HYPERLEDGER SAWTOOTH

App Development

What is a Blockchain?

What is Hashing?

Creates a deterministic, fixed-length, digest of some arbitrary data

Even slightly different data produces a completely different hash

```
sha256('Hello, World!')
// dffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688a362182986f

sha256('Hello, World?')
// f16c3bb0532537acd5b2e418f2b1235b29181e35cffee7cc29d84de4a1d62e4d
```

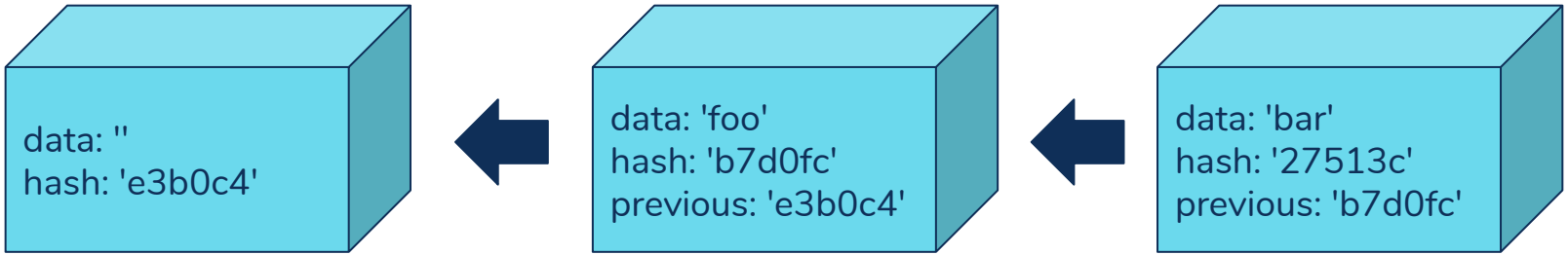
Basic Blockchain Structure

Hashes link discrete "blocks" of data

Hash is generated by combining the data with previous hash

"Genesis" block only one allowed to have no previous hash

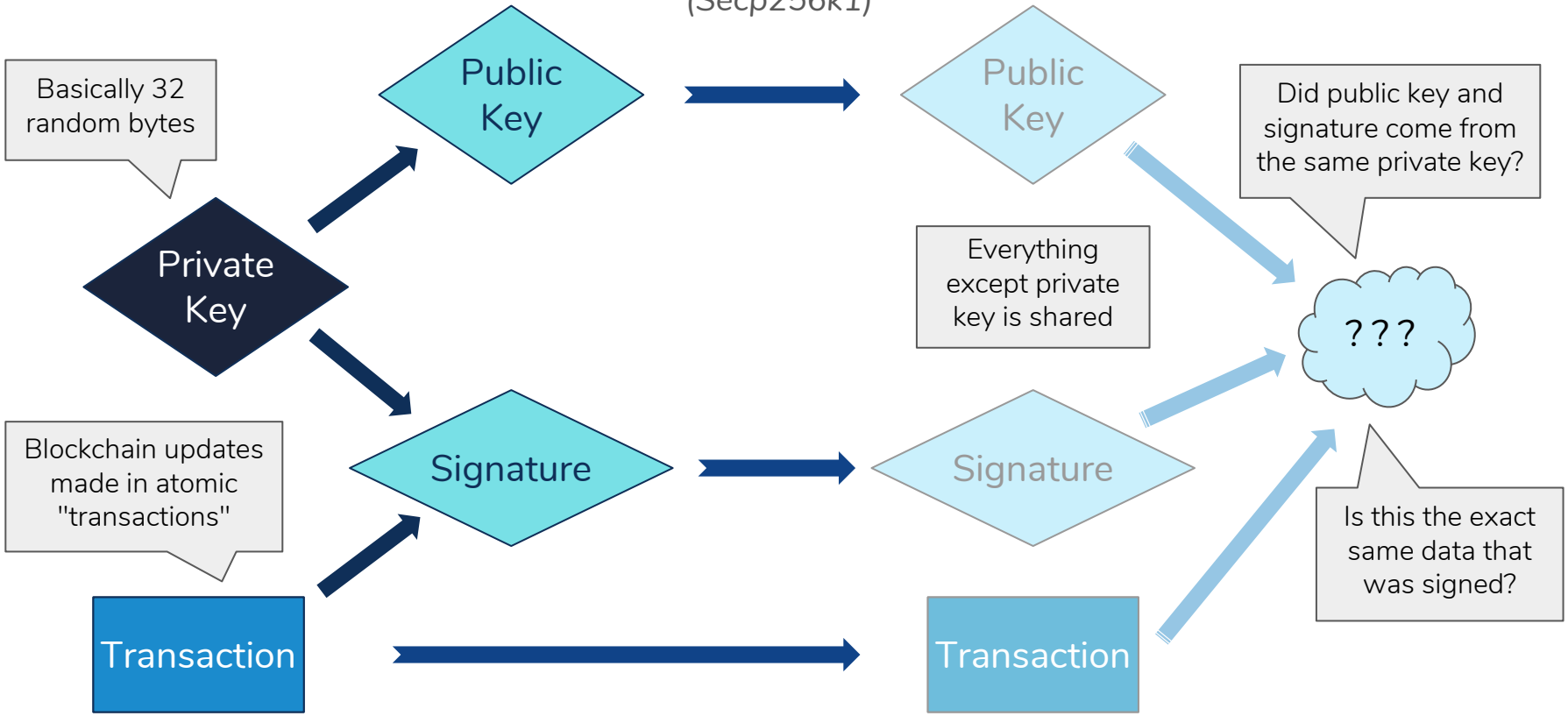
Immutable. Cannot alter blocks without altering **every** later block



