
eventsourcing Documentation

Release 5.0.0

John Bywater

Jan 25, 2018

Contents

1	Overview	3
1.1	Background	3
1.2	Quick start	4
1.3	Installation	7
1.4	Features	7
1.5	Design	8
1.6	Infrastructure	8
1.7	Domain model	25
1.8	Application	38
1.9	Snapshotting	44
1.10	Stand-alone example	46
1.11	Notifications	56
1.12	Deployment	71
1.13	Release notes	79
1.14	Module docs	80
	Python Module Index	117

A library for event sourcing in Python.

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. It is common for the state of a software application to be modelled across a set of entities or aggregates.

Therefore, this library provides mechanisms for event sourced applications, a way for domain events to be stored and replayed, and a style for coding entities with both state that is driven by events and behaviour that drives events.

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*, has information about *deployment*, and has 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 make a simple event sourced application using classes from the library. It shows the general story, which is elaborated over the following pages.

- *Define model*
- *Configure environment*
- *Run application*

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

```
$ pip install event sourcing[sqlalchemy]
```

1.2.1 Define model

Define a domain model aggregate.

The class `World` defined below is a subclass of `AggregateRoot`.

The `World` has a property called `history`. It also has an event sourced attribute called `ruler`.

It has a command method called `make_it_so` which triggers a domain event of type `SomethingHappened` which is defined as a nested class. The domain event class `SomethingHappened` has a `mutate()` method, which happens to append triggered events to the history.

```
from event sourcing.domain.model.aggregate import AggregateRoot
from event sourcing.domain.model.decorators import attribute

class World(AggregateRoot):

    def __init__(self, ruler=None, **kwargs):
        super(World, self).__init__(**kwargs)
        self._history = []
        self._ruler = ruler

    @property
    def history(self):
        return tuple(self._history)

    @attribute
    def ruler(self):
        """A mutable event-sourced attribute."""
```



```
def make_it_so(self, something):
    self.__trigger_event__(World.SomethingHappened, what=something)

class SomethingHappened(AggregateRoot.Event):
    def mutate(self, obj):
        obj._history.append(self)
```

This class can be used and completely tested without any infrastructure.

Although every aggregate is a “little world”, developing a more realistic domain model would involve defining attributes, command methods, and domain events particular to a concrete domain.

Basically, you can understand everything if you understand that command methods, such as `make_it_so()` in the example above, should not update the state of the aggregate directly with the results of their work, but instead trigger events using domain event classes which have `mutate()` methods that can update the state of the aggregate using the values given to the event when it was triggered. By refactoring the updating of the aggregate state, from the command method, to a domain event object class, triggered events can be stored and replayed to obtain persistent aggregates.

1.2.2 Configure environment

Generate cipher key (optional).

```
from eventsourcing.utils.random import encode_random_bytes

# Keep this safe.
cipher_key = encode_random_bytes(num_bytes=32)
```

Configure environment variables.

```
import os

# Optional cipher key (random bytes encoded with Base64).
os.environ['CIPHER_KEY'] = cipher_key

# SQLAlchemy-style database connection string.
os.environ['DB_URI'] = 'sqlite:///memory:'
```

1.2.3 Run application

With the `SimpleApplication` from the library, you can create, read, update, and delete `World` aggregates that are persisted in the database identified above.

The code below demonstrates many of the features of the library, such as optimistic concurrency control, data integrity, and application-level encryption.

```
from eventsourcing.application.simple import SimpleApplication
from eventsourcing.exceptions import ConcurrencyError

# Construct simple application (used here as a context manager).
with SimpleApplication() as app:

    # Call library factory method.
    world = World.__create__(ruler='gods')
```

```

# Execute commands.
world.make_it_so('dinosaurs')
world.make_it_so('trucks')

version = world.__version__ # note version at this stage
world.make_it_so('internet')

# Assign to event-sourced attribute.
world.ruler = 'money'

# View current state of aggregate.
assert world.ruler == 'money'
assert world.history[2].what == 'internet'
assert world.history[1].what == 'trucks'
assert world.history[0].what == 'dinosaurs'

# Publish pending events (to persistence subscriber).
world.__save__()

# Retrieve aggregate (replay stored events).
copy = app.repository[world.id]
assert isinstance(copy, World)

# View retrieved state.
assert copy.ruler == 'money'
assert copy.history[2].what == 'internet'
assert copy.history[1].what == 'trucks'
assert copy.history[0].what == 'dinosaurs'

# Verify retrieved state (cryptographically).
assert copy.__head__ == world.__head__

# Discard aggregate.
world.__discard__()

# Repository raises key error (when aggregate not found).
assert world.id not in app.repository
try:
    app.repository[world.id]
except KeyError:
    pass
else:
    raise Exception("Shouldn't get here")

# Get historical state (at version from above).
old = app.repository.get_entity(world.id, at=version)
assert old.history[-1].what == 'trucks' # internet not happened
assert len(old.history) == 2
assert old.ruler == 'gods'

# Optimistic concurrency control (no branches).
old.make_it_so('future')
try:
    old.__save__()
except ConcurrencyError:
    pass
else:
    raise Exception("Shouldn't get here")

```

```
# Check domain event data integrity (happens also during replay).
events = app.event_store.get_domain_events(world.id)
last_hash = ''
for event in events:
    event.__check_hash__()
    assert event.__previous_hash__ == last_hash
    last_hash = event.__event_hash__

# Verify sequence of events (cryptographically).
assert last_hash == world.__head__

# Check records are encrypted (values not visible in database).
record_manager = app.event_store.record_manager
items = record_manager.get_items(world.id)
for item in items:
    assert item.originator_id == world.id
    assert 'dinosaurs' not in item.state
    assert 'trucks' not in item.state
    assert 'internet' not in item.state
```

1.3 Installation

Use pip to install the library from the [Python Package Index](#).

```
$ pip install eventsourcing
```

If you want to use [SQLAlchemy](#), then install the library with the ‘sqlalchemy’ option. Also install your chosen database driver.

```
$ pip install eventsourcing[sqlalchemy]
$ pip install pycopg2
```

Similarly, if you want to use [Apache Cassandra](#), then please install with the ‘cassandra’ option.

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

Running the install command with again 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 a record manager to map domain events to database records in ways that can be easily extended and replaced.

Data integrity - Sequences of events can be hash-chained, and the entire sequence of events checked for integrity. If the last hash can be independently validated, then so can the entire sequence. Events records can be encrypted with an authenticated encryption algorithm, so you cannot lose information in transit or at rest, or get database corruption without being able to detect it.

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 record manager.

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, and an example database table. Plus lots of examples in the documentation.

1.5 Design

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

The infrastructure layer encapsulates infrastructural services required by an event sourced application, in particular an event store.

The domain layer contains independent domain model classes. Nothing in the domain layer depends on anything in the infrastructure layer.

The application layer 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 repository, from which example entities can be retrieved. It also has a factory method to create 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. Event players use the event store to retrieve the stored events, and the model mutator functions to project entities from sequences of events.

Functionality such as mapping events to a database, or snapshotting, is implemented as strategy objects, and injected into dependents by constructor parameter, making it easy to substitute custom classes for defaults.

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.

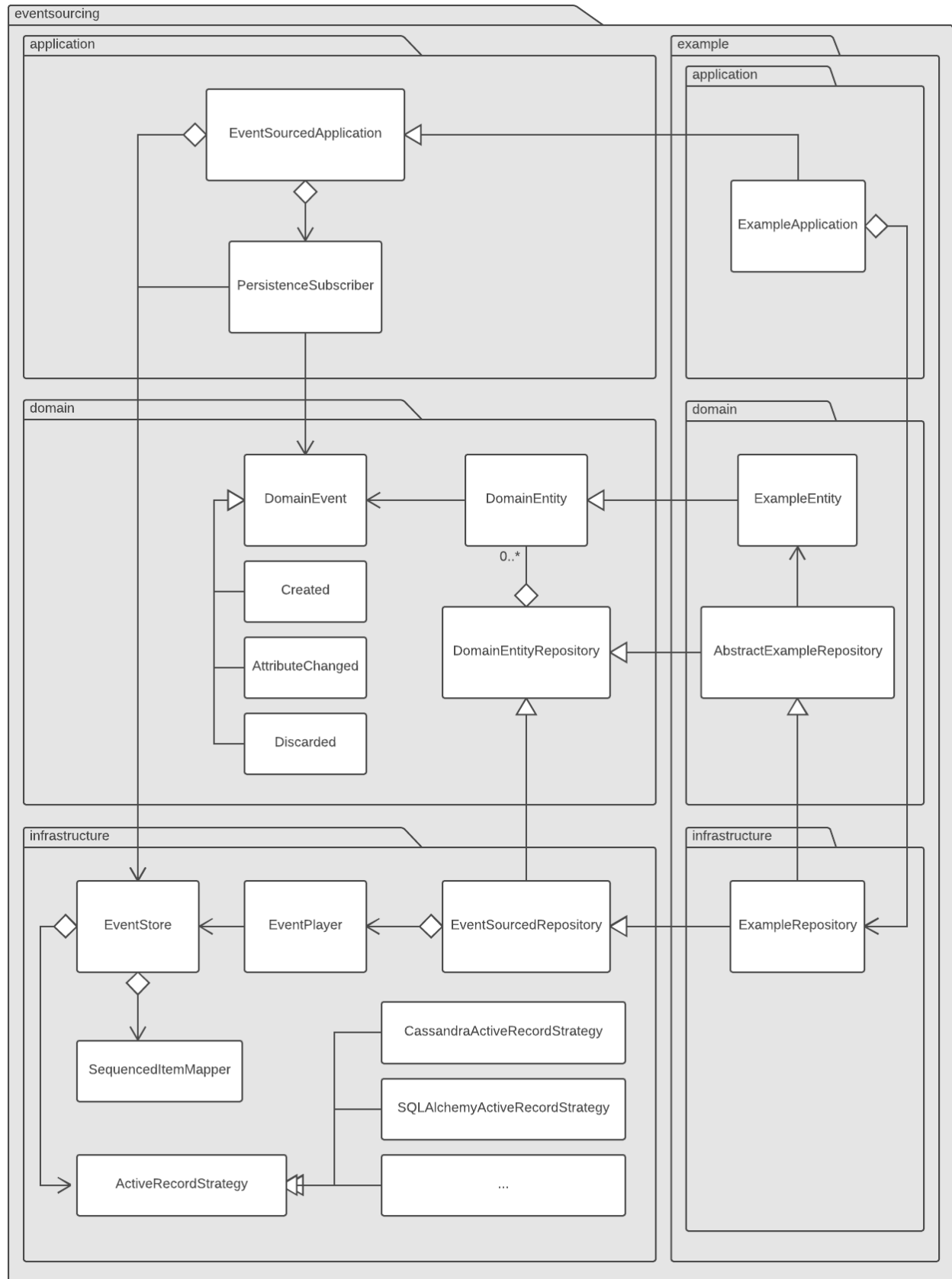
1.6 Infrastructure

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 a record manager.

The sequenced item mapper converts objects such as domain events to sequenced items, and the record manager writes sequenced items to database records. The sequenced item mapper and the record manager operate by reflection off a common sequenced item type.

- *Sequenced item type*
 - *SequencedItem namedtuple*



- *StoredEvent namedtuple*
- *Sequenced item mapper*
 - *Custom JSON transcoding*
 - *Application-level encryption*
- *Record managers*
 - *SQLAlchemy*
 - * *SQLAlchemy dialects*
 - * *MySQL*
 - * *PostgreSQL*
 - * *SQLite*
 - *Django ORM*
 - * *Django backends*
 - *Contiguous record IDs*
 - *Cassandra*
 - *Sequenced item conflicts*
- *Event store*
 - *Optimistic concurrency control*
 - *Event store factory*
 - *Timestamped event store*
 - * *TimeUUIDs*

1.6.1 Sequenced item type

Sequenced item types are declared as named tuples (`namedtuple` from `collections`).

Below is an example of a sequenced item named tuple.

```
from collections import namedtuple

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

The fields can be named differently, however a suitable database table will have matching column names.

Whatever the names of the fields, the first field of a sequenced item will represent the identity of a sequence to which an item belongs. The second field will represent the position of the item in its sequence. The third field will represent a topic to which the item pertains. And the fourth field will represent the data associated with the item.

SequencedItem namedtuple

The library provides a sequenced item named tuple called *SequencedItem*.

```
from event_sourcing.infrastructure.sequenceditem import SequencedItem
```

Like in the example above, the library's `SequencedItem` namedtuple has four fields. 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 a dimension of concern to which the item pertains. The `data` holds the data associated with the item.

A sequenced item is just a tuple, and can be used as such. In the example below, a sequenced item happens to be constructed with a UUID to identify a sequence. The item has also been given an integer position value, it has a topic that happens to correspond to a domain event class in the library. The item's data is a JSON object in which `foo` is `bar`.

```
from uuid import uuid4

sequence1 = uuid4()

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

As expected, the attributes of the sequenced item object are simply the values given when the object was constructed.

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

StoredEvent namedtuple

The library provides a sequenced item named tuple 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 event sourcing.infrastructure.sequenceditem import StoredEvent

aggregate1 = uuid4()

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

assert stored_event1.originator_id == aggregate1
assert stored_event1.originator_version == 0
assert stored_event1.event_type == 'event sourcing.domain.model.events#DomainEvent'
assert stored_event1.state == '{"foo":"bar"}'
```

1.6.2 Sequenced item mapper

The event store uses a sequenced item mapper to map between sequenced items and application-level objects such as domain events.

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

```
from event sourcing . infrastructure . sequenceditemmapper import SequencedItemMapper
```

The `SequencedItemMapper` has a constructor arg `sequenced_item_class`, which defaults to the library's sequenced item named tuple `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.foo == 'bar'
```

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

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

If the names of the first two fields of the sequenced item named tuple (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`.

```
from event sourcing . domain . model . events import DomainEvent

domain_event1 = DomainEvent(
    originator_id=aggregatel,
    originator_version=1,
    foo='baz',
)

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

assert domain_event1.foo == 'baz'

assert sequenced_item_mapper.to_sequenced_item(domain_event1).sequence_id ==
↳aggregatel
```

Alternatively, a sequenced item named tuple type that is different from the default `SequencedItem` namedtuple, for example the library's `StoredEvent` namedtuple, can be passed with the constructor arg `sequenced_item_class`.

```
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
```


Since the alternative `StoredEvent` namedtuple can be used instead of the default `SequencedItem` namedtuple, so it is possible to use a custom named tuple. 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.utils.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.utils.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. If a cipher is given, then the state field of every sequenced item will be encrypted before being sent to the database. The data retrieved from the database will be decrypted and verified, which protects against tampering.

The library provides an AES cipher object class called `AESCipher`. It uses the AES cipher from the Python Cryptography Toolkit, as forked by the actively maintained [PyCryptodome project](#).

The `AESCipher` class uses AES in GCM mode, which is a padding-less, authenticated encryption mode. Other AES modes aren't supported by this class, at the moment.

The `AESCipher` constructor arg `cipher_key` is required. The key 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. However, the `utils` package does contain a function `encode_random_bytes()` that may help to generate a unicode key string, representing random bytes encoded with Base64. A companion function `decode_random_bytes()` decodes the unicode key string into a sequence of bytes.

```

from event sourcing.utils.cipher.aes import AESCipher
from event sourcing.utils.random import encode_random_bytes, decode_random_bytes

# Unicode string representing 256 random bits encoded with Base64.
cipher_key = encode_random_bytes(num_bytes=32)

# Construct AES-256 cipher.
cipher = AESCipher(cipher_key=decode_random_bytes(cipher_key))

# Encrypt some plaintext (using nonce arguments).
ciphertext = cipher.encrypt('plaintext')
assert ciphertext != 'plaintext'

# Decrypt some ciphertext.
plaintext = cipher.decrypt(ciphertext)
assert plaintext == 'plaintext'

```

The `SequencedItemMapper` has constructor arg `cipher`, which can be used to pass in a cipher object, and thereby enable encryption.

```

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

# 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 not encrypted, necessarily. However, by encrypting the state of the item within the application, potentially sensitive information, for example personally identifiable information, will be encrypted in transit to the database, at rest in the database, and in all backups and other copies.

1.6.3 Record managers

The event store uses a record manager to write sequenced items to database records.

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

A record manager is constructed with a `sequenced_item_class` and a matching `record_class`. The field names of a suitable record class will match the field names of the sequenced item named tuple.

SQLAlchemy

The library has a record manager for SQLAlchemy provided by the object class `SQLAlchemyRecordManager`.

To run the example below, please install the library with the ‘sqlalchemy’ option.

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

The library provides record classes for SQLAlchemy, such as `IntegerSequencedRecord` and `StoredEventRecord`. The `IntegerSequencedRecord` class matches the default `SequencedItem` namedtuple. The `StoredEventRecord` class matches the alternative `StoredEvent` namedtuple. There is also a `TimestampSequencedRecord` and a `SnapshotRecord`.

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

```
from eventsourcing.infrastructure.sqlalchemy.records 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 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 record classes which inherit from your Base class. If so, it may help to refer to the library 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 `record_class`, the `SQLAlchemyRecordManager` requires a scoped session object, passed using the constructor arg `session`. For convenience, the `SQLAlchemyDatabase` has a thread-scoped session facade set as its `session` attribute. You may wish to use a different scoped session facade, such as a request-scoped session object provided by a Web framework.

With the database setup, the `SQLAlchemyRecordManager` can be constructed, and used to store events using SQLAlchemy.

```
from eventsourcing.infrastructure.sqlalchemy.manager import SQLAlchemyRecordManager

record_manager = SQLAlchemyRecordManager(
    sequenced_item_class=StoredEvent,
    record_class=StoredEventRecord,
    session=datastore.session,
    contiguous_record_ids=True
)
```

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

```
record_manager.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 = record_manager.list_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
```

SQLAlchemy dialects

The databases supported by core [SQLAlchemy dialects](#) are Firebird, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, and Sybase. This library's infrastructure classes for SQLAlchemy have been tested with MySQL, PostgreSQL, and SQLite.

MySQL

For MySQL, the Python package [mysqlclient](#) can be used.

```
$ pip install mysqlclient
```

The `uri` for MySQL would look something like this.

```
mysql://username:password@localhost/eventsourcing
```

PostgreSQL

For PostgreSQL, the Python package [psycopg2](#) can be used.

```
$ pip install psycopg2
```

The `uri` for PostgreSQL would look something like this.

```
postgresql://username:password@localhost:5432/eventsourcing
```

SQLite

SQLite is shipped with core Python packages, so nothing extra needs to be installed.

The `uri` for a temporary SQLite database might look something like this.

```
sqlite:////tmp/eventsourcing.db
```

Please note, the library's SQLAlchemy infrastructure defaults to using an in memory SQLite database, which is the fastest way to run the library, and is recommended as a convenience for development.

Django ORM

The library has a record manager for the Django ORM provided by `DjangoRecordManager` class.

