JSON string

   *return JSON.stringify(**msg**);*

   Hence, the email ID is included in query response if and only if **msg.flag =** -1

Before moving to **queryAllMsgs** section, perform the following steps:

1. Query all messages using **user1** wallet
   $ node query.js -1 user1

queryAllMsgs txn : Takes no input and proceeds as follows:

1. Construct a range iterator over a set of keys in the ledger using ***getStateByRange*** function. Here *startKey = 0* and *endKey = 99999*

   *const iterator = await ctx.stub.**getStateByRange**(startKey, endKey);*

2. *iterator* is used to iterate over all **(msgID, msg)** pairs in the range *startKey* ≤ **msgID** < *endKey* using a while loop. Note: Code lines for error handling have been omitted for clarity.

```
while (true) {
const res = await iterator.next();
const Key = res.value.key;                    // msgID
msg = JSON.parse(res.value.value.toString('utf8'));
……....
}
```

3. Construct an array **allResults** as follows:

```
For each msg
    IF msg.flag ≠ -1
        THEN
            delete msg.emailID in msg
    delete msg.userID in msg
    delete msg.flag in msg
    delete msg.flaggers in msg
    Add msg to allResults
```

Important: Any changes to **msg** do not reflect in the world state unless **putState()** is used.

4. Return **allResults** as JSON string

```
return JSON.stringify(allResults);
```

++

Part 4: Miscellaneous

1. *first-network* has chaincode installed on only 2 peers: *peer0.org1* and *peer0.org2*. A query proposal sent to a peer which does not have chaincode installed results in a failure. Fix this by installing chaincode on all peers. See this commit.

2. * **initLedger** is the first txn that is invoked when the network is started. When a chaincode container is restarted after a crash/shutdown. the chaincode can correctly restore its global variables *users* and *msgID* using an **initledger** txn.

    1. What are **$HELLO$** messages?

Note that running *registerUser.js* only creates a wallet and registers user to the CA. The chaincode remains unaware of this user until the user invokes the chaincode by submitting its first txn.

A workaround is to have the client immediately submit a **createMsg** txn with **msgText = $HELLO$** using the user's wallet at the time of user registration.

$HELLO$ msgs have the following property:

1. They are not shown in query responses.
2. They cannot be flagged.

2. **initLedger** txn: It takes no input and proceeds as follows:

    1. Construct a range iterator over a set of keys in the ledger using ***getStateByRange*** function. Here *startKey = 0* and *endKey = 99999*

        *const iterator = await ctx.stub.**getStateByRange**(startKey, endKey);*

    2. *iterator* is used to iterate over all **(msgID, msg)** pairs in the range *startKey* ≤ **msgID** < *endKey* using a while loop.

    3. For each **msg**
        IF **msg.msgText = $HELLO$**
            push **msg.userID** in *users* array
        Increment **msgID** by 1

After an **initLedger** txn, the global variables of the chaincode are correctly restored.

++