sha256(data + previous) // first six chars displayed

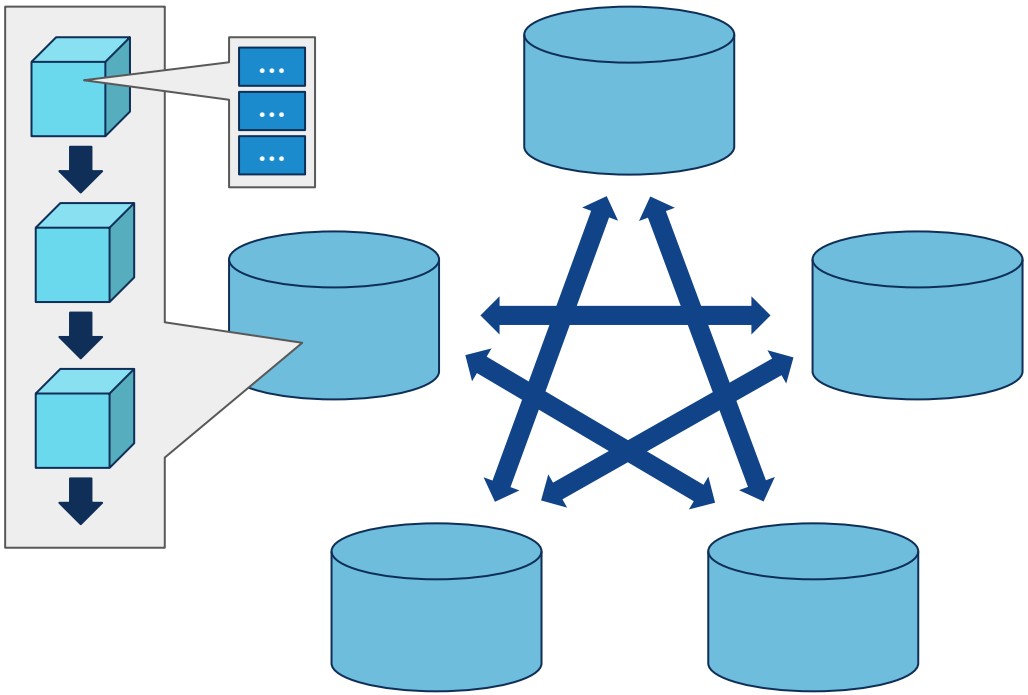
Signing

(Secp256k1)



Consensus

(Byzantine Fault Tolerance)



Use a lottery to determine who gets to create the next block

Always prefer "longest" chain, bad actors need 51% to catch up

Bitcoin and Ethereum use **Proof of Work** to randomly choose a "leader"

Sawtooth features **Proof of Elapsed Time** (and others)

Why Blockchain?

Share a database between mutually distrusting entities

No reliance on trusted third parties

Immutable transaction history

High availability

- Crash fault tolerant
- Byzantine fault tolerant
- Liveness

Why NOT Blockchain?

