Hyperledger Sawtooth for Application Developers

Module 4: Overview of Application Development

MODULE 4: Overview of Application Development

This module describes application development process for Hyperledger Sawtooth. It explains how to create a Sawtooth application with a transaction processor, client front-end app, a custom REST API, and an event subscriber with an off-chain reporting database for storing state data locally.

Contents

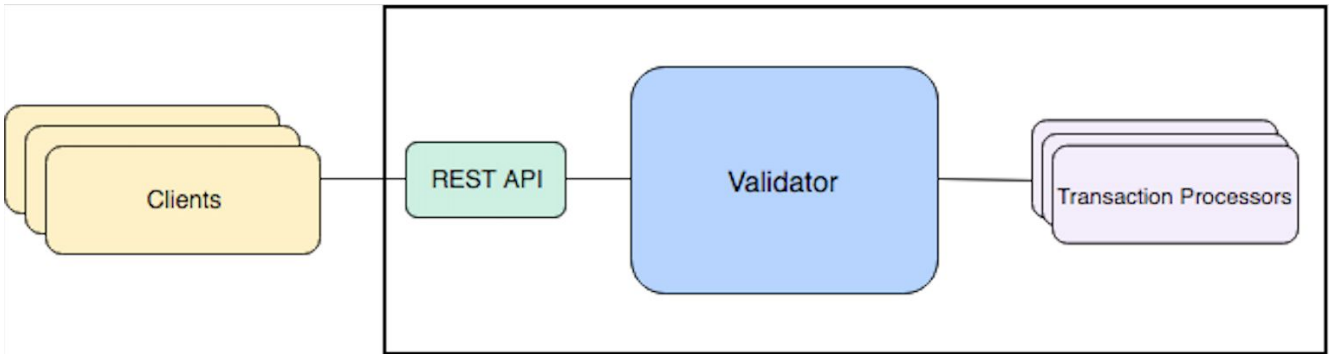
Review of Sawtooth Concepts

This section reviews the important Sawtooth concepts that are needed for application development. You will also learn about event subscription with a reporting database.

Contents

Sawtooth Applications

As described in the previous module, Sawtooth separates the application level from the core system (the Sawtooth validator).



The components of a Sawtooth application include:

- A **client** that handles the front-end logic for your application. The client structures (serializes) the transaction data, which is also called the *payload*. Then the client wraps the transaction payload in a Sawtooth batch and submits it to the validator.
- A **transaction processor** that defines the server-side business logic for your application. The transaction processor decodes payload data and uses an apply function to make state changes.
- An optional REST API or RESTful service to communicate between the client and validator. An application can use the standard Sawtooth REST API or provide a custom REST API.

  ★ A REST API is optional because the client could use ZeroMQ instead of HTTP/JSON to communicate directly with the validator. This advanced topic is not covered in this course.

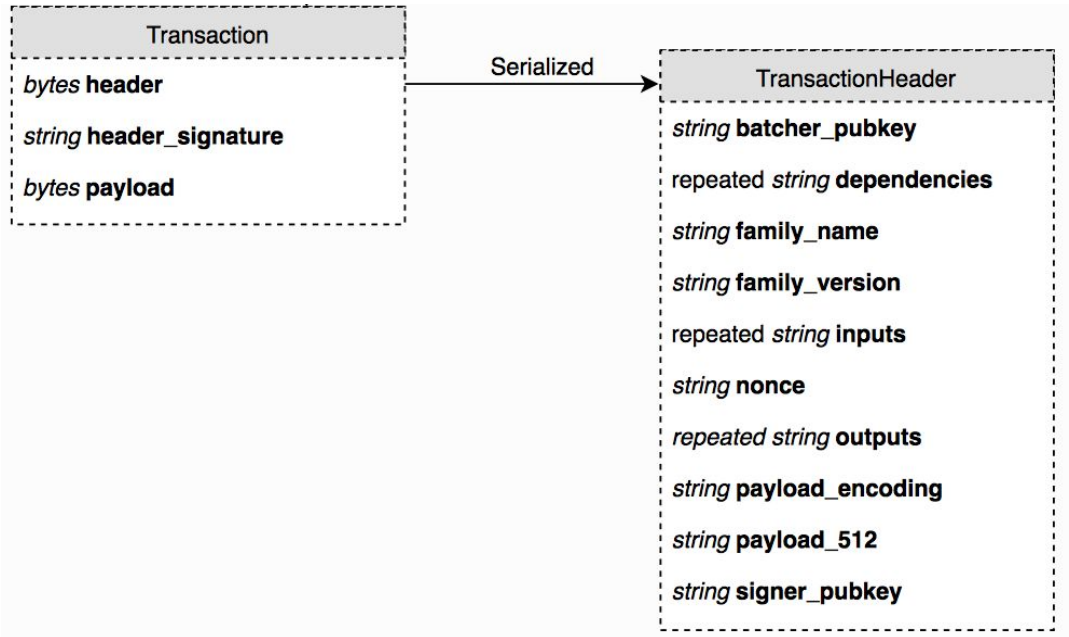
Transactions, Batches, and Blocks

Transactions represent changes to blockchain state and are wrapped in *batches*, which are then wrapped in *blocks*.