To run the example below, please install the library with the 'django' option.

```
$ pip install eventsourcing[django]
```

For the `DjangoRecordManager`, the `IntegerSequencedRecord` from `eventsourcing.infrastructure.django.models` matches the `SequencedItem` namedtuple. The `StoredEventRecord` from the same module matches the `StoredEvent` namedtuple. There is also a `TimestampSequencedRecord` and a `SnapshotRecord`. These are all Django models.

The package `eventsourcing.infrastructure.django` is a little Django app. To involve its models in your Django project, simply include the application in your project's list of `INSTALLED_APPS`.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'eventsourcing.infrastructure.django'
]
```

Alternatively, import or write the classes you want into one of your own Django app's `models.py`.

The Django application at `eventsourcing.infrastructure.django` has database migrations that will add four tables, one for each of the record classes mentioned above. So if you use the application directly in `INSTALLED_APPS` then the app's migrations will be picked up by Django.

If, instead of using the app directly, you import some of its model classes into your own application's `models.py`, you will need to run `python manage.py makemigrations` before tables for event sourcing can be created by Django. This way you can avoid creating tables you won't use.

The library has a little Django project for testing the library's Django app, it is used in this example to help run the library's Django app.

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'eventsourcing.tests.djangoproject.
↳ djangoproject.settings'
```

This Django project is simply the files that `django-admin.py startproject` generates, with the SQLite database set to be in memory, and with the library's Django app added to the `INSTALLED_APPS` setting.

With the environment variable `DJANGO_SETTINGS_MODULE` referring to the Django project, Django can be started. If you aren't running tests with the Django test runner, you may need to run `django.setup()`.

```
import django

django.setup()
```

Before using the database, make sure the migrations have been applied, so the necessary database tables exist.

An alternative to `python manage.py migrate` is the `call_command()` function, provided by Django. If you aren't running tests with the Django test runner, this can help e.g. to setup an SQLite database in memory before each test by calling it in the `setUp()` method of a test case.

```
from django.core.management import call_command

call_command('migrate')
```

So long as a table exists for its record class, the `DjangoRecordManager` can be used to store events using the Django ORM.

```
from eventsourcing.infrastructure.django.manager import DjangoRecordManager
from eventsourcing.infrastructure.django.models import StoredEventRecord

django_record_manager = DjangoRecordManager(
    record_class=StoredEventRecord,
    sequenced_item_class=StoredEvent,
    contiguous_record_ids=True
)
```

```
results = django_record_manager.list_items(aggregate1)
assert len(results) == 0

django_record_manager.append(stored_event1)

results = django_record_manager.list_items(aggregate1)
assert results[0] == stored_event1
```

Django backends

The supported [Django backends](#) are PostgreSQL, MySQL, SQLite, and Oracle. This library’s Django infrastructure classes have been tested with PostgreSQL, MySQL, SQLite.

Contiguous record IDs

The `contiguous_record_ids` argument, used in the examples above, is optional, and is by default `False`. If set to a `True` value, and if the record class has an ID field, then the records will be inserted (using an “insert select from” query) that generates a table of records with IDs that form a contiguous integer sequence.

Application events recorded in this way can be accurately followed as a single sequence without overbearing complexity to mitigate gaps and race conditions. This feature is only available on the relational record managers (Django and SQLAlchemy, not Cassandra).

If the record ID is merely auto-incrementing, as it is when the the library’s integer sequenced record classes are used without this feature being enabled, then gaps could be generated. Whenever there is contention in the aggregate sequence (record ID) that causes the unique record ID constraint to be violated, the transaction will be rolled back, and an ID that was issued could be discarded and lost. Other greater IDs may already have been issued. The complexity for followers is that a gap may be permanent or temporary. It may be that a gap is eventually filled by a transaction that was somehow delayed. Although some database appear to have auto-incrementing functionality that does not lead to gaps even with transactions being rolled back, I don’t understand when this happens and when it doesn’t and so feel unable to reply on it, at least at the moment. It appears to be an inherently unreliable situation that could probably be mitigated satisfactorily by followers if they need to project the application events accurately, but only with increased complexity.

Each relational record manager has a raw SQL query with an “insert select from” statement. If possible, the raw query is compiled when the record manager object is constructed. When a record is inserted, the new field values are bound to the raw query and executed within a transaction. When executed, the query firstly selects the maximum ID from all records currently existing in the table (as visible in its transaction), and then attempts to insert a record with an ID value of the max existing ID plus one (the next unused ID). The record table must have a unique constraint for the ID, so that records aren’t overwritten by this query. The record ID must also be indexed, so that the max value can be identified efficiently. The b-tree commonly used for database indexes supports this purpose well. The transaction isolation level must be at least “read committed”, which is true by default for MySQL and PostgreSQL.

It is expected that the performance of the “insert select from” statement will not be dominated by the “select max from” clause, but rather by the work needed to update the table indexes during the insert, which would also be required when executing a more conventional “insert values” statement. Hence it is anticipated that the maximum rate of inserting records will not be reduced greatly by enabling this feature. Initial performance testing seems to confirm this expectation.

Any resulting contention in the record ID will raise an exception so that the query can be retried. An attempt is made to disambiguate this integrity error from those caused by the constraint on the position field of a sequenced item. If the database error can be identified as a record ID conflict, then the library exception class `RecordIDConflict` will be raised. Otherwise the library exception class `SequencedItemConflict` will be raised. `RecordIDConflict`

inherits from `SequencedItemConflict`. A `RecordIDConflict` can be retried with good hope that a retry will be successful. A `SequencedItemConflict` that is not a `RecordIDConflict` means an application command needs to be retried. If a `SequencedItemConflict` is raised due to a record ID conflict, then retrying the application command is a reasonable response. The `RecordIDConflict` exception exists as an optimisation, to allow the query to be retried without the cost of re-executing the command. Hence, failures to distinguish integrity errors caused by record ID conflicts from other integrity errors (which in the current implementation depends on the name of the index in the record class conforming to expectations and being included in the database exception) are not catastrophic.

Cassandra

The library has a record manager for Apache Cassandra provided by the `CassandraRecordManager` class.

```
from eventsourcing.infrastructure.cassandra.manager import CassandraRecordManager
```

To run the example below, please install the library with the ‘cassandra’ option.

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

It takes a while to build the driver. If you want to do that last step quickly, set the environment variable `CASS_DRIVER_NO_CYTHON`.

```
$ CASS_DRIVER_NO_CYTHON=1 pip install eventsourcing[cassandra]
```

For the `CassandraRecordManager`, the `IntegerSequencedRecord` from `eventsourcing.infrastructure.cassandra.models` matches the `SequencedItem` namedtuple. The `StoredEventRecord` from the same module matches the `StoredEvent` namedtuple. There is also a `TimestampSequencedRecord`, a `TimeuuidSequencedRecord`, and a `SnapshotRecord`.

The `CassandraDatastore` and `CassandraSettings` can be used in the same way as `SQLAlchemyDatastore` and `SQLAlchemySettings` above. Please investigate library class `CassandraSettings` for information about configuring away from default settings.

```
from eventsourcing.infrastructure.cassandra.datastore import CassandraDatastore, \
    ↪CassandraSettings
from eventsourcing.infrastructure.cassandra.records import StoredEventRecord

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

With the database setup, the `CassandraRecordManager` can be constructed, and used to store events using Apache Cassandra.

```
from eventsourcing.infrastructure.cassandra.manager import CassandraRecordManager

cassandra_record_manager = CassandraRecordManager(
    record_class=StoredEventRecord,
    sequenced_item_class=StoredEvent,
)

results = cassandra_record_manager.list_items(aggregate1)
assert len(results) == 0
```



```
cassandra_record_manager.append(stored_event1)

results = cassandra_record_manager.list_items(aggregate1)
assert results[0] == stored_event1

cassandra_datastore.drop_tables()
cassandra_datastore.close_connection()
```

Sequenced item conflicts

It is a common feature of the record manager classes 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.

```
from eventsourcing.exceptions import SequencedItemConflict

# Fail to append an item at the same position in the same sequence as a previous item.
try:
    record_manager.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, a unique constraint is used that involves both the sequence and the position columns. The Django ORM strategy works in the same way.

With Cassandra the position is the primary key in the sequence partition, and the “IF NOT EXISTS” feature is applied. The Cassandra database management system implements the Paxos protocol, and can thereby accomplish linearly-scalable distributed optimistic concurrency control, guaranteeing sequential consistency of the events of an entity despite the database being distributed. 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.

1.6.4 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 record manager, both are discussed in detail in the sections above.

```
from eventsourcing.infrastructure.eventstore import EventStore

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

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 named tuple from a domain events, and it uses the `record_manager` to write a sequenced item to a database.

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

```
event_store.append(
    DomainEvent(
        originator_id=aggregatel,
        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 `record_manager` 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(aggregatel)
```

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].foo == 'bar'
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(aggregatel, lte=0)
assert len(result) == 1, result
assert result[0].foo == 'bar'

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

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

# Get the last event in the sequence.
result = event_store.get_domain_events(aggregatel, limit=1, is_ascending=False)
assert len(result) == 1, result
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=aggregatel,
            originator_version=1,
            foo='baz',
        )
    )
except ConcurrencyError:
    pass
else:
    raise Exception("ConcurrencyError not raised")
```

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

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

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

errors = []

@retry(ConcurrencyError, max_attempts=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 named tuple, and the record class `StoredEventRecord`. The optional args `sequenced_item_class` and `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 record class `TimestampSequencedRecord`.

```
from uuid import uuid4

from event_sourcing.infrastructure.sqlalchemy.records import TimestampSequencedRecord
from event_sourcing.utils.times import decimaltimestamp

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

# Construct event store for timestamp sequenced events.
timestamped_event_store = factory.construct_sqlalchemy_eventstore(
    sequenced_item_class=SequencedItem,
    record_class=TimestampSequencedRecord,
    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=decimaltimestamp(),
)

# 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 < decimaltimestamp()
```

Please note, optimistic concurrent control doesn't work with timestamped sequenced items to maintain consistency of a domain entity, because each event is likely to have a unique timestamp, and so branches can occur without restraint. Optimistic concurrency control will prevent one timestamp sequenced event from overwriting another. For this reason, although domain events are usefully 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. It remains that timestamp sequenced items can happen to occur at the same timestamp, in which case there would be a concurrency error exception, and the event could be retried with a later timestamp.

TimeUUIDs

If throughput is so high that such conflicts are too frequent, the library also supports sequencing items by TimeUUID, which includes a random component that makes it very unlikely two events will conflict. This feature currently works with Apache Cassandra only. Tests exist in the library, other documentation is forthcoming.

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.

- *Domain events*
 - *Publish-subscribe*
 - *Event library*
 - *Custom events*
- *Domain entities*
 - *Entity library*
 - *Naming style*
 - *Entity events*
 - *Factory method*
 - *Triggering events*
 - *Data integrity*
 - *Discarding entities*
 - *Custom entities*
 - *Custom attributes*
 - *Custom commands*
 - *Custom events*
- *Aggregate root*

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 event_sourcing.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 event_sourcing.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 event_sourcing.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 decimal import Decimal
from uuid import UUID

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

# 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, AttributeChanged, Discarded

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

def is_attribute_changed(event):
    return isinstance(event, AttributeChanged)

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”.

```

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

```

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

```

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.

```

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

assert done.timestamp > seen.timestamp

```

So long as the entity event classes inherit ultimately from library class `QualnameABC`, which `DomainEvent` does, the utility functions `get_topic()` and `resolve_topic()` can work with domain events defined as inner or nested classes in all versions of Python. These functions are used in the `DomainEntity.Created` event class, and in the infrastructure class `SequencedItemMapper`. The requirement to inherit from `QualnameABC` actually only applies when using nested classes in Python 2.7 with the utility functions `get_topic()` and `resolve_topic()`. Events classes that are not nested, or that will not be run with Python 2.7, do not need to inherit from `QualnameABC` in order to work with these two functions (and hence the library domain and infrastructure classes which use those functions).

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 has a base class for domain entities called `DomainEntity`, which has an `id` attribute.

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

entity_id = uuid4()

entity = DomainEntity(id=entity_id)

assert entity.id == entity_id
```

Entity library

The library also has a domain entity class called `VersionedEntity`, which extends the `DomainEntity` class with a `__version__` attribute.

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

entity = VersionedEntity(id=entity_id, __version__=1)

assert entity.id == entity_id
assert entity.__version__ == 1
```

The library also has a domain entity class called `TimestampedEntity`, which extends the `DomainEntity` class with attributes `__created_on__` and `__last_modified__`.

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

entity = TimestampedEntity(id=entity_id, __created_on__=123)

assert entity.id == entity_id
assert entity.__created_on__ == 123
assert entity.__last_modified__ == 123
```

There is also a `TimestampedVersionedEntity` that has `id`, `__version__`, `__created_on__`, and `__last_modified__` attributes.

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

entity = TimestampedVersionedEntity(id=entity_id, __version__=1, __created_on__=123)

assert entity.id == entity_id
assert entity.__created_on__ == 123
assert entity.__last_modified__ == 123
assert entity.__version__ == 1
```

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

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

Naming style

The double leading and trailing underscore naming style, seen above, is used consistently in the library's domain entity and event base classes for attribute and method names, so that developers can begin with a clean namespace.

The intention is that the library functionality is included in the application by aliasing these library names with names that work within the project’s ubiquitous language.

This style breaks PEP8, but it seems worthwhile in order to keep the “normal” Python object namespace free for domain modelling. It is a style used by other libraries (such as SQLAlchemy and Django) for similar reasons.

The exception is the `id` attribute of the domain entity base class, which is assumed to be required by all domain entities (or aggregates) in all domains.

Entity events

The library’s domain entity classes have domain events defined as inner classes: `Event`, `Created`, `AttributeChanged`, and `Discarded`.

```
DomainEntity.Event
DomainEntity.Created
DomainEntity.AttributeChanged
DomainEntity.Discarded
```

The domain event class `DomainEntity.Event` is a super type of the others. The others also inherit from the library base classes `Created`, `AttributeChanged`, and `Discarded`. All these domain events classes are subclasses of `DomainEvent`.

```
assert issubclass(DomainEntity.Created, DomainEntity.Event)
assert issubclass(DomainEntity.AttributeChanged, DomainEntity.Event)
assert issubclass(DomainEntity.Discarded, DomainEntity.Event)

assert issubclass(DomainEntity.Created, Created)
assert issubclass(DomainEntity.AttributeChanged, AttributeChanged)
assert issubclass(DomainEntity.Discarded, Discarded)

assert issubclass(DomainEntity.Event, DomainEvent)
```

These entity event classes can be freely constructed, with suitable arguments.

All events need an `originator_id`. Events of versioned entities also need an `originator_version`. Events of timestamped entities generate a current `timestamp` value, unless one is given. `Created` events also need an `originator_topic`. The other events need an `__previous_hash__`. `AttributeChanged` events also need `name` and `value`.

All the events of `DomainEntity` use SHA-256 to generate an `event_hash` from the event attribute values when constructed for the first time. Events can be chained together by constructing each subsequent event to have its `__previous_hash__` as the `event_hash` of the previous event.

```
from eventsourcing.utils.topic import get_topic

entity_id = UUID('b81d160d-d7ef-45ab-a629-c7278082a845')

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

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

```

    originator_id=entity_id,
    __previous_hash__=created.__event_hash__,
)

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

entity_discarded = VersionedEntity.Discarded(
    originator_version=3,
    originator_id=entity_id,
    __previous_hash__=attribute_b_changed.__event_hash__,
)

```

The events have a `__mutate__()` function, which can be used to mutate the state of a given object appropriately.

For example, the `DomainEntity.Created` event mutates to an entity instance. The class that is instantiated is determined by the `originator_topic` attribute of the `DomainEntity.Created` event.

A domain event's `__mutate__()` method normally requires an `obj` argument, but that is not required for `DomainEntity.Created` events. The default is `None`, but if a value is provided it must be callable that returns an object, such as a domain entity class. If a domain entity class is provided, the `originator_topic` will be ignored.

```

entity = created.__mutate__()

assert entity.id == entity_id

```

As another example, when a versioned entity is mutated by an event of the `VersionedEntity` class, the entity version number is set to the event `originator_version`.

```

assert entity.__version__ == 0

entity = attribute_a_changed.__mutate__(entity)
assert entity.__version__ == 1
assert entity.a == 1

entity = attribute_b_changed.__mutate__(entity)
assert entity.__version__ == 2
assert entity.b == 2

```

Similarly, when a timestamped entity is mutated by an event of the `TimestampedEntity` class, the `__last_modified__` attribute of the entity is set to have the event's timestamp value.

Factory method

The `DomainEntity` has a class method `__create__()` which can return new entity objects. When called, it constructs the `Created` event of the concrete class with suitable arguments such as a unique ID, and a topic representing the concrete entity class, and then it projects that event into an entity object using the event's `__mutate__()` method. Then it publishes the event, and then it returns the new entity to the caller. This technique works correctly for subclasses of both the entity and the event class.

```
entity = DomainEntity.__create__()
assert entity.id
assert entity.__class__ is DomainEntity

entity = VersionedEntity.__create__()
assert entity.id
assert entity.__version__ == 0
assert entity.__class__ is VersionedEntity

entity = TimestampedEntity.__create__()
assert entity.id
assert entity.__created_on__
assert entity.__last_modified__
assert entity.__class__ is TimestampedEntity

entity = TimestampedVersionedEntity.__create__()
assert entity.id
assert entity.__created_on__
assert entity.__last_modified__
assert entity.__version__ == 0
assert entity.__class__ is TimestampedVersionedEntity
```

Triggering events

Commands methods will construct, apply, and publish events, using the results from working on command arguments. The events need to be constructed with suitable arguments.

To help trigger events in an extensible manner, the `DomainEntity` class has a method called `__trigger_event__()`, that is extended by subclasses in the library, which can be used in command methods to construct, apply, and publish events with suitable arguments. The events' `__mutate__()` methods update the entity appropriately.

For example, triggering an `AttributeChanged` event on a timestamped, versioned entity will cause the attribute value to be updated, but it will also cause the version number to increase, and it will update the last modified time.

```
entity = TimestampedVersionedEntity.__create__()
assert entity.__version__ == 0
assert entity.__created_on__ == entity.__last_modified__

# Trigger domain event.
entity.__trigger_event__(entity.AttributeChanged, name='c', value=3)

# Check the event was applied.
assert entity.c == 3
assert entity.__version__ == 1
assert entity.__last_modified__ > entity.__created_on__
```

The command method `__change_attribute__()` triggers an `AttributeChanged` event. In the code below, the attribute `full_name` is set to 'Mr Boots'. A subscriber receives the event.

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

entity = VersionedEntity.__create__(entity_id)
```

```
# Change an attribute.
entity.__change_attribute__(name='full_name', value='Mr Boots')

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

# Check two events were published.
assert len(received_events) == 2

first_event = received_events[0]
assert first_event.__class__ == VersionedEntity.Created
assert first_event.originator_id == entity_id
assert first_event.originator_version == 0

last_event = received_events[1]
assert last_event.__class__ == VersionedEntity.AttributeChanged
assert last_event.name == 'full_name'
assert last_event.value == 'Mr Boots'
assert last_event.originator_version == 1

# Check the event hash is the current entity head.
assert last_event.__event_hash__ == entity.__head__

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

Data integrity

Domain events that are triggered in this way are hash-chained together by default.