Wrong use case:

- Internal-only business model
- A centralized blockchain is just bad database

Active areas of research:

- Transaction throughput
- "Private" transactions

What is Sawtooth?

What is Sawtooth?

Began as a hardware experiment in **Intel**'s labs

Contributed to **Linux Foundation** as a part of Hyperledger

1.0 released February 2018

General purpose blockchain platform

Designed as a permissioned blockchain for consortiums

Not tied to specific hardware

Permissioned Blockchains

Targeted at small(ish) groups

Endpoints are restricted, not available to the general public

Not centralized

No single canonical peer-to-peer network (deploy your own!)

No need for a currency (different incentives)

Why Sawtooth?

Highly modularized

Designed to scale:

- On-chain settings
- Smart contracts
- Parallel transaction processing

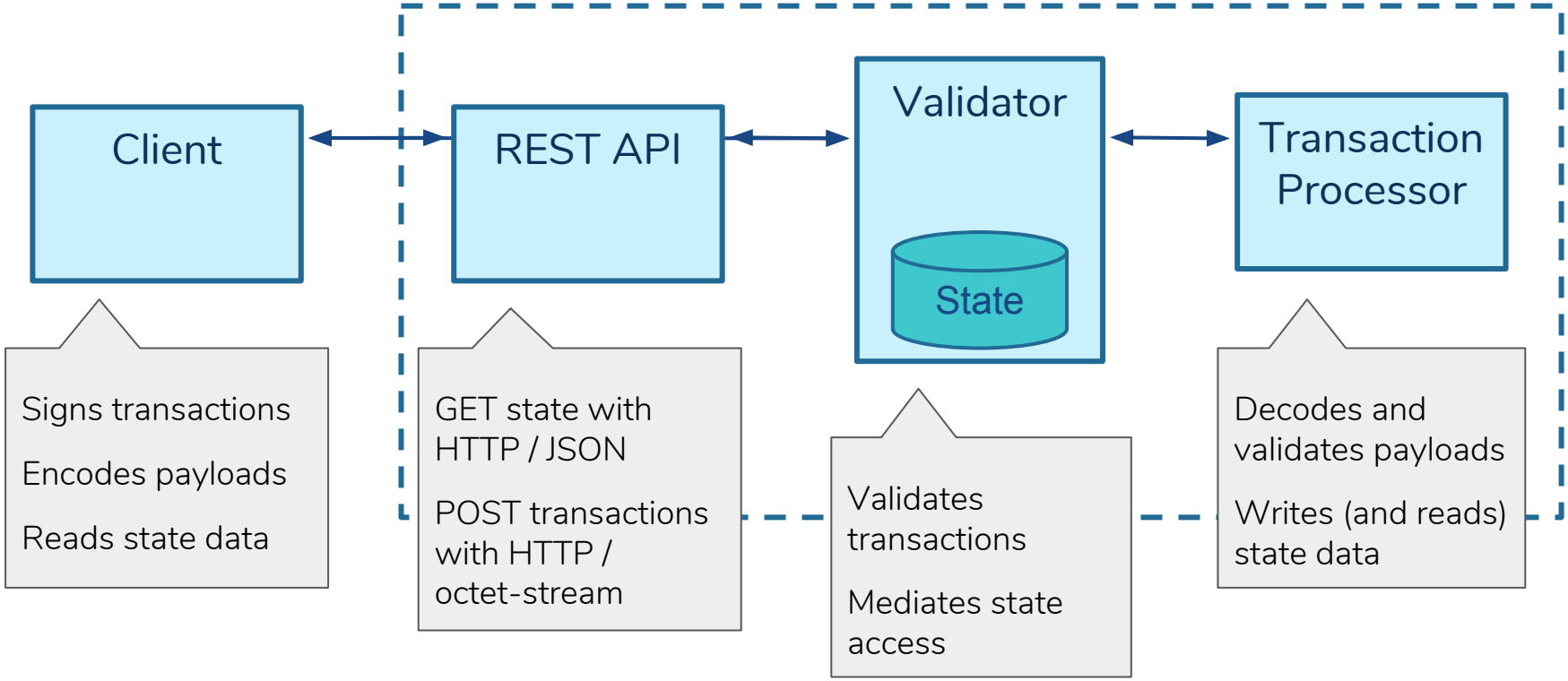
Byzantine Fault Tolerance (PoET)