- **Transaction**: A single change in the state of the blockchain. Each Sawtooth transaction is a protocol buffer (protobuf) object that includes a payload, a transaction header, and a header signature derived from signing the transaction header. The transaction header contains the application's family name and version, plus several other details.
- **Batch**: The wrapper for a set of transactions. All transactions within a Sawtooth batch are either committed to state together or are not committed at all, which makes batches the atomic unit of state change in Sawtooth. Each batch is a protobuf object that contains a list of transactions and a header that includes a signature from the batch creator (often the same as the transaction creator).
- **Block**: A group of Sawtooth batches. A block has a header that includes a timestamp, a signer, and a hash (unique block ID). After a block is committed, the header also identifies the previous block in the blockchain.

Transaction Payload and Header

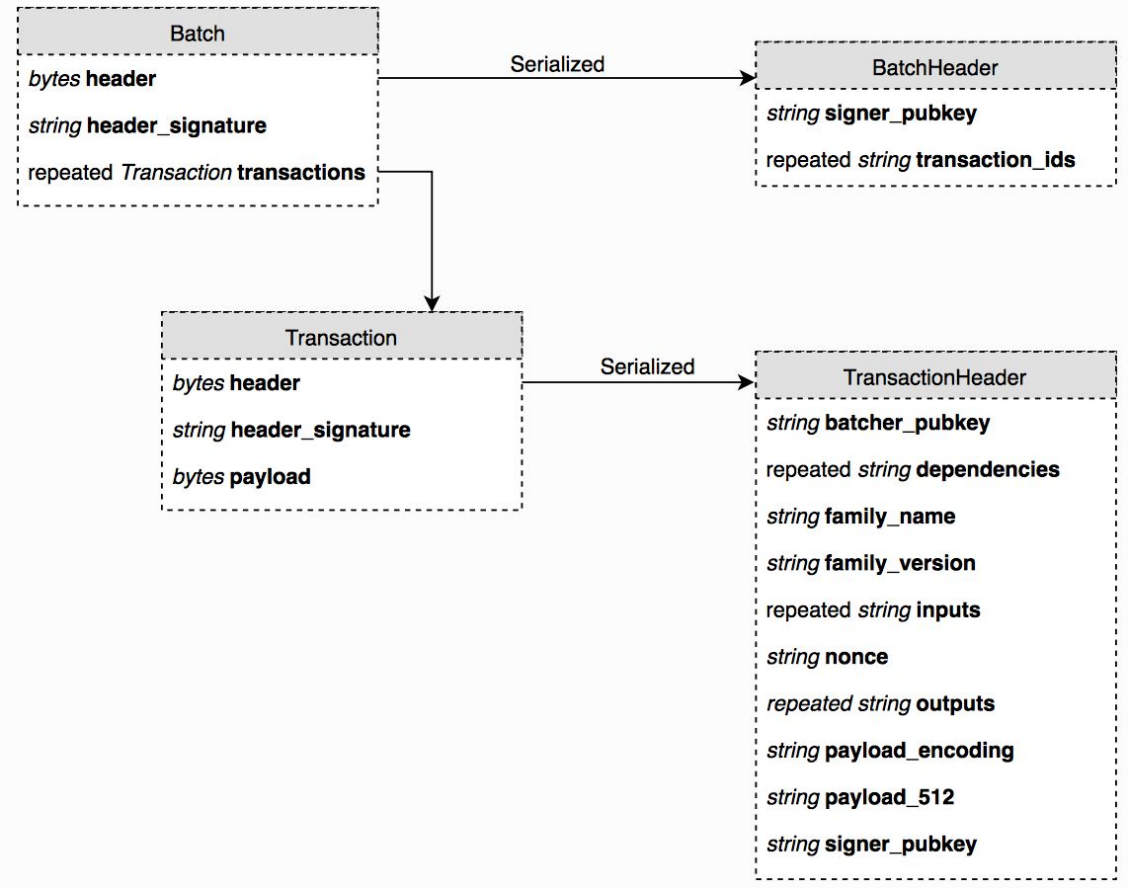
The transaction payload is used during transaction execution as a way to convey the change that should be applied to state. Only the application processing the transaction will deserialize the payload. To all other components of the system, the payload is just a sequence of bytes.



In the transaction header, the `payload_sha512` field contains a SHA-512 hash of the payload bytes. As part of the header, `payload_sha512` is signed and later verified, while the payload field is not. To verify the payload field matches the header, a SHA-512 of the payload field can be compared to `payload_sha512`.