The state of each event, including the hash of the last event, is hashed using SHA-256. Before an event is applied to an entity, it is validated in itself (the event hash represents the state of the event) and as a part of the chain (the previous event hash is included in the next event state). If the sequence of events is accidentally damaged in any way, then a `DataIntegrityError` will almost certainly be raised from the domain layer when the sequence is replayed.

The hash of the last event applied to an entity is available as an attribute called `__head__`.

```
# Entity's head hash is determined exclusively
# by the entire sequence of events and SHA-256.
assert entity.__head__ ==
↳ 'ae7688000c38b2bd504b3eb3cd8e015144dd9a3c4992951c87cef9cce047f86c'

# Entity's head hash is simply the event hash
# of the last event that mutated the entity.
assert entity.__head__ == last_event.__event_hash__
```

A different sequence of events will almost certainly result a different head hash. So the entire history of an entity can be verified by checking the head hash. This feature could be used to protect against tampering.

The hashes can be salted by setting environment variable `SALT_FOR_DATA_INTEGRITY`, perhaps with random bytes encoded as Base64.

```
from eventsourcing.utils.random import encode_random_bytes

# Keep this safe.
```

```
salt = encode_random_bytes(num_bytes=32)

# Configure environment (before importing library).
import os
os.environ['SALT_FOR_DATA_INTEGRITY'] = salt
```

Discarding entities

The entity method `__discard__()` can be used to discard the entity, by triggering 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")
```

Custom entities

The library entity classes can be subclassed.

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

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 triggers 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):
        """Full name of the user."""
```

In the code below, after the entity has been created, assigning to the `full_name` attribute causes the entity to be updated. An `AttributeChanged` event is 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 = User.__create__(full_name='Mrs Boots')

# Publish an AttributeChanged event.
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[0].originator_version == 0
assert received_events[0].originator_id == user.id

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

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

Custom commands

The entity base classes can be extended with custom command methods. In general, the arguments of a command will be used to perform some work. Then, the result of the work will be used to trigger a domain event that represents what happened. Please note, command methods normally have no return value.

For example, the `set_password()` method of the `User` entity below is given a raw password. It creates an encoded string from the raw password, and then uses the `__change_attribute__()` method to trigger an `AttributeChanged` event for the `_password` attribute with the encoded password.

```
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)

        # Change private _password attribute.
        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', __version__=0)

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

```

Custom events

Custom events can be defined as inner or nested classes of the custom entity class. In the code below, the entity class `World` has a custom event called `SomethingHappened`.

Custom event classes can extend the `__mutate__()` method, so it affects entities in a way that is specific to that type of event. More conveniently, event classes can implement a `mutate()` method, which avoids the need to call the super method and return the obj. For example, the `SomethingHappened` event class has a `mutate()` method which simply appends the event object to the entity's `history` attribute.

Custom events are normally triggered by custom commands. In the example below, the command method `make_it_so()` triggers the custom event `SomethingHappened`.

```

from event_sourcing.domain.model.decorators import mutator

class World(VersionsEntity):

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

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

        # Trigger event with the results of the work.
        self.__trigger_event__(World.SomethingHappened, what=what_happened)

    class SomethingHappened(VersionsEntity.Event):
        """Published when something happens in the world."""
        def mutate(self, obj):
            obj.history.append(self)

```

A new world can now be created, using the `__create__()` method. The command `make_it_so()` can be used to make things happen in this world. When something happens, the history of the world is augmented with the new event.

```

world = World.__create__()

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

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."

In this situation, one aggregate command may result in many events. In order to construct a consistency boundary, we need to prevent the situation where other threads pick up only some of the events, but not all of them, which could present the aggregate in an inconsistent, or unusual, and perhaps unworkable state.

In other words, we need to avoid the situation where some of the events have been stored successfully but others have not been. If the events from a command were stored in a series of independent database transactions, then some would be written before others. If another thread needs the aggregate and gets its events whilst a series of new event are being written, it would not receive some of the events, but not the events that have not yet been written. Worse still, events could be lost due to an inconvenient database server problem, or sudden termination of the client. Even worse, later events in the series could fall into conflict because another thread has started appending events to the same sequence, potentially causing an incoherent state that would be difficult to repair.

Therefore, to implement the aggregate as a consistency boundary, all the events from a command on an aggregate 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. If all the events from an aggregate are to be written to a database as a single atomic operation, then they must have been published by the entity as a single list.

The library has a domain entity class called `AggregateRoot` that can be useful in a domain driven design, especially where a single command can cause many events to be published. The `AggregateRoot` entity class extends `TimestampedVersionedEntity`. It overrides the `__publish__()` method of the base class, so that triggered events are published only to a private list of pending events, rather than directly to the publish-subscribe mechanism. It also adds a method called `__save__()`, which publishes all pending events to the publish-subscribe mechanism as a single list.

It can be subclassed by custom aggregate root entities. In the example below, the entity class `World` inherits from `AggregateRoot`.

```
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_event__(World.SomethingHappened, what=something)

    class SomethingHappened(AggregateRoot.Event):
```

```
def mutate(self, obj):
    obj.history.append(self)
```

The World aggregate root has a command method `make_things_so()` which publishes `SomethingHappened` events. The `mutate()` method of the `SomethingHappened` class simply appends the event (`self`) to the aggregate object `obj`.

We can see the events that are published by subscribing to the handler `receive_events()`.

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

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

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

# State of aggregate object has changed
# but no events have been published yet.
assert len(received_events) == 0
assert world.history[0].what == 'dinosaurs'
assert world.history[1].what == 'trucks'
assert world.history[2].what == 'internet'
```

Events are pending, and will not be published until the `__save__()` method is called.

```
# Has pending events.
assert len(world.__pending_events__) == 4

# Publish pending events.
world.__save__()

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

# No longer any pending events.
assert len(world.__pending_events__) == 0

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

1.8 Application

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

- *Overview*
- *Simple application*
- *Custom application*

- *Run the code*
- *Stored events*
- *Sequenced items*
- *Database records*
- *Close*

1.8.1 Overview

An application object normally has repositories and policies. A repository allows aggregates to be retrieved by ID, using a dictionary-like interface. Whereas aggregates implement commands that publish events, obversely, policies subscribe to events and then execute commands as events are received. An application can be well understood by understanding its policies, aggregates, commands, and events.

An application object can have methods (“application services”) which provide a relatively simple interface for client 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.

To run the examples below, please install the library with the ‘sqlalchemy’ option.

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

1.8.2 Simple application

The library provides a simple application class `SimpleApplication` which can be constructed directly.

Its `uri` attribute is an SQLAlchemy-style database connection string. An SQLAlchemy thread-scoped session facade will be setup using the `uri` value.

```
uri = 'sqlite:///memory:'
```

As you can see, 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; or for a MySQL database. See SQLAlchemy’s `create_engine()` documentation for details. You may need to install drivers for your database management system (such as `psycopg2` or `mysqlclient`).

```
sqlite:///tmp/mydatabase
postgresql://scott:tiger@localhost:5432/mydatabase
mysql://scott:tiger@hostname/dbname
```

Encryption is optionally enabled in `SimpleApplication` with a suitable AES key (16, 24, or 32 random bytes encoded as Base64).

```
from eventsourcing.utils.random import encode_random_bytes

# Keep this safe (random bytes encoded with Base64).
cipher_key = encode_random_bytes(num_bytes=32)
```

These values can be given to the application object as constructor arguments `uri` and `cipher_key`. Alternatively, the `uri` value can be set as environment variable `DB_URI`, and the `cipher_key` value can be set as environment variable `CIPHER_KEY`.

```
from eventsourcing.application.simple import SimpleApplication

app = SimpleApplication(
    uri='sqlite:///memory:',
    cipher_key=cipher_key
)
```

As an alternative to providing a URI, an already existing SQLAlchemy session can be passed in with a constructor argument called `session`, for example a session object provided by a framework extension such as [Flask-SQLAlchemy](#).

Once constructed, the `SimpleApplication` will have an event store, provided by the library's `EventStore` class, for which it uses the library's infrastructure classes for SQLAlchemy.

```
app.event_store
```

The `SimpleApplication` uses the library function `construct_sqlalchemy_eventstore()` to construct its event store, for integer-sequenced items with SQLAlchemy.

To use different infrastructure for storing events, subclass the `SimpleApplication` class and override the method `setup_event_store()`. You can read about the available alternatives in the [infrastructure layer](#) documentation.

The `SimpleApplication` also has a persistence policy, provided by the library's `PersistencePolicy` class.

```
app.persistence_policy
```

The persistence policy appends domain events to its event store whenever they are published.

The `SimpleApplication` also has a repository, an instance of the library's `EventSourcedRepository` class.

```
app.repository
```

Both the repository and persistence policy use the event store.

The aggregate repository is generic, and can retrieve all aggregates in an application, regardless of their class.

The `SimpleApplication` can be used as a context manager. The example below uses the `AggregateRoot` class directly to create a new aggregate object that is available in the application's repository.

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

obj = AggregateRoot.__create__()
obj.__change_attribute__(name='a', value=1)
assert obj.a == 1
obj.__save__()

# Check the repository has the latest values.
copy = app.repository[obj.id]
assert copy.a == 1

# Check the aggregate can be discarded.
copy.__discard__()
assert copy.id not in app.repository

# Check optimistic concurrency control is working ok.
from eventsourcing.exceptions import ConcurrencyError
```

```
try:
    obj.__change_attribute__(name='a', value=2)
    obj.__save__()
except ConcurrencyError:
    pass
else:
    raise Exception("Shouldn't get here")
```

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 cause a `ConcurrencyError` exception to be raised.

The `SimpleApplication` has a `notification_log` attribute, which can be used to follow the application events as a single sequence.

```
# Follow application event notifications.
from eventsourcing.interface.notificationlog import NotificationLogReader
reader = NotificationLogReader(app.notification_log)
notification_ids = [n['id'] for n in reader.read()]
assert notification_ids == [1, 2, 3], notification_ids

# - create two more aggregates
obj = AggregateRoot.__create__()
obj.__save__()

obj = AggregateRoot.__create__()
obj.__save__()

# - get the new event notifications from the reader
notification_ids = [n['id'] for n in reader.read()]
assert notification_ids == [4, 5], notification_ids
```

1.8.3 Custom application

The `SimpleApplication` class can be extended.

The example below shows a custom application class `MyApplication` that extends `SimpleApplication` with application service `create_aggregate()` that can create new `CustomAggregate` entities.

```
class MyApplication(SimpleApplication):
    def create_aggregate(self, a):
        return CustomAggregate.__create__(a=1)
```

The application code above depends on an entity class called `CustomAggregate`, which is defined below. It extends the library's `AggregateRoot` entity with an event sourced, mutable attribute `a`.

```
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):
        """Mutable attribute a."""
```

For more sophisticated domain models, please read about the custom entities, commands, and domain events that can be developed using classes from the library's *domain model layer*.

Run the code

The custom application object can be constructed.

```
# Construct application object.
app = MyApplication(uri='sqlite:///memory:')
```

The application service aggregate factor method `create_aggregate()` can be called.

```
# Create aggregate using application service, and save it.
aggregate = app.create_aggregate(a=1)
aggregate.__save__()
```

Existing aggregates can be retrieved by ID using the repository's dictionary-like interface.

```
# Aggregate is in the repository.
assert aggregate.id in app.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 once pending events have been published.

```
# Change attribute value.
aggregate.a = 2
aggregate.a = 3

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

# Retrieve again from repository.
aggregate = app.repository[aggregate.id]

# Check attribute has new value.
assert aggregate.a == 3
```

The aggregate can be discarded. After being saved, a discarded aggregate will no longer be available in the repository.

```
# Discard the aggregate.
aggregate.__discard__()

# Check discarded aggregate no longer exists in repository.
assert aggregate.id not in app.repository
```

Attempts to retrieve an aggregate that does not exist will cause a `KeyError` to be raised.

```
# Fail to get aggregate from dictionary-like interface.
try:
    app.repository[aggregate.id]
except KeyError:
    pass
```

```
else:
    raise Exception("Shouldn't get here")
```

Stored events

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

```
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, by using the method `get_items()` of the event store's record manager.

```
items = app.event_store.record_manager.list_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 '"a":1' in items[0].state, items[0].state
assert '"timestamp":' in items[0].state

assert items[1].originator_id == aggregate.id
assert items[1].event_type == 'eventsourcing.domain.model.aggregate#AggregateRoot.
↳AttributeChanged'
assert '"name": "_a"' in items[1].state
assert '"timestamp":' in items[1].state

assert items[2].originator_id == aggregate.id
assert items[2].event_type == 'eventsourcing.domain.model.aggregate#AggregateRoot.
↳AttributeChanged'
assert '"name": "_a"' in items[2].state
assert '"timestamp":' in items[2].state

assert items[3].originator_id == aggregate.id
```

```
assert items[3].event_type == 'event sourcing.domain.model.aggregate#AggregateRoot.
↳ Discarded'
assert '"timestamp":' in items[3].state
```

In this example, the `cipher_key` was not set, so the stored data is visible.

Database records

Of course, it is also possible to just use the record class directly to obtain records. After all, it's just an SQLAlchemy ORM object.

```
app.event_store.record_manager.record_class
```

The `query` property of the SQLAlchemy record manager is a convenient way to get a query object from the session for the record class.

```
event_records = app.event_store.record_manager.query.all()

assert len(event_records) == 4
```

Close

If the application isn't being used as a context manager, then 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, such as when this documentation is tested as part of the library's test suite).

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

1.9 Snapshotting

Snapshots provide a fast path for obtaining the state of an entity or aggregate that skips replaying some or all of the entity's events.

If the library repository class `EventSourcedRepository` is constructed with a snapshot strategy object, it will try to get the closest snapshot to the required version of a requested entity, and then replay only those events that will take the snapshot up to the state at that version.

Snapshots can be taken manually. To automatically generate snapshots, a snapshotting policy can take snapshots whenever a particular condition occurs, for example after every ten events.

- *Domain*
- *Application*
- *Run the code*

1.9.1 Domain

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

```
from event sourcing.example.domainmodel import Example, create_new_example
```

1.9.2 Application

The library class `SnapshottingApplication`, extends `SimpleApplication` by setting up infrastructure for snapshotting, such as a snapshot store, a dedicated table for snapshots, and a policy to take snapshots every so many events. It is recommended not to co-mingle saved snapshots with the entity event sequence. It can also make sense to use a completely separate table for snapshot records, especially if the table of domain events is also being used as application log (e.g. it has an auto-incrementing integer primary key column).

```
from event sourcing.application.simple import SnapshottingApplication
```

1.9.3 Run the code

In the example below, snapshots of entities are taken every period number of events.

```
with SnapshottingApplication(period=2) as app:

    # Create an entity.
    entity = 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.repository
    assert app.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.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.repository
snapshot = app.snapshot_strategy.get_snapshot(entity.id)
assert snapshot.state is None

try:
    app.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__'] == 1 # one behind
assert snapshot.state['_foo'] == 'bar2'

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

# Get historical entities.
entity = app.repository.get_entity(entity.id, at=0)
assert entity.__version__ == 0
assert entity.foo == 'bar1', entity.foo

entity = app.repository.get_entity(entity.id, at=1)
assert entity.__version__ == 1
assert entity.foo == 'bar2', entity.foo

entity = app.repository.get_entity(entity.id, at=2)
assert entity.__version__ == 2
assert entity.foo == 'bar3', entity.foo

entity = app.repository.get_entity(entity.id, at=3)
assert entity.__version__ == 3
assert entity.foo == 'bar4', entity.foo

```

1.10 Stand-alone example

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*
 - *Domain events*
 - *Publish-subscribe*
 - *Domain entity*

- *Mutator function*
- *Run the code*
- *Infrastructure*
 - *Database table*
 - *Event store*
 - *Entity repository*
 - *Run the code*
 - *Sequenced items*
- *Application*
 - *Persistence policy*
 - *Application object*
- *Run the code*

1.10.1 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):
    """
    Supertype for domain event objects.
    """
    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, an if-else block is used to structure the mutator function. The library has a function decorator `mutator()` that allows a default mutator function to register handlers for different types of event, much 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'
```

1.10.2 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).

To run the code in this section, please install the library with the ‘sqlalchemy’ option.

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

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

Base = declarative_base()

class IntegerSequencedRecord(Base):
    __tablename__ = 'integer_sequenced_items'

    id = Column(BigInteger().with_variant(Integer, "sqlite"), primary_key=True)

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

    # Position (index) of item in sequence.
    position = Column(BigInteger(), nullable=False)

    # 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', unique=True),
```

The library has a class `IntegerSequencedRecord` 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=Base,
    settings=SQLAlchemySettings(uri='sqlite:///memory:'),
)

datastore.setup_connection()
datastore.setup_table(IntegerSequencedRecord)
```

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
```

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 a record manager to persist the sequenced items into a particular database management system. The record manager uses a record class to manipulate records in a particular database table.

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

```
from eventsourcing.infrastructure.eventstore import EventStore
from eventsourcing.infrastructure.sqlalchemy.manager import SQLAlchemyRecordManager
from eventsourcing.infrastructure.sequenceditemmapper import SequencedItemMapper

record_manager = SQLAlchemyRecordManager(
    session=datastore.session,
    record_class=IntegerSequencedRecord,
)

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

event_store = EventStore(
    record_manager=record_manager,
    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`.

In this example, the repository is given an event store object. The repository is also given the mutator function `mutate()` defined above.

```
from eventsourcing.infrastructure.eventsourcedrepository import EventSourcedRepository

example_repository = EventSourcedRepository(
    event_store=event_store,
    mutator_func=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 record manager. 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.record_manager.list_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
```

1.10.3 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(
            record_manager=SQLAlchemyRecordManager(
                record_class=IntegerSequencedRecord,
                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_func=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.

1.10.4 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
```

1.11 Notifications

This section discusses how to use notifications to propagate the domain events of an application.

Basically, if the domain events of an application can be placed in a single sequence, the application sequence can be propagated as notifications in a standard way, and projections can be updated as the notifications are received.

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 suggests the simple logic of an ascending sequence of integers can allow others to progress along the application’s sequence of events.

It is the third option that is pursued below.