Global state agreement

Develop in a variety of languages:

- Javascript
- Python
- Rust
- Solidity
- Go
- Java

Architecture



The Transaction Processor

Encoding State (and payloads)

Any encoding scheme can be used

Should produce raw binary (i.e. Buffer or Uint8Array)

Must be deterministic

We'll use sorted (sorted) JSON for simplicity

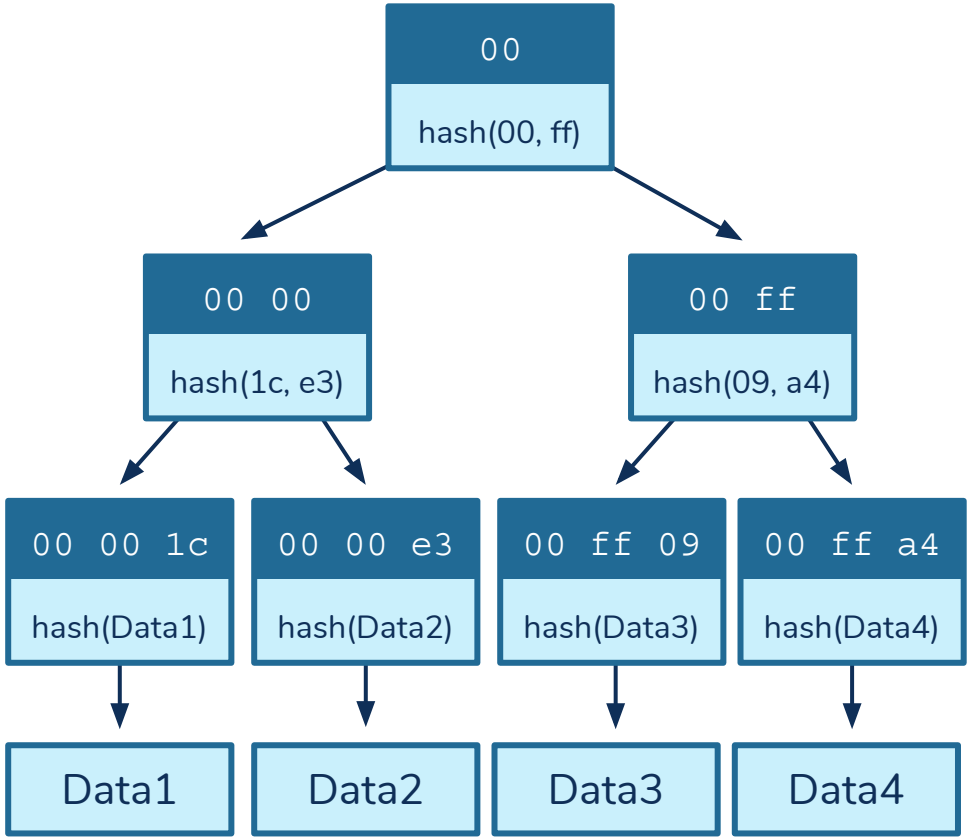
Other schemes are more predictable, more storage/CPU efficient (like Protobufs)

Encoding State (example)

```
const encode = obj => {
  const sortedKeys = Object.keys(obj).sort();
  const json = JSON.stringify(obj, sortedKeys);
  return Buffer.from(json);
};
```

(Warning! This won't work for nested objects!)

Addressing (the merkle tree)



State is stored in a "merkle tree" or "hash tree" so it can be verified

Each node is addressed with a single byte (i.e. two hex characters)

