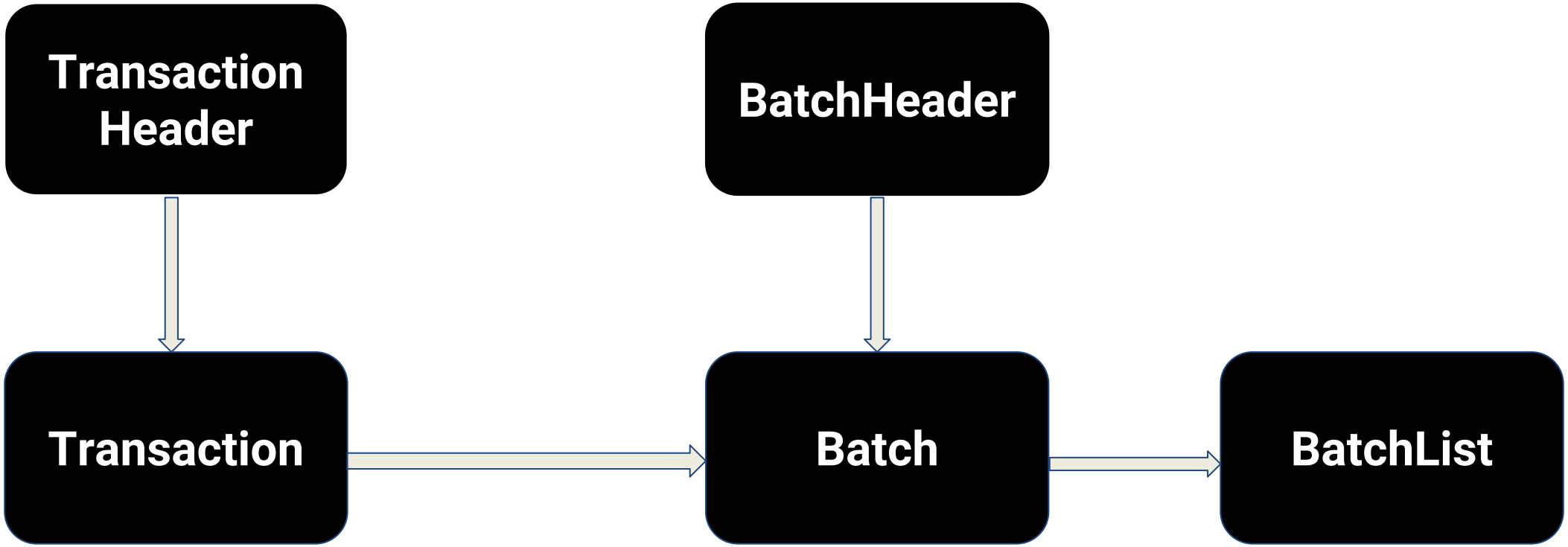
Writing a Client

Sawtooth : Client Writing

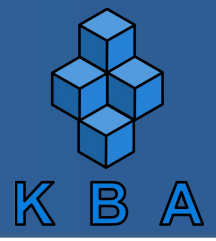
- Create Transaction using payload ,header and signature
- Converting Payload into Bytes
- Create Transaction header
- Create Batch from Transaction(s)
- Create BatchList from Batch(es) convert it into bytes
- Creating and using a Signer
- Address Generation example
- Post and Get the data using the Rest API

Building the Transaction



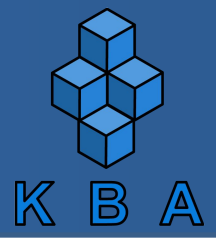
Create the Transaction

- Transactions are the basis for individual changes of state to the sawtooth blockchain.
- TransactionHeader is constructed, its bytes are then used to create a signature.
- This header signature also acts as the ID of the transaction.
- The header bytes, the header signature, and the payload bytes are all used to construct the complete Transaction.



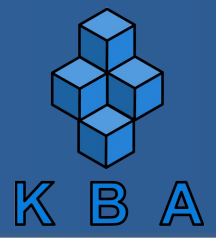
Create a Transaction

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

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



Converting Payload to Bytes

- Transaction Payload is the part of transmitted data that is the actual intended message.
- The payload is used during transaction execution as a way to convey the change which should be applied to state.
- The logic for converting to bytes rests entirely within the particular Transaction Processor itself.
- It can be using methods like csv, *utf8*, *cbor* or *protobuf*.



Converting Payload to Bytes

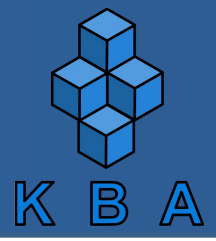
```
const {TextEncoder} = require('text-encoding/lib/encoding')

payload = values;

var enc = new TextEncoder('utf8');

const payloadBytes = enc.encode(payload)
```

Create the Transaction Header

- A TransactionHeader contains information for routing a transaction to the correct transaction processor.
- Input and Output state addresses for read and write, references to previous transactions it depends on, and the public keys associated with its signature.
- The header references the payload through a SHA-512 hash of the payload bytes.
- Payload field can be verified by performing SHA-512 hash of the payload field and comparing it with the payload SHA-512 field in the TransactionHeader.