- *Application sequence*
 - *Timestamps*
 - *Contiguous integers*
 - *Record managers*
 - *BigArray*
- *Notification logs*
- *Local notification logs*
 - *RecordManagerNotificationLog*
 - *BigArrayNotificationLog*
- *Remote notification logs*
 - *NotificationLogView*
 - *Notification API*
 - *RemoteNotificationLog*
- *Notification log reader*
- *Updating projections*
 - *Synchronous update*
 - *Asynchronous update*

Before continuing with code examples, let's setup an event store, and a database, which are needed by the examples below.

```
from eventsourcing.infrastructure.sqlalchemy.manager import SQLAlchemyRecordManager
from eventsourcing.infrastructure.sqlalchemy.records 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

# Setup the database.
datastore = SQLAlchemyDatastore(
    settings=SQLAlchemySettings(),
    tables=[StoredEventRecord],
)
datastore.setup_connection()
datastore.setup_tables()

# Setup the record manager.
record_manager = SQLAlchemyRecordManager(
    session=datastore.session,
    record_class=StoredEventRecord,
    sequenced_item_class=StoredEvent,
```

```

    contiguous_record_ids=True,
)

# Setup a sequenced item mapper.
sequenced_item_mapper = SequencedItemMapper(
    sequenced_item_class=StoredEvent,
)

# Setup the event store.
event_store = EventStore(
    record_manager=record_manager,
    sequenced_item_mapper=sequenced_item_mapper
)

# Set up a persistence policy.
persistence_policy = PersistencePolicy(
    event_store=event_store,
)

```

Please note, the SQLAlchemyRecordManager has its `contiguous_record_ids` option enabled.

The infrastructure classes are explained in other sections of this documentation.

1.11.1 Application sequence

The fundamental concern here is to accomplish perfect accuracy when propagating the events of an application, so that events are not missed, duplicated, jumbled, or unnecessarily delayed.

The events of an application sequence could be sequenced with timestamps and integers. Sequencing the application events by timestamp is supported by the relational timestamp sequenced record classes, in that their position column is indexed. However, the notification logs only work with integer sequences.

Timestamps

If time itself was ideal, then timestamps would be ideal. Each event could then have a timestamp that could be used to index and iterate through the events of the application. However, there are many clocks, and each runs slightly differently from the other.

If the timestamps of the application events are created by different clocks, then it is possible to write events in an order that creates consistency errors when reconstructing the application state. Hence it is also possible for new records to be written with a timestamp that is earlier than the latest one, which makes following the application sequence tricky.

A “jitter buffer” can be used, otherwise any events timestamped by a relatively retarded clock, and hence positioned behind events that were inserted earlier, could be missed. The delay, or the length of the buffer, must be greater than the differences between clocks, but how do we know for sure what is the maximum difference between the clocks?

Of course, there are lots of remedies. Clocks can be synchronised, more or less. A timestamp server could be used, and hybrid monotonically increasing timestamps can be implemented. Furthermore, the risk of simultaneous timestamps can be mitigated by using a random component to the timestamp, as with UUID v1, at the expense of randomizing the order of otherwise simultaneous events.

Such techniques are common, widely discussed, and entirely legitimate approaches to the complications encountered when using timestamps to sequence events. The big advantage of using timestamps is that you don’t need to generate a sequence of integers, and applications can be distributed and scaled without performance being limited by a fragile single-threaded auto-incrementing integer-sequence bottleneck.

In support of this approach, the library's relational record classes for timestamp sequenced items, in particular the `TimestampSequencedRecord` classes for SQLAlchemy and Django, index their position field, which is a timestamp, and so this index can be used to get all application events in certain order. Following this sequence will be as reliable as the timestamps given to the events. So if you use this class in this way, do make sure your clocks are in sync.

Todo: Code example.

(An improvement could be to have a timestamp field that is populated by the database server, and index that instead of the application event's timestamp which would vary according to the variation between the clock of application servers. Code changes and other suggestions are always welcome.)

Contiguous integers

To make propagation perfectly accurate (which is defined here as reproducing the application's sequence of events perfectly, without any risk of gaps or duplicates or jumbled items, or race conditions), we can generate and follow a contiguous sequence of integers. Two such techniques are described below.

The first technique uses the library's relational record managers with record classes that have an indexed integer ID column. Record IDs can be used to place all the application's event records in a single sequence. This technique is recommended for enterprise applications, and at least the earlier stages of more ambitious projects. There is an inherent limit to the rate at which an application can write events using this technique, which essentially follows from the need to write events in series. The rate limit is the reciprocal of the time it takes to write one event record.

Given the rate limit, it would take quite a long time to fill up a properly provisioned database table. Nevertheless, if the volume of domain event records in your system inclines you towards partitioning the table of stored events, or if your database works in this way (Cassandra) the table would need to be partitioned by sequence ID, and so maintaining an index of record IDs across such partitions, and hence sequencing the events of an application in this way, will be problematic.

To proceed without an indexed record ID column, the library class `BigArray` can be used to sequence all the events of an application. This technique can be used as an alternative to using a native database index of record IDs, especially in situations where a normal database index across all records is generally discouraged (e.g. in Cassandra), or where records do not have an integer ID or timestamp that can be indexed (e.g. all the library's record classes for Cassandra, and the `IntegerSequencedNoIDRecord` for SQLAlchemy, or when storing an index for a large number of records in a single partition is undesirable for infrastructure or performance reasons, or is not supported by the database.

Record managers

A relational record manager can function as an application sequence, especially when its record class has an ID field, and more so when the `contiguous_record_ids` option is enabled. This technique ensures that whenever an entity or aggregate command returns successfully, any events will already have been simultaneously placed in both the aggregate's and the application's sequence. Importantly, if inserting an event hits a uniqueness constraint and the transaction is rolled back, the event will not appear in either sequence.

This approach provides perfect accuracy with great simplicity for followers, but it has a maximum total rate at which records can be written into the database. In particular, the `contiguous_record_ids` feature executes an "insert select from" SQL statement that generates contiguous record IDs when records are inserted, on the database-side as a clause in the insert statement, by selecting the maximum existing ID in the table, adding one, and inserting that value, along with the event data.

Because the IDs must be unique, applications may experience the library's `ConcurrencyError` exception if they happen to insert records simultaneously with others. Record ID conflicts are retried a finite number of times by the library before a `ConcurrencyError` exception is raised. But with a load beyond the capability of a service, increased congestion will be experienced by the application as an increased frequency of concurrency errors.

Please note, without the `contiguous_record_ids` feature enabled, the ID columns of library record classes should be merely auto-incrementing, and so the record IDs can anyway be used to get all the records in the order they were written. However, auto-incrementing the ID can lead to a sequence of IDs that has gaps, a non-contiguous sequence, which could lead to race conditions and missed items. The gaps would need to be negotiated, which is relatively complicated. To keep things relatively simple, a record manager that does not have the `contiguous_record_ids` feature enabled cannot be used with the library's `RecordManagerNotificationLog` class (introduced below). If you want to sequence the application events with a non-contiguous sequence, then you will need to write something that can negotiate the gaps.

To use contiguous IDs to sequence the events of an application, simply use a relational record manager with an `IntegerSequencedRecord` that has an ID, such as the `StoredEventRecord` record class, and with a `True` value for its `contiguous_record_ids` constructor argument. The `record_manager` above was constructed in this way. The records can then be obtained using the `all_records()` method of the record manager. The record IDs will form a contiguous sequence, suitable for the `RecordManagerNotificationLog`.

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

all_records = record_manager.all_records()

assert len(all_records) == 0, all_records

first_entity = VersionedEntity.__create__()

all_records = record_manager.all_records(start=0, stop=5)

assert len(all_records) == 1, all_records
```

The local notification log class `RecordManagerNotificationLog` (see below) can adapt record managers, presenting the application sequence as notifications in a standard way.

BigArray

This is a long section, and can be skipped if you aren't currently required to scale beyond the limits of a database table that has indexed record IDs.

To support ultra-high capacity requirements, the application sequence must be capable of having a very large number of events, neither swamping an individual database partition (in Cassandra) nor distributing things across table 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 for normal usage.

The library class `BigArray` satisfies these requirements quite well, by spanning across many such partitions. 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, limited in 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 default 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. The performance can be improved by using an integer sequence generator, but departing from using the current max ID risks creating gaps in the sequence of IDs.

```
from uuid import uuid4
from eventsourcing.domain.model.array import BigArray
from eventsourcing.infrastructure.repositories.array import BigArrayRepository

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

big_array = repo[uuid4()]
big_array.append('item0')
big_array.append('item1')
big_array.append('item2')
big_array.append('item3')
```

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 big_array.get_next_position() == 4

big_array[4] = 'item4'
try:
    big_array[4] = 'item4a'
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 factoring out 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, for example during development, 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 by 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 big array object.

```
big_array[next(integers)] = 'item5'
big_array[next(integers)] = 'item6'

assert big_array.get_next_position() == 7
```

Items can be read from the big array 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 big_array[0] == 'item0'
assert list(big_array[5:7]) == ['item5', 'item6']
```

The big array can be written to by a persistence policy. References to events could be assigned to the big array 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 sequence. To do that, construct the application logging policy object before the normal application persistence policy. Also, make sure the application log policy excludes the events published by the big array (otherwise there will be an infinite recursion). If the event fails to write, then the application sequence will have a dangling reference, which followers will have to cope with.

Alternatively, if the database supports transactions across different tables (not Cassandra), the big array assignment and the event record insert can be done in the same transaction, so they both appear or neither does. This will help to avoid some complexity for followers. The library currently doesn't have any code that writes to both in the same transaction.

Todo: Code example of policy that places application domain events in a big array.

If the big array item is not assigned in the same separate transaction as the event record is inserted, commands that fail to insert the event record after the event has been assigned to the big array (due to an operation error or a concurrency error) should probably raise an exception, so that the command is seen to have failed and so may be retried. An event would then be in the application sequence but not in the aggregate sequence, which is effectively a dangling reference, one that may or may not be satisfied later. If the event record insert failed due to an operational error, and the command is retried, a new event at the same position in the same sequence may be published, and so it would appear twice in the application sequence. And so, whilst dangling references in the application log can perhaps be filtered out by followers after a delay, care should be taken by followers to deduplicate events.

It may also happen that an item fails to be assigned to the big array. In this case, an ID that was issued by an integer sequence generator would be lost. The result would be a gap, that would need to be negotiated by followers.

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.

Since we can imagine there is quite a lot of noise in the sequence, it may be useful to process the application sequence within the context by constructing another sequence that does not have duplicates or gaps, and then propagating that sequence.

The local notification log class `BigArrayNotificationLog` (see below) can adapt big arrays, presenting the assigned items as notifications in a standard way. Gaps in the array will result in notification items of `None`. But where there are gaps, there can be race conditions, where the gaps are filled. Only a contiguous sequence, which has no gaps, can exclude gaps being filled later.

1.11.2 Notification logs

As described in *Implementing Domain Driven Design*, a notification log presents a sequence of notification items in linked sections.

Sections are obtained from a notification log using Python's "square brackets" sequence index syntax. The key is a section ID. A special section ID called "current" can be used to obtain a section which contains the latest notification (see examples below).

Each section contains a limited number items, the size is fixed by the notification log's `section_size` constructor argument. When the current section is full, it is considered to be an archived section.

All the archived sections have an ID for the next section. Similarly, all sections except the first have an ID for the previous section.

A client can get the current section, go back until it reaches the last notification it has already received, and then go forward until all existing notifications have been received.

The section ID numbering scheme follows Vaughn Vernon's book. Section IDs are strings: a string formatted with two integers separated by a comma. The integers represent the first and last number of the items included in a section.

The classes below can be used to present a sequence of items, such the domain events of an application, in linked sections. They can also be used to present other sequences for example a projection of the application sequence, where the events are rendered in a particular way for a particular purpose, such as analytics.

A local notification log could be presented by an API in a serialized format e.g. JSON or Atom XML. A remote notification log could then access the API and provide notification items in the standard way. The serialized section documents could then be cached using standard cacheing mechanisms.

1.11.3 Local notification logs

RecordManagerNotificationLog

A relational record manager can be adapted by the library class *RecordManagerNotificationLog*, which will then present the application's events as notifications.

The *RecordManagerNotificationLog* is constructed with a *record_manager*, and a *section_size*.

```
from eventsourcing.interface.notificationlog import RecordManagerNotificationLog

# Construct notification log.
notification_log = RecordManagerNotificationLog(record_manager, section_size=5)

# Get the "current" section from the record notification log.
section = notification_log['current']
assert section.section_id == '6,10', section.section_id
assert section.previous_id == '1,5', section.previous_id
assert section.next_id == None
assert len(section.items) == 4, len(section.items)

# Get the first section from the record notification log.
section = notification_log['1,5']
assert section.section_id == '1,5', section.section_id
assert section.previous_id == None, section.previous_id
assert section.next_id == '6,10', section.next_id
assert len(section.items) == 5, len(section.items)
```

The sections of the record notification log each have notification items that reflect the recorded domain event. The items (notifications) in the sections from *RecordManagerNotificationLog* are Python dicts with three key-values: *id*, *topic*, and *data*.

The record manager uses its *sequenced_item_class* to identify the actual names of the record fields containing the topic and the data, and constructs the notifications (the dicts) with the values of those fields. The notification's data is simply the record data, so if the record data was encrypted, the notification data will also be encrypted. The keys of the event notification do not reflect the sequenced item class being used in the record manager.

The *topic* value can be resolved to a Python class, such as a domain event class. An object instance, such as a domain event object, can then be reconstructed using the notification's *data*.

In the code below, the function *resolve_notifications* shows how that can be done (this function doesn't exist in the library).

```
def resolve_notifications(notifications):
    return [
        sequenced_item_mapper.from_topic_and_data(
            topic=notification['topic'],
            data=notification['data']
        ) for notification in notifications
    ]

# Resolve a section of notifications into domain events.
domain_events = resolve_notifications(section.items)

# Check we got the first entity's "created" event.
assert isinstance(domain_events[0], VersionedEntity.Created)
assert domain_events[0].originator_id == first_entity.id
```

If the notification data was encrypted by the sequenced item mapper, the sequence item mapper will decrypt the data before reconstructing the domain event. In this example, the sequenced item mapper does not have a cipher, so the notification data is not encrypted.

The library's SimpleApplication has a notification_log that uses this RecordManagerNotificationLog class.

BigArrayNotificationLog

You can skip this section if you skipped the section about BigArray.

A big array can be adapted by the library class *BigArrayNotificationLog*, which will then present the items assigned to the array as notifications.

The BigArrayNotificationLog is constructed with a big_array, and a section_size.

```
from eventsourcing.interface.notificationlog import BigArrayNotificationLog

# Construct notification log.
big_array_notification_log = BigArrayNotificationLog(big_array, section_size=5)

# Get the "current" section from the big array notification log.
section = big_array_notification_log['current']
assert section.section_id == '6,10', section.section_id
assert section.previous_id == '1,5', section.previous_id
assert section.next_id == None
assert len(section.items) == 2, len(section.items)

# Check we got the last two items assigned to the big array.
assert section.items == ['item5', 'item6']

# Get the first section from the notification log.
section = big_array_notification_log['1,10']
assert section.section_id == '1,5', section.section_id
assert section.previous_id == None, section.previous_id
assert section.next_id == '6,10', section.next_id
assert len(section.items) == 5, len(section.items)

# Check we got the first five items assigned to the big array.
assert section.items == ['item0', 'item1', 'item2', 'item3', 'item4']
```

Please note, for simplicity, the items in this example are just strings ('item0' etc). If the big array is being used to sequence the events of an application, it is possible to assign just the item's sequence ID and position, and let followers

get the actual event using those references.

1.11.4 Remote notification logs

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.

This library has a pair of classes that can help to present a notification log remotely.

The `RemoteNotificationLog` class has the same interface for getting sections as the local notification log classes described above, but instead of using a local datasource, it requests serialized sections from a Web API.

The `NotificationLogView` class serializes sections from a local notification log, and can be used to implement a Web API that presents notifications to a network.

Alternatively to presenting domain event data and topic information, a Web API could present only the event's sequence ID and position values, requiring clients to obtain the domain event from the event store using those references. If the notification log uses a big array, and the big array is assigned with only sequence ID and position values, the big array notification log could be used directly with the `NotificationLogView` to notify of domain events by reference rather than by value. However, if the notification log uses a record manager, then a notification log adapter would be needed to convert the events into the references.

If a notification log would then receive and would also return only sequence ID and position information to its caller. The caller could then proceed by obtaining the domain event from the event store. Another adapter could be used to perform the reverse operation: adapting a notification log that contains references to one that returns whole domain event objects. Such adapters are not currently provided by this library.

NotificationLogView

The library class `NotificationLogView` presents sections from a local notification log, and can be used to implement a Web API.

The `NotificationLogView` class is constructed with a local `notification_log` object and an optional `json_encoder_class` (which defaults to the library's `ObjectJSONEncoder` class, used explicitly in the example below).

The example below uses the record notification log, constructed above.

```
import json

from eventsourcing.interface.notificationlog import NotificationLogView
from eventsourcing.utils.transcoding import ObjectJSONEncoder, ObjectJSONDecoder

view = NotificationLogView(
    notification_log=notification_log,
    json_encoder_class=ObjectJSONEncoder
)

section_json, is_archived = view.present_section('1,5')

section_dict = json.loads(section_json, cls=ObjectJSONDecoder)

assert section_dict['section_id'] == '1,5'
assert section_dict['next_id'] == '6,10'
assert section_dict['previous_id'] == None
assert section_dict['items'] == notification_log['1,5'].items
assert len(section_dict['items']) == 5
```

```

item = section_dict['items'][0]
assert item['id'] == 1
assert '__event_hash__' in item['data']
assert item['topic'] == 'eventsourcing.domain.model.entity#VersionedEntity.Created'

assert section_dict['items'][1]['topic'] == 'eventsourcing.domain.model.array'
↪#ItemAssigned'
assert section_dict['items'][2]['topic'] == 'eventsourcing.domain.model.array'
↪#ItemAssigned'
assert section_dict['items'][3]['topic'] == 'eventsourcing.domain.model.array'
↪#ItemAssigned'
assert section_dict['items'][4]['topic'] == 'eventsourcing.domain.model.array'
↪#ItemAssigned'

# Resolve the notifications to domain events.
domain_events = resolve_notifications(section_dict['items'])

# Check we got the first entity's "created" event.
assert isinstance(domain_events[0], VersionedEntity.Created)
assert domain_events[0].originator_id == first_entity.id

```

Notification API

A Web application could identify a section ID from an HTTP request path, and respond by returning an HTTP response with JSON content that represents that section of a notification log.

The example uses the library's `NotificationLogView` to serialize the sections of the record notification log (see above).

```

def notification_log_wsgi(envIRON, start_response):

    # Identify section from request.
    section_id = environ['PATH_INFO'].strip('/')

    # Construct notification log view.
    view = NotificationLogView(notification_log)

    # Get serialized section.
    section, is_archived = view.present_section(section_id)

    # Start HTTP response.
    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]
    start_response(status, headers)

    # Return body.
    return [(line + '\n').encode('utf8') for line in section.split('\n')]

```

A more sophisticated application might include an ETag header when responding with the current section, and a Cache-Control header when responding with archived sections.

A more standard approach would be to use Atom (`application/atom+xml`) which is a common standard for producing RSS feeds and thus a great fit for representing lists of events, rather than `NotificationLogView`. However, just as this library doesn't (currently) have any code that generates Atom feeds from domain events, there isn't any code that reads domain events from atom feeds. So if you wanted to use Atom rather than `NotificationLogView` in

your API, then you will also need to write a version of `RemoteNotificationLog` that can work with your Atom API.

