Helloworld Application

This application shows how to use the Sawtooth JavaScript SDK to develop a simple application.This is a simple Hyperledger application with very basic functionality like write a text data to the blockchain network and also read that data from the same network.

The client is responsible for creating and signing transactions, combining those transactions into batches, and submitting them to the validator.

Information about Helloworld Transaction Family

FAMILY_NAME ='HelloWorld' ;
FAMILY_VERSION ='1.0';

Tasks to be completed:

1. Create a signer
2. Create an address
3. Serialize the payload
4. Steps for creating Transaction
   4.1 Create TransactionHeader
   4.2 Create Transaction
5. Steps for creating Batch
   5.1 Create BatchHeader
   5.2 Create Batch from transaction list
6. Create BatchList from Batches
7. Send BatchList to validator via the REST API
8. Write GET state call to validator via the REST API

Step 1 : Create Signer

In order to confirm the identity and sign the information send to the validator we need a key (256-bit).This key is the only way to prove the identity on the blockchain.

- First we need a private key it's basically just a random set of 32 bytes.
- Here variable 'privateKey' is a private key of a user that passes from UI side.
- For our exercise, we will be hard-coding the private key.
- Sawtooth uses the **secp256k1 ECDSA** standard for signing, which means that almost any set of 32 bytes is a valid key.
- It is fairly simple to generate a valid key using the SDK's signing module.
- By using that private key we generate a public key.

```
const {CryptoFactory, createContext } = require('sawtooth-sdk/signing')
const {Secp256k1PrivateKey} = require('sawtooth-sdk/signing/secp256k1')

privateKey="66ad89d0ff29b0267fba72ea8d40ef7975e10f8acde8d50d20cdf56ba9599c5e";

constructor(){
   const context = createContext('secp256k1');
   const secp256k1pk = Secp256k1PrivateKey.fromHex(privateKey.trim());
   this.signer = new CryptoFactory(context).newSigner(secp256k1pk);
   this.publicKey = this.signer.getPublicKey().asHex();

}
```

Step 2 : Address Generation

A state address in Sawtooth is 35 bytes long, typically expressed as 70 hexadecimal characters. By convention, the first six characters are reserved for a namespace for the transaction family. The remaining 64 characters are up to each family to define.

All helloworld application transactions have the same 6 hex digit prefix, which is the first 6 hex characters of the SHA-512 hash of family name ('Helloworld') (that is, "8ae6ae"). And remaining 64 hex characters of each client address is different, which is first 64 hex characters of the Sha-512 hash of client's Public Key (c2879b0209cfa70c3f464ab38177341707edac6b2bc58e974141d2fd46b1d44a).

```
const {createHash} = require('crypto')

function hash(v) {
 return createHash('sha512').update(v).digest('hex');
}

this.address =  hash("HelloWorld").substr(0, 6) + hash(this.publicKey).substr(0, 64);
```

Step 3 : Serialize the payload

Clients convert the payload to an encoded bytes format. In the helloworld application, the payload is a plain text entered by the user. We will use simple 'utf8' to encode our payload to bytes. We are doing this because the sawtooth transaction payload should be in bytes. You can use any kind of encoding format for your payload.

The serialized payload would look something like

100,102,102,100,102,100

```
const {TextEncoder, TextDecoder} = require('text-encoding/lib/encoding')

var payload = values;
 var encode =new TextEncoder('utf8');
 const payloadBytes = encode.encode(payload)
```

Step 4 : Steps for creating Transaction

Transactions **are the basis for individual changes of state to the Sawtooth blockchain.Transaction contains binary payload, a binary-encoded *TransactionHeader and a signature of that header***

4.1 Create TransactionHeader

A TransactionHeader contains information about a transaction, including details for routing a transaction to the correct transaction processor. Transaction Header is encoded using the *TransactionHeader* protobuf (available in the sawtooth sdk) to create transactionHeaderBytes.

```
const protobuf = require('sawtooth-sdk/protobuf')

const transactionHeaderBytes = protobuf.TransactionHeader.encode({
familyName: ,
familyVersion: ,
inputs: ,
outputs: ,
signerPublicKey: ,
batcherPublicKey: ,
dependencies: [],
payloadSha512:
}).finish()
```

4.2 Create Transaction

1.Once the TransactionHeader is constructed, its bytes are then used to create a signature. Resulting signature is stored in the header signature also act as an ID of the transaction.

2. The header bytes, the header signature, and the payload bytes are used to create complete Transaction. Transactions should be encoded using the *Transaction* protobuf (available in the SDK)

```
const transaction = protobuf.Transaction.create({
  header: transactionHeaderBytes,
  headerSignature: this.signer.sign(transactionHeaderBytes),
  payload: payloadBytes
 })
```

Step 5 : Steps for Creating Batch

After creating transaction they must be wrapped in a *Batch*.

Similar to the TransactionHeader, there is a *BatchHeader* for each Batch.

5.1 Create BatchHeader

As Batches are much simpler than Transactions, a BatchHeader needs only the public key of the signer and the list of Transaction IDs, in the same order they are listed in the Batch.

```
const transactions = [transaction];
const batchHeaderBytes = protobuf.BatchHeader.encode({
signerPublicKey: this.signer.getPublicKey().asHex(),
transactionIds: transactions.map((txn) => txn.headerSignature),
}).finish();
```

5.2 : Create Batch from transaction list

Creating a Batch is similar to creating a transaction. A batch contain header, header signature , transactions. BatchHeader bytes is signed by the signer and resulting signature is the headerSignature. It also act as an ID of the batch.

```
const batchSignature = this.signer.sign(batchHeaderBytes);
 const batch = protobuf.Batch.create({
   header: batchHeaderBytes,
   headerSignature: batchSignature,
   transactions: transactions,
 })
```

Step 6 : Create BatchList from Batches

In order to submit Batches to the validator, Batches must be collected into a BatchList . Multiple batches can be submitted in one BatchList.

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

Step 7 : Send BatchList to validator via the REST API

```
const fetch = require('node-fetch');


try{
  let resp =await fetch('http://rest-api:8008/batches', {
method: 'POST',
    headers: {
 'Content-Type': 'application/octet-stream'
    },
    body: batchListBytes
     })
       console.log("response", resp);
    }
     catch(error) {
       console.log("error in fetch", error);
   }
```

Step 8 : Write GET state call to validator via the REST API

We can query the data that we have written to the sawtooth state, using the state endpoint of the REST API.

```
try{
  var geturl = 'http://rest-api:8008/state/'+this.address
  let response=await fetch(geturl, {
    method: 'GET',
  })
  let responseJson = await response.json();
    var data = responseJson.data;
    var newdata = new Buffer(data, 'base64').toString();
    return newdata;
  }
  catch(error) {
    console.error(error);
  }
```

Additional Lab Exercises

1. **Send multiple data from client to transaction processor**

   Existing application sends single data (ie a text) to the transaction processor for storing.Instead of sending single data, modify your client application to send multiple fields of data. For example create two text boxes for entering your first name and last name. Combine these two strings and store this in the state. You should also display the stored data.

2. **Create batch by combining two transactions**

   Existing application sends one transaction per batch to transaction processor for storing.Instead of sending one transaction per batch from client application , send multiple transactions in a single batch from client to transaction processor