Any arbitrary data can be stored at any address you like

Prefixes can be used to quickly fetch all child nodes

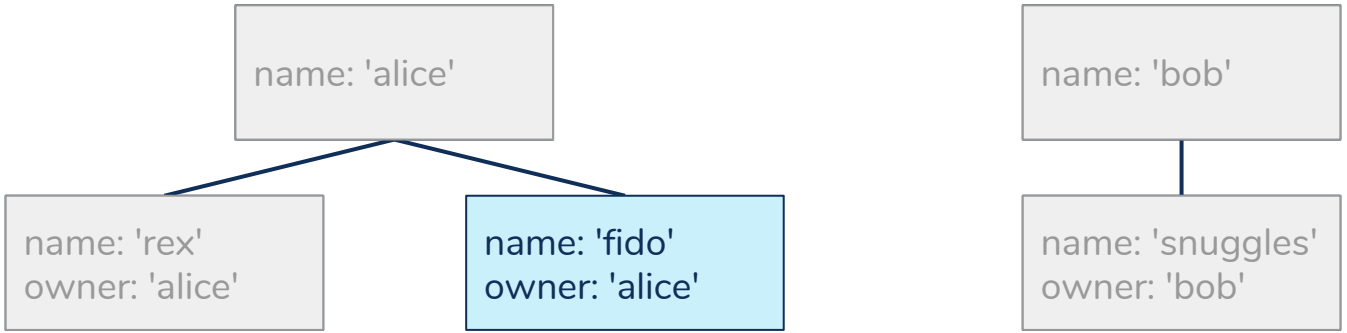
Sawtooth addresses are 70 hex characters long (35 bytes)

Addressing (example)

	Namespace	Type	Owner (optional)	Identifier
owner	sha512('pets').slice(0, 6)	'00'	--	sha512(name).slice(0, 62)
pet	sha512('pets').slice(0, 6)	'01'	sha512(owner).slice(0, 16)	sha512(name).slice(0, 46)

```
102fe1 01 408b27d3097eea5a 1f443e224ee0de0afdc46cd68820de1494687677bed1d3
```

'pets' 'alice' 'fido'



Transaction Processor

Communicates with validator over ZMQ

The SDK handles most of the networking details

Transaction processors manage one or more transaction handlers

Transaction Processor (example)

```
const { TransactionProcessor } = require('sawtooth-sdk/processor');
const PetsHandler = require('./handler');

const VALIDATOR_URL = 'tcp://localhost:4004';

const tp = new TransactionProcessor(VALIDATOR_URL);
tp.addHandler(new PetsHandler());
tp.start();
```

index.js

Transaction Handler

Handles transactions specific to one version of one "family"

"Transaction family" is a general term, referring to the shared business logic of an app

Implements an `apply()` method, which is called for each transaction

If the payload is invalid, throw an `InvalidTransaction` error to be caught and logged by the SDK

Transaction Handler (example)

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

module.exports = class PetsHandler extends TransactionHandler {
  constructor() {
    super('pets', [ '0.1' ], [ '102fe1' ]);
  }
  apply(txn, context) {
    . . .
  }
}
```

handler.js

Transaction Handler (example)

```
apply(txn, context) {
  console.log(txn.header)  // { signerPublicKey: '034f35...', ... }
  console.log(txn.headerSignature)  // '5f7a93ac...'
  console.log(txn.payload)  // <Buffer e7 bd 6c ... >

  console.log(context.getState)  // [Function: getState]
  console.log(context.setState)  // [Function: setState]
}
```

apply()

Transaction Handler (example)

```
const { InvalidTransaction } = require('sawtooth-sdk/processor/exceptions');

  apply(txn, context) {
    let payload = null;
    try {
      payload = decode(txn.payload);
    } catch (err) {
      throw new InvalidTransaction('Unable to decode payload');
    }
  }
```

apply()

Transaction Handler (example)

```
const { InvalidTransaction } = require('sawtooth-sdk/processor/exceptions');

  apply(txn, context) {
    . . .
    if (payload.action === 'CREATE_OWNER') {
      return createOwner(context, payload, txn.header.signerPublicKey);
    }

    throw new InvalidTransaction('Unknown action');
  }
```

apply()

Accessing State

The `getState()` method takes an array of state addresses (or prefixes) and returns an object with the fetched state.

The `setState()` method takes an object with addresses as keys and encoded updates as values.

Both methods returns Promises.

Transaction Handler (example)