RemoteNotificationLog

The library class `RemoteNotificationLog` can be used in the same way as the local notification logs above. The difference is that rather than accessing a database using a `BigArray` or record manager, it makes requests to an API.

The `RemoteNotificationLog` class is constructed with a `base_url`, a `notification_log_id` and a `json_decoder_class`. The JSON decoder must be capable of decoding JSON encoded by the API. Hence, the JSON decoder must match the JSON encoder used by the API.

The default `json_decoder_class` is the library's `ObjectJSONDecoder`. This encoder matches the default `json_encoder_class` of the library's `NotificationLogView` class, which is the library's `ObjectJSONEncoder` class. If you want to extend the JSON encoder classes used here, just make sure they match, otherwise you will get decoding errors.

The `NotificationLogReader` can use the `RemoteNotificationLog` in the same way that it uses a local notification log object. Just construct it with a remote notification log object, rather than a local notification log object, then read notifications in the same way (as described above).

If the API uses a `NotificationLogView` to serialise the sections of a local notification log, the remote notification log object functions effectively as a proxy for a local notification log on a remote node.

```
from eventsourcing.interface.notificationlog import RemoteNotificationLog

remote_notification_log = RemoteNotificationLog("base_url")
```

If a server were running at “base_url” the `remote_notification_log` would function in the same way as the local notification logs described above, returning section objects for section IDs using the square brackets syntax.

If the section objects were created by a `NotificationLogView` that had the `notification_log` above, we could obtain all the events of an application across an HTTP connection, accurately and without great complication.

See `test_notificationlog.py` for an example that uses a Flask app running in a local HTTP server to get notifications remotely using these classes.

1.11.5 Notification log reader

The library object class `NotificationLogReader` effectively functions as an iterator, yielding a continuous sequence of notifications that it discovers from the sections of a notification log, local or remote.

A notification log reader object will navigate the linked sections of a notification log, backwards from the “current” section of the notification log, until reaching the position it seeks. The position, which defaults to 0, can be set directly with the reader's `seek()` method. Hence, by default, the reader will navigate all the way back to the first section.

After reaching the position it seeks, the reader will then navigate forwards, yielding as a continuous sequence all the subsequent notifications in the notification log.

As it navigates forwards, yielding notifications, it maintains position so that it can continue when there are further notifications. This position could be persisted, so that position is maintained across invocations, but that is not a feature of the `NotificationLogReader` class, and would have to be added in a subclass or client object.

The `NotificationLogReader` supports slices. The position is set indirectly when a slice has a start index.

All the notification logs discussed above (local and remote) have the same interface, and can be used by `NotificationLogReader` progressively to obtain unseen notifications.

The example below happens to yield notifications from a big array notification log, but it would work equally well with a record notification log, or with a remote notification log.

Todo: Maybe just use “obj.read()” rather than “list(obj)”, so it’s more file-like.

```
from eventsourcing.interface.notificationlog import NotificationLogReader

# 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 = reader.read()
assert len(all_notifications) == 9

# Resolve the notifications to domain events.
domain_events = resolve_notifications(all_notifications)

# Check we got the first entity's created event.
assert isinstance(domain_events[0], VersionedEntity.Created)
assert domain_events[0].originator_id == first_entity.id

# Check the position has advanced.
assert reader.position == 9

# Read all subsequent notifications (should be none).
subsequent_notifications = list(reader)
assert subsequent_notifications == []

# Publish two more events.
VersionedEntity.__create__()
VersionedEntity.__create__()

# Read all subsequent notifications (should be two).
subsequent_notifications = reader.read()
assert len(subsequent_notifications) == 2

# Check the position has advanced.
assert reader.position == 11

# Read all subsequent notifications (should be none).
subsequent_notifications = reader.read()
len(subsequent_notifications) == 0

# Publish three more events.
VersionedEntity.__create__()
VersionedEntity.__create__()
last_entity = VersionedEntity.__create__()

# Read all subsequent notifications (should be three).
```

```
subsequent_notifications = reader.read()
assert len(subsequent_notifications) == 3

# Check the position has advanced.
assert reader.position == 14

# Resolve the notifications.
domain_events = resolve_notifications(subsequent_notifications)
last_domain_event = domain_events[-1]

# Check we got the last entity's created event.
assert isinstance(last_domain_event, VersionedEntity.Created), last_domain_event
assert last_domain_event.originator_id == last_entity.id

# Read all subsequent notifications (should be none).
subsequent_notifications = reader.read()
assert subsequent_notifications == []

# Check the position has advanced.
assert reader.position == 14
```

The position could be persisted, and the persisted value could be used to initialise the reader's position when reading is restarted.

In this way, the events of an application can be followed with perfect accuracy and without lots of complications. This seems to be an inherently reliable approach to following the events of an application.

1.11.6 Updating projections

Once the events of an application can be followed reliably, they can be used to update projections of the application state.

Synchronous update

You may wish to update a view of an aggregate synchronously whenever an event is published. You may wish simply to subscribe to the events of the aggregate. Then, whenever an event occurs, the projection can be updated.

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

The most simple implementation of a projection would consume an event synchronously as it is published by updating the view without considering whether the event was a duplicate or previous events were missed. This may be perfectly adequate for projecting events that are by design independent, such as tracking all 'Created' events so the extent aggregate IDs are available in a view.

It is also possible for a synchronous update to refer to an application log and catch up if necessary, perhaps after an error or because the projection is new to the application and needs to initialise.

Of course, it is possible to access aggregates and other views when updating a view, especially to avoid bloating events with redundant information that might be added to avoid such queries.

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

Todo: Code example showing "Projection" class using a notification log reader and (somehow) stateful position in the log, to follow application events and update a view.

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 somehow 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.

It is possible to use the decorator in a downstream application, in which domain events are republished following the application sequence asynchronously. The decorator would be called synchronously with the republishing of the event. In this case, if the view update routine somehow fails to update, the position of the downstream application in the upstream sequence would not advance until the view is restored to working order, after which the view will be updated as if there had been no failure.

Asynchronous update

Updates can be triggered by pushing notifications to messaging infrastructure, and having the remote components subscribe. De-duplication would involve tracking which events have already been received.

To keep the messaging infrastructure stable, it may be sufficient simply to identify the domain event, perhaps with its sequence ID and position.

If anything goes wrong with messaging infrastructure, such that a notification is sent but not received, remote components can detect they have missed a notification and pull the notifications they have missed. A pull mechanism, such as that described above, can be used to catch up.

The same mechanism can be used when materialized views (or other kinds of projections) are developed after the application has been initially deployed and require initialising from an established application sequence, or after changes need to be reinitialised from scratch, or updated after being offline for some reason.

Todo: Something about pumping events to a message bus, following the application sequence.

Todo: Something about republishing events in a downstream application that has subscribers such as the decorator above. Gives opportunity for sequence to be reconstructed in the application before being published (but then what if several views are updated and the last one fails? are they all updated in the same a transaction, are do they each maintain their own position in the sequence, or does the application just have one subscriber and one view?)

Todo: So something for a view to maintain its position in the sequence, perhaps version the view updates (event sourced or snapshots) if there are no transactions, or use a dedicated table if there are transactions.

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

1.12 Deployment

This section gives an overview of the concerns that arise when using an event sourcing 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 `event_sourcing.example.interface.flaskapp`, which is tested both stand-alone and with uWSGI.

- *Application object*
 - *Lazy initialization*
- *Database connection*
 - *SQLAlchemy*
 - *Cassandra*
- *Web interfaces*
 - *uWSGI*
 - *Flask*
 - * *Flask with SQLAlchemy*
 - * *Flask with Cassandra*
 - *Django*
 - * *Django ORM*
 - * *Django with Cassandra*
 - *Zope*
 - * *Zope with SQLAlchemy*
- *Task queues*
 - *Celery*
 - *Redis Queue*

1.12.1 Application object

In general you need one, and only one, instance of your application object in each process. If your event sourcing application object has any policies, for example if it 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 exception if `init_application()` has not already been called.

```
from event sourcing.application.simple import SimpleApplication

def construct_application(**kwargs):
    return SimpleApplication(**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)

def get_application():
    if _application is None:
        raise AssertionError("init_application() must be called first")
    return _application
```

The expected behaviour is demonstrated below.

```
try:
    get_application()
except AssertionError:
    pass
else:
    raise Exception("Shouldn't get here")

init_application()

get_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.

```
def close_application():
    global _application
    if _application is not None:
        _application.close()
        _application = None
```

The expected behaviour is demonstrated below.

```
close_application()
close_application()
```

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.

```
import threading

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

get_application()
get_application()
get_application()

close_application()
```

1.12.2 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.

1.12.3 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.

```
from uwsgidecorators import postfork

@postfork
def init_process():
    # Set up database connection.
    database = {}
    # Construct eventsourcing application.
    init_application()
```

Other decorators are available.

Flask

Flask with 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 record class(es) 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

# Construct Flask application.
application = Flask(__name__)

# Construct Flask-SQLAlchemy object.
```

```
db = SQLAlchemy(application)

# Define database table using Flask-SQLAlchemy library.
class StoredEventRecord(db.Model):
    __tablename__ = 'integer_sequenced_items'

    id = db.Column(db.BigInteger().with_variant(db.Integer, "sqlite"), primary_
↳key=True)

    # Sequence ID (e.g. an entity or aggregate ID).
    originator_id = db.Column(UUIDType(), nullable=False)

    # Position (index) of item in sequence.
    originator_version = db.Column(db.BigInteger(), nullable=False)

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

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

    # Index.
    __table_args__ = db.Index('index', 'originator_id', 'originator_version',
↳unique=True),

# Construct event sourcing application with Flask-SQLAlchemy table and session.
@application.before_first_request
def before_first_request():
    init_application(
        session=db.session,
        stored_event_record_class=StoredEventRecord,
    )
```

For a working example using Flask and SQLAlchemy, please refer to the library module `event_sourcing.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.

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
```

The [Flask-Cassandra](#) project serves a similar function to Flask-SQLAlchemy.

Django

When deploying an event sourcing application with Django, just remember that there must only be one instance of the application in any given process, otherwise its subscribers will be registered too many times. There are perhaps three different processes to consider. Firstly, running the test suite for your Django project or app. Secondly, running the Django project with WSGI (or equivalent). Thirdly, running the Django project from a task queue worker, such as RabbitMQ.

For the first case, if your application needs to be created fresh for each test, it is recommended to have a base test case class, which initialises the application during `setUp()` and closes the application during `tearDown()`. Another option is to use a yield fixture in pytest with the application object yielded whilst acting as a context manager. Just make sure the application is constructed once, and then closed if it is constructed again.

Of course if you only have one application object to test, then you could perhaps just create it at the start of the test suite. If so, closing the application doesn't matter, because no other application object will be created before the process ends.

For the second case, it is recommended to construct the application object from the project's `wsgi.py` file, which doesn't get used when running Django from a test suite, or from a task queue worker. Views can then get the application object freely. Closing the application doesn't matter, because it will be used until the process ends.

For the third case, it is recommended to construct the application in a suitable signal from the task queue framework, so that the application is constructed before request threads begin. Jobs can then get the application object freely. Closing the application doesn't matter, because it will be used until the process ends.

In each case, to make things very clear for other developers of your code, it is recommended to construct the application object with a module level function called `init_application()` that assigns to a module level variable, and then obtain the application object with another module level function called `get_application()`, which raises an exception if the application has not been constructed.

Django ORM

If you wish to use eventsourcing with Django ORM, the simplest way is having your application's event store use this library's `DjangoRecordManager`, and making sure the record classes (Django models) are included in your Django project. See [infrastructure doc](#) for more information.

The independent project [djangoevents](#) by [Applause](#) is a Django app that provides a more integrated approach to event sourcing in a Django project. It also uses the Django ORM to store events. Using `djangoevents` is well documented in its 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 much older version of this library which isn't recommended for new projects.

Django with Cassandra

If you wish to use eventsourcing with Django and Cassandra, regardless of any event sourcing, you may wish to use [Django-Cassandra](#). The library's Cassandra classes use the Cassandra Python library which the Django-Cassandra project integrates into Django. So you can easily develop an event sourcing application using the capabilities of this library, and then write views in Django, and use the Django-Cassandra project as a means of integrating Django as an Web interface to an event sourced application that uses 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 (which is what Django-Cassandra tries to make easy).

Zope

Zope with SQLAlchemy

The [Zope-SQLAlchemy](#) project serves a similar function to Flask-SQLAlchemy.

1.12.4 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.

1.13 Release notes

It is the aim of the project that releases with the same major version number are backwards compatible, within the scope of the documented examples. New major versions indicate a backward incompatible changes have been introduced since the previous major version.

Version 4.x series was released after quite a lot of refactoring made things backward-incompatible. Object namespaces for entity and event classes was cleaned up, by moving library names to double-underscore prefixed and postfixed names. Domain events can be hashed, and also hash-chained together, allowing entity state to be verified. Created events were changed to have `originator_topic`, which allowed other things such as mutators and repositories to be greatly simplified. Mutators are now by default expected to be implemented on entity event classes. Event timestamps were changed from floats to decimal objects, an exact number type. Cipher was changed to use AES-GCM to allow verification of encrypted data retrieved from a database.

Also, the record classes for SQLAlchemy were changed to have an auto-incrementing ID, to make it easy to follow the events of an application, for example when updating view models, without additional complication of a separate application log. This change makes the SQLAlchemy library classes ultimately less “scalable” than the Cassandra classes, because an auto-incrementing ID must operate from a single thread. Overall, it seems like a good trade-off for early-stage development. Later, when the auto-incrementing ID bottleneck would otherwise throttle performance, “scaling-up” could involve switching application infrastructure to use a separate application log.

Also, support for Django ORM was added in version 4.1.0.

Version 3.x series was a released after quite of a lot of refactoring made things backwards-incompatible. Documentation was greatly improved, in particular with pages reflecting the architectural layers of the library (infrastructure, domain, application).

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.

1.14 Module docs

This document describes the packages, modules, classes, functions and other code details of the library.

- [genindex](#)
- [modindex](#)

1.14.1 eventsourcing

The eventsourcing package contains packages for the application layer, the domain layer, the infrastructure layer, and the interface layer. There is also a module for exceptions, an example package, and a utils package.

application

The application layer brings together the domain and infrastructure layers.

base

```
class eventsourcing.application.base.ApplicationWithEventStores (entity_record_manager=None,  
                                                                log_record_manager=None,  
                                                                snap-  
                                                                shot_record_manager=None,  
                                                                al-  
                                                                ways_encrypt=False,  
                                                                cipher=None)
```

Bases: `object`

Event sourced application object class.

Can construct event stores using given database records. Supports three different event stores: for log events, for entity events, and for snapshot events.

```
close ()
```

```
construct_event_store (event_sequence_id_attr, event_position_attr, record_manager, al-  
                        ways_encrypt=False, cipher=None)
```

```
construct_sequenced_item_mapper (sequenced_item_class, event_sequence_id_attr,  
                                  event_position_attr, json_encoder_class=<class  
                                  'eventsourcing.utils.transcoding.ObjectJSONEncoder'>,  
                                  json_decoder_class=<class 'eventsourc-  
                                  ing.utils.transcoding.ObjectJSONDecoder'>, al-  
                                  ways_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 ()
```

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*)

```
class eventsourcing.application.policies.SnapshottingPolicy (repository,      pe-  
                                                         riod=2)
```

Bases: object

close()

condition (*event*)

take_snapshot (*event*)

simple

```
class eventsourcing.application.simple.SimpleApplication (persist_event_type=None,  
                                                         uri=None,          ses-  
                                                         sion=None,        ci-  
                                                         pher_key=None,  
                                                         stored_event_record_class=None,  
                                                         setup_table=True,  
                                                         contigu-  
                                                         ous_record_ids=True)
```

Bases: object

close()

drop_table()

setup_cipher (*cipher_key*)

setup_datastore (*session, uri*)

setup_event_store()

setup_persistence_policy (*persist_event_type*)

setup_repository (***kwargs*)

setup_table()

```
class eventsourcing.application.simple.SnapshottingApplication (period=10,  
                                                         snap-  
                                                         shot_record_class=None,  
                                                         **kwargs)
```

Bases: `eventsourcing.application.simple.SimpleApplication`

close()

setup_event_store()

setup_persistence_policy (*persist_event_type*)

```
setup_repository (**kwargs)
setup_table ()
```

domain.model

The domain layer contains a domain model, and optionally services that work across different entities or aggregates. The domain model package contains classes and functions that can help develop an event sourced domain model.

aggregate

Base classes for aggregates in a domain driven design.

```
class eventsourcing.domain.model.aggregate.AggregateRoot (**kwargs)
    Bases: eventsourcing.domain.model.entity.TimestampedVersionedEntity
    Root entity for an aggregate in a domain driven design.

class AttributeChanged (originator_version, **kwargs)
    Bases: eventsourcing.domain.model.aggregate.Event, eventsourcing.domain.
    model.entity.AttributeChanged
    Published when an AggregateRoot is changed.

class Created (originator_version=0, **kwargs)
    Bases: eventsourcing.domain.model.aggregate.Event, eventsourcing.domain.
    model.entity.Created
    Published when an AggregateRoot is created.

class Discarded (originator_version, **kwargs)
    Bases: eventsourcing.domain.model.aggregate.Event, eventsourcing.domain.
    model.entity.Discarded
    Published when an AggregateRoot is discarded.

class Event (originator_version, **kwargs)
    Bases: eventsourcing.domain.model.entity.Event
    Supertype for aggregate events.

__publish__ (event)
    Appends event to internal collection of pending events.

__save__ ()
    Publishes pending events for others in application.
```

array

A kind of collection, indexed by integer. Doesn't need to replay all events to exist.

```
class eventsourcing.domain.model.array.AbstractArrayRepository (array_size=10000,
                                                                *args,
                                                                **kwargs)
    Bases: eventsourcing.domain.model.entity.AbstractEntityRepository
    Repository for sequence objects.
```

`__getitem__ (array_id)`
Returns sequence for given ID.

class `eventsourcing.domain.model.array.AbstractBigArrayRepository`
Bases: `eventsourcing.domain.model.entity.AbstractEntityRepository`

Repository for compound sequence objects.

`__getitem__ (array_id)`
Returns sequence for given ID.

subrepo
Sub-sequence repository.

class `eventsourcing.domain.model.array.Array (array_id, repo)`
Bases: `object`

`__getitem__ (item)`
Returns item at index, or items in slice.

`__len__ ()`
Returns length of array.

`__setitem__ (index, item)`
Sets item in array, at given index.

Won't overrun the end of the array, because the position is fixed to be less than `base_size`.

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 partition. 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 base the underlying array size. Write time on average, and read time given an index, is constant 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.

__len__()

Returns length of array.

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 event sourcing.domain.model.array.ItemAssigned(*item, index, *args, **kwargs*)

Bases: event sourcing.domain.model.entity.Event

Occurs when an item is set at a position in an array.

index

item

collection

Decorators useful in domain models based on the classes in this library.

event sourcing.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.

event sourcing.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_attempts=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()
```

entity

Base classes for domain entities of different kinds.

