

**EAS-IPM**

**Design Document – Life Insurance Network**

**VERSION 1.0**

****

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Prepared By / Last Updated By** | **Reviewed By** | **Approved By** |
| **Name** | Nisha M. R |  | Vinod Kumar Pai |
| **Role** | Sr. Developer |  | Program Manager |
| **Signature** |  |  |  |
| **Date** |  |  |  |

# Introduction

* 1. **Purpose**

The purpose of this document is to provide an explanation of the Life Insurance Network

* 1. **Background**

Life Insurance Network provides accurate details about an individual’s insurance details with multiple insurers using the PAN card number. All participants are notified through the ledger.

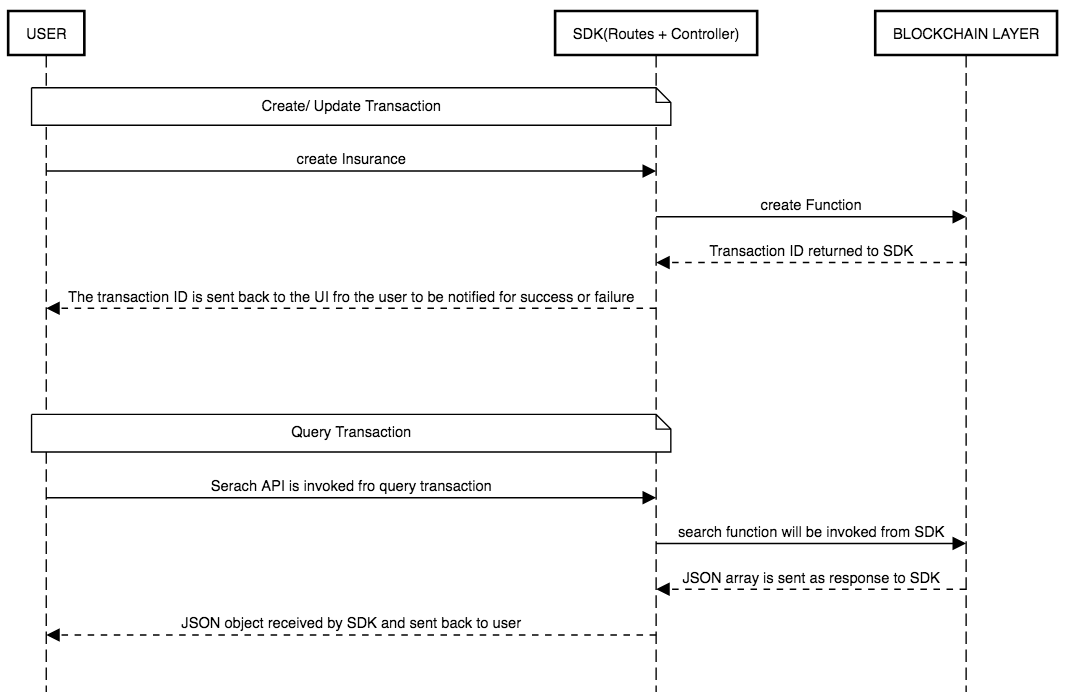
* 1. **Requirements and Dependencies**

The following were used to build the application.

1. Hyperledger Fabric – 1.4
2. Node JS - v8.11.3
3. GoLang - go1.12
4. Angular JS
5. IDE – Visual Studio Code
   1. **Network Architecture**

The application in divided into 3 layers.

1. User Interaction layer
2. SDK Layer
3. Blockchain Layer



* + 1. **User Interaction Layer**

The user interaction layer includes a presentation layer. This layer is built using HTML/Angular JS and CSS

* + 1. **SDK Layer**

This is the middle/interface layer which is the gateway to the blockchain. The middle layer provides the communication between user interaction layer and the blockchain layer. This layer is built using Node js. Java SDK is also available. The transaction submitted by the user is routed to controller file which is the node layer via a routes.js file. The controller file will have the orderer address and peer address which are participating in the network. The named of installed chaincode, the transaction and the arguments of the transaction is send to the chaincode in the form of request. The transaction is sent using sendTransactionProposal. The proposal response will be sent back to sdk by the chaincode and this response can be sent back to UI. For query transaction, the request is sent to chaincode using querybyChaincode.

Since private data is used, the collections\_config.json file path should be specified along with request. This json file will be taken into account for saving the needed data to both the organization’s peers.