Batch Structure

Transactions are always wrapped inside of a batch.



State and Addressing

Sawtooth manages the changes to the blockchain *state* using *addresses*.

State: The Sawtooth blockchain, as represented as a log of all changes that have occurred since the genesis block. Each validator node has its own copy of state in a local database. As blocks are published, each validator updates its state database to reflect the current blockchain status.

Global state: Represents the consensus of all participants on the state of the blockchain. Also called *global state agreement*.

Addressing: Sawtooth represents blockchain state as a tree on each validator. State is split into application-specific *namespaces* (subset of addresses), so that each application knows which portion of state it is allowed to change. Also, application developers can define, share, and reuse global state data between applications.

Sawtooth Application Design

This section explains the key concepts of Sawtooth application design.

Contents

- Application Workflow
- Application Design Considerations
  - Transaction Signing
    - Client Signing Model
    - Server Signing Model
  - State Address Format
  - Event Subscription
  - Programming Language Choice

Application Workflow

In Hyperledger Sawtooth, the application workflow looks something like this:

1. The user performs an action, such as initializing a value (for example, “set A to 1”)
2. The client takes that action, also called the transaction, and does the following:
    a. Encodes (serializes) it in a payload, which could be as simple as `{"A": 1}`
    b. Wraps that payload in a signed transaction, then in a signed batch containing one or more transactions
    c. Submits the batch to the validator
3. The validator confirms that the batch and transaction are valid
4. The transaction processor receives the transaction and:
    a. Verifies the signer
    b. Unwraps the transaction and decodes (deserializes) the payload
    c. Verifies that the action is valid (for example, ensures that A ***can*** be set to 1)
    d. Modifies state in a way that satisfies the action (for example, address `...000000a` becomes `1`)
5. Later, the client might read that state address and decode it for display to the user

Application Design Considerations

The core Sawtooth system handles the details of running and validating the blockchain. An application doesn't need to verify signatures and hashes, validate blocks, or confirm consensus.

As the application developer, you are responsible only for building the client and the transaction processor, plus optional components such as a custom REST API (if you choose not to use the Sawtooth REST API) and an event subscriber.

A Sawtooth application must determine the following items:

- What does the transaction payload look like?
- How is the payload serialized and deserialized?
- What is the format of data that is stored in state?
- Which addresses in state are used to store the application data?

You must decide how to represent your business logic as discrete transaction payloads, and determine how those payloads will change state data. Most importantly, you must ensure that the client and transaction processor agree on how to encode payloads and state. The next module provides specific details for the application design details using the example Sawtooth Simple Supply application.

★ We recommend using an application specification document to record your design decisions before building the application. For examples, see the Transaction family (application) specifications in the Sawtooth documentation and the Simple Supply specification document.

Transaction Signing

Although fetching data from a reporting database works well with typical RESTful HTTP/JSON interfaces, submitting updates to the blockchain using a REST API requires special consideration for the signing model. These two models have been used in the past:

- Client signing model
- Server signing model

Client Signing Model

The flow for a client signing model is as follows:

1. The client builds transactions and batches and signs them using the user's private key
2. The client serializes the batches and submits them to the single POST /batches route on the REST API
3. The REST API sends the batches directly to the validator

Pros:

- User identity is fully verifiable since the transactions are signed locally using the user's private key
- Even if the server is fully compromised, there is no way to falsify transactions from a certain user

Cons:

- RESTful endpoints for submitting transactions will not be possible since the server only acts as a middleman, maintaining a connection with and submitting serialized data to the validator
- Each client must implement transaction creation and signing functionality

This model fully takes advantage of blockchain identity verification, leaving the onus on the user to protect their own private key. Examples of this model include Sawtooth Supply Chain, as well as the Sawtooth REST API.

Server Signing Model

The flow for a server signing model is as follows:

1. The client submits update requests as JSON objects to traditional RESTful interfaces
2. The server creates and signs transactions based on the submitted JSON
3. The server handles batching/serialization and sends the transaction to the validator

Pros:

- The server maintains RESTful interfaces, which means that interacting with the server is no different than if the server was backed by a traditional database.
- The client does not have to handle any signing or transaction creation.

Cons:

- Because the server is in charge of signing all transactions, the identity verification guarantee of blockchain is somewhat compromised.

A naive approach of the server signing model would sign all transactions from a single key owned by the server. This would negate all of the identity verification advantages of blockchain as the server would be the single source of truth, the same as with a traditional database.

This disadvantage can be mitigated by generating a public/private key pair for each user on the server and storing the (encrypted) private key in a key escrow service. The server can then retrieve and decrypt the private key of an authenticated user and sign transactions on their behalf. While this is better than the former approach, it is still only as secure as the server's security mechanisms. Simple Supply uses this approach, which is described in detail in the following sections. For another example, see Sawtooth Marketplace.

