Hyperledger Sawtooth for Application Developers

Module 5: Creating an Application: Sawtooth Simple Supply

[MODULE 5] Creating an Application: Sawtooth Simple Supply

This module presents an example application, Sawtooth Simple Supply, that demonstrates the fundamentals of application development. The module starts with the design decisions for Simple Supply, then walks you through the process of developing a simple, yet full-stack application.

Contents

Cloning the Simple Supply Repository

To get started, clone the Github repository hyperledger/education-sawtooth-simple-supply. This repository contains the code for the Simple Supply example application.

1. Open a terminal window and navigate to your projects directory.
2. Clone the repository.

```
$ git clone https://github.com/hyperledger/education-sawtooth-simple-supply.git
```

3. Navigate to the project root directory.

```
$ cd education-sawtooth-simple-supply
```

As you go through this module, you can follow along with the Simple Supply code in this repository.

About Simple Supply

Sawtooth Simple Supply is a basic example of asset transfer with location tracking. It allows users to track the provenance and location of goods as they move through a supply chain.

- ★ Sawtooth Simple Supply is a simplified version of Sawtooth Supply Chain, which is a distributed application that traces the provenance and other contextual information of any asset.

Simple Supply includes a client web app, Curator, that tracks artwork loans between museums, galleries, and private owners.

Vincent Van Gogh, Irises
Digital image courtesy of the Getty's Open Content Program

Each participant can add a Sawtooth node to the blockchain network, then registering owners and their artwork (including the location). To make a loan, an owner transfers the art and changes its location.

Storing this information on the blockchain means that another client app (not in this simple example) could provide more features such as telling owners whether their loaned art is in transit or has arrived; informing artists where their work is being displayed; or helping art lovers find an exhibit of their favorite paintings.

About Simple Supply (continued)

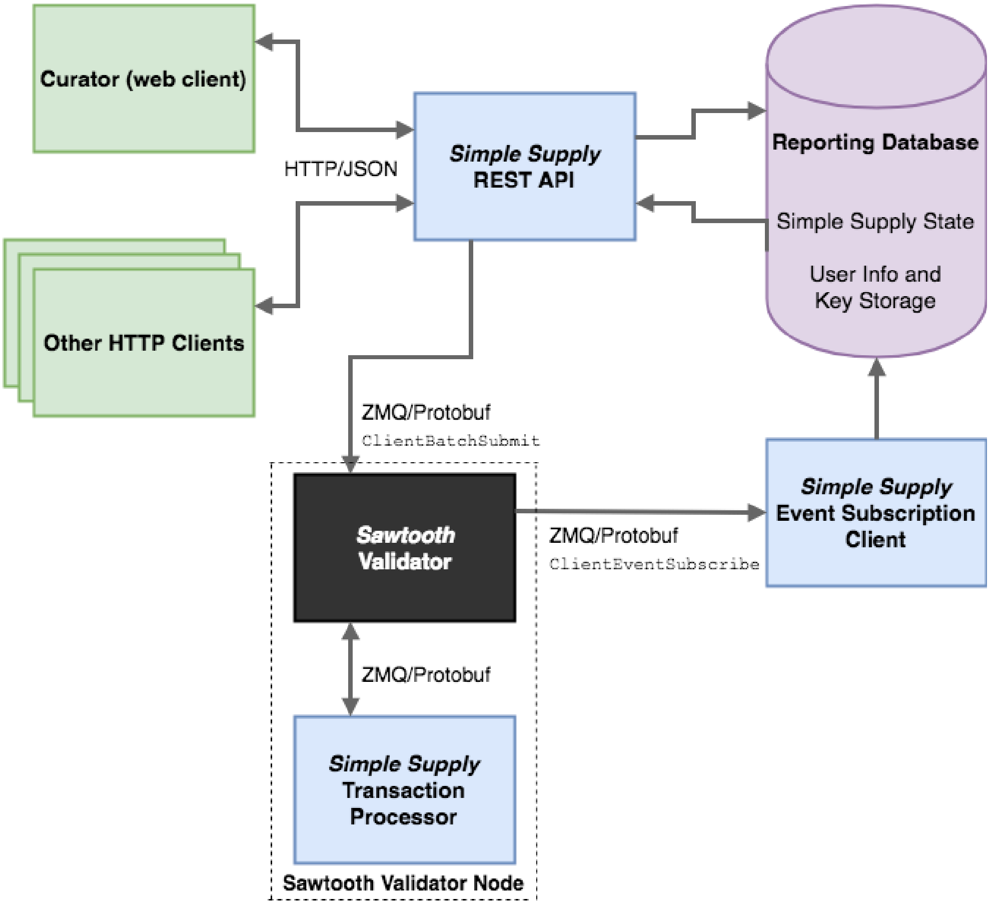
The Simple Supply application includes these components:

- The Simple Supply transaction processor, which interfaces with a Sawtooth validator in order to validate transactions. The transaction processor is written using the Sawtooth Python 3 SDK.
- The Curator client, a lightweight web app that submits requests to register new owners and artwork, transfer artwork to a different owner, and change the artwork's location. The client then submits these requests as HTTP to the REST API. Curator is written in JavaScript.
- A custom REST API that handles some client functions (such as creating batches and transactions from the requests submitted by Curator) as well as communicating with the validator. The REST API manages user information and public/private key pairs, which are stored in an off-chain reporting database. This REST API is written in Python.
- An event subscriber, which parses Sawtooth events and stores state data in the off-chain reporting database. The event subscriber is written in Python.

Simple Supply Architecture

Module 5 > Creating an Application > About SImple Supply > Simple Supply Architecture

This architecture diagram shows the communication between the Simple Supply components: Curator web client, custom REST API, event subscriber, reporting database, and Sawtooth validator.



The Event Subscriber

The Simple Supply application includes an event subscriber that registers to receive the following Sawtooth events.

- `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

★ For more information about events, such as defining application-specific events, see Events and Transaction Receipts in the Sawtooth documentation.

The event subscriber stores the information in a local (off-chain) ***reporting database*** for later use. This allows the Simple Supply REST API to query the reporting database for state data instead of using slower queries to the validator. Simple Supply also uses this PostgreSQL database to store user information such as hashed passwords and encrypted signing keys.

Designing the Simple Supply Application

Before looking at the Simple Supply code for each component, it is important to consider the application design:

- **Business logic**: What are Simple Supply's transaction execution rules?
- **Addressing scheme**: Which addresses in state are used to store the application data?
- **State encoding**: What is the format of data that is stored in state?
- **Payload encoding**: What does the transaction payload look like?

In Simple Supply, an asset is called a *record*. A record contains a unique identifier and lists containing the history of its owners and location. An *agent* can own, transfer, and update records. Each agent is associated with a public and private key that is used to sign transactions. Note that the agent object in state does not actually store the private key; instead, the private key is encrypted and stored in the off-chain database.