Controller.js syntax

module.exports = (function() {

return{

    registerDrug: function(req, res){

        var channel = fabric\_client.newChannel('mychannel');

        var peer1 = fabric\_client.newPeer('grpc://localhost:7051');

        var peer2 = fabric\_client.newPeer('grpc://localhost:8051');

        var peer3 = fabric\_client.newPeer('grpc://localhost:9051');

        var peer4 = fabric\_client.newPeer('grpc://localhost:10051');

        channel.addPeer(peer1);

        channel.addPeer(peer2);

        channel.addPeer(peer3);

        channel.addPeer(peer4);

        var order = fabric\_client.newOrderer('grpc://localhost:7050')

        channel.addOrderer(order);

var collectionsConfigPath = path.join(os.homedir(),'/Desktop/WORKING/fabric-samples-release-1.2/chaincode/insurance\_private/collections\_config.json');

var input = req.body;

       var first\_name = input.firstName;

        var last\_name = input.lastName;

        var pan = input.pan;

        var dob = input.dob;

        var annual\_income = input.annualIncome;

        var company\_name = input.companyName;

        var requested\_amount = input.requestedAmount;

        var insured\_amount = input.insuredAmount;

        var claim\_status = input.claimStatus;

        var date\_of\_approval = input.dateOfApproval

        var comment = input.comments;

        var nominee = input.nominee;

const request = {

                //targets : --- letting this default to the peers assigned to the channel

                chaincodeId: 'insurance\_private',

                fcn: 'create',

                args: [first\_name, last\_name, pan, dob, annual\_income,company\_name,requested\_amount,insured\_amount,claim\_status,date\_of\_approval,comment,nominee],

                chainId: 'mychannel',

                txId: tx\_id,

                'collections-config': collectionsConfigPath

            };

        // send the transaction proposal to the peers

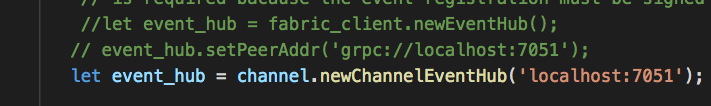
        return channel.sendTransactionProposal(request);

    }

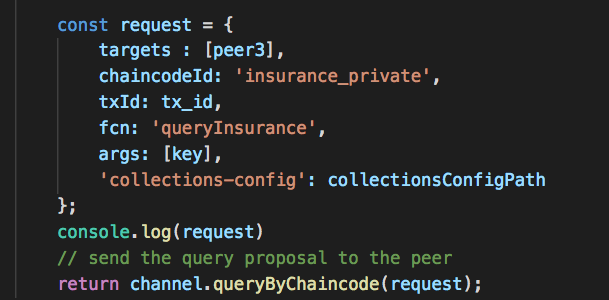
}

});

Here peer1 and peer 2 belongs to org1 and peer3 and peer4 belongs to org2.We have to specify the peer to which is acting as committing peer.



For search transaction, the target peer has to be specified along with request.



* + 1. **Blockchain Layer**

In the blockchain layer we have the chaincode or smart contract. The chaincode is written in Golang. The call to chaincode is done through SDK. The explanation of chaincode and the functions implemented are explained in later section.

* 1. **Application Participants**

1. ABC Insurance
2. XYZ Insurance
   1. **Application Description**

There are two participants in the network as mentioned in Application participants section.

An applicant can create insurance with either of the insurance company, ABC Insurance or XYZ Insurance using their pan card details.

ABC insurance allows for the coverage up to 3 times the annual income of the applicant and XYZ allows up to 5 times the annual income. When the applicant creates the insurance for the first time, the details will be stored in ledger and can be fetched using the pan card number. Insurance for the applicant will be rejected if salary variation greater than 30 percent is detected when applying for the insurance second time.

There is UI parameter called user risk. If the policy is rejected for more than 5 times for a particular applicant then he/she will be declared as high-risk user.

The nominee details for insurance created by one company will not be displayed to other. It will be explained in private data section.

In each of the organizations, one pair is defined as anchor peer. A channel named “mychannel” is created and the all the peers are added to the channel. There is a script file to start the hyperledger fabric network and to install the chaincode. While running the script file the peer, chaincode, orderer and CA containers are created.

The commands to install, instantiate and invoke chaincode needs to be specified in the shell script file in the first network of fabric-samples (utils.sh).