The entity module provides base classes for domain entities.

class eventsourcing.domain.model.entity.**AbstractEntityRepository**

Bases: *eventsourcing.domain.model.entity.AbstractEventPlayer*

__contains__(entity_id)

Returns True or False, according to whether or not entity exists.

__getitem__(entity_id)

Returns entity for given ID.

event_store

Returns event store object used by this repository.

get_entity (*entity_id*, *at=None*)

Returns entity for given ID.

take_snapshot (*entity_id*, *lt=None*, *lte=None*)

Takes snapshot of entity state, using stored events. :return: Snapshot

class event sourcing.domain.model.entity.**AbstractEventPlayer**

Bases: object

class event sourcing.domain.model.entity.**DomainEntity** (*id*)

Bases: *event sourcing.domain.model.events.QualnameABC*

Base class for domain entities.

class **AttributeChanged** (***kwargs*)

Bases: event sourcing.domain.model.entity.Event, *event sourcing.domain.model.events.AttributeChanged*

Published when a DomainEntity is discarded.

class **Created** (*originator_topic*, ***kwargs*)

Bases: event sourcing.domain.model.entity.Event, *event sourcing.domain.model.events.Created*

Published when an entity is created.

originator_topic

class **Discarded** (***kwargs*)

Bases: *event sourcing.domain.model.events.Discarded*, event sourcing.domain.model.entity.Event

Published when a DomainEntity is discarded.

class **Event** (***kwargs*)

Bases: *event sourcing.domain.model.events.EventWithOriginatorID*, *event sourcing.domain.model.events.DomainEvent*

Supertype for events of domain entities.

__check_obj__ (*obj*)

Checks obj state before mutating.

__assert_not_discarded__ ()

Raises exception if entity has been discarded already.

__change_attribute__ (*name*, *value*)

Changes named attribute with the given value, by triggering an AttributeChanged event.

__discard__ ()

Discards self, by triggering a Discarded event.

__publish__ (*event*)

Publishes given event for subscribers in the application.

Parameters **event** – domain event or list of events

__publish_to_subscribers__ (*event*)

Actually dispatches given event to publish-subscribe mechanism.

Parameters **event** – domain event or list of events

__trigger_event__ (*event_class*, ***kwargs*)
Constructs, applies, and publishes a domain event.

id
Entity ID allows an entity instance to be referenced and distinguished from others, even though its state may change over time.

class `eventsourcing.domain.model.entity.TimestampedEntity` (*__created_on__*, ***kwargs*)

Bases: `eventsourcing.domain.model.entity.DomainEntity`

class `AttributeChanged` (***kwargs*)

Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.AttributeChanged`

Published when a `TimestampedEntity` is changed.

class `Created` (*originator_topic*, ***kwargs*)

Bases: `eventsourcing.domain.model.entity.Created`, `eventsourcing.domain.model.entity.Event`

Published when a `TimestampedEntity` is created.

class `Discarded` (***kwargs*)

Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.Discarded`

Published when a `TimestampedEntity` is discarded.

class `Event` (***kwargs*)

Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.events.EventWithTimestamp`

Supertype for events of timestamped entities.

__mutate__ (*obj*)

Update obj with values from self.

class `eventsourcing.domain.model.entity.TimestampedVersionedEntity` (*__created_on__*, ***kwargs*)

Bases: `eventsourcing.domain.model.entity.TimestampedEntity`, `eventsourcing.domain.model.entity.VersionedEntity`

class `AttributeChanged` (*originator_version*, ***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` (*originator_version=0*, ***kwargs*)

Bases: `eventsourcing.domain.model.entity.Created`, `eventsourcing.domain.model.entity.Created`, `eventsourcing.domain.model.entity.Event`

Published when a `TimestampedVersionedEntity` is created.

class `Discarded` (*originator_version*, ***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 (originator_version, **kwargs)
    Bases:      eventsourcing.domain.model.entity.Event,  eventsourcing.domain.
                model.entity.Event

    Supertype for events of timestamped, versioned entities.

class eventsourcing.domain.model.entity.TimeuuidedEntity (event_id, **kwargs)
    Bases: eventsourcing.domain.model.entity.DomainEntity

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__=None,
                                                         **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.Created,  eventsourcing.domain.
                model.entity.Event

    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

    Supertype for events of versioned entities.

    __check_obj__ (obj)
        Also checks the event's originator version follows this entity's version.

    __trigger_event__ (event_class, **kwargs)
        Triggers domain event with entity's next version number.

```

events

Base classes for domain events of different kinds.

```

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.

__eq__ (*other*)
 Tests for equality of two event objects.

__hash__ ()
 Computes a Python integer hash for an event, using its event hash string if available.
 Supports equality and inequality comparisons.

__init__ (**kwargs)
 Initialises event attribute values directly from constructor kwargs.

__mutate__ (*obj*)
 Update obj with values from self.
 Can be extended, but subclasses must call super method, and return an object.

Parameters *obj* – object to be mutated

Returns mutated object

__ne__ (*other*)
 Negates the equality test.

__repr__ ()
 Returns string representing the type and attribute values of the event.

__setattr__ (*key, value*)
 Inhibits event attributes from being updated by assignment.

mutate (*obj*)
 Convenience for use in custom models, to update obj with values from self without needing to call super method and return obj (two extra lines).

Can be overridden by subclasses. Any value returned by this method will be ignored.

Please note, subclasses that extend mutate() might not have fully completed that method before this method is called. To ensure all base classes have completed their mutate behaviour before mutating an event in a concrete class, extend mutate() instead of overriding this method.

Parameters *obj* – object to be mutated

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*)

snapshot

Snapshotting is implemented in the domain layer as an event.

class eventsourcing.domain.model.snapshot.**AbstractSnapshot**

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.
           AbstractSnapshot

    state
        State of the snapshotted entity.

    topic
        Path to the class of the snapshotted entity.
```

timebucketedlog

Time-bucketed logs allow a sequence of the items that is sequenced by timestamp to be split across a number of different database partitions, which avoids one partition becoming very large (and then unworkable).

```
class eventsourcing.domain.model.timebucketedlog.MessageLogged (message, origi-
                                                                nator_id)
    Bases: eventsourcing.domain.model.events.EventWithTimestamp, eventsourcing.
           domain.model.events.EventWithOriginatorID, eventsourcing.domain.model.
           events.Logged

    message

class eventsourcing.domain.model.timebucketedlog.Timebucketedlog (name,
                                                                    bucket_size=None,
                                                                    **kwargs)
    Bases: eventsourcing.domain.model.entity.TimestampedVersionedEntity

    class BucketSizeChanged (originator_version, **kwargs)
        Bases: eventsourcing.domain.model.timebucketedlog.Event, eventsourcing.
               domain.model.entity.AttributeChanged

    class Event (originator_version, **kwargs)
        Bases: eventsourcing.domain.model.entity.Event

        Supertype for events of time-bucketed log.

    class Started (originator_version=0, **kwargs)
        Bases: eventsourcing.domain.model.entity.Created, eventsourcing.domain.
               model.timebucketedlog.Event

    append_message (message)

    bucket_size

    name

    started_on

class eventsourcing.domain.model.timebucketedlog.TimebucketedlogRepository
    Bases: eventsourcing.domain.model.entity.AbstractEntityRepository

    get_or_create (log_name, bucket_size)
        Gets or creates a log.

        Return type Timebucketedlog
```

```

eventsourcing.domain.model.timebucketedlog.bucket_duration(bucket_size)
eventsourcing.domain.model.timebucketedlog.bucket_starts(timestamp, bucket_size)
eventsourcing.domain.model.timebucketedlog.make_timebucket_id(log_id,
                                                                timestamp,
                                                                bucket_size)
eventsourcing.domain.model.timebucketedlog.next_bucket_starts(timestamp,
                                                                bucket_size)
eventsourcing.domain.model.timebucketedlog.previous_bucket_starts(timestamp,
                                                                    bucket_size)
eventsourcing.domain.model.timebucketedlog.start_new_timebucketedlog(name,
                                                                      bucket_size=None)
eventsourcing.domain.model.timebucketedlog.timestamp_from_datetime(dt)

```

infrastructure

The infrastructure layer adapts external devices in ways that are useful for the application, such as the way an event store encapsulates a database.

base

Abstract base classes for the infrastructure layer.

```

class eventsourcing.infrastructure.base.AbstractRecordManager(record_class, se-
                                                                quenced_item_class=<class
                                                                'eventsourc-
                                                                ing.infrastructure.sequenceditem.Sequence
                                                                contiguous-
                                                                ous_record_ids=False)

```

Bases: object

```

all_items()
    Returns all stored items from all sequences (possibly in chronological order, depending on database).

all_records(start=None, stop=None, *args, **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.

list_items(*args, **kwargs)

raise_index_error(eq)

raise_sequenced_item_conflict()

```



```
class eventsourcing.infrastructure.base.RelationalRecordManager (*args,
                                                                **kwargs)
    Bases: eventsourcing.infrastructure.base.AbstractRecordManager

    _prepare_insert (tmpl)
        Compile SQL statement with placeholders for bind parameters.

    _write_records (records)
        Actually creates records in the database.

    append (sequenced_item_or_items)

    from_record (record)
        Returns a sequenced item, from given database record.

    get_items (sequence_id, gt=None, gte=None, lt=None, lte=None, limit=None,
               query_ascending=True, results_ascending=True)
        Returns items of a sequence.

    get_max_record_id ()
        Return maximum ID of existing records.

    get_records (sequence_id, gt=None, gte=None, lt=None, lte=None, limit=None,
                  query_ascending=True, results_ascending=True)
        Returns records for a sequence.

    insert_select_max
        SQL statement that inserts records with contiguous IDs, by selecting max ID from indexed table records.

    insert_values
        SQL statement that inserts records without ID.

    raise_after_integrity_error (e)

    static raise_record_id_conflict ()
        Raises RecordIDConflict exception.

    record_table_name
        Returns table name - used in raw queries, and to detect record ID conflicts.

        Return type str

    to_record (sequenced_item)
        Returns a database record, from given sequenced item.

    write_records (records)
        Calls _write_records() implemented by concrete classes.

        Retries call in case of a RecordIDConflict.
```

cassandra

Classes for event sourcing with Apache Cassandra.

```
class eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore (tables,
                                                                            *args,
                                                                            **kwargs)

    Bases: eventsourcing.infrastructure.datastore.Datastore

    close_connection ()

    drop_table (*)

    drop_tables ()
```

```

setup_connection()
setup_tables()
truncate_tables()

```

```

class eventsourcing.infrastructure.cassandra.datastore.CassandraSettings (hosts=None,
                                                                           port=None,
                                                                           pro-
                                                                           to-
                                                                           col_version=None,
                                                                           de-
                                                                           fault_keyspace=None,
                                                                           con-
                                                                           sis-
                                                                           tency=None,
                                                                           repli-
                                                                           ca-
                                                                           tion_factor=None,
                                                                           user-
                                                                           name=None,
                                                                           pass-
                                                                           word=None)

```

Bases: *eventsourcing.infrastructure.datastore.DatastoreSettings*

```

CONSISTENCY_LEVEL = 'LOCAL_QUORUM'
DEFAULT_KEYSPACE = 'eventsourcing'
HOSTS = ['127.0.0.1']
PORT = 9042
PROTOCOL_VERSION = 3
REPLICATION_FACTOR = 1

```

```

class eventsourcing.infrastructure.cassandra.factory.CassandraInfrastructureFactory (record_m
se-
quenced_
in-
te-
ger_sequ
times-
tamp_seq
snap-
shot_reco
con-
tigu-
ous_reco
ses-
sion=Non

```

Bases: *eventsourcing.infrastructure.factory.InfrastructureFactory*

```

integer_sequenced_record_class
    alias of IntegerSequencedRecord
record_manager_class
    alias of CassandraRecordManager

```

snapshot_record_class
alias of SnapshotRecord

timestamp_sequenced_record_class
alias of TimestampSequencedRecord

class eventsourcing.infrastructure.cassandra.manager.**CassandraRecordManager** (*record_class*,
sequenced_item_class=
'eventsourcing.infrastructure.se-
quential_record_ids=False

Bases: *eventsourcing.infrastructure.base.AbstractRecordManager*

all_items()

all_records (*start=None, stop=None, *args, **kwargs*)

all_sequence_ids()

append (*sequenced_item_or_items*)

delete_record (*record*)

filter (***kwargs*)

from_record (*record*)

Returns a sequenced item instance, from given database record.

get_item (*sequence_id, eq*)

get_items (*sequence_id, gt=None, gte=None, lt=None, lte=None, limit=None,*
query_ascending=True, results_ascending=True)

to_record (*sequenced_item*)

Returns an database record, from given sequenced item.

datastore

Base classes for concrete datastore classes.

class eventsourcing.infrastructure.datastore.**Datastore** (*settings*)

Bases: object

close_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`

django

A Django application for event sourcing with the Django ORM.

class `eventsourcing.infrastructure.django.manager.DjangoRecordManager` (*convert_position_float_to_dec*
**args,*
***kwargs*)

Bases: `eventsourcing.infrastructure.base.RelationalRecordManager`

`_prepare_insert` (*tmpl*)

With transaction isolation level of “read committed” this should generate records with a contiguous sequence of integer IDs, using an indexed ID column, the database-side SQL max function, the insert-select-from form, and optimistic concurrency control.

`all_items` ()

Returns all items across all sequences.

`all_records` (*start=None, stop=None, *args, **kwargs*)

Returns all records in the table.

`delete_record` (*record*)

Permanently removes record from table.

`get_field_kwargs` (*record*)

`get_item` (*sequence_id, eq*)

`get_max_record_id` ()

`get_records` (*sequence_id, gt=None, gte=None, lt=None, lte=None, limit=None,*
query_ascending=True, results_ascending=True)

`record_table_name`

eventplayer

Base classes for event players of different kinds.

class `eventsourcing.infrastructure.eventplayer.EventPlayer` (*event_store, snap-*
shot_strategy=None,
use_cache=False,
mutator_func=None)

Bases: `eventsourcing.domain.model.entity.AbstractEventPlayer`

`event_store`

`static mutate` (*initial, event*)

`replay_events` (*initial_state, domain_events*)

Evolves initial state using the sequence of domain events and a mutator function.

eventsourcedrepository

Base classes for event sourced repositories (not abstract, can be used directly).

```
class eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository (event_store,
                                                                    snap-
                                                                    shot_strategy=
                                                                    use_cache=False,
                                                                    mu-
                                                                    ta-
                                                                    tor_func=None)

Bases:  eventsourcing.infrastructure.eventplayer.EventPlayer, eventsourcing.
        domain.model.entity.AbstractEntityRepository

__contains__ (entity_id)
    Returns a boolean value according to whether entity with given ID exists.

__getitem__ (entity_id)
    Returns entity with given ID.

get_entity (entity_id, at=None)
    Returns entity with given ID, optionally until position.

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.

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.
```

eventstore

The event store provides the application-level interface to the event sourcing persistence mechanism.

```
class eventsourcing.infrastructure.eventstore.AbstractEventStore
    Bases: object

    Abstract base class for event stores. Defines the methods expected of an event store by other classes in the
    library.

    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 (record_manager, se-
                                                         quenced_item_mapper)
    Bases: eventsourcing.infrastructure.eventstore.AbstractEventStore
```

Event store appends domain events to stored sequences. It uses a record manager to map named tuples to database records, and it uses a sequenced item mapper to map named tuples to application-level objects.

__init__ (*record_manager*, *sequenced_item_mapper*)

Initialises event store object.

Parameters

- **record_manager** – record manager
- **sequenced_item_mapper** – sequenced item mapper

all_domain_events ()

Gets all domain events in the event store.

Returns map object, yielding a sequence of domain events

append (*domain_event_or_events*)

Appends given domain event, or list of domain events, to their sequence.

Parameters **domain_event_or_events** – domain event, or list of domain events

get_domain_event (*originator_id*, *eq*)

Gets a domain event from the sequence identified by *originator_id* at position *eq*.

Parameters

- **originator_id** – ID of a sequence of events
- **eq** – get item at this position

Returns domain event

get_domain_events (*originator_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *is_ascending=True*, *page_size=None*)

Gets domain events from the sequence identified by *originator_id*.

Parameters

- **originator_id** – ID of a sequence of events
- **gt** – get items after this position
- **gte** – get items at or after this position
- **lt** – get items before this position
- **lte** – get items before or at this position
- **limit** – get limited number of items
- **is_ascending** – get items from lowest position
- **page_size** – restrict and repeat database query

Returns list of domain events

get_most_recent_event (*originator_id*, *lt=None*, *lte=None*)

Gets a domain event from the sequence identified by *originator_id* at the highest position.

Parameters

- **originator_id** – ID of a sequence of events
- **lt** – get highest before this position
- **lte** – get highest at or before this position

Returns domain event

iterator_class

alias of SequencedItemIterator

to_sequenced_item (*domain_event*)

Maps domain event to sequenced item namedtuple.

Parameters *domain_event* – application-level object

Returns namedtuple: sequence item namedtuple

integersequencegenerators

Different ways of generating sequences of integers.

class eventsourcing.infrastructure.integersequencegenerators.base.**AbstractIntegerSequenceGenerator**

Bases: object

__next__ ()

Returns the next item in the container.

next ()

Python 2.7 version of the iterator protocol.

class eventsourcing.infrastructure.integersequencegenerators.base.**SimpleIntegerSequenceGenerator**

Bases: *eventsourcing.infrastructure.integersequencegenerators.base.AbstractIntegerSequenceGenerator*

AbstractIntegerSequenceGenerator

class eventsourcing.infrastructure.integersequencegenerators.redisincr.**RedisIncr** (*redis=None*, *key=None*)

Bases: *eventsourcing.infrastructure.integersequencegenerators.base.AbstractIntegerSequenceGenerator*

AbstractIntegerSequenceGenerator

Generates a sequence of integers, using Redis' INCR command.

Maximum number is 2**63, or 9223372036854775807, the maximum value of a 64 bit signed integer.

iterators

Different ways of getting sequenced items from a datastore.

class eventsourcing.infrastructure.iterators.**AbstractSequencedItemIterator** (*record_manager*, *sequence_id*, *page_size=None*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*, *is_ascending=True*)

Bases: object

DEFAULT_PAGE_SIZE = 1000

__iter__ ()

Yields a continuous sequence of items.

_inc_page_counter ()

Increments the page counter.

Each query result as a page, even if there are no items in the page. This really counts queries.

- it is easy to divide the number of events by the page size if the “correct” answer is required
- there will be a difference in the counts when the number of events can be exactly divided by the page size, because there is no way to know in advance that a full page is also the last page.

`_inc_query_counter()`
 Increments the query counter.

```
class eventsourcing.infrastructure.iterators.GetEntityEventsThread(record_manager,
                                                                    se-
                                                                    quence_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(record_manager,
                                                                    se-
                                                                    quence_id,
                                                                    page_size=None,
                                                                    gt=None,
                                                                    gte=None,
                                                                    lt=None,
                                                                    lte=None,
                                                                    limit=None,
                                                                    is_ascending=True)
```

Bases: `eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator`

`__iter__()`
 Yields a continuous sequence of items from “pages” of sequenced items retrieved using the record manager.