The transaction actions in Simple Supply are:

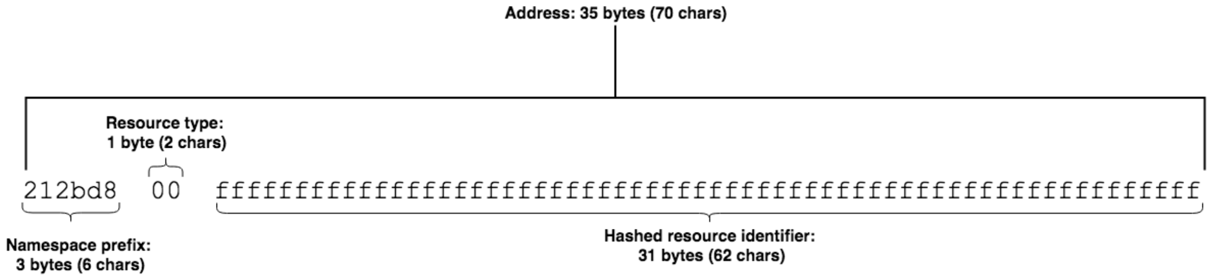
- Create agent
- Create record
- Update record (change its geographical location)
- Transfer record

Later topics show how to implement the 'create agent' transaction, as well as any overhead for each of the application components (such as metadata for the transaction processor). The other transactions are similar so aren't shown in detail in this course.

Addressing Scheme

Module 5 > Creating a Sawtooth Application > Designing the Simple Supply Application > Addressing Scheme

A Sawtooth address is 35 bytes (70 hex characters). Simple Supply uses a very simple addressing scheme with three parts.


Address: 35 bytes (70 chars)
Resource type:
1 byte (2 chars)
212bd8
00
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
Namespace prefix:
3 bytes (6 chars)
Hashed resource identifier:
31 bytes (62 chars)

- Bytes 1-3: The first 6 characters of the hashed application name. (The set of addresses that start with these bytes is the application's *namespace*.) Simple Supply uses the SHA-512 hash of the string `simple_supply`, as follows:

  ```
  >>> hashlib.sha512('simple_supply'.encode('utf-8')).hexdigest()[:6]
  '212bd8'
  ```

- Byte 4: Resource type.
  - Agent: `00`
  - Record: `01`
- The final 31 bytes are determined by the resource type:
  - Agent: The first 62 characters of the hash of its public key
  - Record: The first 62 characters of the hash of its identifier

This addressing scheme allows Simple Supply to quickly determine which type of resource is stored at any address by looking at the fourth byte of the address.

State Encoding

Applications serialize payloads to make the data “opaque” to the core Sawtooth system. The validator sees the data as simply a string of bytes. Only the transaction processor that handles the transaction will deserialize the payload.

When interacting with the blockchain, a transaction processor sets and retrieves state data by making calls against a version of state that the validator provides.

- `get_state(address)` returns the byte array found at the address
- `set_state(address, data)` sets the byte array stored at that address

Likewise, when a client sends a transaction to the validator, it must serialize the payload data.

For this reason, the encoding scheme must be *deterministic*; serialization and deserialization must always produce the exact same results. Even the slightest difference in the state entities across platforms or executions (such as the keys being in a different order or rounding inconsistencies for floating-point numbers) can break the global state agreement. Avoid data structures that don’t enforce an ordered serialization (such as sets, maps, dicts, or JSON objects). Also take care to avoid data types that may be interpreted differently across platforms

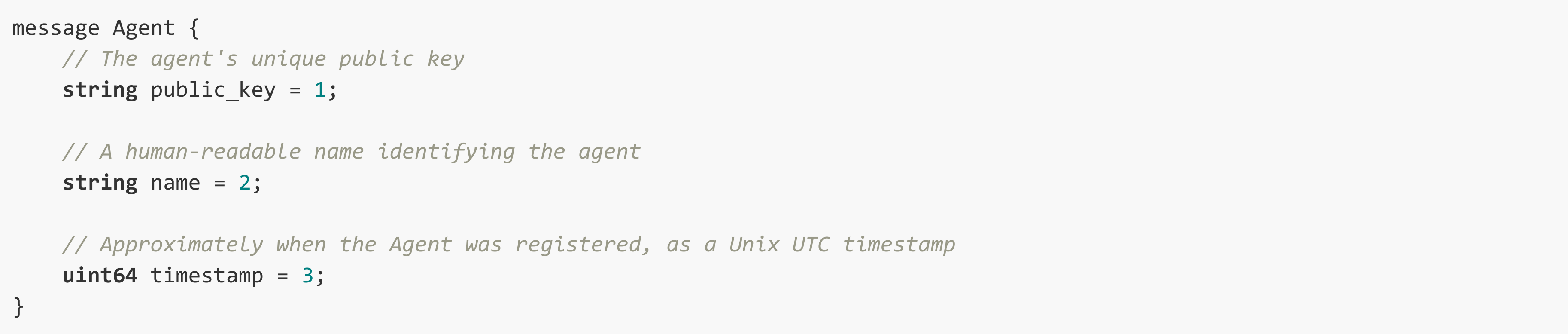
Simple Supply uses protocol buffers (protobufs) to encode all objects before storing them in state (both payloads and state data. Using protobufs here is convenient because Simple Supply uses protobufs to serialize batches and transactions. Although protobufs don’t fully guarantee determinism, they serialize and deserialize identically for the purposes of Simple Supply. The Simple Supply protobuf messages are in education-sawtooth-simple-supply/protos.

State Data: Agents

Module 5 > Creating a Sawtooth Application > Designing the Simple Supply Application > State Data: Agents

An agent is identified by the public key of the user who created the agent. Simple Supply stores the public key on the blockchain. Simple Supply also tracks a user-specified name for the agent, and the time when the agent was registered (when the CreateAgentAction transaction was submitted to the validator).

The protobuf message for an agent stored in state is defined in protos/agent.proto:



```
message Agent {
    // The agent's unique public key
    string public_key = 1;

    // A human-readable name identifying the agent
    string name = 2;

    // Approximately when the Agent was registered, as a Unix UTC timestamp
    uint64 timestamp = 3;
}
```

State Data: Records

A record is a collection of data about a specific asset that has been registered on the blockchain. A record has a record ID, which is a unique identifier determined by the application or an external resource (such as a serial number or other meaningful data). A record also has time-stamped lists of owners and location updates.