Syntax:

peer chaincode install -n insurance\_private

-v ${VERSION} -l ${LANGUAGE} -p ${CC\_SRC\_PATH} >&log.txt

  peer chaincode instantiate -o orderer.example.com:7050 -C $CHANNEL\_NAME -n insurance\_private -l ${LANGUAGE} -v ${VERSION} -c '{"Args":["init"]}' -P "OR ('Org1MSP.member','Org2MSP.member')" --collections-config $GOPATH/src/github.com/chaincode/insurance\_private/collections\_config.json

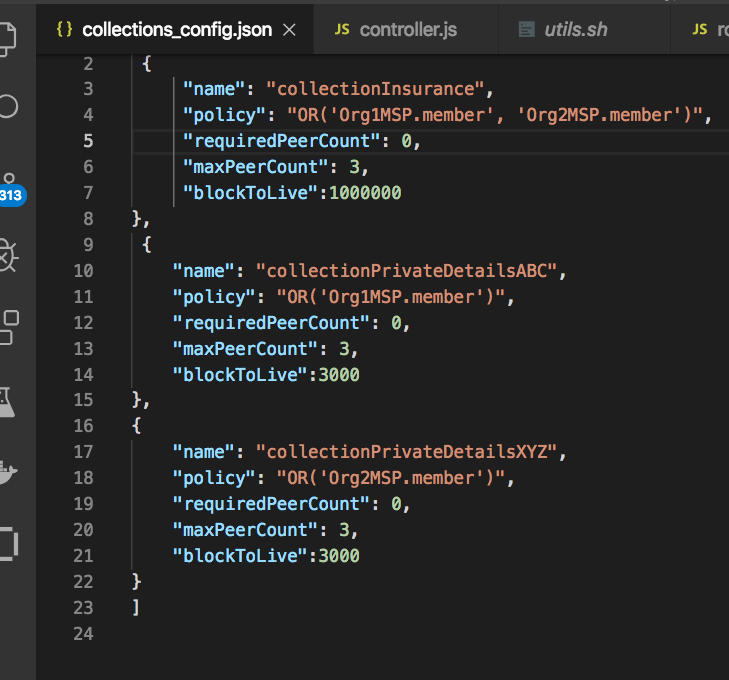
 peer chaincode invoke -o orderer.example.com:7050 --tls $CORE\_PEER\_TLS\_ENABLED --cafile $ORDERER\_CA -C mychannel -n insurance\_private $PEER\_CONN\_PARMS -c '{"Args":["create", "JOE", "DANIEL", "PANABC12", "JAN 10 2000", "5000", "ABC\_INSURANCE", "10000", "10000", "true", "Oct 23 2018","Coverage for Parents","cris"]}' >&log.txt

* 1. **Private Data**

There will be cases when one organization of a channel want to keep their data private from other organizations on that channel. That’s why private data is introduced in hyperledger fabric since version 1.2, so that there is no need to create separate channel for each of the organization. Private date can be implemented from within the channel.

[Private Data](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data/private-data.html)

For implementing private data, there is a collection\_config.json file which tells the details of the organization that implement the private data.



**name** - Name of the collection

**policy** - The distribution policy defines which organizations’ peers are allowed to persist the collection data expressed using the “signature” policy syntax, with each member being included in an “OR” signature policy list.

**requiredPeerCount** - Minimum number of peers (across authorized organizations) that each endorsing peer must successfully disseminate private data to before the peer signs the endorsement and returns the proposal response back to the client.

**maxPeerCount** - For data redundancy purposes, the maximum number of other peers (across authorized organizations) that each endorsing peer will attempt to distribute the private data to.

**blockToLive** - Represents how long the data should live on the private database in terms of blocks.

In this application, nominee is specified each time an insurance is created. Nominee details is specific to one insurance company only. The details of nominee details created by ABC Insurance for an applicant should not be displayed to XYZ Insurance. For this purpose, Private data collection is created for ABC Insurance and XYZ Insurance. “**collectionInsurance**” is used to store the details which are common for both the insurance companies and other two collections specified are for each of the organizations.

* 1. **Chaincode**

The chaincode is written in Golang. The packages needed for implementation are imported from “Shim”. The important functions defined for chaincode are

1. Init

Init function is called during initialization or update done on the chaincode. Init function takes ChaincodeStubInterface as parameter. When we call this function from CLI, it responds back with the peer response.

//Init - initialization of chaincode

func (t \*PharmaChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {

    return shim.Success(nil)

}

1. Invoke

All the logic implementation functions are written here. These implementation functions are called transactions. There are insert, update and query transactions. The transaction list is explained later.