```
class eventsourcing.infrastructure.iterators.ThreadedSequencedItemIterator(record_manager,
                                                                    se-
                                                                    quence_id,
                                                                    page_size=None,
                                                                    gt=None,
                                                                    gte=None,
                                                                    lt=None,
                                                                    lte=None,
                                                                    limit=None,
                                                                    is_ascending=True)
```

Bases: `eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator`

`start_thread()`

repositories

Repository base classes for entity classes defined in the library.


```

class eventsourcing.infrastructure.repositories.array.ArrayRepository (array_size=10000,
                                                                    *args,
                                                                    **kwargs)

    Bases:
        eventsourcing.domain.model.array.AbstractArrayRepository,
        eventsourcing.infrastructure.eventourcedrepository.EventSourcedRepository

class eventsourcing.infrastructure.repositories.array.BigArrayRepository (array_size=10000,
                                                                    *args,
                                                                    **kwargs)

    Bases:
        eventsourcing.domain.model.array.AbstractBigArrayRepository,
        eventsourcing.infrastructure.eventourcedrepository.EventSourcedRepository

    subrepo
    subrepo_class
        alias of ArrayRepository

class eventsourcing.infrastructure.repositories.collection_repo.CollectionRepository (event_st
                                                                    snap-
                                                                    shot_str
                                                                    use_cac
                                                                    mu-
                                                                    ta-
                                                                    tor_func

    Bases:
        eventsourcing.infrastructure.eventourcedrepository.
        EventSourcedRepository,
        eventsourcing.domain.model.collection.
        AbstractCollectionRepository

    Event sourced repository for the Collection domain model entity.

class eventsourcing.infrastructure.repositories.timebucketedlog_repo.TimebucketedlogRepo (ev
                                                                    sn
                                                                    sh
                                                                    us
                                                                    m
                                                                    ta
                                                                    to

    Bases:
        eventsourcing.infrastructure.eventourcedrepository.
        EventSourcedRepository,
        eventsourcing.domain.model.timebucketedlog.
        TimebucketedlogRepository

    Event sourced repository for the Example domain model entity.

```

sequenceditem

The persistence model for storing events.

```

class eventsourcing.infrastructure.sequenceditem.SequencedItem (sequence_id,
                                                                    position, topic,
                                                                    data)

    Bases: tuple

    __getnewargs__ ()
        Return self as a plain tuple. Used by copy and pickle.

    static __new__ (_cls, sequence_id, position, topic, data)
        Create new instance of SequencedItem(sequence_id, position, topic, data)

```

```

__repr__()
    Return a nicely formatted representation string

_asdict()
    Return a new OrderedDict which maps field names to their values.

classmethod __make__(iterable, new=<built-in method __new__ of type object at 0xa385c0>,
                    len=<built-in function len>)
    Make a new SequencedItem object from a sequence or iterable

__replace__(_self, **kws)
    Return a new SequencedItem object replacing specified fields with new values

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

    other_names

    position

    sequence_id

    topic

class eventsourcing.infrastructure.sequenceditem.StoredEvent(originator_id,
                                                            originator_version,
                                                            event_type, state)
    Bases: tuple

    __getnewargs__()
        Return self as a plain tuple. Used by copy and pickle.

    static __new__(_cls, originator_id, originator_version, event_type, state)
        Create new instance of StoredEvent(originator_id, originator_version, event_type, state)

    __repr__()
        Return a nicely formatted representation string

    _asdict()
        Return a new OrderedDict which maps field names to their values.

    classmethod __make__(iterable, new=<built-in method __new__ of type object at 0xa385c0>,
                        len=<built-in function len>)
        Make a new StoredEvent object from a sequence or iterable

    __replace__(_self, **kws)
        Return a new StoredEvent object replacing specified fields with new values

    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

sequenceditemmapper

The sequenced item mapper maps sequenced items to application-level objects.

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*=
'eventsourcing.infrastructure.sequenceditemmapper.SequencedItem',
sequence_id_attr_name=
position_attr_name=None,
json_encoder_class=None,
json_decoder_class=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*)

from_sequenced_item(*sequenced_item*)
Reconstructs domain event from stored event topic and event attrs. Used in the event store when getting domain events.

from_topic_and_data(*topic*, *data*)

to_sequenced_item(*domain_event*)
Constructs a sequenced item from a domain event.

eventsourcing.infrastructure.sequenceditemmapper.**reconstruct_object**(*obj_class*,
obj_state)

snapshotting

Snapshotting avoids having to replay an entire sequence of events to obtain the current state of a projection.

class `event sourcing.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 *AbstractSnapshot*

class `event sourcing.infrastructure.snapshotting.EventSourcedSnapshotStrategy` (*event_store*)

Bases: `event sourcing.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*

`event sourcing.infrastructure.snapshotting.entity_from_snapshot` (*snapshot*)

Reconstructs domain entity from given snapshot.

sqlalchemy

Classes for event sourcing with SQLAlchemy.

class `event sourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore` (*base=<class*
'sqlalchemy.ext.declarative.declarative_base',
ta-
bles=None,
con-
nec-
tion_strategy='plain',
ses-
sion=None,
***kwargs*)

Bases: `event sourcing.infrastructure.datastore.Datastore`

close_connection ()

drop_table (*table*)

drop_tables ()

session

setup_connection ()

setup_table (*table*)

```

setup_tables (tables=None)

truncate_tables ()

class eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemySettings (uri=None)
    Bases: eventsourcing.infrastructure.datastore.DatastoreSettings

class eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory (session, *args, **kwargs)
    Bases: eventsourcing.infrastructure.factory.InfrastructureFactory

construct_record_manager (**kwargs)

integer_sequenced_record_class
    alias of IntegerSequencedWithIDRecord

record_manager_class
    alias of SQLAlchemyRecordManager

snapshot_record_class
    alias of SnapshotRecord

timestamp_sequenced_record_class
    alias of TimestampSequencedNoIDRecord

eventsourcing.infrastructure.sqlalchemy.factory.construct_sqlalchemy_eventstore (session, sequence_id_attr, position_attr_name, json_encoder_class, json_decoder_class, cipher=None, record_class=None, contiguous_record_ids)

class eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager (session, *args, **kwargs)
    Bases: eventsourcing.infrastructure.base.RelationalRecordManager

    _prepare_insert (tmpl)
        With transaction isolation level of “read committed” this should generate records with a contiguous sequence of integer IDs, assumes an indexed ID column, the database-side SQL max function, the insert-select-from form, and optimistic concurrency control.

    all_items ()
        Returns all items across all sequences.

    all_records (start=None, stop=None, *args, **kwargs)
        Returns all records in the table.

        Intended to support getting all application domain events in order, especially if the records have contiguous IDs.

```

delete_record(*record*)

Permanently removes record from table.

filter(***kwargs*)

get_item(*sequence_id*, *eq*)

get_items(*sequence_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*,
query_ascending=True, *results_ascending=True*)

get_max_record_id()

get_records(*sequence_id*, *gt=None*, *gte=None*, *lt=None*, *lte=None*, *limit=None*,
query_ascending=True, *results_ascending=True*)

query

record_table_name

class eventsourcing.infrastructure.sqlalchemy.records.**IntegerSequencedNoIDRecord**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in *kwargs*.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

data

position

sequence_id

topic

eventsourcing.infrastructure.sqlalchemy.records.**IntegerSequencedRecord**

alias of *IntegerSequencedWithIDRecord*

class eventsourcing.infrastructure.sqlalchemy.records.**IntegerSequencedWithIDRecord**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in *kwargs*.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

data

id

position

sequence_id

topic

class eventsourcing.infrastructure.sqlalchemy.records.**SnapshotRecord**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

data
position
sequence_id
topic

```
class eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord (**kwargs)
    Bases: sqlalchemy.ext.declarative.api.Base
```

```
    __init__ (**kwargs)
        A simple constructor that allows initialization from kwargs.
```

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

event_type
id
originator_id
originator_version
state

```
class eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedNoIDRecord (**kwargs)
    Bases: sqlalchemy.ext.declarative.api.Base
```

```
    __init__ (**kwargs)
        A simple constructor that allows initialization from kwargs.
```

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

data
position
sequence_id
topic

```
eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedRecord
    alias of TimestampSequencedNoIDRecord
```

```
class eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedWithIDRecord (**kwargs)
    Bases: sqlalchemy.ext.declarative.api.Base
```

```
    __init__ (**kwargs)
        A simple constructor that allows initialization from kwargs.
```

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

data

```

    id
    position
    sequence_id
    topic

```

timebucketedlog_reader

Reader for timebucketed logs.

```

class eventsourcing.infrastructure.timebucketedlog_reader.TimebucketedlogReader(log,
                                                                              event_store,
                                                                              page_size=50)

    Bases: object

    get_events(gt=None, gte=None, lt=None, lte=None, limit=None, is_ascending=False,
              page_size=None)

    get_messages(gt=None, gte=None, lt=None, lte=None, limit=None, is_ascending=False,
                page_size=None)

eventsourcing.infrastructure.timebucketedlog_reader.get_timebucketedlog_reader(log,
                                                                              event_store)

    Return type TimebucketedlogReader

```

interface

The interface layer uses an application to service client requests.

notificationlog

Notification log is a pull-based mechanism for updating other applications.

```

class eventsourcing.interface.notificationlog.AbstractNotificationLog
    Bases: object

    Presents a sequence of sections from a sequence of notifications.

    __getitem__(section_id)
        Get section of notification log.

        Return type Section

class eventsourcing.interface.notificationlog.BigArrayNotificationLog(big_array,
                                                                    sec-
                                                                    tion_size)

    Bases: eventsourcing.interface.notificationlog.LocalNotificationLog

    get_items(start, stop, next_position=None)

    get_next_position()

class eventsourcing.interface.notificationlog.LocalNotificationLog(section_size)
    Bases: eventsourcing.interface.notificationlog.AbstractNotificationLog

    Presents a sequence of sections from a sequence of notifications.

    static format_section_id(first_item_number, last_item_number)

```


get_items (*start, stop, next_position=None*)
Returns items for section.

Return type list

get_next_position ()
Returns items for section.

Return type int

class eventsourcing.interface.notificationlog.**NotificationLogReader** (*notification_log*)
Bases: object

get_items (*stop_index=None, advance_by=None*)

next ()
Python 2.7 version of the iterator protocol.

read (*advance_by=None*)

seek (*position*)

class eventsourcing.interface.notificationlog.**NotificationLogView** (*notification_log, json_encoder_class=None*)
Bases: object

present_section (*section_id*)

class eventsourcing.interface.notificationlog.**RecordManagerNotificationLog** (*record_manager, notification_size*)
Bases: *eventsourcing.interface.notificationlog.LocalNotificationLog*

get_items (*start, stop, next_position=None*)

get_next_position ()

class eventsourcing.interface.notificationlog.**RemoteNotificationLog** (*base_url, json_decoder_class=None*)
Bases: *eventsourcing.interface.notificationlog.AbstractNotificationLog*

deserialize_section (*section_json*)

get_json (*section_id*)

get_resource (*doc_url*)

make_notification_log_url (*section_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.

utils

The utils package contains common functions that are used in more than one layer.

cipher

class `eventsourcing.utils.cipher.aes.AESCipher(cipher_key)`

Bases: `object`

Cipher strategy that uses Crypto library AES cipher in GCM mode.

__init__ (*cipher_key*)

Initialises AES cipher strategy with *cipher_key*.

Parameters *cipher_key* – 16, 24, or 32 random bytes

decrypt (*ciphertext*)

Return plaintext for given ciphertext.

encrypt (*plaintext*)

Return ciphertext for given plaintext.

time

`eventsourcing.utils.times.datetime_from_timestamp(t)`

Returns a datetime from a decimal UNIX timestamp.

Parameters *t* – timestamp, either Decimal or float

Returns `datetime.datetime` object

`eventsourcing.utils.times.decimaltimestamp(t=None)`

A UNIX timestamp as a Decimal object (exact number type).

Returns current time when called without args, otherwise converts given floating point number *t* to a Decimal with 9 decimal places.

Parameters *t* – Floating point UNIX timestamp (“seconds since epoch”).

Returns A Decimal with 6 decimal places, representing the given floating point or the value returned by `time.time()`.

`eventsourcing.utils.times.decimaltimestamp_from_uuid(uuid_arg)`

Return a floating point unix timestamp.

Parameters *uuid_arg* –

Returns Unix timestamp in seconds, with microsecond precision.

Return type `float`

`eventsourcing.utils.times.timestamp_long_from_uuid(uuid_arg)`

Returns an integer value representing a unix timestamp in tenths of microseconds.

Parameters *uuid_arg* –

Returns Unix timestamp integer in tenths of microseconds.

Return type `int`

topic

`eventsourcing.utils.topic.get_topic(domain_class)`

Returns a string describing a class.

Args: *domain_class*: A class.

Returns: A string describing the class.

`eventsourcing.utils.topic.resolve_attr(obj, path)`

A recursive version of `getattr` for navigating dotted paths.

Args: `obj`: An object for which we want to retrieve a nested attribute. `path`: A dot separated string containing zero or more attribute names.

Returns: The attribute referred to by `obj.a1.a2.a3...`

Raises: `AttributeError`: If there is no such attribute.

`eventsourcing.utils.topic.resolve_topic(topic)`

Return class described by given topic.

Args: `topic`: A string describing a class.

Returns: A class.

Raises: `TopicResolutionError`: If there is no such class.

transcoding

class `eventsourcing.utils.transcoding.ObjectJSONDecoder` (*object_hook=None*,
***kwargs*)

Bases: `json.decoder.JSONDecoder`

classmethod `from_jsonable(d)`

class `eventsourcing.utils.transcoding.ObjectJSONEncoder` (*sort_keys=True*,
**args*,
***kwargs*)

Bases: `json.encoder.JSONEncoder`

default (*obj*)

`eventsourcing.utils.transcoding.json_dumps(obj, cls)`

`eventsourcing.utils.transcoding.json_loads(s, cls)`

exceptions

A few exception classes are defined by the library to indicate particular kinds of error.

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.DataIntegrityError`

Bases: `ValueError`, `eventsourcing.exceptions.EventSourcingError`

Raised when a sequenced item data is damaged (hash doesn't match data)

- exception** `event sourcing.exceptions.DatasourceSettingsError`
 Bases: `event sourcing.exceptions.EventSourcingError`
 Raised when an error is detected in settings for a datasource.
- exception** `event sourcing.exceptions.EntityIsDiscarded`
 Bases: `AssertionError`
 Raised when access to a recently discarded entity object is attempted.
- exception** `event sourcing.exceptions.EntityVersionNotFound`
 Bases: `event sourcing.exceptions.EventSourcingError`
 Raise when accessing an entity version that does not exist.
- exception** `event sourcing.exceptions.EventHashError`
 Bases: `event sourcing.exceptions.DataIntegrityError`
 Raised when an event's seal hash doesn't match the hash of the state of the event.
- exception** `event sourcing.exceptions.EventSourcingError`
 Bases: `Exception`
 Base event sourcing exception.
- exception** `event sourcing.exceptions.HeadHashError`
 Bases: `event sourcing.exceptions.DataIntegrityError`, `event sourcing.exceptions.MismatchedOriginatorError`
 Raised when applying an event with hash different from aggregate head.
- exception** `event sourcing.exceptions.MismatchedOriginatorError`
 Bases: `event sourcing.exceptions.ConsistencyError`
 Raised when applying an event to an inappropriate object.
- exception** `event sourcing.exceptions.MutatorRequiresTypeNotInstance`
 Bases: `event sourcing.exceptions.ConsistencyError`
 Raised when mutator function received a class rather than an entity.
- exception** `event sourcing.exceptions.OriginatorIDError`
 Bases: `event sourcing.exceptions.MismatchedOriginatorError`
 Raised when applying an event to the wrong entity or aggregate.
- exception** `event sourcing.exceptions.OriginatorVersionError`
 Bases: `event sourcing.exceptions.MismatchedOriginatorError`
 Raised when applying an event to the wrong version of an entity or aggregate.
- exception** `event sourcing.exceptions.ProgrammingError`
 Bases: `event sourcing.exceptions.EventSourcingError`
 Raised when programming errors are encountered.
- exception** `event sourcing.exceptions.RecordIDConflict`
 Bases: `event sourcing.exceptions.SequencedItemConflict`
 Raised when a record ID conflict is detected.
- exception** `event sourcing.exceptions.RepositoryKeyError`
 Bases: `KeyError`, `event sourcing.exceptions.EventSourcingError`
 Raised when using entity repository's dictionary like interface to get an entity that does not exist.

exception `event sourcing.exceptions.SequencedItemConflict`

Bases: `event sourcing.exceptions.EventSourcingError`

Raised when a sequence error occurs e.g. trying to save a version that already exists.

exception `event sourcing.exceptions.TimeSequenceError`

Bases: `event sourcing.exceptions.EventSourcingError`

Raised when a time sequence error occurs e.g. trying to save a timestamp that already exists.

exception `event sourcing.exceptions.TopicResolutionError`

Bases: `event sourcing.exceptions.EventSourcingError`

Raised when unable to resolve a topic to a Python class.

example

A simple, unit-tested, event sourced application.

application

class `event sourcing.example.application.ExampleApplication` (***kwargs*)

Bases: `event sourcing.application.base.ApplicationWithPersistencePolicies`

Example event sourced application with entity factory and repository.

create_new_example (*foo=*"", *a=*"", *b=*"")

Entity object factory.

`event sourcing.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()`.

`event sourcing.example.application.construct_example_application` (***kwargs*)

Application object factory.

`event sourcing.example.application.get_example_application()`

Returns single global instance of application.

To be called when handling a worker request, if required.

`event sourcing.example.application.init_example_application` (***kwargs*)

Constructs single global instance of application.