See the `Record` protobuf in protos/record.proto:

```
message Record {
    message Owner {
        // Public key of the agent who owns the record
        string agent_id = 1;

        // Approximately when the owner was updated, as a Unix UTC timestamp
        uint64 timestamp = 2;
    }

    message Location {
        // Coordinates are expected to be in millionths of a degree
        sint64 latitude = 1;
        sint64 longitude = 2;

        // Approximately when the location was updated, as a Unix UTC timestamp
        uint64 timestamp = 3;
    }

    // The user-defined natural key which identifies the object in the
    // real world (for example a serial number)
    string record_id = 1;

    // Ordered oldest to newest by timestamp
    repeated Owner owners = 2;
    repeated Location locations = 3;
}
```

State Data: Containers

As described in “Addressing Scheme”, above, Simple Supply stores agent and record state entities at addresses based on a hash of the agent’s public key or record’s ID. Because there is no guarantee that the hashes are unique, Simple Supply must handle the case where two unique entities have the same state address.

To solve this problem, Simple Supply actually stores *containers* at each state address, rather than the entity itself. This approach allows the application to store multiple state entities at a single address.

The data stored at each agent address in state is defined by this message in protos/agent.proto:

```
message AgentContainer {
    repeated Agent entries = 1;
}
```

The state data for records is defined by a similar message in protos/record.proto.

Payload Encoding

The `SimpleSupplyPayload` object has six fields: an action tag, a timestamp, and four fields that contain other objects with the data for specific actions. This object is defined in protos/payload.proto:

```
message SimpleSupplyPayload{
    enum Action {
        CREATE_AGENT = 0;
        CREATE_RECORD = 1;
        UPDATE_RECORD = 2;
        TRANSFER_RECORD = 3;
    }

    // Whether the payload contains a create agent, create record,
    // update record, or transfer record action
    Action action = 1;

    // The transaction handler will read from just one of these fields
    // according to the action
    CreateAgentAction create_agent = 2;
    CreateRecordAction create_record = 3;
    UpdateRecordAction update_record = 4;
    TransferRecordAction transfer_record = 5;

    // Approximately when transaction was submitted, as a Unix UTC timestamp
    uint64 timestamp = 6;
}
```

Payload Encoding (continued)

When the client creates a transaction, it puts the UTC time in the `timestamp` field, chooses the proper enum value for the `action` field, and fills in the field that corresponds to the action. The other three fields are blank.

For example, if the action is `CREATE_AGENT`, the client fills in the `create_agent` field. The `CreateAgentAction` message is defined in protos/payload.proto:

```
message CreateAgentAction {
    // A human-readable name identifying the new agent
    string name = 1;
}
```

Note that the client does not provide the public key for the new agent, even though the key is included in the `Agent` state entity, because the public key can be read from the `signer_public_key` field of the transaction header.

This approach means that each user must register as an agent using their own public key. For an application that uses a different registration method (for example, if a manager can create new accounts for subordinates), it might be necessary to add a public_key field to the `CreateAgentAction` message.

★ For the other actions, see the Simple Supply specification (PDF).

Creating the Transaction Processor

When the validator receives a transaction from a client, the validator confirms that the signature is valid, then send the transaction to the transaction processor to be executed.

A transaction processor has two main components:

- The `TransactionProcessor` class is a generic class for communicating with a validator and routing transaction processing requests to a registered handler. This class uses ZMQ and channels to handle requests concurrently. Each SDK provides an implementation of this class.
- The `TransactionHandler` class is a base class that defines the business logic for the application. The application extends the `TransactionHandler` class with application-specific logic.

The handler gets called in two ways:

- With an `apply` method
- With various transaction processor metadata methods

The Simple Supply transaction processor is defined in processor/simple_supply_tp.

The next topic describes `apply` and its helper functions, which make up the majority of the handler. The metadata, which is used to connect the handler to the processor, is described later.

Implementing the `apply` Method

Transaction processing is done in the `apply` method and its helper methods. The `apply` method must:

1. Unpack the action data from the transaction
2. Determine which action is specified
3. Create new values or update existing values in state

The `apply` method has two arguments:

- `transaction` holds the action (for example, create a new agent). This argument's structure resembles the transaction that the validator received, although the transaction header was deserialized before it was sent to the transaction processor. `apply` uses the following parts:
    - `transaction.payload`: Serialized payload received by the validator
    - `transaction.header.signer_public_key`: Public key used to sign the transaction (which doubles as the ID of the agent sending the transaction)
- `context` holds the blockchain state and has methods for getting and setting values, including `get_state` and `set_state`.

If everything goes well, the validator's state will be updated with the new or changed values. If the transaction cannot be executed or is invalid, the handler will return an `InvalidTransaction` error.

Implementing the apply Method (continued)

The `apply` method for Simple Supply is defined in processor/simple_supply_tp/handler.py. The following topics describe how to implement the transaction logic for the “create agent” action.

```python
def apply(self, transaction, context):
    header = transaction.header
    payload = SimpleSupplyPayload(transaction.payload)
    state = SimpleSupplyState(context)
    if payload.action == payload_pb2.SimpleSupplyPayload.CREATE_AGENT:
        _create_agent(
            state=state,
            public_key=header.signer_public_key,
            payload=payload)
    elif payload.action == payload_pb2.SimpleSupplyPayload.CREATE_RECORD:
        # ...
    elif payload.action == payload_pb2.SimpleSupplyPayload.TRANSFER_RECORD:
        # ...
    elif payload.action == payload_pb2.SimpleSupplyPayload.UPDATE_RECORD:
        # ...
    else:
        raise InvalidTransaction('Unhandled action')

def _create_agent(state, public_key, payload):
    if state.get_agent(public_key):
        raise InvalidTransaction('Agent with the public key {} already exists'.format(public_key))
    state.set_agent(
        public_key=public_key,
        name=payload.data.name,
        timestamp=payload.timestamp)
# ...
```

Decoding the Payload

To keep the application as modular as possible, Simple Supply defines a `SimpleSupplyPayload` helper class. This class decodes the payload, makes sure that it is properly formatted, and simplifies access to the transaction data. See processor/simple_supply_tp/payload.py.

```python
class SimpleSupplyPayload(object):
    def __init__(self, payload):
        self._transaction = payload_pb2.SimpleSupplyPayload()
        self._transaction.ParseFromString(payload)

    @property
    def action(self):
        return self._transaction.action

    @property
    def data(self):
        if self._transaction.HasField('create_agent') and \
            self._transaction.action == \
                payload_pb2.SimpleSupplyPayload.CREATE_AGENT:
            return self._transaction.create_agent
        # ...
        raise InvalidTransaction('Action does not match payload data')

    @property
    def timestamp(self):
        return self._transaction.timestamp
```

Getting and Setting State

Module 5 > Creating a Sawtooth Application > Creating the Transaction Processor > Getting and Setting State

As with the payload, Simple Supply uses a helper class to serialize and deserialize state data. Because all state entities are stored in containers, this helper class also ensures that the `get_agent` method fetches the right data at the address.

