Transaction Processor:

When transactions comes through from clients and go toward the rest api its the job of the validator to route out those to the right transaction processor/transaction family which does the business logic. With regard to the code on the repository we will walk through the core steps/procedures followed for our Transaction processor section:

- we import the transaction processor function from the library and create a new object which connects to specific validator (tcp://localhost:4004)(imported from env.).

```
const { TransactionProcessor } = require('sawtooth-sdk/processor')
const transactionProcessor = new
TransactionProcessor(env.validatorUrl)
const loginHandler = require('./int')
```

- In env.js file we prepare to sort out the routing from which we connect to the sawtooth server (`'tcp://localhost:4004'`)

```
const env = require('./env')
dotenv.config()
dotenv.config()
const env = {
validatorUrl: process.env.VALIDATOR_URL || 'tcp://localhost:4004'
}
 module.exports = env
```

- followed by the import we add a handler function on the tp object which is imported from int,js file to init.

```
const { TransactionHandler } =
require('sawtooth-sdk/processor/handler')
const loginHandler = require('./int')
```

- finally by using start() we initiate our transaction processor,while listening to 4004 port.

```
transactionProcessor.start()
```

On the sequel we have int.js where we receive and store the payload which is being sent from the payload.

- we initialize it by loading up with class handler and hash function(crypto) {createHash} for it's purpose along with constants to held it Transaction Family,

```
const { TransactionHandler } =
require('sawtooth-sdk/processor/handler')
const { createHash } = require('crypto');
const cbor = require('cbor')

const TP_FAMILY = 'loginForm'
const TP_NAMESPACE = hash(TP_FAMILY).substring(0, 6);
const TP_VERSION = '1.0';
```

- signers public key, also updates the input output where the addresses can be read from the ledger in the transaction which makes data accessible/readable, we have the apply method to define all the business logic. we register our family Name,version and namespace.

- next we we decode the payload that we receive from client here, again we use cbor to serve its purpose, core aspect of the whole comes the role state address as paraphrased in the sawtooth documentation, Sawtooth stores its data within a Merkle tree comprising of 35 bytes, represented as 70 Hex characters. in short the first 6 character are mended with the namespace for the transaction processor to take place.

```
let header = transactionProcessorRequest.header
 const publicKey = header.signerPublicKey
 console.log('Public KEY' + publicKey)
 const address = TP_NAMESPACE + hash(publicKey).slice(-64);
```

- the transaction processor request object comprising of its header,header signature and payload,getting the payload from the header furthermore we call the apply method where all our business logics are being stored. We have transactionprocessorreequest and context,one checks on the validation of the transaction the other the storage.

```
   apply(transactionProcessorRequest, context)
```

```
        console.log('apply')
```

- Since the data we receive are in serialized manner and encoded we need to decode it, for this we use cbor.

```
let payload = cbor.decode(transactionProcessorRequest.payload)
       console.log(payload)
```

.

- finally we have the function for storing our data to the state, here we have the context address and payload, we encode the payload then create a variable where we place the key on the address and payload is where our data are being processed and stored.

```
class loginHandler extends TransactionHandler {
constructor() {
 console.log(`Iniitialising the Transaction Family with the name
${TP_FAMILY} and with the address ${TP_NAMESPACE}`)
 super(TP_FAMILY, [TP_VERSION], [TP_NAMESPACE]);
}
```

Client:

Client serves as a middleware it is the process of encoding the informations to be submitted which later validates and moves down to Transaction processor where the logic start it mostly serves as a data pre-processor.

- We will be using Hapi framework for connecting it to our login server (localhost:5000) from which we connected/created its path/server, Alongside using MongoDB to store on the data we sent. We will be using mongoose for convenience as to connect it with it's schema for better object storage.

```
const Hapi = require('hapi')
const mongoose = require('mongoose')
const server = new Hapi.server({
host: 'localhost',
port: 5000,
routes: {
```

```
        cors: true
      }
})
```

- For setting up the environment its optional we can either have in a separate file or make it work on server.js itself(since we run on it'server.js',visually we can see the output of host and server,along with keys which are also executed simultaneously)for accessibility/readability we find it more convenient.

  In env.js we set up the environment
  dotenv then .env file(where public/private keys being stored) which then generates process.env

```
const dotenv = require('dotenv')

const { leafHash } = require('./sawtooth-client')
```

Furthermore creating const env Where we store the object values to private,public(which is generated by private key),

```
dotenv.config()
const env = {
privateKey: process.env.PRIVATE_KEY || '',
publicKey: process.env.PUBLIC_KEY || '',
restApiUrl: process.env.REST_API_URL || 'http://localhost:8008',
familyName: 'loginForm',
familyPrefix: leafHash('loginForm').substring(0, 6),
familyVersion: '1.0'
}

module.exports = env
```

(Which we would be exporting it to server.js and init.js)

- In init.js it generates the key which is deployed from env.js

```
const { createContext, CryptoFactory } =
require('sawtooth-sdk/signing')
```

```
const fs = require('fs')
const path = require('path')

const env = require('./env')
const context = createContext('secp256k1')
const privateKey = context.newRandomPrivateKey()


const output =
`PRIVATE_KEY=${privateKey.asHex()}\nPUBLIC_KEY=${signer.getPublicKey
().asHex()}\nREST_API_URL=http://localhost:8008`
```

(as stated above we will generate random key from which it will generate public key)

We need signer Inorder to identify/confirm the identity of the transaction and for sending the information to the validator we need to generate key,which in our case is the only way to proving the identity of the block/payload.

```
const signer = new CryptoFactory(context).newSigner(privateKey)
```