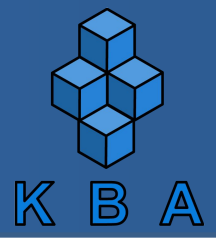
TransactionHeader Fields

TransactionHeader(Deserialized)
batcherPublicKey string **signerPublicKey** string **familyName** string **familyVersion** string **inputs** [string] **outputs** [string] **dependencies** [string] **nonce** string **payloadSha512** string



Create the Transaction Header

```
const transactionHeaderBytes = protobuf.TransactionHeader.encode({
    familyName: 'helloworld',
    familyVersion: '1.0',
    inputs: [97fd77f7eabdd2c77fd852cf6c16a7a4429ab27aff394493ab7c41609c759ddb696f2e],
    outputs: [97fd77f7eabdd2c77fd852cf6c16a7a4429ab27aff394493ab7c41609c759ddb696f2e],
    signerPublicKey: this.signer.getPublicKey().asHex(),
    nonce: //a random string ,
    batcherPublicKey: this.signer.getPublicKey().asHex(),
    dependencies: [],
    payloadSha512: this.f(payloadBytes),
    }).finish();
```



Creating a Batch

- When one or more Transaction instances are ready, they must be wrapped in a *Batch*. Batches are the atomic unit of change in Sawtooth's state.

- When a Batch is submitted to a validator, each Transaction in it will be applied (in order), or *no* Transactions will be applied.

Creating a Batch

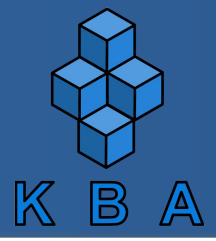
- Creating a Batch is similar to creating a transaction.
- The header is signed, and the resulting signature acts as the Batch's ID.
- The Batch is then constructed out of the header bytes, the header signature, and the transactions that make up the batch.

Batch
header *bytes* **headerSignature** *string* **transactions** *[Transaction]*

BatchHeader Fields

- Even if the Transactions are not dependent on any others, they cannot be submitted directly to the validator.

BatchHeader *(deserialized)*
signerPublicKey *string* **transactionIds** *[string]*

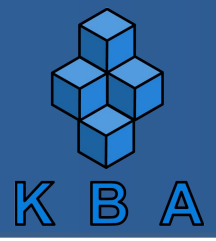


Create the BatchHeader

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

- 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();
```



Create the Batch

```
const signature = signer.sign(batchHeaderBytes)

const batch = protobuf.Batch.create({
header: batchHeaderBytes,
headerSignature: signature,
transactions: transactions
});
```

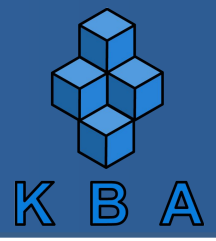
Encode the Batch(es) in a BatchList

- In order to submit Batches to the validator, they must be collected into a *BatchList*.

- Multiple batches can be submitted in one BatchList, though the Batches themselves don't necessarily need to depend on each other.

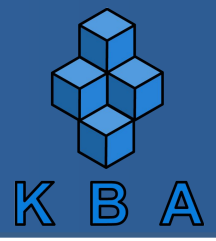
- Unlike Batches, a BatchList is not atomic. Batches from other clients may be interleaved with yours.

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



Creating a Signer

- 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.
- A *Signer* wraps a private key and provides some convenient methods for signing bytes and getting the private key's associated public key.



Creating a Signer

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

const context = createContext('secp256k1')
const privateKey =
55006afc3d3204844b7ebedbafdf2c4b453e5fd1e7c6558e832d75cf221
c61bb
const signer = CryptoFactory(context).newSigner(privateKey)
```

- https://sawtooth.hyperledger.org/docs/core/releases/1.0/javascript_sdk/signing/index.html
- Using the signer

Address Generation Example

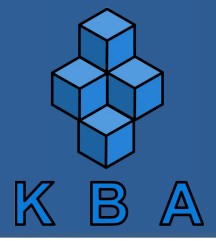
- All *HelloWorld* application transactions have the same 6-hex digit prefix, which is the first 6 hex characters of the SHA-512 hash of the family name "*HelloWorld*"
- Here the addresses would start with "*8ae6ae*"
- The remaining 64 characters can be calculated by sha512 hashing the client's public key and taking the first 64 hex characters from the hashed output.

```
const {createHash} = require('crypto')
function hash(v) {
return createHash('sha512').update(v).digest('hex');
}
address = hash("HelloWorld").substr(0,6) + hash(this.publicKey).substr(0,64);
```

Submitting Batches to the Validator(REST API)

POST/batches

- The prescribed way to submit Batches to the validator is via the REST API.
- This is an independent process that runs alongside a validator, allowing clients to communicate using HTTP/JSON standards.
- Simply send a *POST* request to the */batches* endpoint, with a *"Content-Type"* header of*"application/octet-stream"*, and the *body* as a serialized *BatchList*



POST Method(REST API)

```
resp = await fetch('http://rest-api:8008/batches', {
        method: 'POST',
        headers: {
                'Content-Type': 'application/octet-stream'
            },
        body: batchListBytes
    })
```

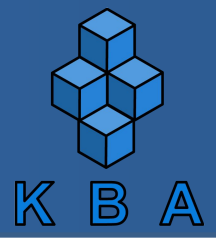
GET Method(REST API)

GET/state/{address}

- fetches a single resource
- Uses a full 70-char address

GET/state?address={prefix}

- fetches multiple encoded entities from state
- use an address prefix to filter the results



GET Method

```
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;
   }
```

THANK YOU