The SimpleSupplyState class is defined in processor/simple_supply_tp/state.py.

```
class SimpleSupplyState(object):
    def __init__(self, context, timeout=2):
        self._context = context
        self._timeout = timeout

    def get_agent(self, public_key):
        """Gets the agent associated with the public_key
        Args:
            public_key (str): The public key of the agent
        Returns:
            agent_pb2.Agent: Agent with the provided public_key
        """
        address = addresser.get_agent_address(public_key)
        state_entries = self._context.get_state(
            addresses=[address], timeout=self._timeout)
        if state_entries:
            container = agent_pb2.AgentContainer()
            container.ParseFromString(state_entries[0].data)
            for agent in container.entries:
                if agent.public_key == public_key:
                    return agent
        return None
    # ...
```

Getting and Setting State (continued)

When updating state, `set_agent` must either create a new container or reuse an existing container at an address. See processor/simple_supply_tp/state.py.

```
class SimpleSupplyState(object):
    # ...
    def set_agent(self, public_key, name, timestamp):
    """Creates a new agent in state
    Args:
        public_key (str): The public key of the agent
        name (str): The human-readable name of the agent
        timestamp (int): Unix UTC timestamp of when the agent was created
    """
    address = addresser.get_agent_address(public_key)
    agent = agent_pb2.Agent(
        public_key=public_key, name=name, timestamp=timestamp)
    container = agent_pb2.AgentContainer()
    state_entries = self._context.get_state(
        addresses=[address], timeout=self._timeout)
    if state_entries:
        container.ParseFromString(state_entries[0].data)

    container.entries.extend([agent])
    data = container.SerializeToString()
    updated_state = {}
    updated_state[address] = data
    self._context.set_state(updated_state, timeout=self._timeout)
```

Generating Addresses

Because both the client and transaction processor must generate addresses for state entities, it is a good idea to put the address-generation functionality in its own library. In Simple Supply, the addressing library defines the metadata fields and methods for creating an address based on the identifier of a state entity.

See addressing/simple_supply_addressing/addresser.py

```
FAMILY_NAME = 'simple_supply'
FAMILY_VERSION = '0.1'
NAMESPACE = hashlib.sha512(FAMILY_NAME.encode('utf-8')).hexdigest()[:6]
AGENT_PREFIX = '00'
RECORD_PREFIX = '01'

# ...

def get_agent_address(public_key):
    return NAMESPACE + AGENT_PREFIX + hashlib.sha512(
        public_key.encode('utf-8')).hexdigest()[:62]

def get_record_address(record_id):
    return NAMESPACE + RECORD_PREFIX + hashlib.sha512(
        record_id.encode('utf-8')).hexdigest()[:62]

# ...
```

Setting Metadata

After implementing the transaction logic, Simple Supply needs to set up the `SimpleSupplyHandler` class and its metadata. The metadata consists of information about what kinds of transactions the transaction processor can handle. This information is read by the validator upon registration of the transaction processor.

See processor/simple_supply_tp/handler.py.

```
class SimpleSupplyHandler(TransactionHandler):

    @property
    def family_name(self):
        return addresser.FAMILY_NAME

    @property
    def family_versions(self):
        return [addresser.FAMILY_VERSION]

    @property
    def namespaces(self):
        return [addresser.NAMESPACE]

    def apply(self, transaction, context):
    # ...
```

Creating the main Method

Module 5 > Creating a Sawtooth Application > Creating the Transaction Processor > Creating the main Method

The final piece of the transaction processor is the `main` method. This function uses the `argparse` library to get the validator's URL and logging preferences (such as verbosity).

Simple Supply's main method is defined in processor/simple_supply_tp/main.py:

```
def main(args=None):
    if args is None:
        args = sys.argv[1:]
    opts = parse_args(args)
    processor = None
    try:
        init_console_logging(verbose_level=opts.verbose)

        processor = TransactionProcessor(url=opts.connect)
        handler = SimpleSupplyHandler()
        processor.add_handler(handler)
        processor.start()
    except KeyboardInterrupt:
        pass
    except Exception as err:
        print("Error: {}".format(err))
    finally:
        if processor is not None:
            processor.stop()
```

Creating the REST API

The Simple Supply REST API handles transaction creation and signing logic. This custom REST API has several routes for submitting transaction requests and fetching data for use by a user-facing client application. Simple Supply includes an example lightweight web app called Curator that uses these routes, but you could also use tools like Postman or `cURL` to make requests to the REST API. For more information, see the specification for the Simple Supply REST API.

Simple Supply's REST API uses a server-side signing model. For an application that chooses to use the standard Sawtooth REST API with the client-side signing, the client must handle functions such as submitting and signing transaction requests and fetching data.

The Simple Supply REST API is implemented using the aiohttp library, which has a few different classes:

- `RouteHandler`: Services incoming HTTP requests, handling auth, fetching requested data from the `Database`, and sending transaction data to the `Messenger`
- `Messenger`: Manages the connection to the validator and uses the `transaction_creation` module to build transactions
- `Database`: Handles connection to the reporting database

★ This section describes how to implement the "create agent" action. The other actions are implemented in a similar way. Refer to the Simple Supply code for details about the "create record", "update record", and "transfer record" actions.

Create Agent Workflow

The Simple Supply REST API has the following basic workflow for the “create agent” action:

1. Create a new agent using the POST /agents route (enter a name and password).
2. Generate a new public/private key pair for the agent.
3. Encrypt the private key and store it in the auth table in the reporting database, along with the agent’s public key and hashed password.
   ★ Key encryption and storage is not a blockchain operation. The Simple Supply client provides this functionality as a convenience for storing private keys.
4. Use the `Messenger` class to format the transaction and send it to the validator
5. If all operations are successful, the REST API returns an auth token

Route Handler

The `RouteHandler` functionality for Create Agent is defined in rest_api/simple_supply_rest_api/route_handler.py:

```python
async def create_agent(self, request):
    body = await decode_request(request)
    required_fields = ['name', 'password']
    validate_fields(required_fields, body)

    public_key, private_key = self._messenger.get_new_key_pair()

    await self._messenger.send_create_agent_transaction(
        private_key=private_key,
        name=body.get('name'),
        timestamp=get_time())

    encrypted_private_key = encrypt_private_key(
        request.app['aes_key'], public_key, private_key)
    hashed_password = hash_password(body.get('password'))

    await self._database.create_auth_entry(
        public_key, encrypted_private_key, hashed_password)

    token = generate_auth_token(
        request.app['secret_key'], public_key)

    return json_response({'authorization': token})
```

Generating Keys

To confirm user identity and sign the information sent to the validator, each user needs a 256-bit key. For signing, Sawtooth uses the ECDSA standard algorithm with secp256k1 curve, 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` object wraps a private key and provides some convenient methods for signing bytes and getting the private key's associated public key. For the full signing API, see the Python SDK sawtooth_signing package in the Sawtooth documentation.