```
const createOwner = (context, { name }, publicKey) => {
  const address = getOwnerAddress(publicKey);
  return context.getState([ address ]).then(state => {
    if (state[address].length > 0) {
      throw new InvalidTransaction('Owner already exists');
    }
    const update = {};
    update[address] = encode({ key: publicKey, name });
    return context.setState(update);
  }
}
```

createOwner()

The Client

What is a client?

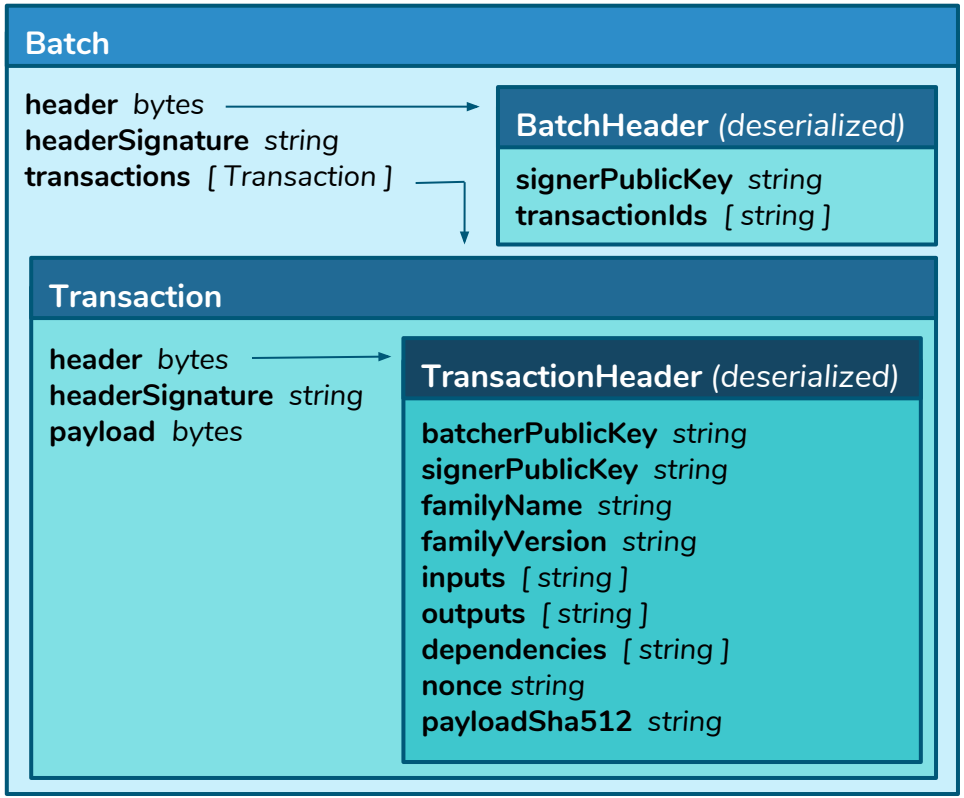
Can be simple or complex:

- a CLI
- a web app
- a whole ecosystem with a custom REST API and a local copy of state in a database

Responsible for:

- encoding payloads
- creating and signing **transactions** and **batches**
- submitting transactions
- fetching and displaying state data for users

Transactions and Batches



Provided by the SDK as protobufs (using protobuf.js)

Signed and encoded by the client

Every transaction in a batch must be valid, or the batch is invalid

Inputs/outputs can be prefixes, but ideally are full addresses

The nonce can be any random string

Creating a signed Transaction

```
import { Transaction, TransactionHeader } from 'sawtooth-sdk/protobuf';

const createTransaction = (privateKey, payload) => {
  const encodedPayload = encode(payload);
  const header = TransactionHeader.encode({ . . . }).finish();
  return Transaction.create({
    header,
    headerSignature: sign(privateKey, header),
    payload: encodedPayload
  });
};
```

createTransaction()

Using the REST API

A pre-built Sawtooth component for generic blockchain requests

Proxied with Docker to the `/api` route in your web app

Three routes you will need:

- POST /batches
- GET /state?address={prefix}
- GET /state/{address}

Responses are wrapped in a JSON envelope. State entities are at the "data" key.

REST API Routes

POST /batches

- Submit transactions to the validator
- Encode in a BatchList
- Use the header `Content-Type: application/octet-stream`

GET /state?address={prefix}

- Fetches multiple encoded entities from state
- Use an address prefix to filter the results

GET /state/{address}

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

REST API (example)

```
import axios from 'axios';

const fetchOwners = () => {
  return axios.get('/api/state?address=102fe100').then(response => {
    return response.data.data.map(entity => {
      const decoded = decode(entity.data);
      decoded.address = entity.address;
      return decoded;
    });
  });
};
```

fetchOwners()

That's it! Thank you!

Hyperledger Cryptomoji
https://github.com/hyperledger/education-cryptomoji

App Developer's Guide
https://sawtooth.hyperledger.org/docs/core/releases/1.0/app_developers_guide.html

Sawtooth Chat
https://chat.hyperledger.org/channel/sawtooth