- Moving on enclave is another important element where we generate the key we use to 256k1 standard for signing Here the private key is the random set of 32 digit which the user passes.

```
const { createHash, randomBytes } = require('crypto')
const secp256k1 = require('secp256k1/elliptic')

const createPrivateKey = () => {
   let privateKey
   do {
       privateKey = randomBytes(32)
   } while (!secp256k1.privateKeyVerify(privateKey))
   return privateKey
}...
```

- In user.js we have the routing of our api server along with payload, We will be using bcrypt for hashing the data, and import sawtooth -client factory which sends our payload to validator then to transaction processor.

```
let jwt = require('jsonwebtoken')
const bcrypt = require('bcryptjs')
const User = require('../models/User')
const mongoose = require('mongoose')
const { EnclaveFactory } = require('../enclave')
const { SawtoothClientFactory } = require('../sawtooth-client')
const request = require('request')
const env = require('../env') ...
```

Note: User.js file comprises of the schema(mongoose) of our loginForm.

- Payload/udataser is a very important component in client side it is the actual raw data like in these case data we receive from the form manipulating the changes which is required to be applied to the state.

```
      const payload = userData
      const postpayload = await intkeyTransactor.post(payload)
      console.log('Response from the sawtooth client',
postpayload.status, postpayload.statusText)
```

- Sawtooth-client.js is basically the heart of the client where we set up the transaction before sending it to validator which then passes it down to the Transaction processor .

```
const { randomBytes, createHash } = require('crypto')
const axios = require('axios')
const cbor = require('cbor')
const protobuf = require('sawtooth-sdk/protobuf')
const fs = require('fs')
const leafHash = (input) => {
```

```
    return
createHash('sha512').update(input).digest('hex').toLowerCase().slice
(0, 64)
}
```

In brief ,crypto is an algorithm used for encrypting/securing user data, Axios is where we make the ajax/http request, cbor for hashing, protobuf is how we serialize the structured data. Cbor is where we encode and parse our data.

- We will looking at the flow of preparation of transaction in brief, transactionclientfactory functions which keeps track of the process in a hierarchical manner.

Firstly we have the transactionoption where we record/configure of the transaction which is to happen such as in term of encryption,encoding etc.

```
newTransactor(transactorOptions) {
const _familyNamespace = transactorOptions.familyNamespace ||
leafHash(transactorOptions.familyName).substring(0, 6)
const _familyVersion = transactorOptions.familyVersion || '1.0'
const _familyEncoder = transactorOptions.familyEncoder || cbor.encode
        return {
            async post(payload, txnOptions) {
```

- Second step is to encode our payload to byte, payload is the text we entered in our form for its purpose we use _familyencoder

```
const payloadBytes = _familyEncoder(payload)
```

- TransactionHeader comprises of the informations of transaction, routing of transaction to correct transaction processor.

```
const transactionHeaderBytes = protobuf.TransactionHeader.encode({
                        familyName: transactorOptions.familyName,
                        familyVersion: _familyVersion,
                        inputs: [_familyNamespace],
                        outputs: [_familyNamespace],
                                                signerPublicKey:
factoryOptions.enclave.publicKey.toString('hex'),
                                               batcherPublicKey:
factoryOptions.enclave.publicKey.toString('hex'),
                        dependencies: [],
```

```
                        nonce: randomBytes(32).toString('hex'),
                                                    payloadSha512:
createHash('sha512').update(payloadBytes).digest('hex'),
                        ...txnOptions
                    }).finish()
```

- After transactionHeader it is then followed by creating the transaction which redirects us to the current routing address.

- Sign the txn header. This signature will also be the txn address.

```
const
=txnSignature=factoryOptions.enclave.sign(transactionHeaderBytes).toString
('hex')
```

- We create the transaction where the actual header,header signature and payload,we create the current transaction array.

```
const transaction = protobuf.Transaction.create({
                        header: transactionHeaderBytes,
                        headerSignature: txnSignature,
                        payload: payloadBytes
                    })
const transactions = [transaction]
                    const batchHeaderBytes =
protobuf.BatchHeader.encode({
                        signerPublicKey:
factoryOptions.enclave.publicKey.toString('hex'),
                        transactionIds: transactions.map((txn) =>
txn.headerSignature),
                    }).finish()
```

  Note: Similarly like how we created transaction header we create the transaction bytes as well and encode them.

- We sign the batch header and create the batch.

```
const transactions = [transaction]
                    const batchHeaderBytes =
protobuf.BatchHeader.encode({
```

```
                        signerPublicKey:
factoryOptions.enclave.publicKey.toString('hex'),
                        transactionIds: transactions.map((txn) =>
txn.headerSignature),
                    }).finish()
```

- We then follow it up with the batches which are then submitted to the batchlist. With the inclusion of all the header,headerbytes and transaction the header has all the batch ids of the transaction.

```
const batchSignature =
factoryOptions.enclave.sign(batchHeaderBytes).toString('hex')
                    const batch = protobuf.Batch.create({
                        header: batchHeaderBytes,
                        headerSignature: batchSignature,
                        transactions: transactions
                    })


                    const batchListBytes =
protobuf.BatchList.encode({
                        batches: [batch]
                    }).finish()
```

- Signing the batchheader into a list and finally posting them to validator via the REST api,which allows our client to communicate with the validator.

```
                    try {
                        const res = await axios({
                            method: 'post',
                            baseURL: factoryOptions.restApiUrl,
                            url: '/batches',
                            headers: { 'Content-Type':
'application/octet-stream' },
                            data: batchListBytes
                        })
```

```
        return res
    } catch (err) {
        console.log('error', err)
```

In conclusion, The client section can easily be followed by crosschecking and executing the codes attached. .