State Address Format

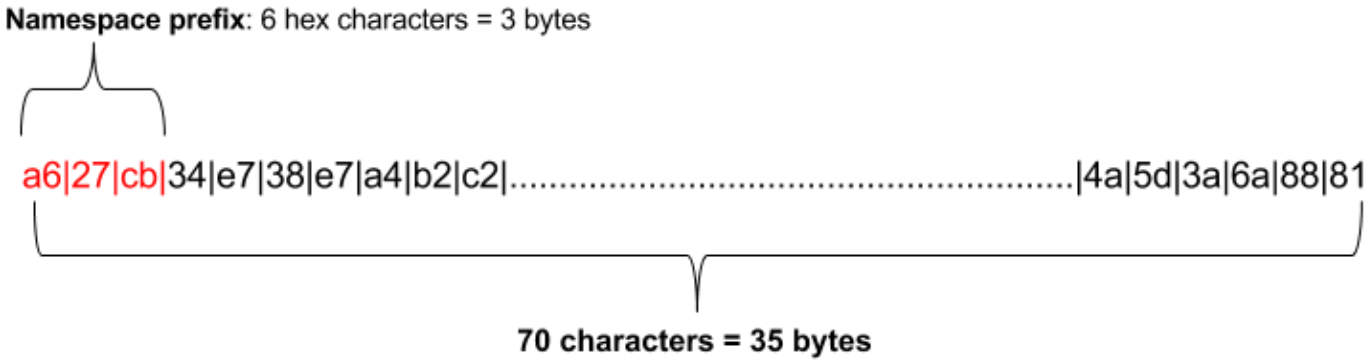
The address scheme for global state is an important part of application design. Sawtooth stores state data for all applications in a single instance of a database (a Merkle-Radix tree) on each validator. Data is stored in leaf nodes, and each node is accessed using an addressing scheme that is composed of 35 bytes, represented as 70 hex characters.

The state is split into namespaces, which allows application developers to define, share, and reuse global state data between transaction processors. The namespace identifies a set of addresses that “belong” to an application.

For more information, see Global State in the Sawtooth documentation.

State Address Format (continued)

A Sawtooth address begins with a namespace prefix of 6 hex characters representing 3 bytes. The rest of the address, 32 bytes represented as 64 hex characters, is defined by the application.



The recommended way to construct an address is to use the hex-encoded hash values of the string or strings that make up the address elements. Sawtooth applications often determine the namespace prefix by calculating an SHA256 hash of the application name, then using the first 6 characters, as in this Python example:

```
prefix = hashlib.sha256("my_app_name".encode('utf-8')).hexdigest()[:6]
```

However, an application can choose to use any arbitrary scheme for the namespace. For example, Sawtooth Settings uses the namespace prefix `000000`. For the existing namespace values, see the transaction family (application) specifications in the Sawtooth documentation.

Event Subscription

A Sawtooth application can implement event subscription with an *event subscriber* and a local *reporting database* to store state event data. Note that this off-chain reporting database is separate from the state database on each validator node.

A change to the blockchain triggers a Sawtooth event. Events are triggered "on block boundaries"; that is, when a block is committed. Each application can define its own events. By convention, event names include the application name as a prefix. Sawtooth has two core events (identified with the `sawtooth` prefix):

- `sawtooth/commit-block`: Contains information about the committed block: The block ID, number, state root hash, and previous block ID
- `sawtooth/state-delta`: Contains all state changes that occurred for a block at a specific address

In this model, an event subscription client subscribes to both Sawtooth state-delta and block-commit events, then uses the data reported in these events to maintain a local copy of blockchain state, which allows rapid querying of state data.

The first request for event data specifies the genesis block. An application can also request *event catch-up* by issuing a catch-up subscription request with a specific block ID. The Sawtooth validator sends data for all events that have occurred after that block was committed.

An event subscription includes the event type, an address (as a specific address, a range, or a pattern), and optional filters for event attributes. Most event subscriptions use filters to focus on the specific events of interest. For more information, see Subscribing to Events in the Sawtooth documentation.

Programming Language Choice

The Sawtooth platform allows you choose the language or languages for implementing your application components. For example, the Simple Supply application (described in a later module) uses JavaScript for the client web app and Python for the transaction processor and REST API.

Sawtooth provides SDKs in multiple languages, including Python, Javascript, Go, C++, Java, and Rust. For more information, see the list of available SDKs in the Sawtooth documentation.

Module 4: Summary

This module described the principles and guidelines for Sawtooth application development.

IIn the next module, you will examine an example application, Sawtooth Simple Supply, while learning the design and coding details that are important for a successful Sawtooth application.

Pierre Cordier, Chimigramme II
J. Paul Getty Museum
Digital image courtesy of the Getty's Open Content Program