Simple Supply's REST API contains a simple example showing how to generate a signer; see rest_api/simple_supply_rest_api/messaging.py:

```python
from sawtooth_signing import create_context
from sawtooth_signing import CryptoFactory

context = create_context('secp256k1')
private_key = context.new_random_private_key()
signer = CryptoFactory(context).new_signer(private_key)
```

! IMPORTANT: With this server-signing model, the private key is the only way a user can prove their identity on the blockchain. A private key **MUST** be kept secret and secure, because anyone who has a user's private key can sign transactions on their behalf. Also, there is no way to recover a private key if it is lost. If an application uses the server-signing model, it is critically important to use highly secure methods to store users' private keys. Note that Simple Supply does not provide an example of secure private key storage.

Messenger

The `Messenger` component of the Simple Supply REST API handles key generation. It maintains its own private key, as well as generating key pairs for new agents. For the Simple Supply code, see rest_api/simple_supply_rest_api/messaging.py.

```
class Messenger(object):
    def __init__(self, validator_url):
        self._connection = Connection(validator_url)
        self._context = create_context('secp256k1')
        self._crypto_factory = CryptoFactory(self._context)
        self._batch_signer = self._crypto_factory.new_signer(
            self._context.new_random_private_key())

    def open_validator_connection(self):
        self._connection.open()

    def close_validator_connection(self):
        self._connection.close()

    def get_new_key_pair(self):
        private_key = self._context.new_random_private_key()
        public_key = self._context.get_public_key(private_key)
        return public_key.as_hex(), private_key.as_hex()
```

The `get_new_key_pair()` method is called from the `RouteHandler` in order to generate a key pair for a new agent. It returns the public key and private key as hex so they can be stored.

Building the Transaction: `RouteHandler`

After `Messenger` generates keys, the `RouteHandler` calls `send_create_agent_transaction()`. This method generates a transaction signer based on the private key, uses the `transaction_creation` module to create the transaction and batch, and then sends the batch to the validator.

The `send_create_agent_transaction` method is defined in rest_api/simple_supply_rest_api/messaging.py.

```
async def send_create_agent_transaction(self, private_key, name, timestamp):
        transaction_signer = self._crypto_factory.new_signer(
            secp256k1.Secp256k1PrivateKey.from_hex(private_key))

        batch = transaction_creation.make_create_agent_transaction(
            transaction_signer=transaction_signer,
            batch_signer=self._batch_signer,
            name=name,
            timestamp=timestamp)
        await self._send_and_wait_for_commit(batch)
```

The `make_create_agent_transaction()` method creates a `MakeCreateAgent` transaction, and then wraps it in a batch.

Building the Transaction: make_create_agent_transaction

Module 5 > Creating a Sawtooth Application > Creating the REST API > Building the Transaction: make_create_agent_transaction

The `make_create_agent_transaction` function does two things:

- First, it creates the inputs and outputs for the transaction: the state addresses where a transaction processor is allowed to read or write during the execution of the transaction. In this case, the transaction processor only writes to or reads from the address of the agent.
- Next, it creates the payload and serializes it to bytes.

This function is defined in rest_api/simple_supply_rest_api/transaction_creation.py.

```
def make_create_agent_transaction(transaction_signer, batch_signer, name, timestamp):
    agent_address = addresser.get_agent_address(
        transaction_signer.get_public_key().as_hex())
    inputs = [agent_address]
    outputs = [agent_address]
    action = payload_pb2.CreateAgentAction(name=name)
    payload = payload_pb2.SimpleSupplyPayload(
        action=payload_pb2.SimpleSupplyPayload.CREATE_AGENT,
        create_agent=action,
        timestamp=timestamp)
    payload_bytes = payload.SerializeToString()
    return _make_batch(
        payload_bytes=payload_bytes,
        inputs=inputs,
        outputs=outputs,
        transaction_signer=transaction_signer,
        batch_signer=batch_signer)
```

Building the Transaction: `_make_batch`

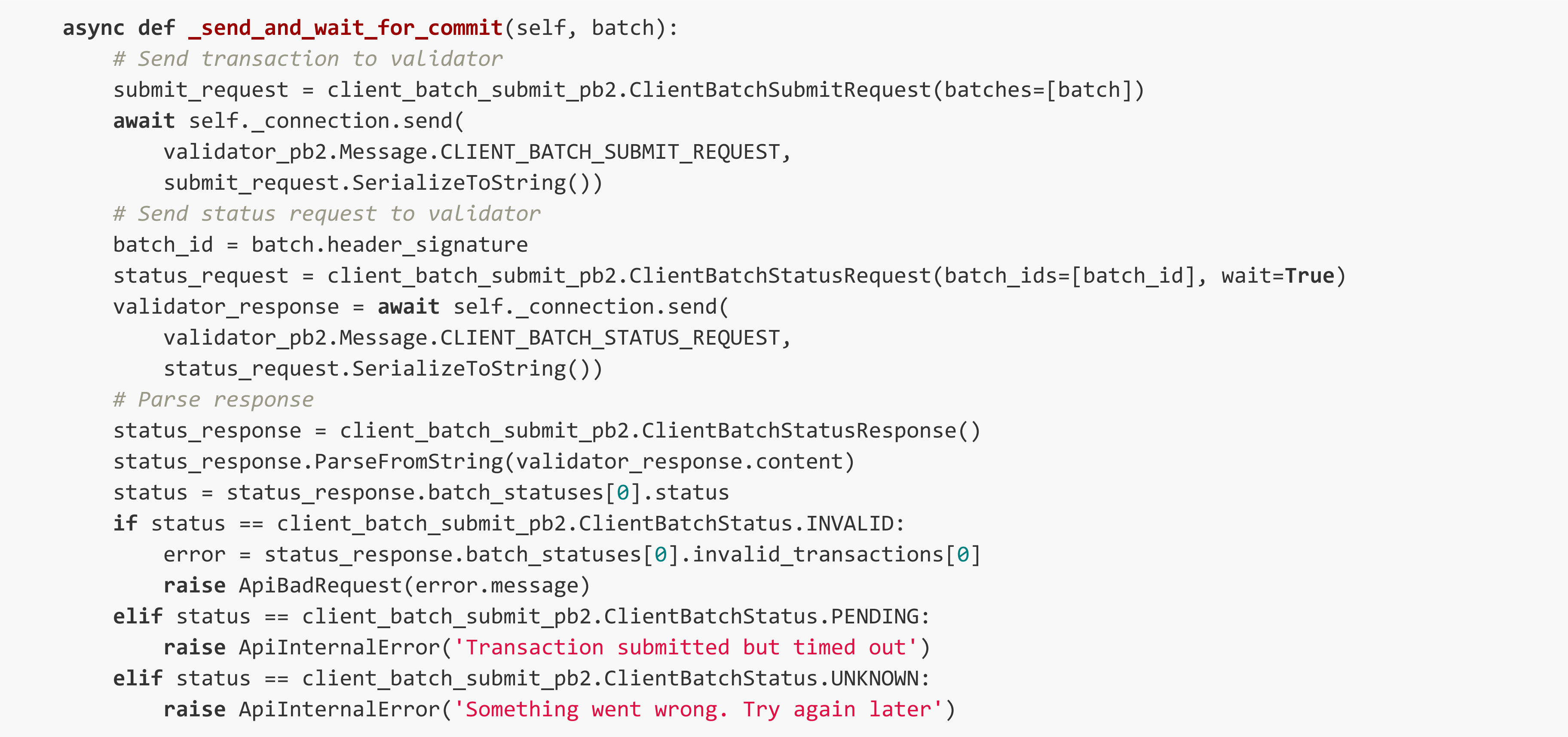
Next, `_make_batch` creates the transaction object and wraps it in a batch. See the code in rest_api/simple_supply_rest_api/transaction_creation.py.

```python
def _make_batch(payload_bytes, inputs, outputs, transaction_signer, batch_signer):
    transaction_header = transaction_pb2.TransactionHeader(
        family_name=addresser.FAMILY_NAME,
        family_version=addresser.FAMILY_VERSION,
        inputs=inputs,
        outputs=outputs,
        signer_public_key=transaction_signer.get_public_key().as_hex(),
        batcher_public_key=batch_signer.get_public_key().as_hex(),
        dependencies=[],
        payload_sha512=hashlib.sha512(payload_bytes).hexdigest())
    transaction_header_bytes = transaction_header.SerializeToString()
    transaction = transaction_pb2.Transaction(
        header=transaction_header_bytes,
        header_signature=transaction_signer.sign(transaction_header_bytes),
        payload=payload_bytes)
    batch_header = batch_pb2.BatchHeader(
        signer_public_key=batch_signer.get_public_key().as_hex(),
        transaction_ids=[transaction.header_signature])
    batch_header_bytes = batch_header.SerializeToString()
    batch = batch_pb2.Batch(
        header=batch_header_bytes,
        header_signature=batch_signer.sign(batch_header_bytes),
        transactions=[transaction])
    return batch
```

Building the Transaction: _send_and_wait_for_commit

Finally, `_send_and_wait_for_commit` wraps the batch in a `ClientBatchSubmitRequest` and sends it to the validator. The batch can be referenced by its header signature, which allows us to query for its status.

See rest_api/simple_supply_rest_api/messaging.py.



```python
    async def _send_and_wait_for_commit(self, batch):
        # Send transaction to validator
        submit_request = client_batch_submit_pb2.ClientBatchSubmitRequest(batches=[batch])
        await self._connection.send(
            validator_pb2.Message.CLIENT_BATCH_SUBMIT_REQUEST,
            submit_request.SerializeToString())
        # Send status request to validator
        batch_id = batch.header_signature
        status_request = client_batch_submit_pb2.ClientBatchStatusRequest(batch_ids=[batch_id], wait=True)
        validator_response = await self._connection.send(
            validator_pb2.Message.CLIENT_BATCH_STATUS_REQUEST,
            status_request.SerializeToString())
        # Parse response
        status_response = client_batch_submit_pb2.ClientBatchStatusResponse()
        status_response.ParseFromString(validator_response.content)
        status = status_response.batch_statuses[0].status
        if status == client_batch_submit_pb2.ClientBatchStatus.INVALID:
            error = status_response.batch_statuses[0].invalid_transactions[0]
            raise ApiBadRequest(error.message)
        elif status == client_batch_submit_pb2.ClientBatchStatus.PENDING:
            raise ApiInternalError('Transaction submitted but timed out')
        elif status == client_batch_submit_pb2.ClientBatchStatus.UNKNOWN:
            raise ApiInternalError('Something went wrong. Try again later')
```

Creating the Event Subscriber

The previous sections showed how to use Sawtooth's client signing and transaction processor SDKs to submit and handle transactions. Next, you will learn how to read data from the blockchain.

Although a client can fetch data directly from the validator, Simple Supply uses a slightly more sophisticated method in order to make fetching data easy and as fast as possible. This method uses an *event subscription client* that subscribes to Sawtooth state delta and block commit events. Sawtooth sends these events whenever the validator's state database is updated.

The events contain the raw state data at the updated addresses. The event subscription client processes the event data and uses it to update the *reporting database*, an off-chain copy of blockchain state. The REST API can query this database when a client needs to get information from the blockchain. This local reporting database allows rapid querying of state data.

If you choose to implement or use a REST API to facilitate communication between the client and validator, you must define the RESTful interfaces for fetching data from the blockchain state and submitting updates to the blockchain.

See protos/events.proto for the Sawtooth event-related protobuf messages, such as `Event` and `EventFilter` protobuf messages.

An event subscription client should operate in the following way:

1. The client establishes a connection to the validator using the Sawtooth SDK's Stream class.
2. The client constructs `EventSubscription` objects that specify the event type that it wants to receive from the validator, with filters to fine-tune the selection.

   ★ For a state-delta subscription, the client should subscribe to both `sawtooth/block-commit` and `sawtooth/state-delta` events, using a filter to specify the application namespace.
3. The client wraps the `EventSubscription` objects in a `ClientEventsSubscribeRequest`, along with the last known block IDs, and sends it to the validator.

   ★ For the first event subscription, send a null block ID in the `known_ids` field to request events for all state updates (starting with the genesis block).
4. After successfully subscribing, the client should listen for incoming events.
5. When the client receives an event, it checks whether the incoming block represents a fork or a duplicate block.
   - If the block is a duplicate, no additional processing is needed.
   - If the block represents a fork, the client should update the reporting database as necessary.
6. After resolving a possible fork, the client transforms raw state data from the event to application-specific objects. (A later topic has more details on this step.)
7. Existing records for application-specific objects are updated as “ended” and new records are inserted as “started” at the new block’s block number.

Storage Schemas for the Reporting Database

A reporting database can have three types of tables:

- A blocks table
- State object tables that are specific to the application
- Other application-specific tables for storing off-chain application information (such as user information and keys)

Updates to the block and state tables should happen atomically, within a single database transaction. This ensures that the block change and all the state changes remain consistent with the state of the validator network.

Blocks Table

The blocks table is a history of the block numbers and IDs in the blockchain. This table is used for fork resolution to compare the block number of a received event against its state root hash (the block ID).

The blocks table should have the following schema:

```
CREATE TABLE blocks (
    block_num  bigint PRIMARY KEY,
    block_id   varchar
);
```

This example is for a PostgreSQL database, but it could be adapted to a different database.

Application Tables

Storage schemas for the state object tables define a pattern for managing the data for a particular application. These tables should follow the guidelines of Type 2 Slowly Changing Dimensions. In this case, there should be three additional columns for each table row: `start_block_num` and `end_block_num`, and a unique `id`.

The `start_block_num` and `end_block_num` columns specify the range in which that state value is set or exists. Values that are valid as of the current block have `end_block_num` set to NULL or `MAX_INTEGER`.

Because there might be state objects in the database that refer to the same object in state at different block heights (with different block numbers), the primary key for each object cannot be the same as the object's natural key in blockchain state. Instead, use a sequence ID column or another unique ID scheme as the primary key. For better query performance, we recommend creating indexes on the natural key of the entry and the `end_block_num`.

For example, the table for state entries in the IntegerKey application (a simple Sawtooth application) could look like this:

```
CREATE TABLE integer_key (
    id bigserial PRIMARY_KEY,
    intkey_name varchar(256),
    intkey_value integer,
    start_block_num integer,
    end_block_num integer,
);

CREATE INDEX integer_key_key_block_num_idx
    ON integer_key (intkey_name, end_block_num NULLS FIRST);
```

Fork Resolution

When the event subscription client receives a new event, it must determine if the data represents a duplicate block, a fork, or new data that can be added to the reporting database normally. The event subscription client follows these steps:

1. Query for the received block's `block_num` in the blocks table
2. If a block with the same `block_num` already exists in the blocks database, the received block is either a duplicate or represents a fork. The client continues with these steps:
   a. Compare the `block_id` of the new block to the existing block
   b. If the block IDs match, they represent the same state, so the block is a duplicate. The client can stop processing this event. In this case, the client doesn't update the reporting database because the information is already there.
   c. If the block IDs are different, the client has detected a fork.

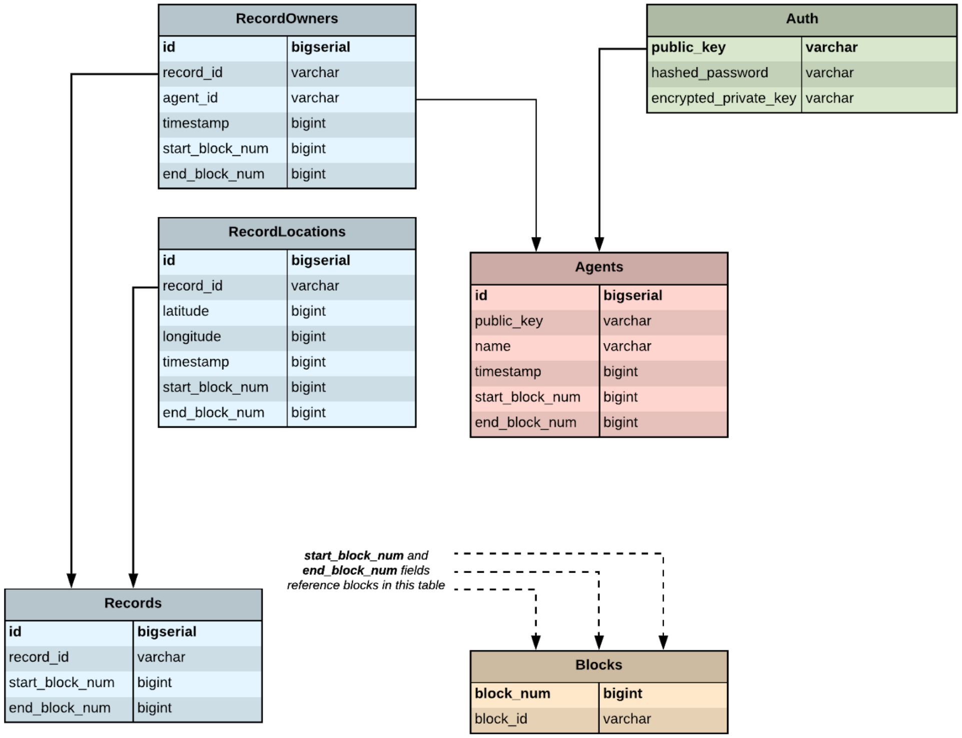
In the case of a fork, the event subscription client must drop the previous fork's data from the reporting database. The following pseudocode represents that process:

```
resolve_fork(existing_block):
  for table in domain_tables:
    delete from table where start_block_num >= existing_block.block_num
    update table set end_block_num = null \
       where end_block_num >= existing_block.block_num
  delete from block where block_num >= existing_block.block_num
```

After resolving the fork, the client updates the reporting database with the state data contained in the event.

Creating the Reporting Database

Before creating the event subscriber component, you need to create a storage schema for the reporting database. The previous topics described the general schema and tables. The Simple Supply reporting database looks like this:



RecordOwners	
id	**bigserial**
record_id	varchar
agent_id	varchar
timestamp	bigint
start_block_num	bigint
end_block_num	bigint

Auth	
public_key	**varchar**
hashed_password	varchar
encrypted_private_key	varchar

RecordLocations	
id	**bigserial**
record_id	varchar
latitude	bigint
longitude	bigint
timestamp	bigint
start_block_num	bigint
end_block_num	bigint

Agents	
id	**bigserial**
public_key	varchar
name	varchar
timestamp	bigint
start_block_num	bigint
end_block_num	bigint

Records	
id	**bigserial**
record_id	varchar
start_block_num	bigint
end_block_num	bigint

Blocks	
block_num	**bigint**
block_id	varchar

start_block_num *and* ***end_block_num*** *fields reference blocks in this table*

Simple Supply implements this database using PostgreSQL. The tables are created in the subscriber's [`Database`]() object when the component starts up.

Constructing the Subscriptions

After defining the schema for the reporting database, start constructing the event subscription. First, create `EventSubscription` objects for each event type to receive, `sawtooth/block-commit` and `sawtooth/state-delta`.

See subscriber/simple_supply_subscriber/subscriber.py.

```
block_sub = EventSubscription(event_type='sawtooth/block-commit')
delta_sub = EventSubscription(
    event_type='sawtooth/state-delta',
    filters=[EventFilter(
        key='address',
        match_string='^{}.*'.format(NAMESPACE),
        filter_type=EventFilter.REGEX_ANY)])
```

For the block subscription object, the subscriber wants to receive any block that comes in. However, for the state delta subscription, it wants only the events with Simple Supply data, so the subscription uses an `EventFilter` with a regular expression to filter for events in the Simple Supply namespace. For more information, see About Event Subscriptions in the Sawtooth documentation.

Submitting the Event Subscription Request

After constructing the `EventSubscription` objects that are wanted, the subscriber wraps them in a `ClientEventsSubscribeRequest` and sends it to the validator. The validator's response is parsed and any errors with the event subscription request are handled.

See subscriber/simple_supply_subscriber/subscriber.py.

```
# ...
request = ClientEventsSubscribeRequest(
    last_known_block_ids=known_ids,
    subscriptions=[block_sub, delta_sub])
response_future = self._stream.send(
    Message.CLIENT_EVENTS_SUBSCRIBE_REQUEST,
    request.SerializeToString())
response = ClientEventsSubscribeResponse()
response.ParseFromString(response_future.result().content)
# ...
self._is_active = True
# ...
```

In this case, the `last_known_block_ids` field is set to `['0000000000000000']`, which is the null block ID. This tells the validator that you want to get events for all state updates, starting with the genesis block.

To re-subscribe to events when the reporting database already contains state data, you request "event catch-up" by sending a list of known block IDs (from the blocks table in the reporting database) so that the validator knows where to start when sending state events.

Listening for Events

After successfully subscribing to events, the event subscription client listens for incoming events. Events are received by subscription clients as an `EventList`.

For convenience, Simple Supply uses an `event_handling` module to separate event handling from subscription and listening functionality.

```
# ...
while self._is_active:
    message_future = self._stream.receive()

    event_list = EventList()
    event_list.ParseFromString(message_future.result().content)
    for handler in self._event_handlers:
        handler(event_list.events)
```

Handling Events

Module 5 > Creating a Sawtooth Application > Creating the Event Subscriber > Handling Events

Each event handler is called with a reference to a Database object that manages the connection to the reporting database. Each time an event is received, the client follows the process described in “Event Subscription Client Operation”.

- If a fork is detected, call Database’s drop_fork method.
- Otherwise, parse the data contained in the events and update the database.

See subscriber/simple_supply_subscriber/event_handling.py.

```python
def _handle_events(database, events):
    block_num, block_id = _parse_new_block(events)
    try:
        is_duplicate = _resolve_if_forked(database, block_num, block_id)
        if not is_duplicate:
            _apply_state_changes(database, events, block_num, block_id)
        database.commit()
    except psycopg2.DatabaseError as err:
        LOGGER.exception('Unable to handle event: %s', err)
        database.rollback()
```

★ As explained above, each update should be handled as a single database transaction.

Once you start making database updates in _apply_state_changes, a transaction is created. After exiting the function, call database.commit() to finalize the transaction, or call database.rollback() to cancel the transaction entirely if there were any errors.

Handling Events (continued)

To apply the state changes to the reporting database, first parse the changes into a StateChangeList, which gives the address and value for each change. Next, determine the type of object that is contained in the event: examine the address (see “Addressing” in the Simple Supply specification document), then use the address to deserialize the state data into an object.

See subscriber/simple_supply_subscriber/event_handling.py.

```python
def _apply_state_changes(database, events, block_num, block_id):
    changes = _parse_state_changes(events)
    for change in changes:
        data_type, resources = deserialize_data(change.address, change.value)
        database.insert_block({'block_num': block_num, 'block_id': block_id})
        if data_type == AddressSpace.AGENT:
            _apply_agent_change(database, block_num, resources)
        if data_type == AddressSpace.RECORD:
            _apply_record_change(database, block_num, resources)
        else:
            LOGGER.warning('Unsupported data type: %s', data_type)
```

Finally, insert the deserialized state objects into the database. See subscriber/simple_supply_subscriber/event_handling.py.

```python
def _apply_agent_change(database, block_num, agents):
    for agent in agents:
        agent['start_block_num'] = block_num
        agent['end_block_num'] = MAX_BLOCK_NUMBER
        database.insert_agent(agent)
```

Updating the Database

Module 5 > Creating a Sawtooth Application > Creating the Event Subscriber > Updating the Database

The `Database` class handles the logic of managing the `start_block_num` and `end_block_num` fields. If there is already an resource with that natural key, we update its `end_block_num` with the current block. See subscriber/simple_supply_subscriber/database.py.

```
def insert_agent(self, agent_dict):
    update_agent = """
    UPDATE agents SET end_block_num = {} WHERE end_block_num = {} AND public_key = '{}'
    """.format(
        agent_dict['start_block_num'],
        agent_dict['end_block_num'],
        agent_dict['public_key'])
    insert_agent = """
    INSERT INTO agents (
    public_key,
    name,
    timestamp,
    start_block_num,
    end_block_num)
    VALUES ('{}', '{}', '{}', '{}', '{}');
    """.format(
        agent_dict['public_key'],
        agent_dict['name'],
        agent_dict['timestamp'],
        agent_dict['start_block_num'],
        agent_dict['end_block_num'])

    with self._conn.cursor() as cursor:
        cursor.execute(update_agent)
        cursor.execute(insert_agent)
```

Fetching from the Database

Module 5 > Creating a Sawtooth Application > Creating the Event Subscriber > Fetching from the Database

After the reporting database contains some event data, the client can make GET requests to the Simple Supply REST API, which will then query the database for the requested information.

See rest_api/simple_supply_rest_api/database.py.

```
async def fetch_agent_resource(self, public_key):
    fetch = """
    SELECT public_key, name, timestamp FROM agents
    WHERE public_key='{0}'
    AND ({1}) >= start_block_num
    AND ({1}) < end_block_num;
    """.format(public_key, LATEST_BLOCK_NUM)

    async with self._conn.cursor(cursor_factory=RealDictCursor) as cursor:
        await cursor.execute(fetch)
        return await cursor.fetchone()
```

The Simple Supply REST API can only fetch **current** state data from the reporting database. A more complex REST API could also query for previous state data by specifying the block height (block number). This feature might be required for an application that needs to return data that will be synchronized at a certain point. See Sawtooth Supply Chain and Sawtooth Marketplace for examples of this functionality.

Testing the Application

The Simple Supply repository includes basic unit tests for this example application. To run them, navigate to the project's root directory and enter this command:

```
$ ./bin/run-tests
```

This command uses a test docker-compose file that opens test containers for the unit texts, then shuts down the docker containers when the unit tests are done.

Of course, your application should also include integration tests that include more complex functions such as sending queries to the REST API and receiving responses. For example integration tests, see Sawtooth Marketplace.

Creating the Client

As discussed earlier, the Simple Supply application uses the server-signing model. This means that any client application that uses the Simple Supply REST API is not in charge of creating or signing transactions. Instead, any HTTP client (such as Postman or cURL) can submit transactions or fetch data by sending requests to the Simple Supply REST API.

Because Simple Supply is intended as a general-use application, where the entities involved are generic agents and records, you could add a domain-specific client to this application platform.

As an example, the Simple Supply repository includes a basic web app called Curator. This app can be used to track the provenance of artwork as it is transferred from different museums or collectors. In this case, records are used to represent works of art, and agents represent museums or collectors own or exhibit the artwork.

The next module describes how to use Curator and shows how the log messages let you monitor the on-chain and off-chain activity of the client, REST API, transaction processor, and Sawtooth validator.

Module 5: Summary

This module walked you through the implementation details of the example Simple Supply application, which demonstrated the important concepts of Sawtooth application development.

The next module shows how to run Simple Supply and examine the log output. It also describes basic application troubleshooting.

Veit Langenbucher Musical Clock with Spinet and Organ (interior view)
Metropolitan Museum of Art
Public domain image