Syntax:

func (t \*PharmaChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    function, args := stub.GetFunctionAndParameters()

    if function == "registerDrug" {

        return t.registerDrug(stub, args)

    }

    return shim.Error("Error invoking function")

}

* + 1. **Assets**

The following assets are defined in the chaincode.

1. Policy

**ID – PolicyNumber**

The details of the policy are stored in this structure. One applicant can create a maximum of 10 policies

1. User

**ID – PAN**

The user details except for nominee are stored in user. Policy struct is passed as an array in this struct

1. insurancePrivateDetails

**ID – PolicyNum**

Private data details are stored in this struct. Identified by policy number

* + 1. **Transactions**

1. **create**

This transaction is invoked while creating new insurance. Private data collections are updated during this transaction.

1. **queryInsurance**

This transaction is called for querying the common details from the collection,” collectionInsurance”.

1. **queryInsurancePrivateDataABC**

This transaction is invoked for querying the private details of ABC Insurance.

1. **queryInsurancePrivateDataXYZ**

This transaction is invoked for querying the private details of XYZ Insurance.

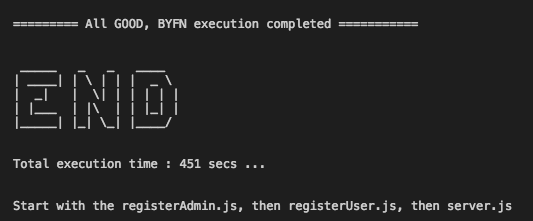
* 1. **Running Network**

For running the application, we have to start the fabric network, install the chaincode and then the node SDK is started. The following steps have to be followed.

1. Navigate to the folder which contain the shell script for starting the fabric. Here there is a startFabric.sh script. When we run this script, it will call the byfn.sh script in the network folder.

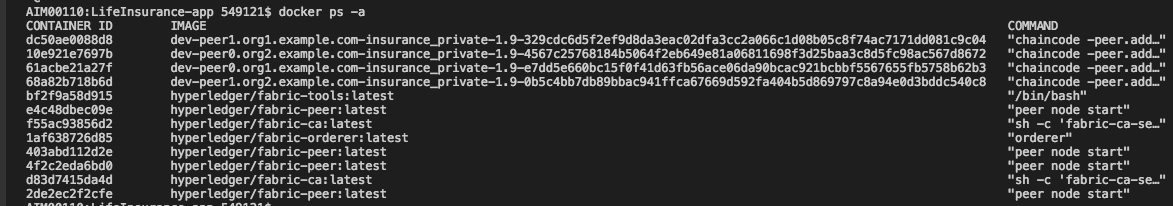
**./startFabric.sh**

If this script is run successfully, the following result will be displayed.



1. We can see the list if containers that are up and running using the following command.

**docker ps –a**



Now the chaincode is installed and instantiated in all the peers of the channel.

1. Node modules

The required node modules need to be installed. This is done using command

**npm install**

This will install the packages that are predefined in package.json. If any error specifying package is not available, we can install that using command npm install.

1. **registerAdmin.js** and **registerUser.js**

The admin and users have to be enrolled for each ca. In the network since this is multi org, there are two ca containers and using the certificate file of ca we have to enroll admin and user.

For enrolling admin

**node registerAdmin.js**

For enrolling user

**node registerUser.js**

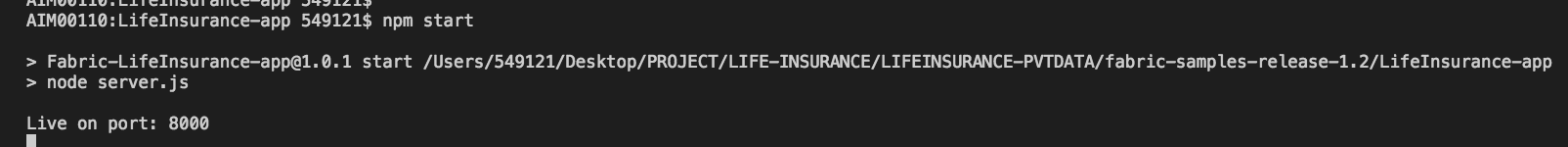
This will create a **hfc-key-store** in the SDK folder and create the user and admin. This is later used by controller when committing a transaction.

1. Now we have to start the node app

Go to the root folder and hit the following command

**npm start** OR **node server.js**

The following will be displayed if the app is started successfully.



We can hit the application using url: <http://localhost:8000/>

