Insurance Claim

Overview:

Blockchain network made using Hyperledger Fabric, written in Golang, deployed on local machine via Docker-Compose. Web backend written in Python, deployed on local machine via Flask. Interface via backend is through REST APIs. The backend and blockchain also communicate through REST APIs.

Transaction Flow:

The following transactions will be entered into the blockchain.

* Insurance Holder buys a product and insures it. Insurance company makes a transaction in blockchain.
* Insurance holder raises a claim.
* Insurance company raises a request for verification of the claim.
* Police agency validate the claim.
* Insurance company decide whether to settle the claim or not.

Interface:

Web Backend APIs

This results in transaction on the blockchain.

1. Insurance company POSTs the insurance purchase.
2. Insurance company can get the dashboard for all purchases via GET insurance purchases.
3. Insurance buyer POSTs the claim requests.
4. Insurance company can get the dashboard for all claim requests via GET claim requests.
5. Insurance company POSTs the verification request.
6. Police agency can get the dashboard for all verification requests via GET verification requests.
7. Police agency POSTs the results of the verifications claimed.
8. Insurance agency can get the dashboard for all completed verifications via GET claimed verifications.
9. Finally the insurance agency POSTs the claim settlement results.
10. The insurance buyer can check the claim settlement result dashboard to know the status of his claim via GET claim settlements.

Blockchain APIs:

1. POST the record
2. GET the record via key

Model Design:

* Insurance Purchase
  + insurancePurchaseId
  + customerName
  + insurerName
  + productName
  + productCost
  + issueDate
  + endDate
  + premiumAmount
  + coveredCost
* Claim Request
  + claimRequestId
  + insurancePurchaseId
  + claimDate
  + claimDetails
* Verification Request
  + verificationRequestId
  + customerName
  + endDate
  + productName
  + claimDetailsHash=claimDate+claimDetails
* Claim Validation
  + claimValidationId
  + verificationRequestId
  + firNumber
  + claimStatus
  + additionalComments
* Claim Settlement
  + claimSettlementId
  + claimValidationId
  + settlementResult
  + additionalComments

Blockchain Implementation Architecture:

To get the entire assignment done in time, I decided to reuse an old hyperledger fabric project of mine. This contains 1 consortium of Volkswagen, which has 2 organizations, MRF and Gabriel. They have 2 peers each, and are a part of a channel, indianChannel. The chaincode is a basic one, which performs read and write operations. The 2 APIs here invoke the chaincode to perform operations according to the parameters set. The default LevelDb is used to store key-value pairs. Ideally, to store the entire model, CouchDb should have been used. But to simplify, I store the sha256 hash of the model as the value, and the id as the key. **The user may calculate the hash on his/her side and ensure the validity of the data by comparing the value for his key on blockchain.**

Steps:

Create the cryptoconfig file and use cryptogen to generate certification for the mentioned organisation and peers. Create the configtx file to create the artifacts, anchor peers and the default channel. Create the docker compose file to get the network up and running. In the docker compose file, create the required containers for orderers, peers and certificate authorities.

I am using the fabric’s goSdk to create the application for this network. So create the config file for the sdk. Write a basic chaincode to read and write from ledger. Write the setup file to create the sdk, channel and install-instantiate the chaincode. Write rest apis in golang which read and write to the ledger on proper authentication.

Backend Implementation:

I have written Rest APIs using Flask in Python, for implementing the interface. To simplify, all storage is in python dictionaries. So it’s temporary, and gets deleted when the session ends. Testing was done via Postman application for all APIs.

Application Architecture:

