eventsourcing Documentation

Release 3.1.0

John Bywater

Jan 25, 2018

Contents

A library for event sourcing in Python.

CHAPTER 1

Overview

What is event sourcing? One definition suggests the state of an event sourced application is determined by a sequence of events. Another definition has event sourcing as a persistence mechanism for domain driven design. In any case, it is common for the state of a software application to be distributed or partitioned across a set of entities or aggregates in a domain model.

Therefore, this library provides mechanisms useful in event sourced applications: a style for coding entity behaviours that emit events; and a way for the events of an entity to be stored and replayed to obtain the entities on demand.

This documentation provides: instructions for *installing* the package, highlights the main *features* of the library, describes the *design* of the software, the *infrastructure layer*, the *domain model layer*, the *application layer*, and has *examples* and some *background* information about the project.

This project is hosted on GitHub. Please register any issues, questions, and requests on GitHub.

1.1 Background

Although the event sourcing patterns are each quite simple, and they can be reproduced in code for each project, they do suggest cohesive mechanisms, for example applying and publishing the events generated within domain entities, storing and retrieving selections of the events in a highly scalable manner, replaying the stored events for a particular entity to obtain the current state, and projecting views of the event stream that are persisted in other models.

Therefore, quoting from Eric Evans' book about domain-driven design:

> *"Partition a conceptually COHESIVE MECHANISM into a separate lightweight framework. Particularly watch for formalisms for well-documented categories of algorithms. Expose the capabilities of the framework with an INTENTION-REVEALING INTERFACE. Now the other elements of the domain can focus on expressing the problem ('what'), delegating the intricacies of the solution ('how') to the framework."*

Inspiration:

- Martin Fowler's article on event sourcing
- Greg Young's discussions about event sourcing, and EventStore system
- Robert Smallshire's brilliant example on Bitbucket

- Various professional projects that called for this approach, for which I didn't want to rewrite the same things each time

See also:

- Evaluation of using NoSQL databases in an event sourcing system by Johan Rothsberg
- Object-relational impedance mismatch page on Wikipedia
- An introduction to event storming by a Steven Lowe, principal consultant developer at ThoughtWorks.

1.2 Quick start

This section shows how to write a very simple event sourced application using classes from the library. It shows the general story, which is elaborated over the following pages.

Please use pip to install the library with the 'sqlalchemy' option.

```
pip install eventsourcing[sqlalchemy]
```

1.2.1 Domain

Firstly, import the example entity class `Example` and its factory function `create_new_example()`.

```
from eventsourcing.example.domainmodel import create_new_example, Example
```

These classes will be used as the domain model in this example.

1.2.2 Infrastructure

Next, setup an SQLite database in memory, using library classes `SQLAlchemyDatastore`, with `SQLAlchemySettings` and `IntegerSequencedItemRecord`.

```
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemySettings,␣
↪SQLAlchemyDatastore
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪IntegerSequencedItemRecord

datastore = SQLAlchemyDatastore(
    settings=SQLAlchemySettings(uri='sqlite:///:memory:'),
    tables=(IntegerSequencedItemRecord,),
)

datastore.setup_connection()
datastore.setup_tables()
```

1.2.3 Application

Finally, define an application object factory, that constructs an application object from library class `ApplicationWithPersistencePolicies`. The application class happens to take an active record strategy object and a session object.

The active record strategy is an instance of class `SQLAlchemyActiveRecordStrategy`. The session object is an argument of the application factory, and will be a normal SQLAlchemy session object.

```
from eventsourcing.application.base import ApplicationWithPersistencePolicies
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy
from eventsourcing.infrastructure.sequenceditem import SequencedItem
from eventsourcing.infrastructure.eventsourcedrepository import EventSourcedRepository

def construct_application(session):
    app = ApplicationWithPersistencePolicies(
        entity_active_record_strategy=SQLAlchemyActiveRecordStrategy(
            active_record_class=IntegerSequencedItemRecord,
            session=session
        )
    )
    app.example_repository = EventSourcedRepository(
        event_store=app.entity_event_store,
        mutator=Example._mutate,
    )
    return app
```

An example repository constructed from class *EventSourcedRepository*, and is assigned to the application object attribute `example_repository`. It is possible to subclass the library application class, and extend it by constructing entity repositories in the `__init__()`, we just didn't do that here.

1.2.4 Run the code

Now, use the application to create, read, update, and delete "example" entities.

```
with construct_application(datastore.session) as app:

    # Create.
    example = create_new_example(foo='bar')

    # Read.
    assert example.id in app.example_repository
    assert app.example_repository[example.id].foo == 'bar'

    # Update.
    example.foo = 'baz'
    assert app.example_repository[example.id].foo == 'baz'

    # Delete.
    example.discard()
    assert example.id not in app.example_repository
```

1.3 Installation

Use pip to install the library from the Python Package Index.

```
pip install eventsourcing
```

If you want to use SQLAlchemy, then please install the library with the 'sqlalchemy' option.

```
pip install eventsourcing[sqlalchemy]
```

Similarly, if you want to use Cassandra, please install with the ‘cassandra’ option.

```
pip install eventsourcing[cassandra]
```

If you want to use encryption, please install with the ‘crypto’ option.

```
pip install eventsourcing[crypto]
```

You can install combinations of options at the same time, for exampe the follow command will install dependencies for Cassandra and for encryption.

```
pip install eventsourcing[cassandra,crypto]
```

Running the install command with different options will just install the extra dependencies associated with that option. If you installed without any options, you can easily install optional dependencies later by running the install command again with the options you want.

1.4 Features

Event store — appends and retrieves domain events. Uses a sequenced item mapper with an active record strategy to map domain events to databases in ways that can be easily extended and replaced.

Optimistic concurrency control — can be used to ensure a distributed or horizontally scaled application doesn’t become inconsistent due to concurrent method execution. Leverages any optimistic concurrency controls in the database adapted by the active record strategy.

Application-level encryption — encrypts and decrypts stored events, using a cipher strategy passed as an option to the sequenced item mapper. Can be used to encrypt some events, or all events, or not applied at all (the default).

Snapshotting — avoids replaying an entire event stream to obtain the state of an entity. A snapshot strategy is included which reuses the capabilities of this library by implementing snapshots as events.

Abstract base classes — suggest how to structure an event sourced application. The library has base classes for application objects, domain entities, entity repositories, domain events of various types, mapping strategies, snapshotting strategies, cipher strategies, etc. They are well factored, relatively simple, and can be easily extended for your own purposes. If you wanted to create a domain model that is entirely stand-alone (recommended by purists for maximum longevity), you might start by replicating the library classes.

Worked examples — a simple example application, with an example entity class, example domain events, an example factory method, an example mutator function, and an example database table.

1.5 Design

The design of the library follows the layered architecture: interfaces, application, domain, and infrastructure.

The domain layer contains a model of the supported domain, and services that depend on that model. The infrastructure layer encapsulates the infrastructural services required by the application.

The application is responsible for binding domain and infrastructure, and has policies such as the persistence policy, which stores domain events whenever they are published by the model.

The example application has an example respository, from which example entities can be retrieved. It also has a factory method to register new example entities. Each repository has an event player, which all share an event store with the persistence policy. The persistence policy uses the event store to store domain events, and the event players use the event store to retrieve the stored events. The event players also share with the model the mutator functions that are used to apply domain events to an initial state.

Functionality such as mapping events to a database, or snapshotting, is factored as strategy objects and injected into dependents by constructor parameter. Application level encryption is a mapping option.

The sequenced item persistence model allows domain events to be stored in wide variety of database services, and optionally makes use of any optimistic concurrency controls the database system may afford.

1.6 Infrastructure

The library's infrastructure layer provides a cohesive mechanism for storing events as sequences of items. The entire mechanism is encapsulated by the library's *EventStore* class.

The event store uses a "sequenced item mapper" and an "active record strategy". The sequenced item mapper and the active record strategy share a common "sequenced item" type. The sequenced item mapper can convert objects such as domain events to sequenced items, and the active record strategy can write sequenced items to a database.

1.6.1 Sequenced items

A sequenced item type provides a common persistence model across the components of the mechanism. The sequenced item type is normally declared as a namedtuple.

```
from collections import namedtuple

SequencedItem = namedtuple('SequencedItem', ['sequence_id', 'position', 'topic', 'data
↪'])
```

The names of the fields are arbitrary. However, the first field of a sequenced item namedtuple represents the identity of a sequence to which an item belongs, the second field represents the position of the item in its sequence, the third field represents a topic to which the item pertains (dimension of concern), and the fourth field represents the data associated with the item.

SequencedItem namedtuple

The library provides a sequenced item namedtuple called *SequencedItem*.

```
from eventsourcing.infrastructure.sequenceditem import SequencedItem
```

The attributes of `SequencedItem` are `sequence_id`, `position`, `topic`, and `data`.

The `sequence_id` identifies the sequence in which the item belongs.

The `position` identifies the position of the item in its sequence.

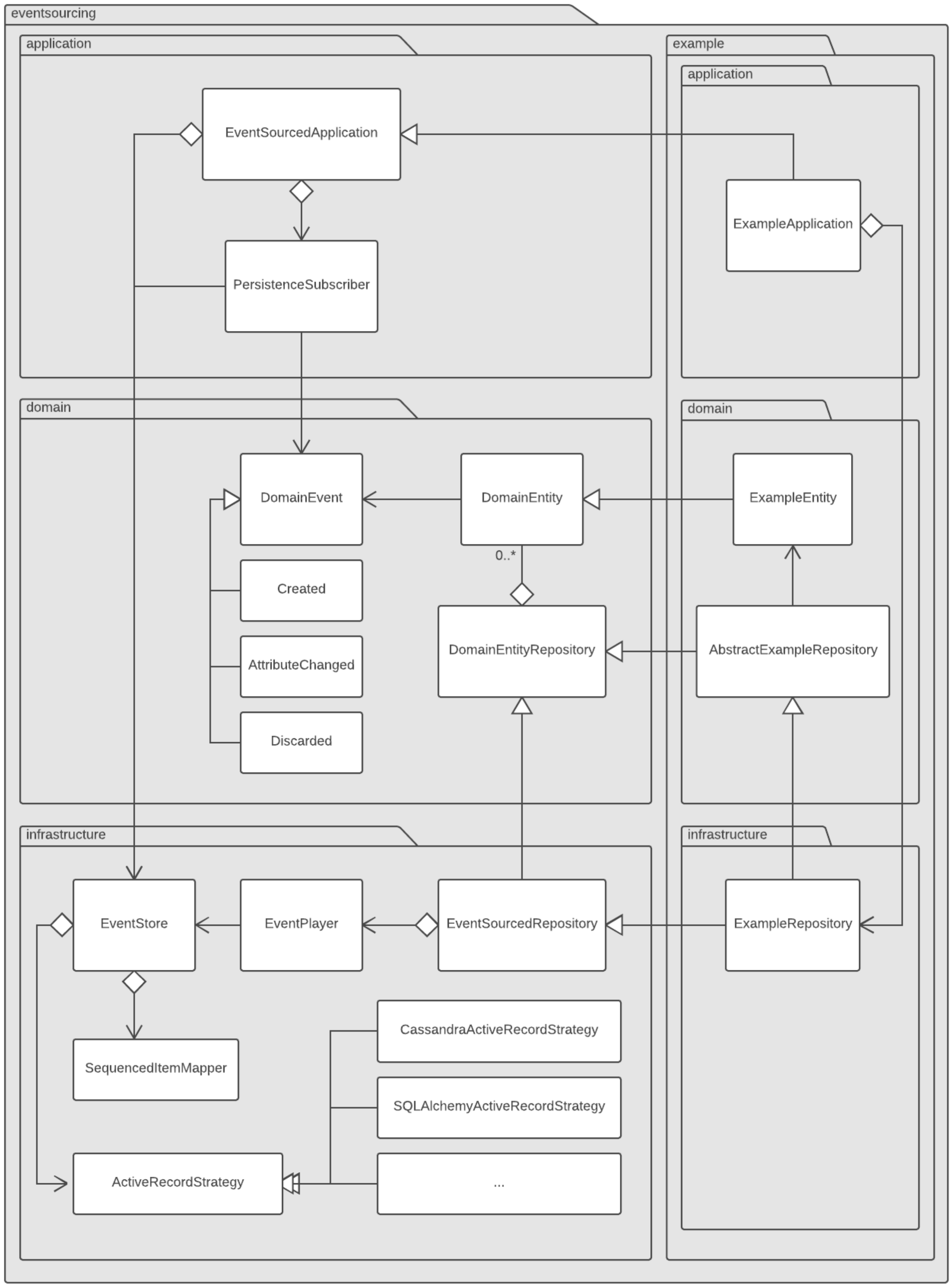
The `topic` identifies the dimension of concern to which the item pertains.

The `data` holds the values of the item, perhaps serialized to JSON, and optionally encrypted.

```
from uuid import uuid4

sequence1 = uuid4()

sequenced_item1 = SequencedItem(
    sequence_id=sequence1,
    position=0,
```


eventsourcing
application
EventSourcedApplication
PersistenceSubscriber
domain
DomainEvent
Created
AttributeChanged
Discarded
DomainEntity
0..*
DomainEntityRepository
infrastructure
EventStore
EventPlayer
EventSourcedRepository
SequencedItemMapper
CassandraActiveRecordStrategy
SQLAlchemyActiveRecordStrategy
...
ActiveRecordStrategy
example
application
ExampleApplication
domain
ExampleEntity
AbstractExampleRepository
infrastructure
ExampleRepository

```
    topic='eventsourcing.domain.model.events#DomainEvent',
    data='{"foo":"bar"}'
)
assert sequenced_item1.sequence_id == sequence1
assert sequenced_item1.position == 0
assert sequenced_item1.topic == 'eventsourcing.domain.model.events#DomainEvent'
assert sequenced_item1.data == '{"foo":"bar"}'
```

StoredEvent namedtuple

As an alternative, the library also provides a sequenced item namedtuple called `StoredEvent`. The attributes of the `StoredEvent` namedtuple are `originator_id`, `originator_version`, `event_type`, and `state`.

The `originator_id` is the ID of the aggregate that published the event, and is equivalent to `sequence_id` above.

The `originator_version` is the version of the aggregate that published the event, and is equivalent to `position` above.

The `event_type` identifies the class of the domain event that is stored, and is equivalent to `topic` above.

The `state` holds the state of the domain event, and is equivalent to `data` above.

```
from eventsourcing.infrastructure.sequenceditem import StoredEvent

aggregate1 = uuid4()

stored_event1 = StoredEvent(
    originator_id=aggregate1,
    originator_version=0,
    event_type='eventsourcing.domain.model.events#DomainEvent',
    state='{"foo":"bar"}'
)
assert stored_event1.originator_id == aggregate1
assert stored_event1.originator_version == 0
assert stored_event1.event_type == 'eventsourcing.domain.model.events#DomainEvent'
assert stored_event1.state == '{"foo":"bar"}'
```

1.6.2 Active record strategies

An active record strategy writes sequenced items to database records.

The library has an abstract base class `AbstractActiveRecordStrategy` with abstract methods `append()` and `get_items()`, which can be used on concrete implementations to read and write sequenced items in a database.

An active record strategy is constructed with a `sequenced_item_class` and a matching `active_record_class`. The field names of a suitable active record class will match the field names of the sequenced item namedtuple.

SQLAlchemy

The library has a concrete active record strategy for SQLAlchemy provided by the object class `SQLAlchemyActiveRecordStrategy`.

```
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy
```

The library also provides active record classes for SQLAlchemy, such as `IntegerSequencedItemRecord` and `StoredEventRecord`. The `IntegerSequencedItemRecord` class matches the default `SequencedItem` namedtuple. The `StoredEventRecord` class matches the alternative `StoredEvent` namedtuple.

The code below uses the namedtuple `StoredEvent` and the active record `StoredEventRecord`.

```
from eventsourcing.infrastructure.sqlalchemy.activerecords import StoredEventRecord
```

Database settings can be configured using `SQLAlchemySettings`, which is constructed with a `uri` connection string. The code below uses an in-memory SQLite database.

```
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemySettings

settings = SQLAlchemySettings(uri='sqlite:///:memory:')
```

To help setup a database connection and tables, the library has object class `SQLAlchemyDatastore`.

The `SQLAlchemyDatastore` is constructed with the `settings` object, and a tuple of active record classes passed using the `tables` arg.

```
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemyDatastore

datastore = SQLAlchemyDatastore(
    settings=settings,
    tables=(StoredEventRecord,)
)
```

Please note, if you have declared your own SQLAlchemy model `Base` class, you may wish to define your own active record classes which inherit from your `Base` class. If so, if may help to refer to the library active record classes to see how SQLALchemy ORM columns and indexes can be used to persist sequenced items.

The methods `setup_connection()` and `setup_tables()` of the datastore object can be used to setup the database connection and the tables.

```
datastore.setup_connection()
datastore.setup_tables()
```

As well as `sequenced_item_class` and a matching `active_record_class`, the `SQLAlchemyActiveRecordStrategy` requires a scoped session object, passed using the constructor arg `session`. For convenience, the `SQLAlchemyDatabase` has a thread-scoped session facade set as its a `session` attribute. You may wish to use a different scoped session facade, such as a request-scoped session object provided by a Web framework.

```
active_record_strategy = SQLAlchemyActiveRecordStrategy(
    sequenced_item_class=StoredEvent,
    active_record_class=StoredEventRecord,
    session=datastore.session,
)
```

Sequenced items (or “stored events” in this example) can be appended to the database using the `append()` method of the active record strategy.

```
active_record_strategy.append(stored_event1)
```

(Please note, since the position is given by the sequenced item itself, the word “append” means here “to add something extra” rather than the perhaps more common but stricter meaning “to add to the end of a document”. That is, the database is deliberately not responsible for positioning a new item at the end of a sequence. So perhaps “save” would be a better name for this operation.)

All the previously appended items of a sequence can be retrieved by using the `get_items()` method.

```
results = active_record_strategy.get_items(aggregate1)
```

Since by now only one item was stored, so there is only one item in the results.

```
assert len(results) == 1
assert results[0] == stored_event1
```

Apache Cassandra

The library also has a concrete active record strategy for Apache Cassandra provided by `CassandraActiveRecordStrategy` class.

Similarly, for the `CassandraActiveRecordStrategy`, the `IntegerSequencedItemRecord` from `eventsourcing.infrastructure.cassandra.activerecords` matches the `SequencedItem` namedtuple. The `StoredEventRecord` from the same module matches the `StoredEvent` namedtuple.

The `CassandraDatastore` class uses the `CassandraSettings` class to setup a Cassandra database.

```
from eventsourcing.infrastructure.cassandra.datastore import CassandraDatastore,␣
↪CassandraSettings
from eventsourcing.infrastructure.cassandra.activerecords import␣
↪CassandraActiveRecordStrategy, StoredEventRecord

cassandra_datastore = CassandraDatastore(
    settings=CassandraSettings(),
    tables=(StoredEventRecord,)
)
cassandra_datastore.setup_connection()
cassandra_datastore.setup_tables()

cassandra_active_record_strategy = CassandraActiveRecordStrategy(
    active_record_class=StoredEventRecord,
    sequenced_item_class=StoredEvent,
)

results = cassandra_active_record_strategy.get_items(aggregate1)
assert len(results) == 0

cassandra_active_record_strategy.append(stored_event1)

results = cassandra_active_record_strategy.get_items(aggregate1)
assert results[0] == stored_event1

cassandra_datastore.drop_tables()
cassandra_datastore.drop_connection()
```

Please refer to `CassandraSettings` class for information about configuring away from default settings.

Sequenced item conflicts

It is a feature of the active record strategy that it isn't possible successfully to append two items at the same position in the same sequence. If such an attempt is made, a `SequencedItemConflict` will be raised by the active record strategy.

```
from eventsourcing.exceptions import SequencedItemConflict

# Fail to append an item at the same position in the same sequence as a previous item.
try:
    active_record_strategy.append(stored_event1)
except SequencedItemConflict:
    pass
else:
    raise Exception("SequencedItemConflict not raised")
```

This feature is implemented using optimistic concurrency control features of the underlying database. With SQLAlchemy, the primary key constraint involves both the sequence and the position columns. With Cassandra the position is the primary key in the sequence partition, and the "IF NOT EXISTS" feature is applied.

1.6.3 Sequenced item mapper

A sequenced item mapper is used by the event store to map between sequenced item namedtuple objects and application-level objects such as domain events.

The library provides a sequenced item mapper object class called `SequencedItemMapper`.

```
from eventsourcing.infrastructure.sequenceditemmapper import SequencedItemMapper
```

The `SequencedItemMapper` has a constructor arg `sequenced_item_class`, which defaults to the library's sequenced item namedtuple `SequencedItem`.

```
sequenced_item_mapper = SequencedItemMapper()
```

The method `from_sequenced_item()` can be used to convert sequenced item objects to application-level objects.

```
domain_event = sequenced_item_mapper.from_sequenced_item(sequenced_item1)

assert domain_event.sequence_id == sequence1
assert domain_event.position == 0
assert domain_event.foo == 'bar'
```

The method `to_sequenced_item()` can be used to convert application-level objects to sequenced item namedtuples.

```
assert sequenced_item_mapper.to_sequenced_item(domain_event) == sequenced_item1
```

If the names of the first two fields of the sequenced item namedtuple (e.g. `sequence_id` and `position`) do not match the names of the attributes of the application-level object which identify a sequence and a position (e.g. `originator_id` and `originator_version`) then the attribute names can be given to the sequenced item mapper using constructor args `sequence_id_attr_name` and `position_attr_name`.

```
sequenced_item_mapper = SequencedItemMapper(
    sequence_id_attr_name='originator_id',
    position_attr_name='originator_version'
)

domain_event1 = sequenced_item_mapper.from_sequenced_item(sequenced_item1)

assert domain_event1.foo == 'bar', domain_event1
assert domain_event1.originator_id == sequence1
assert domain_event1.originator_version == 0
assert sequenced_item_mapper.to_sequenced_item(domain_event1) == sequenced_item1
```

Alternatively, the constructor arg `sequenced_item_class` can be set with a sequenced item namedtuple type that is different from the default `SequencedItem` namedtuple, such as the library's `StoredEvent` namedtuple.

```
sequenced_item_mapper = SequencedItemMapper(
    sequenced_item_class=StoredEvent,
)

domain_event1 = sequenced_item_mapper.from_sequenced_item(stored_event1)

assert domain_event1.foo == 'bar', domain_event1
assert domain_event1.originator_id == aggregate1
assert sequenced_item_mapper.to_sequenced_item(domain_event1) == stored_event1
```

Since the alternative `StoredEvent` namedtuple can be used instead of the default `SequencedItem` namedtuple, so it is possible to use a custom namedtuple. Which alternative you use for your project depends on your preferences for the names in the your domain events and your persistence model.

Please note, it is required of these application-level objects that the "topic" generated by `get_topic()` from the object class is resolved by `resolve_topic()` back to the same object class.

```
from eventsourcing.domain.model.events import Created
from eventsourcing.infrastructure.topic import get_topic, resolve_topic

topic = get_topic(Created)
assert resolve_topic(topic) == Created
assert topic == 'eventsourcing.domain.model.events#Created'
```

Custom JSON transcoding

The `SequencedItemMapper` can be constructed with optional args `json_encoder_class` and `json_decoder_class`. The defaults are the library's `ObjectJSONEncoder` and `ObjectJSONDecoder` which can be extended to support types of value objects that are not currently supported by the library.

The code below extends the JSON transcoding to support sets.

```
from eventsourcing.infrastructure.transcoding import ObjectJSONEncoder,␣
↪ObjectJSONDecoder


class CustomObjectJSONEncoder(ObjectJSONEncoder):
    def default(self, obj):
        if isinstance(obj, set):
            return {'__set__': list(obj)}
        else:
            return super(CustomObjectJSONEncoder, self).default(obj)


class CustomObjectJSONDecoder(ObjectJSONDecoder):
    @classmethod
    def from_jsonable(cls, d):
        if '__set__' in d:
            return cls._decode_set(d)
        else:
            return ObjectJSONDecoder.from_jsonable(d)
    @staticmethod
    def _decode_set(d):
        return set(d['__set__'])


customized_sequenced_item_mapper = SequencedItemMapper(
    json_encoder_class=CustomObjectJSONEncoder,
    json_decoder_class=CustomObjectJSONDecoder,
)

domain_event = customized_sequenced_item_mapper.from_sequenced_item(
    SequencedItem(
        sequence_id=sequence1,
        position=0,
        topic='eventsourcing.domain.model.events#DomainEvent',
        data='{"foo":{"__set__":["bar","baz"]}}'
    )
)
assert domain_event.foo == set(["bar", "baz"])

sequenced_item = customized_sequenced_item_mapper.to_sequenced_item(domain_event)
assert sequenced_item.data.startswith('{"foo":{"__set__":["ba')
```

Application-level encryption

The `SequencedItemMapper` can be constructed with a symmetric cipher object. The library provides an AES cipher object class called `AESCipher`.

The `AESCipher` is given an encryption key, using constructor arg `aes_key`, which must be either 16, 24, or 32 random bytes (128, 192, or 256 bits). Longer keys take more time to encrypt plaintext, but produce more secure ciphertext. Generating and storing a secure key requires functionality beyond the scope of this library.

```
from eventsourcing.infrastructure.cipher.aes import AESCipher

cipher = AESCipher(aes_key=b'01234567890123456789012345678901')  # Key with 256 bits.

ciphertext = cipher.encrypt('plaintext')
plaintext = cipher.decrypt(ciphertext)

assert ciphertext != 'plaintext'
assert plaintext == 'plaintext'
```

If the `SequencedItemMapper` has an optional constructor arg `cipher`. If `always_encrypt` is True, then the `state` field of every stored event will be encrypted with the cipher.

```
# Construct sequenced item mapper to always encrypt domain events.
ciphered_sequenced_item_mapper = SequencedItemMapper(
    sequenced_item_class=StoredEvent,
    cipher=cipher,
    always_encrypt=True,
)

# Domain event attribute ``foo`` has value ``'bar'``.
assert domain_event1.foo == 'bar'
```

```
# Map the domain event to an encrypted stored event namedtuple.
stored_event = ciphered_sequenced_item_mapper.to_sequenced_item(domain_event1)

# Attribute names and values of the domain event are not visible in the encrypted␣
↪``state`` field.
assert 'foo' not in stored_event.state
assert 'bar' not in stored_event.state

# Recover the domain event from the encrypted state.
domain_event = ciphered_sequenced_item_mapper.from_sequenced_item(stored_event)

# Domain event has decrypted attributes.
assert domain_event.foo == 'bar'
```

Please note, the sequence ID and position values are necessarily not encrypted. However, by encrypting the state of the event, sensitive information, such as personally identifiable information, will be encrypted at the level of the application, before being sent to the database, and so it will be encrypted in the database (and in all backups of the database).

1.6.4 Event store

The library's `EventStore` provides an interface to the library's cohesive mechanism for storing events as sequences of items, and can be used directly within an event sourced application to append and retrieve its domain events.

The `EventStore` is constructed with a sequenced item mapper and an active record strategy, both are discussed in detail in the sections above.

```
from eventsourcing.infrastructure.eventstore import EventStore

event_store = EventStore(
    sequenced_item_mapper=sequenced_item_mapper,
    active_record_strategy=active_record_strategy,
)
```

The event store's `append()` method can append a domain event to its sequence. The event store uses the `sequenced_item_mapper` to obtain a sequenced item namedtuple from a domain events, and it uses the `active_record_strategy` to write a sequenced item to a database.

In the code below, a `DomainEvent` is appended to sequence `aggregate1` at position `1`.

```
from eventsourcing.domain.model.events import DomainEvent

event_store.append(
    DomainEvent(
        originator_id=aggregate1,
        originator_version=1,
        foo='baz',
    )
)
```

The event store's method `get_domain_events()` is used to retrieve events that have previously been appended. The event store uses the `active_record_strategy` to read the sequenced items from a database, and it uses the `sequenced_item_mapper` to obtain domain events from the sequenced items.

```
results = event_store.get_domain_events(aggregate1)
```

Since by now two domain events have been stored, so there are two domain events in the results.

```
assert len(results) == 2

assert results[0].originator_id == aggregate1
assert results[0].foo == 'bar'

assert results[1].originator_id == aggregate1
assert results[1].foo == 'baz'
```

The optional arguments of `get_domain_events()` can be used to select some of the items in the sequence.

The `lt` arg is used to select items below the given position in the sequence.

The `lte` arg is used to select items below and at the given position in the sequence.

The `gte` arg is used to select items at and above the given position in the sequence.

The `gt` arg is used to select items above the given position in the sequence.

The `limit` arg is used to limit the number of items selected from the sequence.

The `is_ascending` arg is used when selecting items. It affects how any `limit` is applied, and determines the order of the results. Hence, it can affect both the content of the results and the performance of the method.

```
# Get events below and at position 0.
result = event_store.get_domain_events(aggregate1, lte=0)
assert len(result) == 1, result
assert result[0].originator_id == aggregate1
assert result[0].originator_version == 0
assert result[0].foo == 'bar'

# Get events at and above position 1.
result = event_store.get_domain_events(aggregate1, gte=1)
assert len(result) == 1, result
assert result[0].originator_id == aggregate1
assert result[0].originator_version == 1
assert result[0].foo == 'baz'

# Get the first event in the sequence.
result = event_store.get_domain_events(aggregate1, limit=1)
assert len(result) == 1, result
assert result[0].originator_id == aggregate1
assert result[0].originator_version == 0
assert result[0].foo == 'bar'

# Get the last event in the sequence.
result = event_store.get_domain_events(aggregate1, limit=1, is_ascending=False)
assert len(result) == 1, result
assert result[0].originator_id == aggregate1
assert result[0].originator_version == 1
assert result[0].foo == 'baz'
```

Optimistic concurrency control

It is a feature of the event store that it isn't possible successfully to append two events at the same position in the same sequence. This condition is coded as a `ConcurrencyError` since a correct program running in a single thread wouldn't attempt to append an event that it had already successfully appended.

```
from eventsourcing.exceptions import ConcurrencyError

# Fail to append an event at the same position in the same sequence as a previous␣
↪event.
try:
    event_store.append(
        DomainEvent(
            originator_id=aggregate1,
            originator_version=1,
            foo='baz',
        )
    )
except ConcurrencyError:
    pass
else:
    raise Exception("ConcurrencyError not raised")
```

This feature depends on the behaviour of the active record strategy's `append()` method: the event store will raise a `ConcurrencyError` if a `SequencedItemConflict` is raised by its active record strategy.

If a command fails due to a concurrency error, the command can be retried with the lastest state. The `@retry` decorator can help code retries on commands.

```
from eventsourcing.domain.model.decorators import retry

errors = []

@retry(ConcurrencyError, max_retries=5)
def set_password():
    exc = ConcurrencyError()
    errors.append(exc)
    raise exc

try:
    set_password()
except ConcurrencyError:
    pass
else:
    raise Exception("Shouldn't get here")

assert len(errors) == 5
```

Event store factory

As a convenience, the library function `construct_sqlalchemy_eventstore()` can be used to construct an event store that uses the SQLAlchemy classes.

```
from eventsourcing.infrastructure.sqlalchemy import factory

event_store = factory.construct_sqlalchemy_eventstore(session=datastore.session)
```

By default, the event store is constructed with the `StoredEvent` sequenced item namedtuple, and the active record class `StoredEventRecord`. The optional args `sequenced_item_class` and `active_record_class` can be used to construct different kinds of event store.

Timestamped event store

The examples so far have used an integer sequenced event store, where the items are sequenced by integer version.

The example below constructs an event store for timestamp-sequenced domain events, using the library active record class `TimestampedSequencedItemRecord`.

```
import time
from uuid import uuid4

from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪TimestampSequencedItemRecord

# Setup database table for timestamped sequenced items.
datastore.setup_table(TimestampSequencedItemRecord)

# Construct event store for timestamp sequenced events.
timestamped_event_store = factory.construct_sqlalchemy_eventstore(
    sequenced_item_class=SequencedItem,
    active_record_class=TimestampSequencedItemRecord,
    sequence_id_attr_name='originator_id',
    position_attr_name='timestamp',
    session=datastore.session,
)

# Construct an event.
aggregate_id = uuid4()
event = DomainEvent(
    originator_id=aggregate_id,
    timestamp=time.time(),
)

# Store the event.
timestamped_event_store.append(event)

# Check the event was stored.
events = timestamped_event_store.get_domain_events(aggregate_id)
assert len(events) == 1
assert events[0].originator_id == aggregate_id
assert events[0].timestamp < time.time()
```

Please note, optimistic concurrent control doesn't work to maintain entity consistency, because each event is likely to have a unique timestamp, and so conflicts are very unlikely to arise when concurrent operations appending to the same sequence. For this reason, although domain events can be timestamped, it is not a very good idea to store the events of an entity or aggregate as timestamp-sequenced items. Timestamp-sequenced items are useful for storing events that are logically independent of others, such as messages in a log, things that do not risk causing a consistency error due to concurrent operations.

construct an event store that uses the Apache Cassandra classes.

1.7 Domain model

The library's domain layer has base classes for domain events and entities. These classes show how to write a domain model that uses the library's event sourcing infrastructure. They can also be used to develop an event-sourced application as a domain driven design.

1.7.1 Domain events

The purpose of a domain event is to be published when something happens, normally the results from the work of a command. The library has a base class for domain events called `DomainEvent`.

Domain events can be freely constructed from the `DomainEvent` class. Attributes are set directly from the constructor keyword arguments.

```
from eventsourcing.domain.model.events import DomainEvent

domain_event = DomainEvent(a=1)
assert domain_event.a == 1
```

The attributes of domain events are read-only. New values cannot be assigned to existing objects. Domain events are immutable in that sense.

```
# Fail to set attribute of already-existing domain event.
try:
    domain_event.a = 2
except AttributeError:
    pass
else:
    raise Exception("Shouldn't get here")
```

Domain events can be compared for equality as value objects, instances are equal if they have the same type and the same attributes.

```
DomainEvent(a=1) == DomainEvent(a=1)

DomainEvent(a=1) != DomainEvent(a=2)

DomainEvent(a=1) != DomainEvent(b=1)
```

Publish-subscribe

Domain events can be published, using the library's publish-subscribe mechanism.

The `publish()` function is used to publish events. The `event` arg is required.

```
from eventsourcing.domain.model.events import publish

publish(event=domain_event)
```

The `subscribe()` function is used to subscribe a `handler` that will receive events.

The optional `predicate` arg can be used to provide a function that will decide whether or not the subscribed handler will actually be called when an event is published.

```
from eventsourcing.domain.model.events import subscribe

received_events = []

def receive_event(event):
    received_events.append(event)

def is_domain_event(event):
    return isinstance(event, DomainEvent)
```

```
subscribe(handler=receive_event, predicate=is_domain_event)

# Publish the domain event.
publish(domain_event)

assert len(received_events) == 1
assert received_events[0] == domain_event
```

The `unsubscribe()` function can be used to stop the handler receiving further events.

```
from eventsourcing.domain.model.events import unsubscribe

unsubscribe(handler=receive_event, predicate=is_domain_event)

# Clean up.
del received_events[:]  # received_events.clear()
```

Event library

The library has a small collection of domain event subclasses, such as `EventWithOriginatorID`, `EventWithOriginatorVersion`, `EventWithTimestamp`, `EventWithTimeuuid`, `Created`, `AttributeChanged`, `Discarded`.

Some of these classes provide useful defaults for particular attributes, such as a `timestamp`. Timestamps can be used to sequence events.

```
from eventsourcing.domain.model.events import EventWithTimestamp
from eventsourcing.domain.model.events import EventWithTimeuuid
from uuid import UUID

# Automatic timestamp.
assert isinstance(EventWithTimestamp().timestamp, float)

# Automatic UUIDv1.
assert isinstance(EventWithTimeuuid().event_id, UUID)
```

Some classes require particular arguments when constructed. The `originator_id` can be used to identify a sequence to which an event belongs. The `originator_version` can be used to position the event in a sequence.

```
from eventsourcing.domain.model.events import EventWithOriginatorVersion
from eventsourcing.domain.model.events import EventWithOriginatorID
from uuid import uuid4

# Requires originator_id.
EventWithOriginatorID(originator_id=uuid4())

# Requires originator_version.
EventWithOriginatorVersion(originator_version=0)
```

Some are just useful for their distinct type, for example in subscription predicates.

```
from eventsourcing.domain.model.events import Created, Discarded

def is_created(event):
    return isinstance(event, Created)
```

```python
def is_discarded(event):
    return isinstance(event, Discarded)

assert is_created(Created()) is True
assert is_created(Discarded()) is False
assert is_created(DomainEvent()) is False

assert is_discarded(Created()) is False
assert is_discarded(Discarded()) is True
assert is_discarded(DomainEvent()) is False

assert is_domain_event(Created()) is True
assert is_domain_event(Discarded()) is True
assert is_domain_event(DomainEvent()) is True
```

Custom events

Custom domain events can be coded by subclassing the library's domain event classes.

Domain events are normally named using the past participle of a common verb, for example a regular past participle such as "started", "paused", "stopped", or an irregular past participle such as "chosen", "done", "found", "paid", "quit", "seen".

```python
class SomethingHappened(DomainEvent):
    """
    Published whenever something happens.
    """
```

It is possible to code domain events as inner or nested classes.

```python
class Job(object):

    class Seen(EventWithTimestamp):
        """
        Published when the job is seen.
        """

    class Done(EventWithTimestamp):
        """
        Published when the job is done.
        """
```

Inner or nested classes can be used, and are used in the library, to define the domain events of a domain entity on the entity class itself.

```python
seen = Job.Seen(job_id='#1')
done = Job.Done(job_id='#1')

assert done.timestamp > seen.timestamp
```

1.7.2 Domain entities

A domain entity is an object that is not defined by its attributes, but rather by a thread of continuity and its identity. The attributes of a domain entity can change, directly by assignment, or indirectly by calling a method of the object.

The library provides a domain entity class `VersionedEntity`, which has an `id` attribute, and a `version` attribute.

```
from eventsourcing.domain.model.entity import VersionedEntity

entity_id = uuid4()

entity = VersionedEntity(id=entity_id, version=0)

assert entity.id == entity_id
assert entity.version == 0
```

Entity library

There is a `TimestampedEntity` that has `id` and `created_on` attributes. It also has a `last_modified` attribute which is normally updated as events are applied.

```
from eventsourcing.domain.model.entity import TimestampedEntity

entity_id = uuid4()

entity = TimestampedEntity(id=entity_id, timestamp=123456789)

assert entity.id == entity_id
assert entity.created_on == 123456789
assert entity.last_modified == 123456789
```

There is also a `TimestampedVersionedEntity` that has `id`, `version`, `created_on`, and `last_modified` attributes.

```
from eventsourcing.domain.model.entity import TimestampedVersionedEntity

entity_id = uuid4()

entity = TimestampedVersionedEntity(id=entity_id, version=0, timestamp=123456789)

assert entity.id == entity_id
assert entity.version == 0
assert entity.created_on == 123456789
assert entity.last_modified == 123456789
```

A timestamped, versioned entity is both a timestamped entity and a versioned entity.

```
assert isinstance(entity, TimestampedEntity)
assert isinstance(entity, VersionedEntity)
```

Entity events

The library's domain entities have domain events as inner classes: `Event`, `Created`, `AttributeChanged`, and `Discarded`. These inner event classes are all subclasses of `DomainEvent` and can be freely constructed, with suitable arguments.

```
created = VersionedEntity.Created(
    originator_version=0,
    originator_id=entity_id,
)
```

```
attribute_a_changed = VersionedEntity.AttributeChanged(
    name='a',
    value=1,
    originator_version=1,
    originator_id=entity_id
)

attribute_b_changed = VersionedEntity.AttributeChanged(
    name='b',
    value=2,
    originator_version=2,
    originator_id=entity_id
)

entity_discarded = VersionedEntity.Discarded(
    originator_version=3,
    originator_id=entity_id
)
```

The class `VersionedEntity` has a method `_increment_version()` which can be used, for example by a mutator function, to increment the version number each time an event is applied.

```
entity._increment_version()

assert entity.version == 1
```

Mutator functions

For an application to be event sourced, the state of the application must be mutated by applying domain events.

The entity mutator function `mutate_entity()` can be used to apply a domain event to an entity.

```
from eventsourcing.domain.model.entity import mutate_entity

entity = mutate_entity(entity, attribute_a_changed)

assert entity.a == 1
```

When a versioned entity is updated in this way, the version number is normally incremented.

```
assert entity.version == 2
```

Apply and publish

Events are normally published after they are applied. The method `_apply_and_publish()` can be used to both apply and then publish, so the event mutates the entity and is then received by subscribers.

```
# Apply and publish a domain event.
entity._apply_and_publish(attribute_b_changed)

# Check the event was applied.
assert entity.b == 2
assert entity.version == 3
```

For example, the method `change_attribute()` constructs an `AttributeChanged` event and then calls `_apply_and_publish()`. In the code below, the event is received and checked.

```
entity = VersionedEntity(id=entity_id, version=0)

assert len(received_events) == 0
subscribe(handler=receive_event, predicate=is_domain_event)

# Apply and publish an AttributeChanged event.
entity.change_attribute(name='full_name', value='Mr Boots')

# Check the event was applied.
assert entity.full_name == 'Mr Boots'

# Check the event was published.
assert received_events[0].__class__ == VersionedEntity.AttributeChanged
assert received_events[0].name == 'full_name'
assert received_events[0].value == 'Mr Boots'

# Clean up.
unsubscribe(handler=receive_event, predicate=is_domain_event)
del received_events[:]  # received_events.clear()
```

Discarding entities

The entity method `discard()` can be used to discard the entity, by applying and publishing a `Discarded` event, after which the entity is unavailable for further changes.

```
from eventsourcing.exceptions import EntityIsDiscarded

entity.discard()

# Fail to change an attribute after entity was discarded.
try:
    entity.change_attribute('full_name', 'Mr Boots')
except EntityIsDiscarded:
    pass
else:
    raise Exception("Shouldn't get here")
```

The mutator function will return `None` after mutating an entity with a `Discarded` event.

```
entity = VersionedEntity(id=entity_id, version=3)

entity = mutate_entity(entity, entity_discarded)

assert entity is None
```

That means a sequence of events that ends with a `Discarded` event will result in the same state as an empty sequence of events, when the sequence is replayed by an event player for example.

Custom entities

The library entity classes can be subclassed.

```
from eventsourcing.domain.model.decorators import attribute


class User(VersionedEntity):
    def __init__(self, full_name, *args, **kwargs):
        super(User, self).__init__(*args, **kwargs)
        self.full_name = full_name
```

An entity factory method can construct, apply, and publish the first event of an entity's lifetime. After the event is published, the new entity will be returned by the factory method.

```
def create_user(full_name):
    created_event = User.Created(full_name=full_name, originator_id='1')
    assert created_event.originator_id
    user_entity = mutate_entity(event=created_event, initial=User)
    publish(created_event)
    return user_entity

user = create_user(full_name='Mrs Boots')

assert user.full_name == 'Mrs Boots'
```

Subclasses can extend the entity base classes, by adding event-based properties and methods.

Custom attributes

The library's `@attribute` decorator provides a property getter and setter, which will apply and publish an `AttributeChanged` event when the property is assigned. Simple mutable attributes can be coded as decorated functions without a body, such as the `full_name` function of `User` below.

```
from eventsourcing.domain.model.decorators import attribute


class User(VersionedEntity):

    def __init__(self, full_name, *args, **kwargs):
        super(User, self).__init__(*args, **kwargs)
        self._full_name = full_name

    @attribute
    def full_name(self):
        pass
```

In the code below, after the entity has been created, assigning to the `full_name` attribute causes the entity to be updated, and an `AttributeChanged` event to be published. Both the `Created` and `AttributeChanged` events are received by a subscriber.

```
assert len(received_events) == 0
subscribe(handler=receive_event, predicate=is_domain_event)

# Publish a Created event.
user = create_user('Mrs Boots')
assert user.full_name == 'Mrs Boots'

# Publish an AttributeChanged event.
user.full_name = 'Mr Boots'
```

```python
assert user.full_name == 'Mr Boots'

assert len(received_events) == 2
assert received_events[0].__class__ == VersionedEntity.Created
assert received_events[0].full_name == 'Mrs Boots'

assert received_events[1].__class__ == VersionedEntity.AttributeChanged
assert received_events[1].value == 'Mr Boots'
assert received_events[1].name == '_full_name'

# Clean up.
unsubscribe(handler=receive_event, predicate=is_domain_event)
del received_events[:]  # received_events.clear()
```

Custom commands

The entity base classes can also be extended by adding "command" methods that publish events. In general, the arguments of a command will be used to perform some work. Then, the result of the work will be used to construct a domain event that represents what happened. And then, the domain event will be applied and published.

Methods like this, for example the `set_password()` method of the `User` entity below, normally have no return value. The method creates an encoded string from a raw password, and then uses the `change_attribute()` method to apply and publish an `AttributeChanged` event for the `_password` attribute with the encoded password.

```python
from eventsourcing.domain.model.decorators import attribute


class User(VersionedEntity):

    def __init__(self, *args, **kwargs):
        super(User, self).__init__(*args, **kwargs)
        self._password = None

    def set_password(self, raw_password):
        # Do some work using the arguments of a command.
        password = self._encode_password(raw_password)

        # Construct, apply, and publish an event.
        self.change_attribute('_password', password)

    def check_password(self, raw_password):
        password = self._encode_password(raw_password)
        return self._password == password

    def _encode_password(self, password):
        return ''.join(reversed(password))


user = User(id='1')

user.set_password('password')
assert user.check_password('password')
```

A custom entity can also have custom methods that publish custom events. In the example below, a method `make_it_so()` publishes a domain event called `SomethingHappened`.

Custom mutator

To be applied to an entity, custom event classes must be supported by a custom mutator function. In the code below, the `mutate_world()` mutator function extends the library's `mutate_entity` function to support the event `SomethingHappened`. The `_mutate()` function of `DomainEntity` has been overridden so that `mutate_world()` will be called when events are applied.

```
from eventsourcing.domain.model.decorators import mutator

class World(VersionedEntity):

    def __init__(self, *args, **kwargs):
        super(World, self).__init__(*args, **kwargs)
        self.history = []

    class SomethingHappened(VersionedEntity.Event):
        """Published when something happens in the world."""

    def make_it_so(self, something):
        # Do some work using the arguments of a command.
        what_happened = something

        # Construct an event with the results of the work.
        event = World.SomethingHappened(
            what=what_happened,
            originator_id=self.id,
            originator_version=self.version
        )

        # Apply and publish the event.
        self._apply_and_publish(event)

    @classmethod
    def _mutate(cls, initial, event):
        return mutate_world(event=event, initial=initial)


@mutator
def mutate_world(initial, event):
    return mutate_entity(initial, event)

@mutate_world.register(World.SomethingHappened)
def _(self, event):
    self.history.append(event)
    self._increment_version()
    return self


world = World(id='1')
world.make_it_so('dinosaurs')
world.make_it_so('trucks')
world.make_it_so('internet')

assert world.history[0].what == 'dinosaurs'
assert world.history[1].what == 'trucks'
assert world.history[2].what == 'internet'
```

Reflexive mutator

The `WithReflexiveMutator` class tries to call a function called `mutate()` on the event class itself. This means each event class can define how an entity is mutated by it.

A custom base entity class, for example `Entity` in the code below, may help to adopt this style across all entity classes in an application.

```python
from eventsourcing.domain.model.entity import WithReflexiveMutator


class Entity(WithReflexiveMutator, VersionedEntity):
    """
    Custom base class for domain entities in this example.
    """

class World(Entity):
    """
    Example domain entity, with mutator function on domain event.
    """
    def __init__(self, *args, **kwargs):
        super(World, self).__init__(*args, **kwargs)
        self.history = []

    def make_it_so(self, something):
        what_happened = something
        event = World.SomethingHappened(
            what=what_happened,
            originator_id=self.id,
            originator_version=self.version
        )
        self._apply_and_publish(event)

    class SomethingHappened(VersionedEntity.Event):
        # Define mutator function for entity on the event class.
        def mutate(self, entity):
            entity.history.append(self)
            entity._increment_version()


world = World(id='1')
world.make_it_so('dinosaurs')
world.make_it_so('trucks')
world.make_it_so('internet')

assert world.history[0].what == 'dinosaurs'
assert world.history[1].what == 'trucks'
assert world.history[2].what == 'internet'
```

1.7.3 Aggregate root

The library has a domain entity class called `AggregateRoot` that can be useful in a domain driven design, where a command can cause many events to be published. The `AggregateRoot` class has a `save()` method, which publishes a list of pending events, and overrides the `_publish()` method of the base class to append events to a pending list.

The `AggregateRoot` class inherits from both `TimestampedVersionedEntity` and

WithReflexiveMutator, and can be subclassed to define custom aggregate root entities.

```
from eventsourcing.domain.model.aggregate import AggregateRoot


class World(AggregateRoot):
    """
    Example domain entity, with mutator function on domain event.
    """
    def __init__(self, *args, **kwargs):
        super(World, self).__init__(*args, **kwargs)
        self.history = []

    def make_things_so(self, *somethings):
        for something in somethings:
            self._trigger(World.SomethingHappened, what=something)

    class SomethingHappened(VersionedEntity.Event):
        def mutate(self, entity):
            entity.history.append(self)
            entity._increment_version()


# World factory.
def create_new_world():
    created = World.Created(originator_id=1)
    world = World._mutate(event=created)
    world._publish(created)
    return world
```

An AggregateRoot entity will postpone the publishing of all events, pending the next call to its save() method.

```
assert len(received_events) == 0
subscribe(handler=receive_event)

# Create new entity.
world = create_new_world()
assert isinstance(world, World)

# Command that publishes many events.
world.make_things_so('dinosaurs', 'trucks', 'internet')

assert world.history[0].what == 'dinosaurs'
assert world.history[1].what == 'trucks'
assert world.history[2].what == 'internet'
```

When the save() method is called, all such pending events are published as a single list of events to the publish-subscribe mechanism.

```
# Events are pending actual publishing until the save() method is called.
assert len(received_events) == 0
world.save()

# Pending events were published as a single list of events.
assert len(received_events) == 1
assert len(received_events[0]) == 4
```

Publishing all events from a single command in a single list allows all the events to be written to a database as a single atomic operation.

That avoids the risk that some events will be stored successfully but other events from the same command will fall into conflict and be lost, because another thread has operated on the same aggregate at the same time, causing an inconsistent state that would also be difficult to repair.

It also avoids the risk of other threads picking up only some events caused by a command, presenting the aggregate in an inconsistent or unusual and perhaps unworkable state.

```
unsubscribe(handler=receive_event)
del received_events[:]  # received_events.clear()
```

1.8 Applications

The application layer combines objects from the domain and infrastructure layers.

1.8.1 Repositories and policies

An application object can have repositories, so that aggregates can be retrieved by ID using a dictionary-like interface. In general, aggregates implement commands that publish events.

An application object can also have policies. In general, policies receive events and execute commands.

1.8.2 Application services

An application object can have methods ("application services") which provide a relatively simple interface for clients operations, hiding the complexity and usage of the application's domain and infrastructure layers.

Application services can be developed outside-in, with a test- or behaviour-driven development approach. A test suite can be imagined as an interface that uses the application. Interfaces are outside the scope of the application layer.

1.8.3 Event sourced application

The example code below shows an event sourced application object class. It constructs an event store that uses the library's infrastructure with SQLAlchemy, using library function `construct_sqlalchemy_eventstore()`.

```
from uuid import uuid4

from eventsourcing.application.policies import PersistencePolicy
from eventsourcing.domain.model.aggregate import AggregateRoot
from eventsourcing.infrastructure.eventsourcedrepository import EventSourcedRepository
from eventsourcing.infrastructure.sqlalchemy.factory import construct_sqlalchemy_
↪eventstore


class Application(object):
    def __init__(self, session):
        # Construct event store.
        self.event_store = construct_sqlalchemy_eventstore(
            session=session
        )
        # Construct an event sourced repository.
        self.repository = EventSourcedRepository(
            event_store=self.event_store,
            mutator=CustomAggregate._mutate
        )
        # Construct a persistence policy.
        self.persistence_policy = PersistencePolicy(
            event_store=self.event_store
        )

    def create_aggregate(self, a):
        aggregate_id = uuid4()
        domain_event = CustomAggregate.Created(a=1, originator_id=aggregate_id)
        entity = CustomAggregate._mutate(event=domain_event)
        entity._publish(domain_event)  # Pending save().
        return entity

    def close(self):
        self.persistence_policy.close()
```

The application has a domain model with one domain entity called `CustomAggregate`, defined below. The entity has one attribute, called `a`. It is a subclass of the library's `AggregateRoot` entity class.

Repository

The application has an event sourced repository for `CustomAggregate` instances. It uses the library class `EventSourceRepository`, which uses an event store to get domain events for an aggregate, and the mutator function from the `CustomAggregate` class which it uses to reconstruct an aggregate instance from the events. An application needs one such repository for each type of aggregate in the application's domain model.

Policy

The application object class has a persistence policy. It uses the library class `PersistencePolicy`. The persistence policy appends domain events to an event store whenever they are published.

Aggregate factory

The application also has an application service called `create_aggregate()` which can be used to create new `CustomAggregate` instances. The `CustomAggregate` is a very simple aggregate, which has an event sourced attribute called `a`. To create such an aggregate, a value for `a` must be provided.

```
from eventsourcing.domain.model.decorators import attribute


class CustomAggregate(AggregateRoot):
    def __init__(self, a, **kwargs):
        super(CustomAggregate, self).__init__(**kwargs)
        self._a = a

    @attribute
    def a(self):
        """
        Event sourced attribute 'a'.
        """
```

Database

The library classes `SQLAlchemyDatastore` and `SQLAlchemySettings` can be used to setup a database.

```
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemyDatastore,␣
↪SQLAlchemySettings
from eventsourcing.infrastructure.sqlalchemy.activerecords import StoredEventRecord

# Define database settings.
settings = SQLAlchemySettings(uri='sqlite:///:memory:')

# Setup connection to database.
datastore = SQLAlchemyDatastore(settings=settings)
datastore.setup_connection()

# Setup table in database.
# - done only once
datastore.setup_table(StoredEventRecord)
```

Run the code

After setting up the database connection, the application can be constructed with the session object.

```
# Construct application with session.
app = Application(session=datastore.session)
```

Finally, a new aggregate instance can be created with the application service `create_aggregate()`.

```
# Create aggregate using application service.
aggregate = app.create_aggregate(a=1)

# Don't forget to save!
aggregate.save()

# Aggregate is in the repository.
assert aggregate.id in app.repository

# Remember the aggregate's ID.
aggregate_id = aggregate.id

# Forget the aggregate (will still saved be in the database).
del(aggregate)
```

An existing aggregate can be recovered by ID using the dictionary-like interface of the aggregate repository.

```
# Get aggregate using dictionary-like interface.
aggregate = app.repository[aggregate_id]

assert aggregate.a == 1
```

Changes to the aggregate's attribute `a` are visible in the repository, but only after the aggregate has been saved.

```
aggregate.a = 2
aggregate.a = 3

# Don't forget to save!
aggregate.save()
```

```python
del(aggregate)

aggregate = app.repository[aggregate_id]

assert aggregate.a == 3
```

The aggregate can be discarded. After being saved, a discarded aggregate will not be available in the repository.

```python
aggregate.discard()

# Don't forget to save!
aggregate.save()

# Discarded aggregate no longer in repository.
assert aggregate_id not in app.repository

# Fail to get aggregate from dictionary-like interface.
try:
    app.repository[aggregate_id]
except KeyError:
    pass
else:
    raise Excpetion("Shouldn't get here.")
```

Application events

It is always possible to get the domain events for an aggregate, using the application's event store method `get_domain_events()`.

```python
events = app.event_store.get_domain_events(originator_id=aggregate_id)
assert len(events) == 4

assert events[0].originator_id == aggregate_id
assert isinstance(events[0], CustomAggregate.Created)
assert events[0].a == 1

assert events[1].originator_id == aggregate_id
assert isinstance(events[1], CustomAggregate.AttributeChanged)
assert events[1].name == '_a'
assert events[1].value == 2

assert events[2].originator_id == aggregate_id
assert isinstance(events[2], CustomAggregate.AttributeChanged)
assert events[2].name == '_a'
assert events[2].value == 3

assert events[3].originator_id == aggregate_id
assert isinstance(events[3], CustomAggregate.Discarded)
```

Sequenced items

It is also possible to get the sequenced item namedtuples for an aggregate, using the application's event store's active record strategy method `get_items()`.

```
items = app.event_store.active_record_strategy.get_items(aggregate_id)
assert len(items) == 4

assert items[0].originator_id == aggregate_id
assert items[0].event_type == 'eventsourcing.domain.model.aggregate#AggregateRoot.
↪Created'
assert items[0].state.startswith('{"a":1,"timestamp":')

assert items[1].originator_id == aggregate_id
assert items[1].event_type == 'eventsourcing.domain.model.aggregate#AggregateRoot.
↪AttributeChanged'
assert items[1].state.startswith('{"name":"_a",')

assert items[2].originator_id == aggregate_id
assert items[2].event_type == 'eventsourcing.domain.model.aggregate#AggregateRoot.
↪AttributeChanged'
assert items[2].state.startswith('{"name":"_a",')

assert items[3].originator_id == aggregate_id
assert items[3].event_type == 'eventsourcing.domain.model.aggregate#AggregateRoot.
↪Discarded'
assert items[3].state.startswith('{"timestamp":')
```

Close

It is useful to unsubscribe any handlers subscribed by the policies (avoids dangling handlers being called inappropriately, if the process isn't going to terminate immediately).

```
# Clean up.
app.close()
```

Todo: Something about the library's application class?

Todo: Something about using uuid5 to make UUIDs from things like email addresses.

Todo: Something about using application log to get a sequence of all events.

Todo: Something about using a policy to update views from published events.

Todo: Something about using a policy to update a register of existant IDs from published events.

Todo: Something about having a worker application, that has policies that process events received by a worker.

Todo: Something about having a policy to publish events to worker applications.

Todo: Something like a message queue strategy strategy.

Todo: Something about publishing events to a message queue.

Todo: Something about receiving events in a message queue worker.

Todo: Something about publishing events to a message queue.

Todo: Something about receiving events in a message queue worker.

1.9 Examples

These examples show how to write an event sourced application, with and without classes in this library.

In the first section, a stand-alone event sourced domain model is developed which has no dependencies on the library, along with an application object that has minimal dependencies on library infrastructure classes for storing events. In later sections, more use is made of library classes, in order to demonstrate the other capabilities of the library.

All the examples in this guide follow the layered architecture: application, domain, infrastructure. To create working programs, simply copy all the code snippets from a section into a Python file.

Please feel free to experiment by making variations. The code snippets in this guide are covered by a test case, so please expect everything to work as presented - raise an issue if something goes wrong.

1.9.1 Example application

Install the library with the 'sqlalchemy' option.

```
pip install eventsourcing[sqlalchemy]
```

In this section, an event sourced application is developed that has minimal dependencies on the library.

A stand-alone domain model is developed without library classes, which shows how event sourcing in Python can work. The stand-alone code examples here are simplified versions of the library classes. Infrastructure classes from the library are used explicitly to show the different components involved, so you can understand how to make variations.

Domain

Let's start with the domain model. If the state of an event sourced application is determined by a sequence of events, then we need to define some events.

Domain events

You may wish to use a technique such as “event storming” to identify or decide what happens in your domain. In this example, for the sake of general familiarity let’s assume we have a domain in which things can be “created”, “changed”, and “discarded”. With that in mind, we can begin to write some domain event classes.

In the example below, there are three domain event classes: `Created`, `AttributeChanged`, and `Discarded`. The common aspects of the domain event classes have been pulled up to a layer supertype `DomainEvent`.

```
import time


class DomainEvent(object):
    """
    Layer supertype.
    """
    def __init__(self, originator_id, originator_version, **kwargs):
        self.originator_id = originator_id
        self.originator_version = originator_version
        self.__dict__.update(kwargs)


class Created(DomainEvent):
    """
    Published when an entity is created.
    """
    def __init__(self, **kwargs):
        super(Created, self).__init__(originator_version=0, **kwargs)


class AttributeChanged(DomainEvent):
    """
    Published when an attribute value is changed.
    """
    def __init__(self, name, value, **kwargs):
        super(AttributeChanged, self).__init__(**kwargs)
        self.name = name
        self.value = value


class Discarded(DomainEvent):
    """
    Published when an entity is discarded.
    """
```

Please note, the domain event classes above do not depend on the library. The library does however contain a collection of different kinds of domain event classes that you can use in your models, for example see *`Created`*, *`AttributeChanged`*, and *`Discarded`*.

Publish-subscribe

Since we are dealing with events, let’s define a simple publish-subscribe mechanism for them.

```
subscribers = []

def publish(event):
    for subscriber in subscribers:
        subscriber(event)


def subscribe(subscriber):
    subscribers.append(subscriber)


def unsubscribe(subscriber):
    subscribers.remove(subscriber)
```

Domain entity

Now, let's define a domain entity that publishes the event classes defined above.

The entity class `Example` below has an ID and a version number. It also has a property `foo` with a "setter" method, and a method `discard()` to use when the entity is no longer needed.

The entity methods follow a similar pattern. At some point, each constructs an event that represents the result of the operation. Then each uses a "mutator function" `mutate()` (see below) to apply the event to the entity. Finally, each publishes the event for the benefit of any subscribers, by using the function `publish()`.

```
import uuid


class Example(object):
    """
    Example domain entity.
    """
    def __init__(self, originator_id, originator_version=0, foo=''):
        self._id = originator_id
        self._version = originator_version
        self._is_discarded = False
        self._foo = foo

    @property
    def id(self):
        return self._id

    @property
    def version(self):
        return self._version

    @property
    def foo(self):
        return self._foo

    @foo.setter
    def foo(self, value):
        assert not self._is_discarded

        # Construct an 'AttributeChanged' event object.
        event = AttributeChanged(
            originator_id=self.id,
            originator_version=self.version,
            name='foo',
            value=value,
        )
        # Apply the event to self.
        mutate(self, event)

        # Publish the event for others.
        publish(event)

    def discard(self):
        assert not self._is_discarded

        # Construct a 'Discarded' event object.
        event = Discarded(
            originator_id=self.id,
            originator_version=self.version
        )

        # Apply the event to self.
        mutate(self, event)

        # Publish the event for others.
        publish(event)
```

A factory can be used to create new "example" entities. The function `create_new_example()` below works in a similar way to the entity methods, creating new entities by firstly constructing a `Created` event, then using the function `mutate()` (see below) to construct the entity object, and finally publishing the event for others before returning the new entity object to the caller.

```
def create_new_example(foo):
    """
    Factory for Example entities.
    """
    # Construct an entity ID.
    entity_id = uuid.uuid4()

    # Construct a 'Created' event object.
    event = Created(
        originator_id=entity_id,
        foo=foo
    )

    # Use the mutator function to construct the entity object.
    entity = mutate(None, event)

    # Publish the event for others.
    publish(event=event)

    # Return the new entity.
    return entity
```

The example entity class does not depend on the library. In particular, it doesn't inherit from a "magical" entity base class that makes everything work. The example here just publishes events that it has applied to itself. The library does however contain domain entity classes that you can use to build your domain model, for example the class *AggregateRoot*. The library classes are more developed than the examples here.

Mutator function

The mutator function `mutate()` below handles `Created` events by constructing an object. It handles `AttributeChanged` events by setting an attribute value, and it handles `Discarded` events by marking the entity as discarded. Each handler increases the version of the entity, so that the version of the entity is always one plus the the originator version of the last event that was applied.

When replaying a sequence of events, for example when reconstructing an entity from its domain events, the mutator function is called many times in order to apply each event in the sequence to an evolving initial state.

```
def mutate(entity, event):
    """
    Mutator function for Example entities.
    """
    # Handle "created" events by constructing the entity object.
    if isinstance(event, Created):
        entity = Example(**event.__dict__)
        entity._version += 1
        return entity

    # Handle "value changed" events by setting the named value.
    elif isinstance(event, AttributeChanged):
        assert not entity._is_discarded
        setattr(entity, '_' + event.name, event.value)
        entity._version += 1
        return entity

    # Handle "discarded" events by returning 'None'.
    elif isinstance(event, Discarded):
        assert not entity._is_discarded
        entity._version += 1
        entity._is_discarded = True
        return None
    else:
        raise NotImplementedError(type(event))
```

For the sake of simplicity in this example, we'll use an if-else block to structure the mutator function. The library has a function decorator *mutator()* that allows handlers for different types of event to be registered with a default mutator function, just like singledispatch.

Run the code

Let's firstly subscribe to receive the events that will be published, so we can see what happened.

```
# A list of received events.
received_events = []

# Subscribe to receive published events.
subscribe(lambda e: received_events.append(e))
```

With this stand-alone code, we can create a new example entity object. We can update its property `foo`, and we can discard the entity using the `discard()` method.

```
# Create a new entity using the factory.
entity = create_new_example(foo='bar')
```

```
# Check the entity has an ID.
assert entity.id

# Check the entity has a version number.
assert entity.version == 1

# Check the received events.
assert len(received_events) == 1, received_events
assert isinstance(received_events[0], Created)
assert received_events[0].originator_id == entity.id
assert received_events[0].originator_version == 0
assert received_events[0].foo == 'bar'

# Check the value of property 'foo'.
assert entity.foo == 'bar'

# Update property 'foo'.
entity.foo = 'baz'

# Check the new value of 'foo'.
assert entity.foo == 'baz'

# Check the version number has increased.
assert entity.version == 2

# Check the received events.
assert len(received_events) == 2, received_events
assert isinstance(received_events[1], AttributeChanged)
assert received_events[1].originator_version == 1
assert received_events[1].name == 'foo'
assert received_events[1].value == 'baz'
```

Infrastructure

Since the application state is determined by a sequence of events, the application must somehow be able both to persist the events, and then recover the entities.

Please note, storing and replaying events to persist and to reconstruct the state of an application is the primary capability of this library. The domain and application and interface capabilities are offered as a supplement to the infrastructural capabilities, and have been added to the library partly as a way of shaping and validating the infrastructure, partly to demonstrate how the core capabilities may be applied, but also as a convenient way of reusing foundational code so that attention can remain on the problem domain (framework).

Database table

Let's start by setting up a simple database table that can store sequences of items. We can use SQLAlchemy directly to define a database table that stores items in sequences, with a single identity for each sequence, and with each item positioned in its sequence by an integer index number.

```
from sqlalchemy.ext.declarative.api import declarative_base
from sqlalchemy.sql.schema import Column, Sequence, Index
from sqlalchemy.sql.sqltypes import BigInteger, Integer, String, Text
from sqlalchemy_utils import UUIDType

ActiveRecord = declarative_base()
```

```
class SequencedItemRecord(ActiveRecord):
    __tablename__ = 'sequenced_items'

    # Sequence ID (e.g. an entity or aggregate ID).
    sequence_id = Column(UUIDType(), primary_key=True)

    # Position (index) of item in sequence.
    position = Column(BigInteger(), primary_key=True)

    # Topic of the item (e.g. path to domain event class).
    topic = Column(String(255))

    # State of the item (serialized dict, possibly encrypted).
    data = Column(Text())

    __table_args__ = Index('index', 'sequence_id', 'position'),
```

The library has a class `IntegerSequencedItemRecord` which is very similar to the above.

Next, create the database table. For convenience, the SQLAlchemy objects can be adapted with the class `SQLAlchemyDatastore`, which provides a simple interface for the two operations we require: `setup_connection()` and `setup_tables()`.

```
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemySettings,␣
↪SQLAlchemyDatastore

datastore = SQLAlchemyDatastore(
    base=ActiveRecord,
    settings=SQLAlchemySettings(uri='sqlite:///:memory:'),
    tables=(SequencedItemRecord,),
)

datastore.setup_connection()
datastore.setup_tables()
```

As you can see from the `uri` argument above, this example is using SQLite to manage an in memory relational database. You can change `uri` to any valid connection string. Here are some example connection strings: for an SQLite file; for a PostgreSQL database; and for a MySQL database. See SQLAlchemy's create_engine() documentation for details. You may need to install drivers for your database management system.

```
sqlite:////tmp/mydatabase

postgresql://scott:tiger@localhost:5432/mydatabase

mysql://scott:tiger@hostname/dbname
```

Similar to the support for storing events in SQLAlchemy, there are classes in the library for Cassandra. The project djangoevents has support for storing events with this library using the Django ORM. Support for other databases such as DynamoDB is forthcoming.

Event store

To support different kinds of sequences in the domain model, and to allow for different database schemas, the library has an event store class *`EventStore`* that uses a "sequenced item mapper" for mapping domain events to "sequenced items" - this library's archetype persistence model for storing events. The sequenced item mapper derives the values of sequenced item fields from the attributes of domain events.

The event store then uses an "active record strategy" to persist the sequenced items into a particular database management system. The active record strategy uses an active record class to manipulate records in a particular database table.

Hence you can use a different database table by substituting an alternative active record class. You can use a different database management system by substituting an alternative active record strategy.

```
from eventsourcing.infrastructure.eventstore import EventStore
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy
from eventsourcing.infrastructure.sequenceditemmapper import SequencedItemMapper

active_record_strategy = SQLAlchemyActiveRecordStrategy(
    session=datastore.session,
    active_record_class=SequencedItemRecord,
)

sequenced_item_mapper = SequencedItemMapper(
    sequence_id_attr_name='originator_id',
    position_attr_name='originator_version'
)

event_store = EventStore(
    active_record_strategy=active_record_strategy,
    sequenced_item_mapper=sequenced_item_mapper
)
```

In the code above, the `sequence_id_attr_name` value given to the sequenced item mapper is the name of the domain events attribute that will be used as the ID of the mapped sequenced item, The `position_attr_name` argument informs the sequenced item mapper which event attribute should be used to position the item in the sequence. The values `originator_id` and `originator_version` correspond to attributes of the domain event classes we defined in the domain model section above.

Entity repository

It is common to retrieve entities from a repository. An event sourced repository for the `example` entity class can be constructed directly using library class *EventSourcedRepository*. The repository is given the mutator function `mutate()` and the event store.

```
from eventsourcing.infrastructure.eventsourcedrepository import EventSourcedRepository

example_repository = EventSourcedRepository(
    event_store=event_store,
    mutator=mutate
)
```

Run the code

Now, let's firstly write the events we received earlier into the event store.

```
# Put each received event into the event store.
for event in received_events:
    event_store.append(event)

# Check the events exist in the event store.
stored_events = event_store.get_domain_events(entity.id)
assert len(stored_events) == 2, (received_events, stored_events)
```

The entity can now be retrieved from the repository, using its dictionary-like interface.

```
retrieved_entity = example_repository[entity.id]
assert retrieved_entity.foo == 'baz'
```

Sequenced items

Remember that we can always get the sequenced items directly from the active record strategy. A sequenced item is tuple containing a serialised representation of the domain event. The library class `SequencedItem` is a Python namedtuple with four fields: `sequence_id`, `position`, `topic`, and `data`.

In this example, an event's `originator_id` attribute is mapped to the `sequence_id` field, and the event's `originator_version` attribute is mapped to the `position` field. The `topic` field of a sequenced item is used to identify the event class, and the `data` field represents the state of the event (normally a JSON string).

```
sequenced_items = event_store.active_record_strategy.get_items(entity.id)

assert len(sequenced_items) == 2

assert sequenced_items[0].sequence_id == entity.id
assert sequenced_items[0].position == 0
assert 'Created' in sequenced_items[0].topic
assert 'bar' in sequenced_items[0].data

assert sequenced_items[1].sequence_id == entity.id
assert sequenced_items[1].position == 1
assert 'AttributeChanged' in sequenced_items[1].topic
assert 'baz' in sequenced_items[1].data
```

These are just default names. If it matters in your context that the persistence model uses other names, then you can use a different sequenced item type which either extends or replaces the fields above.

Application

Although we can do everything at the module level, an application object brings it all together. In the example below, the class `ExampleApplication` has an event store, and an entity repository. The application also has a persistence policy.

Persistence policy

The persistence policy below subscribes to receive events whenever they are published. It uses an event store to store events whenever they are received.

```
class PersistencePolicy(object):
    def __init__(self, event_store):
        self.event_store = event_store
        subscribe(self.store_event)
    def close(self):
        unsubscribe(self.store_event)

    def store_event(self, event):
        self.event_store.append(event)
```

A slightly more developed class *PersistencePolicy* is included in the library.

Application object

As a convenience, it is useful to make the application function as a Python context manager, so that the application can close the persistence policy, and unsubscribe from receiving further domain events.

```
class ExampleApplication(object):
    def __init__(self, session):
        # Construct event store.
        self.event_store = EventStore(
            active_record_strategy=SQLAlchemyActiveRecordStrategy(
                active_record_class=SequencedItemRecord,
                session=session,
            ),
            sequenced_item_mapper=SequencedItemMapper(
                sequence_id_attr_name='originator_id',
                position_attr_name='originator_version'
            )
        )
        # Construct persistence policy.
        self.persistence_policy = PersistencePolicy(
            event_store=self.event_store
        )
        # Construct example repository.
        self.example_repository = EventSourcedRepository(
            event_store=self.event_store,
            mutator=mutate
        )

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.persistence_policy.close()
```

A more developed class *ExampleApplication* can be found in the library. It is used in later sections of this guide.

Run the code

With the application object, we can create more example entities and expect they will be available immediately in the repository.

Please note, an entity that has been discarded by using its `discard()` method cannot subsequently be retrieved from the repository using its ID. In particular, the repository's dictionary-like interface will raise a Python `KeyError` exception instead of returning an entity.

```
with ExampleApplication(datastore.session) as app:

    # Create a new entity.
    example = create_new_example(foo='bar')

    # Read.
    assert example.id in app.example_repository
    assert app.example_repository[example.id].foo == 'bar'

    # Update.
    example.foo = 'baz'
    assert app.example_repository[example.id].foo == 'baz'

    # Delete.
    example.discard()
    assert example.id not in app.example_repository
```

Congratulations. You have created yourself an event sourced application.

1.9.2 Snapshotting

To enable snapshots to be used when recovering an entity from a repository, construct an entity repository that has a snapshot strategy object (see below). It is recommended to store snapshots in a dedicated table.

To automatically generate snapshots, you could perhaps define a snapshotting policy, to take snapshots whenever a particular condition occurs.

Domain

To avoid duplicating code from the previous section, let's use the example entity class `Example` and its factory function `create_new_example()` from the library.

```
from eventsourcing.example.domainmodel import Example, create_new_example
```

Infrastructure

It is recommended not to store snapshots within the entity's sequence of events, but in a dedicated table for snapshots. So let's setup a dedicated table for snapshots using the library class `SnapshotRecord`, as well as a table for the events of the entity.

```
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪IntegerSequencedItemRecord, SnapshotRecord
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemyDatastore,␣
↪SQLAlchemySettings


datastore = SQLAlchemyDatastore(
    settings=SQLAlchemySettings(uri='sqlite:///:memory:'),
    tables=(IntegerSequencedItemRecord, SnapshotRecord,),
)

datastore.setup_connection()
datastore.setup_tables()
```

Application

Policy

Now let's define a snapshotting policy object, so that snapshots of example entities are taken every so many events.

The class `ExampleSnapshottingPolicy` below will take a snapshot of the example entities every `period` number of events, so that there will never be more than `period` number of events to replay when recovering the entity. The default value of `2` is effective in the example below.

```
from eventsourcing.domain.model.events import subscribe, unsubscribe


class ExampleSnapshottingPolicy(object):
    def __init__(self, example_repository, period=2):
        self.example_repository = example_repository
        self.period = period
        subscribe(predicate=self.trigger, handler=self.take_snapshot)

    def close(self):
        unsubscribe(predicate=self.trigger, handler=self.take_snapshot)

    def trigger(self, event):
        return isinstance(event, Example.Event) and not (event.originator_version +␣
↪1) % self.period

    def take_snapshot(self, event):
        self.example_repository.take_snapshot(event.originator_id, lte=event.
↪originator_version)
```

Because the event's `originator_version` is passed to the method `take_snapshot()`, with the argument `lte`, the snapshot will reflect the entity as it existed just after the event was applied. Even if a different thread operates on the same entity before the snapshot is taken, the resulting snapshot is the same as it would have been otherwise.

Application object

The application class below extends the library class *`ApplicationWithPersistencePolicies`*, which constructs the event stores and persistence policies we need. The supertype has a policy to persist snapshots whenever they are taken. It also has as a policy to persist the events of entities whenever they are published.

The example entity repository is constructed from library class *`EventSourcedRepository`* with a snapshot strategy, the integer sequenced event store, and a mutator function. The snapshot strategy is constructed from library class *`EventSourcedSnapshotStrategy`* with an event store for snapshots that is provided by the supertype.

The application's snapshotting policy is constructed with the example repository, which it needs in order to take snapshots.

```
from eventsourcing.application.base import ApplicationWithPersistencePolicies
from eventsourcing.infrastructure.eventsourcedrepository import EventSourcedRepository
from eventsourcing.infrastructure.snapshotting import EventSourcedSnapshotStrategy
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy


class SnapshottedApplication(ApplicationWithPersistencePolicies):

    def __init__(self, session):
        # Construct event stores and persistence policies.
        entity_active_record_strategy = SQLAlchemyActiveRecordStrategy(
            active_record_class=IntegerSequencedItemRecord,
            session=session,
        )
        snapshot_active_record_strategy = SQLAlchemyActiveRecordStrategy(
            active_record_class=SnapshotRecord,
            session=session,
        )
        super(SnapshottedApplication, self).__init__(
            entity_active_record_strategy=entity_active_record_strategy,
            snapshot_active_record_strategy=snapshot_active_record_strategy,
        )

        # Construct snapshot strategy.
        self.snapshot_strategy = EventSourcedSnapshotStrategy(
            event_store=self.snapshot_event_store
        )

        # Construct the entity repository, this time with the snapshot strategy.
        self.example_repository = EventSourcedRepository(
            event_store=self.entity_event_store,
            mutator=Example._mutate,
            snapshot_strategy=self.snapshot_strategy
        )

        # Construct the snapshotting policy.
        self.snapshotting_policy = ExampleSnapshottingPolicy(
            example_repository=self.example_repository,
        )

    def create_new_example(self, foo):
        return create_new_example(foo=foo)

    def close(self):
        super(SnapshottedApplication, self).close()
        self.snapshotting_policy.close()
```

Run the code

The application object can be used in the same way as before. Now snapshots of an example entity will be taken every second event.

```
with SnapshottedApplication(datastore.session) as app:

    # Create an entity.
    entity = app.create_new_example(foo='bar1')

    # Check there's no snapshot, only one event so far.
    snapshot = app.snapshot_strategy.get_snapshot(entity.id)
    assert snapshot is None

    # Change an attribute, generates a second event.
    entity.foo = 'bar2'

    # Check the snapshot.
    snapshot = app.snapshot_strategy.get_snapshot(entity.id)
    assert snapshot.state['_foo'] == 'bar2'

    # Check can recover entity using snapshot.
    assert entity.id in app.example_repository
    assert app.example_repository[entity.id].foo == 'bar2'

    # Check snapshot after five events.
    entity.foo = 'bar3'
    entity.foo = 'bar4'
    entity.foo = 'bar5'
    snapshot = app.snapshot_strategy.get_snapshot(entity.id)
    assert snapshot.state['_foo'] == 'bar4'

    # Check snapshot after seven events.
    entity.foo = 'bar6'
    entity.foo = 'bar7'
    assert app.example_repository[entity.id].foo == 'bar7'
    snapshot = app.snapshot_strategy.get_snapshot(entity.id)
    assert snapshot.state['_foo'] == 'bar6'

    # Check snapshot state is None after discarding the entity on the eighth event.
    entity.discard()
    assert entity.id not in app.example_repository
    snapshot = app.snapshot_strategy.get_snapshot(entity.id)
    assert snapshot.state is None

    try:
        app.example_repository[entity.id]
    except KeyError:
        pass
    else:
        raise Exception('KeyError was not raised')

    # Get historical snapshots.
    snapshot = app.snapshot_strategy.get_snapshot(entity.id, lte=2)
    assert snapshot.state['_version'] == 2  # one behind
    assert snapshot.state['_foo'] == 'bar2'

    snapshot = app.snapshot_strategy.get_snapshot(entity.id, lte=3)
    assert snapshot.state['_version'] == 4
    assert snapshot.state['_foo'] == 'bar4'

    # Get historical entities.
    entity = app.example_repository.get_entity(entity.id, lte=0)
    assert entity.version == 1
    assert entity.foo == 'bar1', entity.foo

    entity = app.example_repository.get_entity(entity.id, lte=1)
    assert entity.version == 2
    assert entity.foo == 'bar2', entity.foo

    entity = app.example_repository.get_entity(entity.id, lte=2)
    assert entity.version == 3
    assert entity.foo == 'bar3', entity.foo

    entity = app.example_repository.get_entity(entity.id, lte=3)
    assert entity.version == 4
    assert entity.foo == 'bar4', entity.foo
```

1.9.3 Aggregates in DDD

Eric Evans' book Domain Driven Design describes an abstraction called "aggregate":

> *"An aggregate is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each aggregate has a root and a boundary."*

Therefore,

> *"Cluster the entities and value objects into aggregates and define boundaries around each. Choose one entity to be the root of each aggregate, and control all access to the objects inside the boundary through the root. Allow external objects to hold references to the root only."*

Which seems to suggest an event sourced aggregate must have a set of events and a mutator function that pertain to a cluster of objects within a boundary. Also an entity that can function as the root of the cluster of objects, with identity distinguishable across the application, and methods that exclusively operate on the objects of the aggregate.

Since one command may result in several events, it is also important never to persist only some events that result from executing a command. And so events must be appended to the event store in a single atomic transaction, so that if some of the events resulting from executing a command cannot be stored then none of them will be stored.

Aggregate root

Let's define an aggregate root using class `TimestampedVersionedEntity` from the library. The `Example` class used in the previous section on snapshotting also derives from `TimestampedVersionedEntity`.

The example aggregate root class below defines (as as inner class) the domain event class `ExampleCreated` which will be published by the aggregate when creating "example" objects, and a method `count_examples()` that can operate on all the "example" objects of the aggregate.

```
from eventsourcing.domain.model.entity import TimestampedVersionedEntity


class ExampleAggregateRoot(TimestampedVersionedEntity):
    """
    Root entity of example aggregate.
    """
    class Event(TimestampedVersionedEntity.Event):
        """Layer supertype."""

    class Created(Event, TimestampedVersionedEntity.Created):
        """Published when aggregate is created."""

    class Discarded(Event, TimestampedVersionedEntity.Discarded):
        """Published when aggregate is discarded."""

    class ExampleCreated(Event):
        """Published when an "example" object in the aggregate is created."""

    def __init__(self, **kwargs):
        super(ExampleAggregateRoot, self).__init__(**kwargs)
        self._pending_events = []
        self._examples = {}
    def count_examples(self):
        return len(self._examples)

    def create_new_example(self):
        assert not self._is_discarded
        event = ExampleAggregateRoot.ExampleCreated(
            example_id=uuid.uuid4(),
            originator_id=self.id,
            originator_version=self.version,
        )
        mutate_aggregate(self, event)
        self._publish(event)

    def _publish(self, event):
        self._pending_events.append(event)

    def save(self):
        publish(self._pending_events[:])
        self._pending_events = []


class Example(object):
    """
    Example entity, exists only within the example aggregate boundary.
    """
    def __init__(self, example_id):
        self._id = example_id

    @property
    def id(self):
        return self._id
```

The methods of the aggregate, and the factory below, are similar to previous examples. But instead of immediately publishing events using the `publish()` function, the events are appended to an internal list of pending events using the aggregate's method `_publish()`. The aggregate then has a `save()` method which is used to publish all the pending events in a single list using the function `publish()`.

As before, we'll also need a factory and a mutator function. The factory function here works in the same way as before.

```
def create_example_aggregate():
    """
    Factory function for example aggregate.
    """
    # Construct event.
    event = ExampleAggregateRoot.Created(originator_id=uuid.uuid4())

    # Mutate aggregate.
    aggregate = mutate_aggregate(aggregate=None, event=event)

    # Publish event to internal list only.
    aggregate._publish(event)

    # Return the new aggregate object.
    return aggregate
```

The mutator function `mutate_aggregate()` below handles events `Created` and `Discarded` similarly to the previous examples. It also handles `ExampleCreated`, by constructing an object class `Example` that it adds to the aggregate's internal collection of examples.

```
def mutate_aggregate(aggregate, event):
    """
    Mutator function for example aggregate.
    """
    # Handle "created" events by constructing the aggregate object.
    if isinstance(event, ExampleAggregateRoot.Created):
        kwargs = event.__dict__.copy()
        kwargs['version'] = kwargs.pop('originator_version')
        kwargs['id'] = kwargs.pop('originator_id')
        aggregate = ExampleAggregateRoot(**kwargs)
        aggregate._version += 1
        return aggregate

    # Handle "example entity created" events by adding a new entity
    # to the aggregate's dict of entities.
    elif isinstance(event, ExampleAggregateRoot.ExampleCreated):
        aggregate._assert_not_discarded()
        entity = Example(example_id=event.example_id)
        aggregate._examples[str(entity.id)] = entity
        aggregate._version += 1
        aggregate._last_modified = event.timestamp
        return aggregate

    # Handle "discarded" events by returning 'None'.
    elif isinstance(event, ExampleAggregateRoot.Discarded):
        aggregate._assert_not_discarded()
        aggregate._version += 1
        aggregate._is_discarded = True
        return None
    else:
        raise NotImplementedError(type(event))
```

Application and infrastructure

Set up a database table using library classes.

```
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemySettings,␣
↪SQLAlchemyDatastore
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪IntegerSequencedItemRecord

datastore = SQLAlchemyDatastore(
    settings=SQLAlchemySettings(uri='sqlite:///:memory:'),
    tables=(IntegerSequencedItemRecord,),
)

datastore.setup_connection()
datastore.setup_tables()
```

Define an application class that uses the domain model code above, and infrastructure and policy classes from the library.

```
import uuid
import time
```

```
from eventsourcing.application.policies import PersistencePolicy
from eventsourcing.domain.model.events import publish
from eventsourcing.infrastructure.eventsourcedrepository import EventSourcedRepository
from eventsourcing.infrastructure.eventstore import EventStore
from eventsourcing.infrastructure.sequenceditemmapper import SequencedItemMapper
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy


class ExampleDDDApplication(object):
    def __init__(self, session):
        self.event_store = EventStore(
            active_record_strategy=SQLAlchemyActiveRecordStrategy(
                session=session,
                active_record_class=IntegerSequencedItemRecord,
            ),
            sequenced_item_mapper=SequencedItemMapper(
                sequence_id_attr_name='originator_id',
                position_attr_name='originator_version',
            )
        )
        self.aggregate_repository = EventSourcedRepository(
            event_store=self.event_store,
            mutator=mutate_aggregate,
        )
        self.persistence_policy = PersistencePolicy(
            event_store=self.event_store,
            event_type=ExampleAggregateRoot.Event
        )

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.persistence_policy.close()
```

Run the code

The application can be used to create new aggregates, and aggregates can be used to create new entities. Events are published in batches when the aggregate's `save()` method is called.

```
with ExampleDDDApplication(datastore.session) as app:

    # Create a new aggregate.
    aggregate = create_example_aggregate()
    aggregate.save()

    # Check it exists in the repository.
    assert aggregate.id in app.aggregate_repository, aggregate.id

    # Check the aggregate has zero entities.
    assert aggregate.count_examples() == 0

    # Check the aggregate has zero entities.
    assert aggregate.count_examples() == 0
    # Ask the aggregate to create an entity within itself.
    aggregate.create_new_example()

    # Check the aggregate has one entity.
    assert aggregate.count_examples() == 1

    # Check the aggregate in the repo still has zero entities.
    assert app.aggregate_repository[aggregate.id].count_examples() == 0

    # Call save().
    aggregate.save()

    # Check the aggregate in the repo now has one entity.
    assert app.aggregate_repository[aggregate.id].count_examples() == 1

    # Create two more entities within the aggregate.
    aggregate.create_new_example()
    aggregate.create_new_example()

    # Save both "entity created" events in one atomic transaction.
    aggregate.save()

    # Check the aggregate in the repo now has three entities.
    assert app.aggregate_repository[aggregate.id].count_examples() == 3

    # Discard the aggregate, but don't call save() yet.
    aggregate.discard()

    # Check the aggregate still exists in the repo.
    assert aggregate.id in app.aggregate_repository

    # Call save().
    aggregate.save()

    # Check the aggregate no longer exists in the repo.
    assert aggregate.id not in app.aggregate_repository
```

The library has an `AggregateRoot` class that is slightly more developed than the code in this example.

1.9.4 Application-level encryption

Install the library with the 'crypto' option.

```
pip install eventsourcing[crypto]
```

To enable encryption, pass in a cipher strategy object when constructing the sequenced item mapper, and set `always_encrypt` to a True value.

Cipher strategy

Let's firstly construct a cipher strategy object. This example uses the library AES cipher strategy `AESCipher`.

The library AES cipher strategy uses the AES cipher from the Python Cryptography Toolkit, by default in CBC mode with 128 bit blocksize and a 16 byte encryption key. It generates a unique 16 byte initialization vector for each encryption. In this cipher strategy, serialized event data is compressed before it is encrypted, which can mean application performance is improved when encryption is enabled.

With encryption enabled, event attribute values are encrypted inside the application before they are mapped to the database. The values are decrypted before domain events are replayed.

```
from eventsourcing.infrastructure.cipher.aes import AESCipher

# Construct the cipher strategy.
aes_key = b'0123456789abcdef'
cipher = AESCipher(aes_key)
```

Application and infrastructure

Set up infrastructure using library classes.

```
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemySettings,␣
↪SQLAlchemyDatastore
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪IntegerSequencedItemRecord

datastore = SQLAlchemyDatastore(
    settings=SQLAlchemySettings(uri='sqlite:///:memory:'),
    tables=(IntegerSequencedItemRecord,),
)

datastore.setup_connection()
datastore.setup_tables()
```

Define a factory that uses library classes to construct an application object.

```
from eventsourcing.example.application import ExampleApplication
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy
from eventsourcing.infrastructure.sequenceditem import SequencedItem

def construct_example_application(session, always_encrypt=False, cipher=None):
    active_record_strategy = SQLAlchemyActiveRecordStrategy(
        active_record_class=IntegerSequencedItemRecord,
        session=session
    )
    app = ExampleApplication(
        entity_active_record_strategy=active_record_strategy,
        always_encrypt=always_encrypt,
        cipher=cipher,
    )
    return app
```

Run the code

Now construct an encrypted application with the cipher. Create an "example" with some "secret information". Check the information is not visible in the database, as it is when the application is not encrypted.

```
# Create a new example entity using an encrypted application.
encrypted_app = construct_example_application(datastore.session, always_encrypt=True,␣
↪cipher=cipher)

with encrypted_app as app:
    secret_entity = app.create_new_example(foo='secret info')

    # With encryption enabled, application state is not visible in the database.
    event_store = app.entity_event_store
    item2 = event_store.active_record_strategy.get_item(secret_entity.id, eq=0)
    assert 'secret info' not in item2.data

    # Events are decrypted inside the application.
    retrieved_entity = app.example_repository[secret_entity.id]
    assert 'secret info' in retrieved_entity.foo

# Create a new example entity using an unencrypted application object.
unencrypted_app = construct_example_application(datastore.session)
with unencrypted_app as app:
    entity = app.create_new_example(foo='bar')

    # Without encryption, application state is visible in the database.
    event_store = app.entity_event_store
    item1 = event_store.active_record_strategy.get_item(entity.id, 0)
    assert 'bar' in item1.data
```

1.9.5 Optimistic concurrency control

Because of the unique constraint on the sequenced item table, it isn't possible to branch the evolution of an entity and store two events at the same version. Hence, if the entity you are working on has been updated elsewhere, an attempt to update your object will raise a concurrency exception.

Application and infrastructure

Set up infrastructure using library classes.

```
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemySettings,␣
↪SQLAlchemyDatastore
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪IntegerSequencedItemRecord

datastore = SQLAlchemyDatastore(
    settings=SQLAlchemySettings(uri='sqlite:///:memory:'),
    tables=(IntegerSequencedItemRecord,),
)

datastore.setup_connection()
datastore.setup_tables()
```

Define a factory that uses library classes to construct an application object.

```
from eventsourcing.example.application import ExampleApplication
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy
from eventsourcing.infrastructure.sequenceditem import SequencedItem

def construct_example_application(session):
    active_record_strategy = SQLAlchemyActiveRecordStrategy(
        active_record_class=IntegerSequencedItemRecord,
        sequenced_item_class=SequencedItem,
        session=session
    )
    app = ExampleApplication(
        entity_active_record_strategy=active_record_strategy
    )
    return app
```

Run the code

Use the application to get two instances of the same entity, and try to change them independently.

```
from eventsourcing.exceptions import ConcurrencyError

with construct_example_application(datastore.session) as app:

    entity = app.create_new_example(foo='bar1')

    a = app.example_repository[entity.id]
    b = app.example_repository[entity.id]

    # Change the entity using instance 'a'.
    a.foo = 'bar2'

    # Because 'a' has been changed since 'b' was obtained,
    # 'b' cannot be updated unless it is firstly refreshed.
    try:
        b.foo = 'bar3'
    except ConcurrencyError:
        pass
    else:
        raise Exception("Failed to control concurrency of 'b'.")

    # Refresh object 'b', so that 'b' has the current state of the entity.
    b = app.example_repository[entity.id]
    assert b.foo == 'bar2'

    # Changing the entity using instance 'b' now works because 'b' is up to date.
    b.foo = 'bar3'
    assert app.example_repository[entity.id].foo == 'bar3'

    # Now 'a' does not have the current state of the entity, and cannot be changed.
    try:
        a.foo = 'bar4'
    except ConcurrencyError:
        pass
    else:
        raise Exception("Failed to control concurrency of 'a'.")
```

Cassandra

The Cassandra database management system, which implements the Paxos protocol, can (allegedly) accomplish linearly-scalable distributed optimistic concurrency control, guaranteeing sequential consistency of the events of an entity. It is also possible to serialize calls to the methods of an entity, but that is out of the scope of this package — if you wish to do that, perhaps something like Zookeeper might help.

1.9.6 Alternative schema

Stored event model

The database schema we have been using so far stores events in a sequence of "sequenced items", and the names in the database schema reflect that design.

Let's say we want instead our database records to called "stored events".

It's easy to do. Just define a new sequenced item class, e.g. `StoredEvent` below, and then supply a suitable active record class. As before, create the table using the new active record class, and pass both to the active record strategy when constructing the application object.

```
from collections import namedtuple

StoredEvent = namedtuple('StoredEvent', ['aggregate_id', 'aggregate_version', 'event_
↪type', 'state'])
```

Then define a suitable active record class.

```
from sqlalchemy.ext.declarative.api import declarative_base
from sqlalchemy.sql.schema import Column, Sequence, Index
from sqlalchemy.sql.sqltypes import BigInteger, Integer, String, Text
from sqlalchemy_utils import UUIDType

Base = declarative_base()

class StoredEventRecord(Base):
    __tablename__ = 'stored_events'

    # Sequence ID (e.g. an entity or aggregate ID).
    aggregate_id = Column(UUIDType(), primary_key=True)

    # Position (timestamp) of item in sequence.
    aggregate_version = Column(BigInteger(), primary_key=True)

    # Type of the event (class name).
    event_type = Column(String(100))

    # State of the item (serialized dict, possibly encrypted).
    state = Column(Text())

    __table_args__ = Index('index', 'aggregate_id', 'aggregate_version'),
```

Application and infrastructure

Then redefine the application class to use the new sequenced item and active record classes.

```
from eventsourcing.application.policies import PersistencePolicy
from eventsourcing.infrastructure.eventsourcedrepository import EventSourcedRepository
from eventsourcing.infrastructure.eventstore import EventStore
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy
from eventsourcing.infrastructure.sequenceditem import SequencedItem
from eventsourcing.infrastructure.sequenceditemmapper import SequencedItemMapper
from eventsourcing.example.domainmodel import Example, create_new_example
```

```
class Application(object):
    def __init__(self, session):
        self.event_store = EventStore(
            active_record_strategy=SQLAlchemyActiveRecordStrategy(
                session=session,
                active_record_class=StoredEventRecord,
                sequenced_item_class=StoredEvent,
            ),
            sequenced_item_mapper=SequencedItemMapper(
                sequenced_item_class=StoredEvent,
                sequence_id_attr_name='originator_id',
                position_attr_name='originator_version',
            )
        )
        self.example_repository = EventSourcedRepository(
            event_store=self.event_store,
            mutator=Example._mutate,
        )
        self.persistence_policy = PersistencePolicy(self.event_store, event_
↪type=Example.Event)

    def create_example(self, foo):
        return create_new_example(foo=foo)

    def close(self):
        self.persistence_policy.close()

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close()
```

Set up the database.

```
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemySettings,␣
↪SQLAlchemyDatastore

datastore = SQLAlchemyDatastore(
    base=Base,
    settings=SQLAlchemySettings(uri='sqlite:///:memory:'),
    tables=(StoredEventRecord,),
)

datastore.setup_connection()
datastore.setup_tables()
```

Run the code

Then you can use the application to create, read, update, and discard. And your events will be stored as "stored events" rather than "sequenced items".

```
with Application(datastore.session) as app:

    # Create.
    example = create_new_example(foo='bar')

    # Read.
    assert example.id in app.example_repository
    assert app.example_repository[example.id].foo == 'bar'

    # Update.
    example.foo = 'baz'
    assert app.example_repository[example.id].foo == 'baz'

    # Delete.
    example.discard()
    assert example.id not in app.example_repository
```

Applause djangoevents project

It is possible to replace more aspects of the library, to make a more customized application. The excellent project djangoevents by Applause is a Django app that provides a neat way of taking an event sourcing approach in a Django project. It allows this library to be used seamlessly with Django, by using the Django ORM to store events. Using djangoevents is well documented in the README file. It adds some nice enhancements to the capabilities of this library, and shows how various components can be extended or replaced. Please note, the djangoevents project currently works with a previous version of this library.

1.9.7 Using Cassandra

Install the library with the 'cassandra' option.

```
pip install eventsourcing[cassandra]
```

Infrastructure

Set up the connection and the database tables, using the library classes for Cassandra.

If you are using default settings, make sure you have a Cassandra server available at port 9042. Please investigate the library class `CassandraSettings` for information about configuring away from default settings.

```
from eventsourcing.infrastructure.cassandra.datastore import CassandraSettings,␣
↪CassandraDatastore
from eventsourcing.infrastructure.cassandra.activerecords import␣
↪IntegerSequencedItemRecord

cassandra_datastore = CassandraDatastore(
    settings=CassandraSettings(),
    tables=(IntegerSequencedItemRecord,),
)

cassandra_datastore.setup_connection()
cassandra_datastore.setup_tables()
```

Application object

Define a factory that uses library classes for Cassandra to construct an application object.

```
from eventsourcing.example.application import ExampleApplication
from eventsourcing.infrastructure.cassandra.activerecords import␣
↪CassandraActiveRecordStrategy

def construct_application():
    active_record_strategy = CassandraActiveRecordStrategy(
        active_record_class=IntegerSequencedItemRecord,
    )
    app = ExampleApplication(
        entity_active_record_strategy=active_record_strategy,
    )
    return app
```

Run the code

The application can be used to create, read, update, and delete entities in Cassandra.

```
with construct_application() as app:

    # Create.
    example = app.create_new_example(foo='bar')

    # Read.
    assert example.id in app.example_repository
    assert app.example_repository[example.id].foo == 'bar'

    # Update.
    example.foo = 'baz'
    assert app.example_repository[example.id].foo == 'baz'

    # Delete.
    example.discard()
    assert example.id not in app.example_repository
```

1.9.8 Everything in one app

In this example, an application is developed that includes all of the aspects introduced in previous sections. The application has aggregates with a root entity that controls a cluster of entities and value objects, and which publishes events in batches. Aggregate events are stored using Cassandra, with application level encryption, and with snapshotting at regular intervals. The tests at the bottom demonstrate that it works.

Domain

Aggregate model

```
from eventsourcing.domain.model.decorators import attribute
from eventsourcing.domain.model.entity import TimestampedVersionedEntity
from eventsourcing.domain.model.events import publish, subscribe, unsubscribe


class ExampleAggregateRoot(TimestampedVersionedEntity):
    """
    Root entity of example aggregate.
    """
    class Event(TimestampedVersionedEntity.Event):
        """Layer supertype."""

    class Created(Event, TimestampedVersionedEntity.Created):
        """Published when aggregate is created."""

    class AttributeChanged(Event, TimestampedVersionedEntity.AttributeChanged):
        """Published when aggregate is changed."""

    class Discarded(Event, TimestampedVersionedEntity.Discarded):
        """Published when aggregate is discarded."""

    class ExampleCreated(Event):
        """Published when an "example" object in the aggregate is created."""

    def __init__(self, foo, **kwargs):
        super(ExampleAggregateRoot, self).__init__(**kwargs)
        self._foo = foo
        self._pending_events = []
        self._examples = {}

    @attribute
    def foo(self):
        pass

    def count_examples(self):
        return len(self._examples)

    def create_new_example(self):
        assert not self._is_discarded
        event = ExampleAggregateRoot.ExampleCreated(
            example_id=uuid.uuid4(),
            originator_id=self.id,
            originator_version=self.version,
        )
        self._apply_and_publish(event)
        self._publish(event)

    def _publish(self, event):
        self._pending_events.append(event)

    def save(self):
        publish(self._pending_events[:])
        self._pending_events = []

    @classmethod
    def _mutate(cls, initial, event):
        return mutate_aggregate(initial or cls, event)


class Example(object):
    """
    Example entity. Controlled by aggregate root.

    Exists only within the aggregate boundary.
    """
    def __init__(self, example_id):
        self._id = example_id

    @property
    def id(self):
        return self._id
```

Aggregate factory

```
def create_example_aggregate(foo):
    """
    Factory function for example aggregate.
    """
    # Construct event.
    event = ExampleAggregateRoot.Created(originator_id=uuid.uuid4(), foo=foo)

    # Mutate aggregate.
    aggregate = mutate_aggregate(ExampleAggregateRoot, event)

    # Publish event to internal list only.
    aggregate._publish(event)

    # Return the new aggregate object.
    return aggregate
```

Mutator function

```
from eventsourcing.domain.model.decorators import mutator
from eventsourcing.domain.model.entity import mutate_entity

@mutator
def mutate_aggregate(aggregate, event):
    """
    Mutator function for example aggregate.
    """
    return mutate_entity(aggregate, event)


@mutate_aggregate.register(ExampleAggregateRoot.ExampleCreated)
def _(aggregate, event):
    # Handle "ExampleCreated" events by adding a new entity to the aggregate's dict␣
↪of entities.
    try:
        aggregate._assert_not_discarded()
    except TypeError:
        raise Exception(aggregate)
    entity = Example(example_id=event.example_id)
    aggregate._examples[str(entity.id)] = entity
    aggregate._version += 1
    aggregate._last_modified = event.timestamp
    return aggregate
```

Infrastructure

```
from eventsourcing.infrastructure.cassandra.datastore import CassandraSettings,␣
↪CassandraDatastore
from eventsourcing.infrastructure.cassandra.activerecords import␣
↪IntegerSequencedItemRecord, SnapshotRecord
import uuid

cassandra_datastore = CassandraDatastore(
    settings=CassandraSettings(),
    tables=(IntegerSequencedItemRecord, SnapshotRecord),
)

cassandra_datastore.setup_connection()
cassandra_datastore.setup_tables()
```

Application

Cipher strategy

```
from eventsourcing.infrastructure.cipher.aes import AESCipher

# Construct the cipher strategy.
aes_key = b'0123456789abcdef'
cipher = AESCipher(aes_key)
```

Snapshotting policy

```
class ExampleSnapshottingPolicy(object):
    def __init__(self, example_repository, period=2):
        self.example_repository = example_repository
        self.period = period
        subscribe(predicate=self.trigger, handler=self.take_snapshot)

    def close(self):
        unsubscribe(predicate=self.trigger, handler=self.take_snapshot)

    def trigger(self, event):
        if isinstance(event, (list)):
            return True
        is_period = not (event.originator_version + 1) % self.period
        is_type = isinstance(event, ExampleAggregateRoot.Event)
        is_trigger = is_type and is_period
        return is_trigger

    def take_snapshot(self, event):
        if isinstance(event, list):
            for e in event:
                if self.trigger(e):
                    self.take_snapshot(e)
        else:
            self.example_repository.take_snapshot(event.originator_id, lte=event.
↪originator_version)
```

Application object

```
from eventsourcing.application.base import ApplicationWithPersistencePolicies
from eventsourcing.infrastructure.eventsourcedrepository import EventSourcedRepository
from eventsourcing.infrastructure.snapshotting import EventSourcedSnapshotStrategy
from eventsourcing.infrastructure.cassandra.activerecords import␣
↪CassandraActiveRecordStrategy


class EverythingApplication(ApplicationWithPersistencePolicies):

    def __init__(self, **kwargs):
        # Construct event stores and persistence policies.
        entity_active_record_strategy = CassandraActiveRecordStrategy(
            active_record_class=IntegerSequencedItemRecord,
        )
        snapshot_active_record_strategy = CassandraActiveRecordStrategy(
            active_record_class=SnapshotRecord,
        )
        super(EverythingApplication, self).__init__(
            entity_active_record_strategy=entity_active_record_strategy,
            snapshot_active_record_strategy=snapshot_active_record_strategy,
            **kwargs
        )

        # Construct snapshot strategy.
        self.snapshot_strategy = EventSourcedSnapshotStrategy(
            event_store=self.snapshot_event_store
        )

        # Construct the entity repository, this time with the snapshot strategy.
        self.example_repository = EventSourcedRepository(
            event_store=self.entity_event_store,
            mutator=ExampleAggregateRoot._mutate,
            snapshot_strategy=self.snapshot_strategy
        )

        # Construct the snapshotting policy.
        self.snapshotting_policy = ExampleSnapshottingPolicy(
            example_repository=self.example_repository,
        )

    def close(self):
        super(EverythingApplication, self).close()
        self.snapshotting_policy.close()
```

Run the code

```
from eventsourcing.exceptions import ConcurrencyError


with EverythingApplication(cipher=cipher, always_encrypt=True) as app:

    ## Check encryption.

    secret_aggregate = create_example_aggregate(foo='secret info')
    secret_aggregate.save()

    # With encryption enabled, application state is not visible in the database.
    event_store = app.entity_event_store

    item2 = event_store.active_record_strategy.get_item(secret_aggregate.id, eq=0)
    assert 'secret info' not in item2.data

    # Events are decrypted inside the application.
    retrieved_entity = app.example_repository[secret_aggregate.id]
    assert 'secret info' in retrieved_entity.foo


    ## Check concurrency control.

    aggregate = create_example_aggregate(foo='bar1')
    aggregate.create_new_example()

    aggregate.save()
    assert app.example_repository[aggregate.id].foo == 'bar1'

    a = app.example_repository[aggregate.id]
    b = app.example_repository[aggregate.id]


    # Change the aggregate using instance 'a'.
    a.foo = 'bar2'
    a.save()
    assert app.example_repository[aggregate.id].foo == 'bar2'

    # Because 'a' has been changed since 'b' was obtained,
    # 'b' cannot be updated unless it is firstly refreshed.
    try:
        b.foo = 'bar3'
        b.save()
        assert app.example_repository[aggregate.id].foo == 'bar3'
    except ConcurrencyError:
        pass
    else:
        raise Exception("Failed to control concurrency of 'b':".format(app.example_
↪repository[aggregate.id]))

    # Refresh object 'b', so that 'b' has the current state of the aggregate.
    b = app.example_repository[aggregate.id]
    assert b.foo == 'bar2'

    # Changing the aggregate using instance 'b' now works because 'b' is up to date.
    b.foo = 'bar3'
    b.save()
    assert app.example_repository[aggregate.id].foo == 'bar3'

    # Now 'a' does not have the current state of the aggregate, and cannot be changed.
    try:
        a.foo = 'bar4'
        a.save()
    except ConcurrencyError:
        pass
    else:
        raise Exception("Failed to control concurrency of 'a'.")


    ## Check snapshotting.

    # Create an aggregate.
    aggregate = create_example_aggregate(foo='bar1')
    aggregate.save()

    # Check there's no snapshot, only one event so far.
    snapshot = app.snapshot_strategy.get_snapshot(aggregate.id)
    assert snapshot is None

    # Change an attribute, generates a second event.
    aggregate.foo = 'bar2'
    aggregate.save()

    # Check the snapshot.
    snapshot = app.snapshot_strategy.get_snapshot(aggregate.id)
    assert snapshot.state['_foo'] == 'bar2'

    # Check can recover aggregate using snapshot.
    assert aggregate.id in app.example_repository
    assert app.example_repository[aggregate.id].foo == 'bar2'

    # Check snapshot after five events.
    aggregate.foo = 'bar3'
    aggregate.foo = 'bar4'
    aggregate.foo = 'bar5'
    aggregate.save()
    snapshot = app.snapshot_strategy.get_snapshot(aggregate.id)
    assert snapshot.state['_foo'] == 'bar4', snapshot.state['_foo']

    # Check snapshot after seven events.
    aggregate.foo = 'bar6'
    aggregate.foo = 'bar7'
    aggregate.save()
    assert app.example_repository[aggregate.id].foo == 'bar7'
    snapshot = app.snapshot_strategy.get_snapshot(aggregate.id)
    assert snapshot.state['_foo'] == 'bar6'

    # Check snapshot state is None after discarding the aggregate on the eighth event.
    aggregate.discard()
    aggregate.save()
    assert aggregate.id not in app.example_repository
    snapshot = app.snapshot_strategy.get_snapshot(aggregate.id)
    assert snapshot.state is None

    try:
        app.example_repository[aggregate.id]
    except KeyError:
        pass
    else:
        raise Exception('KeyError was not raised')

    # Get historical snapshots.
    snapshot = app.snapshot_strategy.get_snapshot(aggregate.id, lte=2)
    assert snapshot.state['_version'] == 2  # one behind
    assert snapshot.state['_foo'] == 'bar2'

    snapshot = app.snapshot_strategy.get_snapshot(aggregate.id, lte=3)
    assert snapshot.state['_version'] == 4
    assert snapshot.state['_foo'] == 'bar4'

    # Get historical entities.
    aggregate = app.example_repository.get_entity(aggregate.id, lte=0)
    assert aggregate.version == 1
    assert aggregate.foo == 'bar1', aggregate.foo

    aggregate = app.example_repository.get_entity(aggregate.id, lte=1)
    assert aggregate.version == 2
    assert aggregate.foo == 'bar2', aggregate.foo

    aggregate = app.example_repository.get_entity(aggregate.id, lte=2)
    assert aggregate.version == 3
    assert aggregate.foo == 'bar3', aggregate.foo

    aggregate = app.example_repository.get_entity(aggregate.id, lte=3)
    assert aggregate.version == 4
    assert aggregate.foo == 'bar4', aggregate.foo
```

1.9.9 Projections and notifications

If a projection is just another mutator function that operates on a sequence of events, and a persistent projection is a snapshot of the resulting state, then the new thing we need for projections of the application state is a sequence of all the events of the application. This section introduces the notification log, and assumes your projections and your persistent projections can be coded using techniques for coding mutator functions and snapshots introduced in previous sections.

Synchronous update

In a simple situation, you may wish to update a view of an aggregate synchronously whenever there are changes. If each view model depends only on one aggregate, you may wish simply to subscribe to the events of the aggregate. Then, whenever an event occurs, the projection can be updated.

The library has a decorator function `subscribe_to()` that can be used for this purpose.

```
@subscribe_to(Todo.Created)
def new_todo_projection(event):
    todo = TodoProjection(id=event.originator_id, title=event.title)
    todo.save()
```

The view model could be saved as a normal record, or stored in a sequence that follows the event originator version numbers, perhaps as snapshots, so that concurrent handling of events will not lead to a later state being overwritten by an earlier state. Older versions of the view could be deleted later.

If the view fails to update after the domain event has been stored, then the view will become inconsistent. Since it is not desirable to delete the event once it has been stored, the command must return normally despite the view update failing, so that the command is not retried. The failure to update will need to be logged, or otherwise handled, in a similar way to failures of asynchronous updates.

The big issue with this approach is that if the first event of an aggregate is not processed, there is no way of knowing the aggregate exists, and so there is nothing that can be used to check for updates to that aggregate.

Asynchronous update

The fundamental concern is to accomplish high fidelity when propagating a stream of events, so that events are neither missed nor are they duplicated. Once the stream of events has been propagated faithfully, it can be republished and subscribers can execute commands as above.

As Vaughn Vernon suggests in his book Implementing Domain Driven Design:

> "at least two mechanisms in a messaging solution must always be consistent with each other: the persistence store used by the domain model, and the persistence store backing the messaging infrastructure used to forward the Events published by the model. This is required to ensure that when the model's changes are persisted, Event delivery is also guaranteed, and that if an Event is delivered through messaging, it indicates a true situation reflected by the model that published it. If either of these is out of lockstep with the other, it will lead to incorrect states in one or more interdependent models."

There are three options, he continues. The first option is to have the messaging infrastructure and the domain model share the same persistence store, so changes to the model and insertion of new messages commit in the same local transaction. The second option is to have separate datastores for domain model and messaging but have a two phase commit, or global transaction, across the two.

The third option is to have the bounded context control notifications. Vaughn Vernon is his book Implementing Domain Driven Design relies on the simple logic of an ascending sequence of integers to allow others to progress along the event stream. That is the approach taken here.

A pull mechanism that allows others to pull events they don't yet have can be used to allow remote components to catch up. The same mechanism can be used if a component is developed after the application has been deployed and so requires initialising from an established application stream, or otherwise needs to be reconstructed from scratch.

As we will see below, updates can be triggered by pushing the notifications to messaging infrastructure, and having the remote components subscribe. If anything goes wrong with messaging infrastructure, such that a notification is not received, remote components can detect they have missed a notification and pull the notifications they have missed.

Application log

In order to update a projection of more than one aggregate, or the application state as a whole, we need a single sequence to log all the events of the application.

We want an application log that follows an increasing sequence of integers. The application log must also be capable of storing a very large sequence of events, neither swamping an individual database partition nor distributing things across partitions without any particular order so that iterating through the sequence is slow and expensive. We also want the application log effectively to have constant time read and write operations.

The library class `BigArray` satisfies these requirements quite well. It is a tree of arrays, with a root array that stores references to the current apex, with an apex that contains references to arrays, which either contain references to lower arrays or contain the items assigned to the big array. Each array uses one database partition, and is limited is size (the array size) to ensure the partition is never too large. The identity of each array can be calculated directly from the index number, so it is possible to identify arrays directly without traversing the tree to discover entity IDs. The capacity of base arrays is the array size to the power of the array size. For a reasonable size of array, it isn't really possible to fill up the base of such an array tree, but the slow growing properties of this tree mean that for all imaginable scenarios, the performance will be approximately constant as items are appended to the big array.

Items can be appended to a big array using the `append()` method. The append() method identifies the next available index in the array, and then assigns the item to that index in the array. A `ConcurrencyError` will be raised if the position is already taken.

The performance of the `append()` method is proportional to the log of the index in the array, to the base of the array size used in the big array, rounded up to the nearest integer, plus one (because of the root sequence that tracks the apex). For example, if the sub-array size is 10,000, then it will take only 50% longer to append the 100,000,000th item to the big array than the 1st one. By the time the 1,000,000,000,000th index position is assigned to a big array, the `append()` method will take only twice as long as the 1st.

That's because the performance of the `append()` method is dominated by the need to walk down the big array's tree of arrays to find the highest assigned index. Once the index of the next position is known, the item can be assigned directly to an array.

```
from uuid import uuid4
from eventsourcing.domain.model.array import BigArray, ItemAssigned
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy
from eventsourcing.infrastructure.sqlalchemy.activerecords import StoredEventRecord
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemyDatastore,␣
↪SQLAlchemySettings
from eventsourcing.infrastructure.eventstore import EventStore
from eventsourcing.infrastructure.repositories.array import BigArrayRepository
from eventsourcing.application.policies import PersistencePolicy
from eventsourcing.infrastructure.sequenceditem import StoredEvent
from eventsourcing.infrastructure.sequenceditemmapper import SequencedItemMapper


datastore = SQLAlchemyDatastore(
    settings=SQLAlchemySettings(),
    tables=[StoredEventRecord],
)
datastore.setup_connection()
datastore.setup_tables()

event_store = EventStore(
        active_record_strategy=SQLAlchemyActiveRecordStrategy(
            session=datastore.session,
            active_record_class=StoredEventRecord,
            sequenced_item_class=StoredEvent,
        ),
        sequenced_item_mapper=SequencedItemMapper(
            sequenced_item_class=StoredEvent,
        )
    )
persistence_policy = PersistencePolicy(
    event_store=event_store,
    event_type=ItemAssigned,
)

array_id = uuid4()

repo = BigArrayRepository(
    event_store=event_store,
    array_size=10000
)

application_log = repo[array_id]
application_log.append('event0')
application_log.append('event1')
application_log.append('event2')
application_log.append('event3')
```

Because there is a small duration of time between checking for the next position and using it, another thread could jump in and use the position first. If that happens, a *ConcurrencyError* will be raised by the *BigArray* object. In such a case, another attempt can be made to append the item.

Items can be assigned directly to a big array using an index number. If an item has already been assigned to the same position, a concurrency error will be raised, and the original item will remain in place. Items cannot be unassigned from an array, hence each position in the array can be assigned once only.

The average performance of assigning an item is a constant time. The worst case is the log of the index with base equal to the array size, which occurs when containing arrays are added, so that the last highest assigned index can be discovered. The probability of departing from average performance is inversely proportional to the array size, since the the larger the array size, the less often the base arrays fill up. For a decent array size, the probability of needing to build the tree is very low. And when the tree does need building, it doesn't take very long (and most of it probably already exists).

```
from eventsourcing.exceptions import ConcurrencyError

assert application_log.get_next_position() == 4

application_log[4] = 'event4'
try:
    application_log[4] = 'event4a'
except ConcurrencyError:
    pass
else:
    raise
```

If the next available position in the array must be identified each time an item is assigned, the amount of contention will increase as the number of threads increases. Using the `append()` method alone will work if the time period of appending events is greater than the time it takes to identify the next available index and assign to it. At that rate, any contention will not lead to congestion. Different nodes can take their chances assigning to what they believe is an unassigned index, and if another has already taken that position, the operation can be retried.

However, there will be an upper limit to the rate at which events can be appended, and contention will eventually lead to congestion that will cause requests to backup or be spilled.

The rate of assigning items to the big array can be greatly increased by centralizing the generation of the sequence of integers. Instead of discovering the next position from the array each time an item is assigned, an integer sequence generator can be used to generate a contiguous sequence of integers. This technique eliminates contention around assigning items to the big array entirely. In consequence, the bandwidth of assigning to a big array using an integer sequence generator is much greater than using the `append()` method.

If the application is executed in only one process, the number generator can be a simple Python object. The library class *`SimpleIntegerSequenceGenerator`* generates a contiguous sequence of integers that can be shared across multiple threads in the same process.

```
from eventsourcing.infrastructure.integersequencegenerators.base import␣
↪SimpleIntegerSequenceGenerator

integers = SimpleIntegerSequenceGenerator()
generated = []
for i in integers:
    if i >= 5:
        break
    generated.append(i)

expected = list(range(5))
assert generated == expected, (generated, expected)
```

If the application is deployed across many nodes, an external integer sequence generator can be used. There are many possible solutions. The library class `RedisIncr` uses Redis' INCR command to generate a contiguous sequence of integers that can be shared be processes running on different nodes.

Using Redis doesn't necessarily create a single point of failure. Redundancy can be obtained using clustered Redis. Although there aren't synchronous updates between nodes, so that the INCR command may issue the same numbers more than once, these numbers can only ever be used once. As failures are retried, the position will eventually reach an unassigned index position. Arrangements can be made to set the value from the highest assigned index. With care, the worst case will be an occasional slight delay in storing events, caused by switching to a new Redis node and catching up with the current index number. Please note, there is currently no code in the library to update or resync the Redis key used in the Redis INCR integer sequence generator.

```
from eventsourcing.infrastructure.integersequencegenerators.redisincr import RedisIncr

integers = RedisIncr()
generated = []
for i in integers:
    generated.append(i)
    if i >= 4:
        break

expected = list(range(5))
assert generated == expected, (generated, expected)
```

The integer sequence generator can be used when assigning items to the application log.

```
application_log[next(integers)] = 'event5'
application_log[next(integers)] = 'event6'

assert application_log.get_next_position() == 7
```

Items can be read from the application log using an index or a slice.

The performance of reading an item at a given index is always constant time with respect to the number of the index. The base array ID, and the index of the item in the base array, can be calculated from the number of the index.

The performance of reading a slice of items is proportional to the size of the slice. Consecutive items in a base array are stored consecutively in the same database partition, and if the slice overlaps more than base array, the iteration proceeds to the next partition.

```
assert application_log[0] == 'event0'
assert list(application_log[5:7]) == ['event5', 'event6']
```

The application log can be written to by a persistence policy. References to events can be assigned to the application log before the domain event is written to the aggregate's own sequence, so that it isn't possible to store an event in the aggregate's sequence that is not already in the application log.

Commands that fail to write to the aggregate's sequence (due to an operation error or concurrency error) after the event has been logged in the application log should probably raise an exception, so that the command is seen to have failed and so may be retried. This leaves an item in the notification log, but not a domain event in the aggregate stream (a dangling reference, that may be satisfied later). If the command failed due to an operational error, the same event maybe published again, and so it would appear twice in the application log. And so whilst events in the application log that aren't in the aggregate sequence can perhaps be ignored by consumers of the application log, care should be taken to deduplicate events.

If writing the event to its aggregate sequence is successful, then it is possible to push a notification about the event to a message queue. Failing to push the notification perhaps should not prevent the command returning normally. Push notifications could also be generated by another process, something that pulls from the application log, and pushes notifications for events that have not already been sent.

Notification log

As described in Implementing Domain Driven Design, a notification log is presented in linked sections. The "current section" is returned by default, and contains the very latest notification and some of the preceding notifications. There are also archived sections that contain all the earlier notifications. When the current section is full, it is considered to be an archived section that links to the new current section.

Readers can navigate the linked sections from the current section backwards until the archived section is reached that contains the last notification seen by the client. If the client has not yet seen any notifications, it will navigate back to the first section. Readers can then navigate forwards, revealing all existing notifications that have not yet been seen.

The library class *NotificationLog* encapsulates the application log and presents linked sections. The library class *NotificationLogReader* is an iterator that yields notifications. It navigates the sections of the notification log, and maintains position so that it can continue when there are further notifications. The position can be set directly with the `seek()` method. The position is set indirectly when a slice is taken with a start index. The position is set to zero when the reader is constructed.

The notification log uses a big array object. In this example, the big array object is directly the application log above. It is possible to project the application log into a custom notification log, perhaps to deduplicate domain events, or to anonymise data, or to send messages to messaging infrastructure with more stateful control.

```
from eventsourcing.interface.notificationlog import NotificationLog,␣
↪NotificationLogReader

# Construct notification log.
notification_log = NotificationLog(application_log, section_size=10)

# Get the "current "section from the notification log (numbering follows Vaughn Vernon
↪'s book)
section = notification_log['current']
assert section.section_id == '1,10'
assert len(section.items) == 7, section.items
assert section.previous_id == None
assert section.next_id == None

# Construct log reader.
reader = NotificationLogReader(notification_log)

# The position is zero by default.
assert reader.position == 0

# The position can be set directly.
reader.seek(10)
assert reader.position == 10

# Reset the position.
reader.seek(0)

# Read all existing notifications.
all_notifications = list(reader)
assert all_notifications == ['event0', 'event1', 'event2', 'event3', 'event4', 'event5
↪', 'event6']

# Check the position has advanced.
assert reader.position == 7

# Read all subsequent notifications (should be none).
subsequent_notifications = list(reader)
assert subsequent_notifications == []
```

```
# Assign more events to the application log.
application_log[next(integers)] = 'event7'
application_log[next(integers)] = 'event8'

# Read all subsequent notifications (should be two).
subsequent_notifications = list(reader)
assert subsequent_notifications == ['event7', 'event8']

# Check the position has advanced.
assert reader.position == 9

# Read all subsequent notifications (should be none).
subsequent_notifications = list(reader)
assert subsequent_notifications == []

# Assign more events to the application log.
application_log[next(integers)] = 'event9'
application_log[next(integers)] = 'event10'
application_log[next(integers)] = 'event11'

# Read all subsequent notifications (should be two).
subsequent_notifications = list(reader)
assert subsequent_notifications == ['event9', 'event10', 'event11']

# Check the position has advanced.
assert reader.position == 12

# Read all subsequent notifications (should be none).
subsequent_notifications = list(reader)
assert subsequent_notifications == []

# Get the "current "section from the notification log (numbering follows Vaughn Vernon
↪'s book)
section = notification_log['current']
assert section.section_id == '11,20'
assert section.previous_id == '1,10'
assert section.next_id == None
assert len(section.items) == 2, len(section.items)

# Get the first section from the notification log (numbering follows Vaughn Vernon's␣
↪book)
section = notification_log['1,10']
assert section.section_id == '1,10'
assert section.previous_id == None
assert section.next_id == '11,20'
assert len(section.items) == 10, section.items
```

The RESTful API design in Implementing Domain Driven Design suggests a good way to present the notification log, a way that is simple and can scale using established HTTP technology.

The library function *present_section()* serializes sections from the notification log for use in a view.

```
import json

from eventsourcing.interface.notificationlog import present_section

content = present_section(application_log, '1,10', 10)
```

```
expected = {
    "items": [
        "event0",
        "event1",
        "event2",
        "event3",
        "event4",
        "event5",
        "event6",
        "event7",
        "event8",
        "event9"
    ],
    "next_id": "11,20",
    "previous_id": None,
    "section_id": "1,10"
}

assert json.loads(content) == expected
```

A Web application view can pick out from the request path the notification log ID and the section ID, and return an HTTP response with the JSON content that results from calling `present_section()`.

The library class `RemoteNotificationLog` issues HTTP requests to a RESTful API that presents sections from the notification log. It has the same interface as `NotificationLog` and so can be used by `NotificationLogReader` progressively to obtain unseen notifications.

Todo: Pulling from remote notification log.

Todo: Publishing and subscribing to remote notification log.

Todo: Deduplicating domain events in receiving context. Events may appear twice in the notification log if there is contention over the command that generates the logged event, or if the event cannot be appended to the aggregate stream for whatever reason and then the command is retried successfully. So events need to be deduplicated. One approach is to have a UUID5 namespace for received events, and use concurrency control to make sure each event is acted on only once. That leads to the question of when to insert the event, before or after it is successfully applied to the context? If before, and the event is not successfully applied, then the event maybe lost. Does the context need to apply the events in order? It may help to to construct a sequenced command log, also using a big array, so that the command sequence can be constructed in a distributed manner. The command sequence can then be executed in a distributed manner. This approach would support creating another application log that is entirely correct.

Todo: Race conditions around reading events being assigned using central integer sequence generator, could potentially read when a later index has been assigned but a previous one has not yet been assigned. Reading the previous as None, when it just being assigned is an error. So perhaps something can wait until previous has been assigned, or until it can safely be assumed the integer was lost. If an item is None, perhaps the notification log could stall for a moment before yielding the item, to allow time for the race condition to pass. Perhaps it should only do it when the item has been assigned recently (timestamp of the ItemAdded event could be checked) or when there have been lots of event since (the highest assigned index could be checked). A permanent None value should be something that occurs very rarely, when an issued integer is not followed by a successful assignment to the big array. A permanent “None” will exist in the sequence if an integer is lost perhaps due to a database operation error that somehow still failed after many retries, or because the client process crashed before the database operation could be executed but after the integer had been issued, so the integer became lost. This needs code.

Todo: Automatic initialisation of the integer sequence generator RedisIncr from getting highest assigned index. Or perhaps automatic update with the current highest assigned index if there continues to be contention after a number of increments, indicating the issued values are far behind. If processes all reset the value whilst they are also incrementing it, then there will be a few concurrency errors, but it should level out quickly. This also needs code.

Todo: Use actual domain event objects, and log references to them. Have an iterator that returns actual domain events, rather than the logged references. Could log the domain events, but their variable size makes the application log less stable (predictable) in its usage of database partitions. Perhaps deferencing to real domain events could be an option of the notification log? Perhaps something could encapsulate the notification log and generate domain events?

Todo: Configuration of remote reader, to allow URL to be completely configurable.

1.9.10 Deployment

This section gives an overview of the concerns that arise when using an eventsourcing application in Web applications and task queue workers. There are many combinations of frameworks, databases, and process models. The complicated aspect is setting up the database configuration to work well with the framework. Your event sourcing application can be constructed just after the database is configured, and before requests are handled.

Please note, unlike the code snippets in the other examples, the snippets of code in this section are merely suggestive, and do not form a complete working program. For a working example using Flask and SQLAlchemy, please refer to the library module `eventsourcing.example.interface.flaskapp`, which is tested both stand-alone and with uWSGI.

Application object

In general you need one, and only one, instance of your application object in each process. If your eventsourcing application object has any policies, for example if is has a persistence policy that will persist events whenever they are published, then constructing more than one instance of the application causes the policy event handlers to be subscribed more than once, so for example more than one attempt will be made to save each event, which won't work.

To make sure there is only one instance of your application object in each process, one possible arrangement (see below) is to have a module with two functions and a variable. The first function constructs an application object and assigns it to the variable, and can perhaps be called when a module is imported, or from a suitable hook or signal designed for setting things up before any requests are handled. A second function returns the application object assigned to the variable, and can be called by any views, or request or task handlers, that depend on the application's services.

Although the first function below must be called only once, the second function can be called many times. The example functions below have been written relatively strictly so that, when it is called, the function `init_application()` will raise an exception if it has already been called, and `get_application()` will raise an exeception if `init_application()` has not already been called.

```
# Your eventsourcing application.
class ExampleApplication(object):
    def __init__(*args, **kwargs):
        pass


def construct_application(**kwargs):
    return ExampleApplication(**kwargs)


_application = None


def init_application(**kwargs):
    global _application
    if _application is not None:
        raise AssertionError("init_application() has already been called")
    _application = construct_application(**kwargs)
```

```python
def get_application():
    if application is None:
        raise AssertionError("init_application() must be called first")
    return application
```

As an aside, if you will use these function also in your test suite, and your test suite needs to set up the application more than once, you will also need a `close_application()` function that closes the application object, unsubscribing any handlers, and resetting the module level variable so that `init_application()` can be called again. If doesn't really matter if you don't close your application at the end of the process lifetime, however you may wish to close any database or other connections to network services.

```python
def close_application():
    global application
    if application is not None:
        application.close()
    application = None
```

Lazy initialization

An alternative to having separate "init" and "get" functions is having one "get" function that does lazy initialization of the application object when first requested. With lazy initialization, the getter will first check if the object it needs to return has been constructed, and will then return the object. If the object hasn't been constructed, before returning the object it will construct the object. So you could use a lock around the construction of the object, to make sure it only happens once. After the lock is obtained and before the object is constructed, it is recommended to check again that the object wasn't constructed by another thread before the lock was acquired.

```python
import threading

application = None

lock = threading.Lock()

def get_application():
    global application
    if application is None:
        lock.acquire()
        try:
            # Check again to avoid a TOCTOU bug.
            if application is None:
                application = construct_application()
        finally:
            lock.release()
    return application
```

Database connection

Typically, your eventsourcing application object will be constructed after its database connection has been configured, and before any requests are handled. Views or tasks can then safely use the already constructed application object.

If your eventsourcing application depends on receiving a database session object when it is constructed, for example if you are using the SQLAlchemy classes in this library, then you will need to create a correctly scoped session object first and use it to construct the application object.

On the other hand, if your eventsourcing application does not depend on receiving a database session object when it is constructed, for example if you are using the Cassandra classes in this library, then you may construct the application object before configuring the database connection - just be careful not to use the application object before the database connection is configured otherwise your queries just won't work.

Setting up connections to databases is out of scope of the eventsourcing application classes, and should be set up in a "normal" way. The documentation for your Web or worker framework may describe when to set up database connections, and your database documentation may also have some suggestions. It is recommended to make use of any hooks or decorators or signals intended for the purpose of setting up the database connection also to construct the application once for the process. See below for some suggestions.

SQLAlchemy

SQLAlchemy has very good documentation about constructing sessions. If you are an SQLAlchemy user, it is well worth reading the documentation about sessions in full. Here's a small quote:

> *Some web frameworks include infrastructure to assist in the task of aligning the lifespan of a Session with that of a web request. This includes products such as Flask-SQLAlchemy for usage in conjunction with the Flask web framework, and Zope-SQLAlchemy, typically used with the Pyramid framework. SQLAlchemy recommends that these products be used as available.*
>
> *In those situations where the integration libraries are not provided or are insufficient, SQLAlchemy includes its own "helper" class known as scoped_session. A tutorial on the usage of this object is at Contextual/Thread-local Sessions. It provides both a quick way to associate a Session with the current thread, as well as patterns to associate Session objects with other kinds of scopes.*

The important thing is to use a scoped session, and it is better to have the session scoped to the request or task, rather than the thread, but scoping to the thread is ok.

As soon as you have a scoped session object, you can construct your eventsourcing application.

Cassandra

Cassandra connections can be set up entirely independently of the application object. See the section about using Cassandra for more information.

Web interfaces

uWSGI

If you are running uWSGI in prefork mode, and not using a Web application framework, please note that uWSGI has a postfork decorator which may help.

Your "wsgi.py" file can have a module-level function decorated with the `@postfork` decorator that initialises your eventsourcing application for the Web application process after child workers have been forked.

```python
from uwsgidecorators import postfork

@postfork
def init_process():
    # Set up database connection.
    database = {}
    # Construct eventsourcing application.
    init_application()
```

Other decorators are available.

Flask with Cassandra

The Cassandra Driver FAQ has a code snippet about establishing the connection with the uWSGI *postfork* decorator, when running in a forked mode.

```
from flask import Flask
from uwsgidecorators import postfork
from cassandra.cluster import Cluster

session = None
prepared = None

@postfork
def connect():
    global session, prepared
    session = Cluster().connect()
    prepared = session.prepare("SELECT release_version FROM system.local WHERE key=?")

app = Flask(__name__)

@app.route('/')
def server_version():
    row = session.execute(prepared, ('local',))[0]
    return row.release_version
```

Flask-Cassandra

The Flask-Cassandra project serves a similar function to Flask-SQLAlchemy.

Flask-SQLAlchemy

If you wish to use eventsourcing with Flask and SQLAlchemy, then you may wish to use Flask-SQLAlchemy. You just need to define your active record class using the model classes from that library, and then use it instead of the library classes in your eventsourcing application object, along with the session object it provides.

The docs snippet below shows that it can work simply to construct the eventsourcing application in the same place as the Flask application object.

The Flask-SQLAlchemy class *SQLAlchemy* is used to set up a session object that is scoped to the request.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy_utils.types.uuid import UUIDType
from eventsourcing.infrastructure.sqlalchemy.activerecords import␣
↪SQLAlchemyActiveRecordStrategy


# Construct Flask application.
application = Flask(__name__)

# Construct Flask-SQLAlchemy object.
db = SQLAlchemy(application)
```

```
# Define database table using Flask-SQLAlchemy library.
class IntegerSequencedItem(db.Model):
    __tablename__ = 'integer_sequenced_items'

    # Sequence ID (e.g. an entity or aggregate ID).
    sequence_id = db.Column(UUIDType(), primary_key=True)

    # Position (index) of item in sequence.
    position = db.Column(db.BigInteger(), primary_key=True)

    # Topic of the item (e.g. path to domain event class).
    topic = db.Column(db.String(255))

    # State of the item (serialized dict, possibly encrypted).
    data = db.Column(db.Text())

    # Index.
    __table_args__ = db.Index('index', 'sequence_id', 'position'),


# Construct eventsourcing application with db table and session.
init_application(
    entity_active_record_strategy=SQLAlchemyActiveRecordStrategy(
        active_record_class=IntegerSequencedItem,
        session=db.session,
    )
)
```

For a working example using Flask and SQLAlchemy, please refer to the library module `eventsourcing.example.interface.flaskapp`, which is tested both stand-alone and with uWSGI. The Flask application method "before_first_request" is used to decorate an application object constructor, just before a request is made, so that the module can be imported by the test suite, without immediately constructing the application.

Django-Cassandra

If you wish to use eventsourcing with Django and Cassandra, you may wish to use Django-Cassandra.

It's also possible to use this library directly with Django and Cassandra. You just need to configure the connection and initialise the application before handling requests in a way that is correct for your configuration.

Django ORM

The excellent project djangoevents by Applause is a Django app that provides a neat way of taking an event sourcing approach in a Django project. It allows this library to be used seamlessly with Django, by using the Django ORM to store events. Using djangoevents is well documented in the README file. It adds some nice enhancements to the capabilities of this library, and shows how various components can be extended or replaced. Please note, the djangoevents project currently works with a previous version of this library.

Zope-SQLAlchemy

The Zope-SQLAlchemy project serves a similar function to Flask-SQLAlchemy.

Task queues

This section contains suggestions about using an eventsourcing application in task queue workers.

Celery

Celery has a worker_process_init signal decorator, which may be appropriate if you are running Celery workers in prefork mode. Other decorators are available.

Your Celery tasks or config module can have a module-level function decorated with the `@worker-process-init` decorator that initialises your eventsourcing application for the Celery worker process.

```
from celery.signals import worker_process_init

@worker_process_init.connect
def init_process(sender=None, conf=None, **kwargs):
    # Set up database connection.
    database = {}
    # Construct eventsourcing application.
    init_application()
```

As an alternative, it may work to use decorator `@task_prerun` with a getter that supports lazy initialization.

```
from celery.signals import task_prerun
@task_prerun.connect
def init_process(*args, **kwargs):
    get_appliation(lazy_init=True)
```

Once the application has been safely initialized once in the process, your Celery tasks can use function `get_application()` to complete their work. Of course, you could just call a getter with lazy initialization from the tasks.

```
from celery import Celery

app = Celery()

# Use Celery app to route the task to the worker.
@app.task
def hello_world():
    # Use eventsourcing app to complete the task.
    app = get_application()
    return "Hello World, {}".format(id(app))
```

Again, the most important thing is configuring the database, and making things work across all modes of execution, including your test suite.

Redis Queue

Redis queue workers are quite similar to Celery workers. You can call `get_application()` from within a job function. To fit with the style in the RQ documentation, you could perhaps use your eventsourcing application as a context manager, just like the Redis connection example.

1.10 Release notes

It is the aim of the project that releases with the same major version number are backwards compatible.

Version 2.x series was a major rewrite that implemented two distinct kinds of sequences: events sequenced by integer version numbers and events sequenced in time, with an archetypal "sequenced item" persistence model for storing events.

Version 1.x series was an extension of the version 0.x series, and attempted to bridge between sequencing events with both timestamps and version numbers.

Version 0.x series was the initial cut of the code, all events were sequenced by timestamps, or TimeUUIDs in Cassandra, because the project originally emerged whilst working with Cassandra.

CHAPTER 2

Reference

- search
- genindex
- modindex

CHAPTER 3

Modules

3.1 eventsourcing

3.1.1 Interface layer

eventsourcing.interface.notificationlog

class eventsourcing.interface.notificationlog.**AbstractNotificationLog**

Bases: object

Presents a sequence of sections from a sequence of notifications.

class eventsourcing.interface.notificationlog.**NotificationLog**(*big_array*, *section_size*)

Bases: *eventsourcing.interface.notificationlog.AbstractNotificationLog*

static format_section_id(*first_item_number*, *last_item_number*)

class eventsourcing.interface.notificationlog.**NotificationLogReader**(*notification_log*)

Bases: object

get_items(*stop_index=None*)

seek(*position*)

class eventsourcing.interface.notificationlog.**RemoteNotificationLog**(*base_url*, *notification_log_id*)

Bases: *eventsourcing.interface.notificationlog.AbstractNotificationLog*

get_json(*section_id*)

get_resource(*doc_url*)

make_notification_log_url(*notification_log_id*)

class `eventsourcing.interface.notificationlog.`**`Section`**(*section_id*, *items*, *previous_id=None*, *next_id=None*)

Bases: `object`

Section of a notification log.

Contains items, and has an ID.

May also have either IDs of previous and next sections of the notification log.

`eventsourcing.interface.notificationlog.`**`deserialize_section`**(*section_json*)

`eventsourcing.interface.notificationlog.`**`present_section`**(*big_array*, *section_id*, *section_size*)

`eventsourcing.interface.notificationlog.`**`serialize_section`**(*section*)

3.1.2 Application layer

`eventsourcing.application.base`

class `eventsourcing.application.base.`**`ApplicationWithEventStores`**(*entity_active_record_strategy=None*, *log_active_record_strategy=None*, *snapshot_active_record_strategy=None*, *always_encrypt=False*, *cipher=None*)

Bases: `object`

`close`()

`construct_event_store`(*event_sequence_id_attr*, *event_position_attr*, *active_record_strategy*, *always_encrypt=False*, *cipher=None*)

`construct_sequenced_item_mapper`(*sequenced_item_class*, *event_sequence_id_attr*, *event_position_attr*, *json_encoder_class=<class 'eventsourcing.infrastructure.transcoding.ObjectJSONEncoder'>*, *json_decoder_class=<class 'eventsourcing.infrastructure.transcoding.ObjectJSONDecoder'>*, *always_encrypt=False*, *cipher=None*)

class `eventsourcing.application.base.`**`ApplicationWithPersistencePolicies`**(***kwargs*)

Bases: `eventsourcing.application.base.ApplicationWithEventStores`

`close`()

`construct_entity_persistence_policy`()

`construct_log_persistence_policy`()

`construct_snapshot_persistence_policy`()

`eventsourcing.application.policies`

class `eventsourcing.application.policies.`**`PersistencePolicy`**(*event_store*, *event_type=None*)

Bases: `object`

Stores events of given type to given event store, whenever they are published.

close()

is_event(*event*)

store_event(*event*)

3.1.3 Domain layer

eventsourcing.domain.model.aggregate

class eventsourcing.domain.model.aggregate.**AggregateRoot**(***kwargs*)

Bases: *eventsourcing.domain.model.entity.WithReflexiveMutator*, *eventsourcing.domain.model.entity.TimestampedVersionedEntity*

Root entity for an aggregate in a domain driven design.

class AttributeChanged(*timestamp=None*, ***kwargs*)

Bases: eventsourcing.domain.model.aggregate.Event, eventsourcing.domain.model.entity.AttributeChanged

Published when an AggregateRoot is changed.

class Created(*timestamp=None*, ***kwargs*)

Bases: eventsourcing.domain.model.aggregate.Event, eventsourcing.domain.model.entity.Created

Published when an AggregateRoot is created.

class Discarded(*timestamp=None*, ***kwargs*)

Bases: eventsourcing.domain.model.aggregate.Event, eventsourcing.domain.model.entity.Discarded

Published when an AggregateRoot is discarded.

class Event(*timestamp=None*, ***kwargs*)

Bases: eventsourcing.domain.model.entity.Event

Layer supertype.

save()

Publishes pending events for others in application.

eventsourcing.domain.model.array

class eventsourcing.domain.model.array.**AbstractArrayRepository**(*array_size=10000*, **args*, ***kwargs*)

Bases: *eventsourcing.domain.model.entity.AbstractEntityRepository*

Repository for sequence objects.

class eventsourcing.domain.model.array.**AbstractBigArrayRepository**(**args*, ***kwargs*)

Bases: *eventsourcing.domain.model.entity.AbstractEntityRepository*

Repository for compound sequence objects.

subrepo

Sub-sequence repository.

class `eventsourcing.domain.model.array.`**`Array`**(*array_id*, *repo*)

Bases: `object`

`append`(*item*)

Sets item in next position after the last item.

`get_item_assigned`(*index*)

`get_items_assigned`(*start_index=None*, *stop_index=None*, *limit=None*, *is_ascending=True*)

`get_last_item_and_next_position`()

`get_next_position`()

class `eventsourcing.domain.model.array.`**`BigArray`**(*array_id*, *repo*)

Bases: `eventsourcing.domain.model.array.Array`

A virtual array holding items in indexed positions, across a number of Array instances.

Getting and setting items at index position is supported. Slices are supported, and operate across the underlying arrays. Appending is also supported.

BigArray is designed to overcome the concern of needing a single large sequence that may not be suitably stored in any single partiton. In simple terms, if events of an aggregate can fit in a partition, we can use the same size partition to make a tree of arrays that will certainly be capable of sequencing all the events of the application in a single stream.

With normal size base arrays, enterprise applications can expect read and write time to be approximately constant with respect to the number of items in the array.

The array is composed of a tree of arrays, which gives the capacity equal to the size of each array to the power of the size of each array. If the arrays are limited to be about the maximum size of an aggregate event stream (a large number but not too many that would cause there to be too much data in any one partition, let's say 1000s to be safe) then it would be possible to fit such a large number of aggregates in the corresponding BigArray, that we can be confident it would be full.

Write access time in the worst case, and the time to identify the index of the last item in the big array, is proportional to the log of the highest assigned index to the base of the underlying array size. Write time on average, and read time given an index, is contant with respect to the number of items in a BigArray.

Items can be appended in log time in a single thread. However, the time between reading the current last index and claiming the next position leads to contention and retries when there are lots of threads of execution all attempting to append items, which inherently limits throughput.

Todo: Not possible in Cassandra, but maybe do it in a transaction in SQLAlchemy?

An alternative to reading the last item before writing the next is to use an integer sequence generator to generate a stream of integers. Items can be assigned to index positions in a big array, according to the integers that are issued. Throughput will then be much better, and will be limited only by the rate at which the database can have events written to it (unless the number generator is quite slow).

An external integer sequence generator, such as Redis' INCR command, or an auto-incrementing database column, may constitute a single point of failure.

`calc_parent`(*i*, *j*, *h*)

Returns get_big_array and end of span of parent sequence that contains given child.

`calc_required_height`(*n*, *size*)

`create_array_id`(*i*, *j*)

`get_item`(*position*)

get_last_array()
Returns last array in compound.

Return type CompoundSequenceReader

get_last_item_and_next_position()

get_slice(*start*, *stop*)

class eventsourcing.domain.model.array.**ItemAssigned**(*item*, *index*, **args*, ***kwargs*)
Bases: eventsourcing.domain.model.entity.Event

Occurs when an item is set at a position in an array.

index

item

eventsourcing.domain.model.decorators

eventsourcing.domain.model.decorators.**attribute**(*getter*)
When used as a method decorator, returns a property object with the method as the getter and a setter defined to call instance method change_attribute(), which publishes an AttributeChanged event.

eventsourcing.domain.model.decorators.**mutator**(*arg=None*)
Structures mutator functions by allowing handlers to be registered for different types of event. When the decorated function is called with an initial value and an event, it will call the handler that has been registered for that type of event.

It works like singledispatch, which it uses. The difference is that when the decorated function is called, this decorator dispatches according to the type of last call arg, which fits better with reduce(). The builtin Python function reduce() is used by the library to replay a sequence of events against an initial state. If a mutator function is given to reduce(), along with a list of events and an initializer, reduce() will call the mutator function once for each event in the list, but the initializer will be the first value, and the event will be the last argument, and we want to dispatch according to the type of the event. It happens that singledispatch is coded to switch on the type of the first argument, which makes it unsuitable for structuring a mutator function without the modifications introduced here.

The other aspect introduced by this decorator function is the option to set the type of the handled entity in the decorator. When an entity is replayed from scratch, in other words when all its events are replayed, the initial state is None. The handler which handles the first event in the sequence will probably construct an object instance. It is possible to write the type into the handler, but that makes the entity more difficult to subclass because you will also need to write a handler for it. If the decorator is invoked with the type, when the initial value passed as a call arg to the mutator function is None, the handler will instead receive the type of the entity, which it can use to construct the entity object.

```
class Entity(object):
    class Created(object):
        pass

@mutator(Entity)
def mutate(initial, event):
    raise NotImplementedError(type(event))

@mutate.register(Entity.Created)
def _(initial, event):
    return initial(**event.__dict__)

entity = mutate(None, Entity.Created())
```

`eventsourcing.domain.model.decorators.`**`random`**`()` → x in the interval [0, 1).

`eventsourcing.domain.model.decorators.`**`retry`**`(`*exc=<class 'Exception'>*, *max_retries=1*, *wait=0*`)`

`eventsourcing.domain.model.decorators.`**`subscribe_to`**`(`*event_class*`)`

Decorator for making a custom event handler function subscribe to a certain event type

event_class: DomainEvent class or its child classes that the handler function should subscribe to

The following example shows a custom handler that reacts to Todo.Created event and saves a projection of a Todo model object.

```
@subscribe_to(Todo.Created)
def new_todo_projection(event):
    todo = TodoProjection(id=event.originator_id, title=event.title)
    todo.save()
```

eventsourcing.domain.model.entity

class `eventsourcing.domain.model.entity.`**`AbstractEntityRepository`**`(`**args*, ***kwargs*`)`

Bases: `object`

`event_store`

Returns event store object used by this repository.

`get_entity``(`*entity_id*`)`

Returns entity for given ID.

class `eventsourcing.domain.model.entity.`**`DomainEntity`**`(`*id*`)`

Bases: *`eventsourcing.domain.model.events.QualnameABC`*

class AttributeChanged`(`*originator_id*, ***kwargs*`)`

Bases: `eventsourcing.domain.model.entity.Event`, *`eventsourcing.domain.model.events.AttributeChanged`*

Published when a DomainEntity is discarded.

class Created`(`*originator_id*, ***kwargs*`)`

Bases: `eventsourcing.domain.model.entity.Event`, *`eventsourcing.domain.model.events.Created`*

Published when a DomainEntity is created.

class Discarded`(`*originator_id*, ***kwargs*`)`

Bases: `eventsourcing.domain.model.entity.Event`, *`eventsourcing.domain.model.events.Discarded`*

Published when a DomainEntity is discarded.

class Event`(`*originator_id*, ***kwargs*`)`

Bases: *`eventsourcing.domain.model.events.EventWithOriginatorID`*, *`eventsourcing.domain.model.events.DomainEvent`*

Layer supertype.

`change_attribute``(`*name*, *value*, ***kwargs*`)`

Changes given attribute of the entity, by constructing and applying an AttributeChanged event.

`discard``(`***kwargs*`)`

`id`

class `eventsourcing.domain.model.entity.`**`TimestampedEntity`**(*timestamp*, ***kwargs*)
Bases: *`eventsourcing.domain.model.entity.DomainEntity`*

class **`AttributeChanged`**(*timestamp=None*, ***kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.AttributeChanged`

Published when a TimestampedEntity is changed.

class **`Created`**(*timestamp=None*, ***kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.Created`

Published when a TimestampedEntity is created.

class **`Discarded`**(*timestamp=None*, ***kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.Discarded`

Published when a TimestampedEntity is discarded.

class **`Event`**(*timestamp=None*, ***kwargs*)
Bases: *`eventsourcing.domain.model.events.EventWithTimestamp`*, `eventsourcing.domain.model.entity.Event`

Layer supertype.

`created_on`

`last_modified`

class `eventsourcing.domain.model.entity.`**`TimestampedVersionedEntity`**(*timestamp*, ***kwargs*)
Bases: *`eventsourcing.domain.model.entity.TimestampedEntity`*, *`eventsourcing.domain.model.entity.VersionedEntity`*

class **`AttributeChanged`**(*timestamp=None*, ***kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.AttributeChanged`, `eventsourcing.domain.model.entity.AttributeChanged`

Published when a TimestampedVersionedEntity is created.

class **`Created`**(*timestamp=None*, ***kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.Created`, `eventsourcing.domain.model.entity.Created`

Published when a TimestampedVersionedEntity is created.

class **`Discarded`**(*timestamp=None*, ***kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.Discarded`, `eventsourcing.domain.model.entity.Discarded`

Published when a TimestampedVersionedEntity is discarded.

class **`Event`**(*timestamp=None*, ***kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.Event`

Layer supertype.

class `eventsourcing.domain.model.entity.`**`TimeuuidedEntity`**(*event_id*, ***kwargs*)
Bases: *`eventsourcing.domain.model.entity.DomainEntity`*

`created_on`

last_modified

class eventsourcing.domain.model.entity.**TimeuuidedVersionedEntity**(*event_id*, ***kwargs*)

Bases: *eventsourcing.domain.model.entity.TimeuuidedEntity*, *eventsourcing.domain.model.entity.VersionedEntity*

class eventsourcing.domain.model.entity.**VersionedEntity**(*version=0*, ***kwargs*)

Bases: *eventsourcing.domain.model.entity.DomainEntity*

class AttributeChanged(*originator_version*, ***kwargs*)

Bases: eventsourcing.domain.model.entity.Event, eventsourcing.domain.model.entity.AttributeChanged

Published when a VersionedEntity is changed.

class Created(*originator_version=0*, ***kwargs*)

Bases: eventsourcing.domain.model.entity.Event, eventsourcing.domain.model.entity.Created

Published when a VersionedEntity is created.

class Discarded(*originator_version*, ***kwargs*)

Bases: eventsourcing.domain.model.entity.Event, eventsourcing.domain.model.entity.Discarded

Published when a VersionedEntity is discarded.

class Event(*originator_version*, ***kwargs*)

Bases: *eventsourcing.domain.model.events.EventWithOriginatorVersion*, eventsourcing.domain.model.entity.Event

Layer supertype.

change_attribute(*name*, *value*, ***kwargs*)

discard(***kwargs*)

version

class eventsourcing.domain.model.entity.**WithReflexiveMutator**(*id*)

Bases: *eventsourcing.domain.model.entity.DomainEntity*

Implements an entity mutator function by dispatching to the event itself all calls to mutate an entity with an event.

This is an alternative to using an independent mutator function implemented with the @mutator decorator, or an if-else block.

eventsourcing.domain.model.entity.**mutate_entity**(*initial*, *event*)

Entity mutator function. Mutates initial state by the event.

Different handlers are registered for different types of event.

eventsourcing.domain.model.events

class eventsourcing.domain.model.events.**AttributeChanged**(***kwargs*)

Bases: *eventsourcing.domain.model.events.DomainEvent*

Can be published when an attribute of an entity is created.

name

value

class `eventsourcing.domain.model.events.`**`Created`**(***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

Can be published when an entity is created.

class `eventsourcing.domain.model.events.`**`Discarded`**(***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

Published when something is discarded.

class `eventsourcing.domain.model.events.`**`DomainEvent`**(***kwargs*)
Bases: *`eventsourcing.domain.model.events.QualnameABC`*

Base class for domain events.

Implements methods to make instances read-only, comparable for equality, have recognisable representations, and hashable.

exception `eventsourcing.domain.model.events.`**`EventHandlersNotEmptyError`**
Bases: `Exception`

class `eventsourcing.domain.model.events.`**`EventWithOriginatorID`**(*originator_id*, ***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

`originator_id`

class `eventsourcing.domain.model.events.`**`EventWithOriginatorVersion`**(*originator_version*, ***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

For events that have an originator version number.

`originator_version`

class `eventsourcing.domain.model.events.`**`EventWithTimestamp`**(*timestamp=None*, ***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

For events that have a timestamp value.

`timestamp`

class `eventsourcing.domain.model.events.`**`EventWithTimeuuid`**(*event_id=None*, ***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

For events that have an UUIDv1 event ID.

`event_id`

class `eventsourcing.domain.model.events.`**`Logged`**(***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

Published when something is logged.

class `eventsourcing.domain.model.events.`**`QualnameABC`**
Bases: `object`

Base class that introduces __qualname__ for objects in Python 2.7.

class `eventsourcing.domain.model.events.`**`QualnameABCMeta`**
Bases: `abc.ABCMeta`

Supplies __qualname__ to object classes with this metaclass.

`eventsourcing.domain.model.events.`**`assert_event_handlers_empty`**()

eventsourcing.domain.model.events.**create_timesequenced_event_id**()

eventsourcing.domain.model.events.**publish**(*event*)

eventsourcing.domain.model.events.**subscribe**(*handler*, *predicate=None*)

eventsourcing.domain.model.events.**unsubscribe**(*handler*, *predicate=None*)

eventsourcing.domain.model.snapshot

class eventsourcing.domain.model.snapshot.**AbstractSnapshop**

Bases: object

originator_id

ID of the snapshotted entity.

originator_version

Version of the last event applied to the entity.

state

State of the snapshotted entity.

topic

Path to the class of the snapshotted entity.

class eventsourcing.domain.model.snapshot.**Snapshot**(*originator_id*, *originator_version*, *topic*, *state*)

Bases: *eventsourcing.domain.model.events.EventWithTimestamp*, *eventsourcing.domain.model.events.EventWithOriginatorVersion*, *eventsourcing.domain.model.events.EventWithOriginatorID*, *eventsourcing.domain.model.snapshot.AbstractSnapshop*

state

State of the snapshotted entity.

topic

Path to the class of the snapshotted entity.

eventsourcing.infrastructure.cipher.aes

eventsourcing.infrastructure.cipher.base

class eventsourcing.infrastructure.cipher.base.**AbstractCipher**

Bases: object

decrypt(*ciphertext*)

Return plaintext for given ciphertext.

encrypt(*plaintext*)

Return ciphertext for given plaintext.

3.1.4 Infrastructure layer

eventsourcing.infrastructure.eventsourcedrepository

class eventsourcing.infrastructure.eventsourcedrepository.**EventSourcedRepository**(*event_store*, *mutator=None*, *snapshot_strategy=*, *use_cache=Fa*, **args*, ***kwargs*)

Bases: *eventsourcing.domain.model.entity.AbstractEntityRepository*

event_store

get_domain_events(*entity_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *is_ascending=True*)
Returns domain events for given entity ID.

get_entity(*entity_id*, *lt=None*, *lte=None*)
Returns entity with given ID, optionally until position.

get_most_recent_event(*entity_id*, *lt=None*, *lte=None*)
Returns the most recent event for the given entity ID.

get_snapshot(*entity_id*, *lt=None*, *lte=None*)
Returns a snapshot for given entity ID, according to the snapshot strategy.

mutator(*initial*, *event*)
Entity mutator function. Mutates initial state by the event.

Different handlers are registered for different types of event.

replay_entity(*entity_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *initial_state=None*, *query_descending=False*)
Reconstitutes requested domain entity from domain events found in event store.

replay_events(*initial_state*, *domain_events*)
Mutates initial state using the sequence of domain events.

take_snapshot(*entity_id*, *lt=None*, *lte=None*)
Takes a snapshot of the entity as it existed after the most recent event, optionally less than, or less than or equal to, a particular position.

eventsourcing.infrastructure.eventplayer

class eventsourcing.infrastructure.eventplayer.**EventPlayer**(*event_store*, *mutator*, *page_size=None*, *is_short=False*, *snapshot_strategy=None*)

Bases: `object`

Reconstitutes domain entities from domain events retrieved from the event store, optionally with snapshots.

get_domain_events(*entity_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *is_ascending=True*)
Returns domain events for given entity ID.

get_most_recent_event (*entity_id*, *lt=None*, *lte=None*)
Returns the most recent event for the given entity ID.

get_snapshot (*entity_id*, *lt=None*, *lte=None*)
Returns a snapshot for given entity ID, according to the snapshot strategy.

replay_entity (*entity_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *initial_state=None*, *query_descending=False*)
Reconstitutes requested domain entity from domain events found in event store.

replay_events (*initial_state*, *domain_events*)
Mutates initial state using the sequence of domain events.

take_snapshot (*entity_id*, *lt=None*, *lte=None*)
Takes a snapshot of the entity as it existed after the most recent event, optionally less than, or less than or equal to, a particular position.

eventsourcing.infrastructure.snapshotting

class eventsourcing.infrastructure.snapshotting.**AbstractSnapshotStrategy**
Bases: object

get_snapshot (*entity_id*, *lt=None*, *lte=None*)
Gets the last snapshot for entity, optionally until a particular version number.

Return type *Snapshot*

take_snapshot (*entity_id*, *entity*, *last_event_version*)
Takes a snapshot of entity, using given ID, state and version number.

Return type *AbstractSnapshop*

class eventsourcing.infrastructure.snapshotting.**EventSourcedSnapshotStrategy** (*event_store*)
Bases: *eventsourcing.infrastructure.snapshotting.AbstractSnapshotStrategy*

Snapshot strategy that uses an event sourced snapshot.

get_snapshot (*entity_id*, *lt=None*, *lte=None*)
Gets the last snapshot for entity, optionally until a particular version number.

Return type *Snapshot*

take_snapshot (*entity_id*, *entity*, *last_event_version*)
Takes a snapshot by instantiating and publishing a Snapshot domain event.

Return type *Snapshot*

eventsourcing.infrastructure.snapshotting.**entity_from_snapshot** (*snapshot*)
Reconstructs domain entity from given snapshot.

eventsourcing.infrastructure.eventstore

class eventsourcing.infrastructure.eventstore.**AbstractEventStore**
Bases: object

all_domain_events ()
Returns all domain events in the event store.

append (*domain_event_or_events*)
Put domain event in event store for later retrieval.

get_domain_event(*originator_id*, *eq*)
Returns a single domain event.

get_domain_events(*originator_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *is_ascending=True*, *page_size=None*)
Returns domain events for given entity ID.

get_most_recent_event(*originator_id*, *lt=None*, *lte=None*)
Returns most recent domain event for given entity ID.

class eventsourcing.infrastructure.eventstore.**EventStore**(*active_record_strategy*, *sequenced_item_mapper*)
Bases: *eventsourcing.infrastructure.eventstore.AbstractEventStore*

all_domain_events()

append(*domain_event_or_events*)

get_domain_event(*originator_id*, *eq*)

get_domain_events(*originator_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *is_ascending=True*, *page_size=None*)

get_most_recent_event(*originator_id*, *lt=None*, *lte=None*)

iterator_class
alias of `SequencedItemIterator`

to_sequenced_item(*domain_event*)

eventsourcing.infrastructure.sequenceditem

class eventsourcing.infrastructure.sequenceditem.**SequencedItem**(*sequence_id*, *position*, *topic*, *data*)
Bases: `tuple`

data
Alias for field number 3

position
Alias for field number 1

sequence_id
Alias for field number 0

topic
Alias for field number 2

class eventsourcing.infrastructure.sequenceditem.**SequencedItemFieldNames**(*sequenced_item_class*)
Bases: `object`

data

position

sequence_id

topic

class eventsourcing.infrastructure.sequenceditem.**StoredEvent**(*originator_id*, *originator_version*, *event_type*, *state*)

Bases: tuple

event_type
Alias for field number 2

originator_id
Alias for field number 0

originator_version
Alias for field number 1

state
Alias for field number 3

eventsourcing.infrastructure.sequenceditemmapper

class eventsourcing.infrastructure.sequenceditemmapper.**AbstractSequencedItemMapper**
Bases: object

from_sequenced_item(*sequenced_item*)
Return domain event from given sequenced item.

to_sequenced_item(*domain_event*)
Returns sequenced item for given domain event.

class eventsourcing.infrastructure.sequenceditemmapper.**SequencedItemMapper**(*sequenced_item_class=
'eventsourc-
ing.infrastructure.sequ
se-
quence_id_attr_name=
po-
si-
tion_attr_name=None*, *json_encoder_class=<
'eventsourc-
ing.infrastructure.trans
json_decoder_class=<
'eventsourc-
ing.infrastructure.trans
al-
ways_encrypt=False*, *ci-
pher=None*, *other_attr_names=()*)

Bases: eventsourcing.infrastructure.sequenceditemmapper.AbstractSequencedItemMapper

Uses JSON to transcode domain events.

construct_item_args(*domain_event*)
Constructs attributes of a sequenced item from the given domain event.

construct_sequenced_item(*item_args*)

deserialize_event_attrs(*event_attrs*, *is_encrypted*)
Deserialize event attributes from JSON, optionally with decryption.

from_sequenced_item(*sequenced_item*)
Reconstructs domain event from stored event topic and event attrs. Used in the event store when getting domain events.

is_encrypted(*domain_event_class*)

serialize_event_attrs(*event_attrs*, *is_encrypted=False*)

to_sequenced_item(*domain_event*)
Constructs a sequenced item from a domain event.

eventsourcing.infrastructure.sequenceditemmapper.**reconstruct_object**(*obj_class*, *obj_state*)

eventsourcing.infrastructure.transcoding

class eventsourcing.infrastructure.transcoding.**ObjectJSONDecoder**(*object_hook=None*, ***kwargs*)
Bases: json.decoder.JSONDecoder

classmethod from_jsonable(*d*)

class eventsourcing.infrastructure.transcoding.**ObjectJSONEncoder**(*sort_keys=True*, **args*, ***kwargs*)
Bases: json.encoder.JSONEncoder

default(*obj*)

eventsourcing.infrastructure.activerecord

class eventsourcing.infrastructure.activerecord.**AbstractActiveRecordStrategy**(*active_record_class se- quenced_item_class 'eventsourc- ing.infrastructure.se*
Bases: object

all_items()
Returns all stored items from all sequences (possibly in chronological order, depending on database).

all_records(*resume=None*, **arg*, ***kwargs*)
Returns all records in the table (possibly in chronological order, depending on database).

append(*sequenced_item_or_items*)
Writes sequenced item into the datastore.

delete_record(*record*)
Removes permanently given record from the table.

get_field_kwargs(*item*)

get_item(*sequence_id*, *eq*)
Reads sequenced item from the datastore.

get_items(*sequence_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *query_ascending=True*, *results_ascending=True*)
Reads sequenced items from the datastore.

raise_index_error(*eq*)

raise_sequenced_item_error(*sequenced_item*, *e*)

eventsourcing.infrastructure.datastore

class eventsourcing.infrastructure.datastore.**Datastore**(*settings*)
Bases: object

drop_connection()
Drops connection to a datastore.

drop_tables()
Drops tables used to store events.

setup_connection()
Sets up a connection to a datastore.

setup_tables()
Sets up tables used to store events.

truncate_tables()
Truncates tables used to store events.

exception eventsourcing.infrastructure.datastore.**DatastoreConnectionError**
Bases: *eventsourcing.infrastructure.datastore.DatastoreError*

exception eventsourcing.infrastructure.datastore.**DatastoreError**
Bases: Exception

class eventsourcing.infrastructure.datastore.**DatastoreSettings**
Bases: object

Base class for settings for database connection used by a stored event repository.

exception eventsourcing.infrastructure.datastore.**DatastoreTableError**
Bases: *eventsourcing.infrastructure.datastore.DatastoreError*

eventsourcing.infrastructure.cassandra.activerecords

eventsourcing.infrastructure.cassandra.datastore

eventsourcing.infrastructure.sqlalchemy.activerecords

eventsourcing.infrastructure.sqlalchemy.datastore

eventsourcing.infrastructure.iterators

class eventsourcing.infrastructure.iterators.**AbstractSequencedItemIterator**(*active_record_strategy*
se-
quence_id,
page_size=None,
gt=None,
gte=None,
lt=None,
lte=None,
limit=None,
is_ascending=True)
Bases: object

DEFAULT_PAGE_SIZE = 1000

class eventsourcing.infrastructure.iterators.**GetEntityEventsThread**(*active_record_strategy*, *sequence_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *page_size=None*, *is_ascending=True*, **args*, ***kwargs*)

Bases: `threading.Thread`

run()

class eventsourcing.infrastructure.iterators.**SequencedItemIterator**(*active_record_strategy*, *sequence_id*, *page_size=None*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *is_ascending=True*)

Bases: `eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator`

class eventsourcing.infrastructure.iterators.**ThreadedSequencedItemIterator**(*active_record_strategy*
sequence_id, *page_size=None*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *is_ascending=True*)

Bases: `eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator`

start_thread()

eventsourcing.infrastructure.repositories.array

class eventsourcing.infrastructure.repositories.array.**ArrayRepository**(*array_size=10000*, **args*, ***kwargs*)

Bases: `eventsourcing.domain.model.array.AbstractArrayRepository`, `eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository`

class eventsourcing.infrastructure.repositories.array.**BigArrayRepository**(*base_size=10000*, **args*, ***kwargs*)

Bases: `eventsourcing.domain.model.array.AbstractBigArrayRepository`, `eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository`

subrepo

subrepo_class
alias of *ArrayRepository*

eventsourcing.infrastructure.integersequencegenerators.base

class eventsourcing.infrastructure.integersequencegenerators.base.**AbstractIntegerSequenceGe**
Bases: object

next()
Python 2.7 version of the iterator protocol.

class eventsourcing.infrastructure.integersequencegenerators.base.**SimpleIntegerSequenceGene**
Bases: *eventsourcing.infrastructure.integersequencegenerators.base.AbstractIntegerSequenceGenerator*

eventsourcing.infrastructure.integersequencegenerators.redisincr

3.1.5 Exception classes

exception eventsourcing.exceptions.**ArrayIndexError**
Bases: IndexError, *eventsourcing.exceptions.EventSourcingError*

Raised when appending item to an array that is full.

exception eventsourcing.exceptions.**ConcurrencyError**
Bases: *eventsourcing.exceptions.EventSourcingError*

Raised when appending events at the wrong version to a versioned stream.

exception eventsourcing.exceptions.**ConsistencyError**
Bases: *eventsourcing.exceptions.EventSourcingError*

Raised when applying an event stream to a versioned entity.

exception eventsourcing.exceptions.**DatasourceSettingsError**
Bases: *eventsourcing.exceptions.EventSourcingError*

Raised when an error is detected in settings for a datasource.

exception eventsourcing.exceptions.**EntityIsDiscarded**
Bases: AssertionError

Raised when access to a recently discarded entity object is attempted.

exception eventsourcing.exceptions.**EntityVersionNotFound**
Bases: *eventsourcing.exceptions.EventSourcingError*

Raise when accessing an entity version that does not exist.

exception eventsourcing.exceptions.**EventSourcingError**
Bases: Exception

Base eventsourcing exception.

exception eventsourcing.exceptions.**MismatchedOriginatorError**
Bases: *eventsourcing.exceptions.ConsistencyError*

Raised when applying an event to an inappropriate object.

exception eventsourcing.exceptions.**MismatchedOriginatorIDError**
Bases: *eventsourcing.exceptions.MismatchedOriginatorError*

Raised when applying an event to the wrong entity or aggregate.

exception eventsourcing.exceptions.**MismatchedOriginatorVersionError**
Bases: *eventsourcing.exceptions.MismatchedOriginatorError*

Raised when applying an event to the wrong version of an entity or aggregate.

exception eventsourcing.exceptions.**MutatorRequiresTypeNotInstance**
Bases: *eventsourcing.exceptions.ConsistencyError*

Raised when mutator function received a class rather than an entity.

exception eventsourcing.exceptions.**ProgrammingError**
Bases: *eventsourcing.exceptions.EventSourcingError*

Raised when programming errors are encountered.

exception eventsourcing.exceptions.**RepositoryKeyError**
Bases: KeyError, *eventsourcing.exceptions.EventSourcingError*

Raised when using entity repository's dictionary like interface to get an entity that does not exist.

exception eventsourcing.exceptions.**SequencedItemConflict**
Bases: *eventsourcing.exceptions.EventSourcingError*

Raised when an integer sequence error occurs e.g. trying to save a version that already exists.

exception eventsourcing.exceptions.**TimeSequenceError**
Bases: *eventsourcing.exceptions.EventSourcingError*

Raised when a time sequence error occurs e.g. trying to save a timestamp that already exists.

exception eventsourcing.exceptions.**TopicResolutionError**
Bases: *eventsourcing.exceptions.EventSourcingError*

Raised when unable to resolve a topic to a Python class.

3.1.6 Example application

eventsourcing.example.interface.flaskapp

eventsourcing.example.application

class eventsourcing.example.application.**ExampleApplication**(***kwargs*)
Bases: *eventsourcing.application.base.ApplicationWithPersistencePolicies*

Example event sourced application with entity factory and repository.

create_new_example(*foo=", a=", b="*)
Entity object factory.

eventsourcing.example.application.**close_example_application**()
Shuts down single global instance of application.

To be called when tearing down, perhaps between tests, in order to allow a subsequent call to init_example_application().

eventsourcing.example.application.**construct_example_application**(***kwargs*)
Application object factory.

`eventsourcing.example.application.`**`get_example_application`**`()`

Returns single global instance of application.

To be called when handling a worker request, if required.

`eventsourcing.example.application.`**`init_example_application`**`(`*`**kwargs`*`)`

Constructs single global instance of application.

To be called when initialising a worker process.

`eventsourcing.example.domainmodel`

class `eventsourcing.example.domainmodel.`**`AbstractExampleRepository`**`(`*`*args, **kwargs`*`)`

Bases: *`eventsourcing.domain.model.entity.AbstractEntityRepository`*

class `eventsourcing.example.domainmodel.`**`Example`**`(`*foo='', a='', b='', **kwargs*`)`

Bases: *`eventsourcing.domain.model.entity.TimestampedVersionedEntity`*

An example event sourced domain model entity.

class **`AttributeChanged`**`(`*timestamp=None, **kwargs*`)`

Bases: `eventsourcing.example.domainmodel.Event`, `eventsourcing.domain.model.entity.AttributeChanged`

Published when an Example is created.

class **`Created`**`(`*timestamp=None, **kwargs*`)`

Bases: `eventsourcing.example.domainmodel.Event`, `eventsourcing.domain.model.entity.Created`

Published when an Example is created.

class **`Discarded`**`(`*timestamp=None, **kwargs*`)`

Bases: `eventsourcing.example.domainmodel.Event`, `eventsourcing.domain.model.entity.Discarded`

Published when an Example is discarded.

class **`Event`**`(`*timestamp=None, **kwargs*`)`

Bases: `eventsourcing.domain.model.entity.Event`

Layer supertype.

class **`Heartbeat`**`(`*timestamp=None, **kwargs*`)`

Bases: `eventsourcing.example.domainmodel.Event`, `eventsourcing.domain.model.entity.Event`

Published when a heartbeat in the entity occurs (see below).

`a`

An example attribute.

`b`

Another example attribute.

`beat_heart``(`*number_of_beats=1*`)`

`count_heartbeats``()`

`foo`

An example attribute.

eventsourcing.example.domainmodel.**create_new_example**(*foo=”*, *a=”*, *b=”*)

Factory method for example entities.

Return type *Example*

eventsourcing.example.domainmodel.**example_mutator**(*initial*, *event*)

eventsourcing.example.domainmodel.**heartbeat_mutator**(*self*, *event*)

eventsourcing.example.infrastructure

class eventsourcing.example.infrastructure.**ExampleRepository**(*event_store*, *mutator=None*, *snapshot_strategy=None*, *use_cache=False*, **args*, ***kwargs*)

Bases: *eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository*, *eventsourcing.example.domainmodel.AbstractExampleRepository*

Event sourced repository for the Example domain model entity.

mutator(*initial=None*, *event=None*)

Python Module Index

Index