To be called when initialising a worker process.

domainmodel

class `event sourcing.example.domainmodel.AbstractExampleRepository`

Bases: `event sourcing.domain.model.entity.AbstractEntityRepository`

class `event sourcing.example.domainmodel.Example` (*foo=*"", *a=*"", *b=*"", ***kwargs*)

Bases: `event sourcing.domain.model.entity.TimestampedVersionedEntity`

An example event sourced domain model entity.

```
class AttributeChanged (originator_version, **kwargs)
    Bases:      eventsourcing.example.domainmodel.Event,  eventsourcing.domain.
                model.entity.AttributeChanged
```

Published when an Example is created.

```
class Created (originator_version=0, **kwargs)
    Bases:      eventsourcing.example.domainmodel.Event,  eventsourcing.domain.
                model.entity.Created
```

Published when an Example is created.

```
class Discarded (originator_version, **kwargs)
    Bases:      eventsourcing.example.domainmodel.Event,  eventsourcing.domain.
                model.entity.Discarded
```

Published when an Example is discarded.

```
class Event (originator_version, **kwargs)
    Bases: eventsourcing.domain.model.entity.Event
```

Supertype for events of example entities.

```
class Heartbeat (originator_version, **kwargs)
    Bases:      eventsourcing.example.domainmodel.Event,  eventsourcing.domain.
                model.entity.Event
```

Published when a heartbeat in the entity occurs (see below).

```
mutate (obj)
    Update obj with values from self.
```

```
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*

infrastructure

```
class eventsourcing.example.infrastructure.ExampleRepository (event_store, snapshot_strategy=None,
                                                                use_cache=False,
                                                                mutator_func=None)
```

```
Bases:      eventsourcing.infrastructure.event_sourcedrepository.
                EventSourcedRepository,      eventsourcing.example.domainmodel.
                AbstractExampleRepository
```

Event sourced repository for the Example domain model entity.

interface

```
class eventsourcing.example.interface.flaskapp.IntegerSequencedItem(**kwargs)
    Bases: sqlalchemy.ext.declarative.api.Model
```

```
    __init__(**kwargs)
```

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

data

id

position

sequence_id

topic

```
eventsourcing.example.interface.flaskapp.hello()
```

```
eventsourcing.example.interface.flaskapp.init_example_application_with_sqlalchemy()
```


e

eventsourcing.application.base, [80](#)
eventsourcing.application.policies, [81](#)
eventsourcing.application.simple, [81](#)
eventsourcing.domain.model.aggregate, [82](#)
eventsourcing.domain.model.array, [82](#)
eventsourcing.domain.model.decorators, [84](#)
eventsourcing.domain.model.entity, [85](#)
eventsourcing.domain.model.events, [88](#)
eventsourcing.domain.model.snapshot, [90](#)
eventsourcing.domain.model.timebucketedlog, [91](#)
eventsourcing.example.application, [113](#)
eventsourcing.example.domainmodel, [113](#)
eventsourcing.example.infrastructure, [114](#)
eventsourcing.example.interface.flaskapp, [115](#)
eventsourcing.exceptions, [111](#)
eventsourcing.infrastructure.base, [92](#)
eventsourcing.infrastructure.cassandra.datastore, [93](#)
eventsourcing.infrastructure.cassandra.factory, [94](#)
eventsourcing.infrastructure.cassandra.manager, [95](#)
eventsourcing.infrastructure.datastore, [95](#)
eventsourcing.infrastructure.django.manager, [96](#)
eventsourcing.infrastructure.eventplayer, [96](#)
eventsourcing.infrastructure.eventsourcedrepository, [97](#)
eventsourcing.infrastructure.eventstore, [97](#)
eventsourcing.infrastructure.integersequencegenerators.base, [99](#)
eventsourcing.infrastructure.integersequencegenerator, [99](#)
eventsourcing.infrastructure.iterators, [99](#)
eventsourcing.infrastructure.repositories.array, [100](#)
eventsourcing.infrastructure.repositories.collection, [101](#)
eventsourcing.infrastructure.repositories.timebucket, [101](#)
eventsourcing.infrastructure.sequenceditem, [101](#)
eventsourcing.infrastructure.sequenceditemmapper, [103](#)
eventsourcing.infrastructure.snapshotting, [104](#)
eventsourcing.infrastructure.sqlalchemy.datastore, [104](#)
eventsourcing.infrastructure.sqlalchemy.factory, [105](#)
eventsourcing.infrastructure.sqlalchemy.manager, [105](#)
eventsourcing.infrastructure.sqlalchemy.records, [106](#)
eventsourcing.infrastructure.timebucketedlog_reader, [108](#)
eventsourcing.interface.notificationlog, [108](#)
eventsourcing.utils.cipher.aes, [110](#)
eventsourcing.utils.times, [110](#)
eventsourcing.utils.topic, [110](#)
eventsourcing.utils.transcoding, [111](#)

Symbols

<code>__assert_not_discarded__()</code>	(eventsourcing.domain.model.entity.DomainEntity method), 86	<code>__getnewargs__()</code>	(eventsourcing.infrastructure.sequenceditem.SequencedItem method), 101	<code>ing.interface.notificationlog.AbstractNotificationLog</code>	method), 108
<code>__change_attribute__()</code>	(eventsourcing.domain.model.entity.DomainEntity method), 86	<code>__getnewargs__()</code>	(eventsourcing.infrastructure.sequenceditem.StoredEvent method), 102		
<code>__check_obj__()</code>	(eventsourcing.domain.model.entity.DomainEntity.Event method), 86	<code>__hash__()</code>	(eventsourcing.domain.model.events.DomainEvent method), 89		
<code>__check_obj__()</code>	(eventsourcing.domain.model.entity.VersionedEntity.Event method), 88	<code>__init__()</code>	(eventsourcing.domain.model.events.DomainEvent method), 89		
<code>__contains__()</code>	(eventsourcing.domain.model.entity.AbstractEntityRepository method), 85	<code>__init__()</code>	(eventsourcing.example.interface.flaskapp.IntegerSequencedItem method), 115		
<code>__contains__()</code>	(eventsourcing.infrastructure.eventsource repository.EventSourceRepository method), 97	<code>__init__()</code>	(eventsourcing.infrastructure.eventstore.EventStore method), 98		
<code>__discard__()</code>	(eventsourcing.domain.model.entity.DomainEntity method), 86	<code>__init__()</code>	(eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedItem method), 106		
<code>__eq__()</code>	(eventsourcing.domain.model.events.DomainEvent method), 89	<code>__init__()</code>	(eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedItem method), 106		
<code>__getitem__()</code>	(eventsourcing.domain.model.array.AbstractArrayRepository method), 82	<code>__init__()</code>	(eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord method), 106		
<code>__getitem__()</code>	(eventsourcing.domain.model.array.AbstractBigArrayRepository method), 83	<code>__init__()</code>	(eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord method), 107		
<code>__getitem__()</code>	(eventsourcing.domain.model.array.Array method), 83	<code>__init__()</code>	(eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord method), 107		
<code>__getitem__()</code>	(eventsourcing.domain.model.entity.AbstractEntityRepository method), 85	<code>__init__()</code>	(eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedItem method), 107		
<code>__getitem__()</code>	(eventsourcing.infrastructure.eventsource repository.EventSourceRepository method), 97	<code>__init__()</code>	(eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedItem method), 107		
<code>__getitem__()</code>	(eventsourcing.infrastructure.eventsource repository.EventSourceRepository method), 97	<code>__init__()</code>	(eventsourcing.utils.cipher.aes.AESCipher method), 110		
		<code>__iter__()</code>	(eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator method), 99		
		<code>__iter__()</code>	(eventsourcing.infrastructure.iterators.SequencedItemIterator method), 100		
		<code>__len__()</code>	(eventsourcing.domain.model.array.Array method), 83		
		<code>__len__()</code>	(eventsourcing.domain.model.array.BigArray method), 84		
		<code>__mutate__()</code>	(eventsourcing...		

ing.domain.model.entity.TimestampedEntity.Event
 method), 87
 __mutate__() (event sourcing.domain.model.events.DomainEvent
 method), 89
 __ne__() (event sourcing.domain.model.events.DomainEvent
 method), 89
 __new__() (event sourcing.infrastructure.sequenceditem.SequencedItem
 static method), 101
 __new__() (event sourcing.infrastructure.sequenceditem.StoredEvent
 static method), 102
 __next__() (event sourcing.infrastructure.integersequencegenerators.base.AbstractIntegerSequenceGenerator
 method), 99
 __publish__() (event sourcing.domain.model.aggregate.AggregateRoot
 method), 82
 __publish__() (event sourcing.domain.model.entity.DomainEntity
 method), 86
 __publish_to_subscribers__() (event sourcing.domain.model.entity.DomainEntity
 method), 86
 __repr__() (event sourcing.domain.model.events.DomainEvent
 method), 89
 __repr__() (event sourcing.infrastructure.sequenceditem.SequencedItem
 method), 101
 __repr__() (event sourcing.infrastructure.sequenceditem.StoredEvent
 method), 102
 __save__() (event sourcing.domain.model.aggregate.AggregateRoot
 method), 82
 __setattr__() (event sourcing.domain.model.events.DomainEvent
 method), 89
 __setitem__() (event sourcing.domain.model.array.Array
 method), 83
 __trigger_event__() (event sourcing.domain.model.entity.DomainEntity
 method), 86
 __trigger_event__() (event sourcing.domain.model.entity.VersionedEntity
 method), 88
 _asdict() (event sourcing.infrastructure.sequenceditem.SequencedItem
 method), 102
 _asdict() (event sourcing.infrastructure.sequenceditem.StoredEvent
 method), 102
 _inc_page_counter() (event sourcing.infrastructure.iterators.AbstractSequencedItemIterator
 method), 99
 _inc_query_counter() (event sourcing.infrastructure.iterators.AbstractSequencedItemIterator
 method), 100
 _make() (event sourcing.infrastructure.sequenceditem.SequencedItem
 class method), 102
 _make() (event sourcing.infrastructure.sequenceditem.StoredEvent
 class method), 102
 _prepare_insert() (event sourcing.infrastructure.base.RelationalRecordManager
 method), 93
 _prepare_insert() (event sourcing.infrastructure.django.manager.DjangoRecordManager
 method), 96
 _prepare_insert() (event sourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager
 method), 105
 _replace() (event sourcing.infrastructure.sequenceditem.SequencedItem
 method), 102
 _replace() (event sourcing.infrastructure.sequenceditem.StoredEvent
 method), 102
 _write_records() (event sourcing.infrastructure.base.RelationalRecordManager
 method), 93

A

a (event sourcing.example.domainmodel.Example attribute), 114
 AbstractArrayRepository (class in event sourcing.domain.model.array), 82
 AbstractBigArrayRepository (class in event sourcing.domain.model.array), 83
 AbstractEntityRepository (class in event sourcing.domain.model.entity), 85
 AbstractEventPlayer (class in event sourcing.domain.model.entity), 86
 AbstractEventStore (class in event sourcing.infrastructure.eventstore), 97
 AbstractExampleRepository (class in event sourcing.example.domainmodel), 113
 AbstractIntegerSequenceGenerator (class in event sourcing.infrastructure.integersequencegenerators.base), 99
 AbstractNotificationLog (class in event sourcing.interface.notificationlog), 108
 AbstractRecordManager (class in event sourcing.infrastructure.base), 92
 AbstractSequencedItemIterator (class in event sourcing.infrastructure.iterators), 99
 AbstractSequencedItemMapper (class in event sourcing.infrastructure.sequenceditemmapper), 103
 AbstractSnapshot (class in event sourcing.domain.model.snapshot), 90

- AbstractSnapshotStrategy (class in event sourcing.infrastructure.snapshotting), 104
 - AESCipher (class in event sourcing.utils.cipher.aes), 110
 - AggregateRoot (class in event sourcing.domain.model.aggregate), 82
 - AggregateRoot.AttributeChanged (class in event sourcing.domain.model.aggregate), 82
 - AggregateRoot.Created (class in event sourcing.domain.model.aggregate), 82
 - AggregateRoot.Discarded (class in event sourcing.domain.model.aggregate), 82
 - AggregateRoot.Event (class in event sourcing.domain.model.aggregate), 82
 - all_domain_events() (event sourcing.infrastructure.eventstore.AbstractEventStore method), 97
 - all_domain_events() (event sourcing.infrastructure.eventstore.EventStore method), 98
 - all_items() (event sourcing.infrastructure.base.AbstractRecordManager method), 92
 - all_items() (event sourcing.infrastructure.cassandra.manager.CassandraRecordManager method), 95
 - all_items() (event sourcing.infrastructure.django.manager.DjangoRecordManager method), 96
 - all_items() (event sourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 105
 - all_records() (event sourcing.infrastructure.base.AbstractRecordManager method), 92
 - all_records() (event sourcing.infrastructure.cassandra.manager.CassandraRecordManager method), 95
 - all_records() (event sourcing.infrastructure.django.manager.DjangoRecordManager method), 96
 - all_records() (event sourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 105
 - all_sequence_ids() (event sourcing.infrastructure.cassandra.manager.CassandraRecordManager method), 95
 - append() (event sourcing.domain.model.array.Array method), 83
 - append() (event sourcing.infrastructure.base.AbstractRecordManager method), 92
 - append() (event sourcing.infrastructure.base.RelationalRecordManager method), 93
 - append() (event sourcing.infrastructure.cassandra.manager.CassandraRecordManager method), 95
 - append() (event sourcing.infrastructure.eventstore.AbstractEventStore method), 97
 - append() (event sourcing.infrastructure.eventstore.EventStore method), 98
 - append_message() (event sourcing.domain.model.timebucketedlog.Timebucketedlog method), 91
 - ApplicationWithEventStores (class in event sourcing.application.base), 80
 - ApplicationWithPersistencePolicies (class in event sourcing.application.base), 80
 - Array (class in event sourcing.domain.model.array), 83
 - ArrayIndexError, 111
 - ArrayRepository (class in event sourcing.infrastructure.repositories.array), 100
 - assert_event_handlers_empty() (in module event sourcing.domain.model.events), 90
 - attribute() (in module event sourcing.domain.model.decorators), 84
 - AttributeChanged (class in event sourcing.domain.model.events), 88
- ## B
- beat_heart() (event sourcing.example.domainmodel.Example attribute), 114
 - beat_heart() (event sourcing.example.domainmodel.Example method), 114
 - BigArray (class in event sourcing.domain.model.array), 84
 - BigArrayNotificationLog (class in event sourcing.interface.notificationlog), 108
 - BigArrayRepository (class in event sourcing.infrastructure.repositories.array), 101
 - bucket_duration() (in module event sourcing.domain.model.timebucketedlog), 91
 - bucket_size (event sourcing.domain.model.timebucketedlog.Timebucketedlog attribute), 91
 - bucket_starts() (in module event sourcing.domain.model.timebucketedlog), 92
- ## C
- calc_parent() (event sourcing.domain.model.array.BigArray method), 84
 - calc_required_height() (event sourcing.domain.model.array.BigArray method), 84
 - CassandraDatastore (class in event sourcing.infrastructure.cassandra.datastore), 93
 - CassandraInfrastructureFactory (class in event sourcing.infrastructure.cassandra.factory), 94

[CassandraRecordManager](#) (class in `eventsourcing.infrastructure.cassandra.manager`), [95](#)
[CassandraSettings](#) (class in `eventsourcing.infrastructure.cassandra.datastore`), [94](#)
[close\(\)](#) (`eventsourcing.application.base.ApplicationWithEventStores` method), [80](#)
[close\(\)](#) (`eventsourcing.application.base.ApplicationWithPersistencePolicies` method), [80](#)
[close\(\)](#) (`eventsourcing.application.policies.PersistencePolicy` method), [81](#)
[close\(\)](#) (`eventsourcing.application.policies.SnapshottingPolicy` method), [81](#)
[close\(\)](#) (`eventsourcing.application.simple.SimpleApplication` method), [81](#)
[close\(\)](#) (`eventsourcing.application.simple.SnapshottingApplication` method), [81](#)
[close_connection\(\)](#) (`eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore` method), [93](#)
[close_connection\(\)](#) (`eventsourcing.infrastructure.datastore.Datastore` method), [95](#)
[close_connection\(\)](#) (`eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore` method), [104](#)
[close_example_application\(\)](#) (in module `eventsourcing.example.application`), [113](#)
[CollectionRepository](#) (class in `eventsourcing.infrastructure.repositories.collection_repo`), [101](#)
[ConcurrencyError](#), [111](#)
[condition\(\)](#) (`eventsourcing.application.policies.SnapshottingPolicy` method), [81](#)
[CONSISTENCY_LEVEL](#) (`eventsourcing.infrastructure.cassandra.datastore.CassandraSettings` attribute), [94](#)
[ConsistencyError](#), [111](#)
[construct_entity_persistence_policy\(\)](#) (`eventsourcing.application.base.ApplicationWithPersistencePolicies` method), [80](#)
[construct_event_store\(\)](#) (`eventsourcing.application.base.ApplicationWithEventStores` method), [80](#)
[construct_example_application\(\)](#) (in module `eventsourcing.example.application`), [113](#)
[construct_item_args\(\)](#) (`eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper` method), [103](#)
[construct_log_persistence_policy\(\)](#) (`eventsourcing.application.base.ApplicationWithPersistencePolicies` method), [80](#)
[construct_record_manager\(\)](#) (`eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyFactory` method), [105](#)
[construct_sequenced_item\(\)](#) (`eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper` method), [103](#)
[construct_sequenced_item_mapper\(\)](#) (`eventsourcing.application.base.ApplicationWithEventStores` method), [80](#)
[construct_snapshot_persistence_policy\(\)](#) (`eventsourcing.application.base.ApplicationWithPersistencePolicies` method), [80](#)
[construct_sqlalchemy_eventstore\(\)](#) (in module `eventsourcing.infrastructure.sqlalchemy.factory`), [105](#)
[count_heartbeats\(\)](#) (`eventsourcing.example.domainmodel.Example` method), [114](#)
[create_array_id\(\)](#) (`eventsourcing.domain.model.array.BigArray` method), [84](#)
[create_new_example\(\)](#) (`eventsourcing.example.application.ExampleApplication` method), [113](#)
[create_new_example\(\)](#) (in module `eventsourcing.example.domainmodel`), [114](#)
[create_timesequenced_event_id\(\)](#) (in module `eventsourcing.domain.model.events`), [90](#)
[Created](#) (class in `eventsourcing.domain.model.events`), [88](#)

D

[data](#) (`eventsourcing.example.interface.flaskapp.IntegerSequencedItem` attribute), [115](#)
[data](#) (`eventsourcing.infrastructure.sequenceditem.SequencedItem` attribute), [102](#)
[data](#) (`eventsourcing.infrastructure.sequenceditem.SequencedItemFieldName` attribute), [102](#)
[data](#) (`eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedNoItem` attribute), [106](#)
[data](#) (`eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithItem` attribute), [106](#)
[data](#) (`eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord` attribute), [107](#)
[data](#) (`eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedItem` attribute), [107](#)
[data](#) (`eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedNoItem` attribute), [107](#)
[DataIntegrityError](#), [111](#)
[DatasourceSettingsError](#), [111](#)
[Datastore](#) (class in `eventsourcing.infrastructure.datastore`), [95](#)
[DatastoreConnectionError](#), [95](#)
[DatastoreError](#), [95](#)
[DatastoreSettings](#) (class in `eventsourcing.infrastructure.datastore`), [96](#)
[DatastoreTableError](#), [96](#)

datetime_from_timestamp() (in module eventsourcing.utils.times), 110

decimaltimestamp() (in module eventsourcing.utils.times), 110

decimaltimestamp_from_uuid() (in module eventsourcing.utils.times), 110

decrypt() (eventsourcing.utils.cipher.aes.AESCipher method), 110

default() (eventsourcing.utils.transcoding.ObjectJSONEncoder method), 111

DEFAULT_KEYSPACE (eventsourcing.infrastructure.cassandra.datastore.CassandraSettings attribute), 94

DEFAULT_PAGE_SIZE (eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator attribute), 99

delete_record() (eventsourcing.infrastructure.base.AbstractRecordManager method), 92

delete_record() (eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager method), 95

delete_record() (eventsourcing.infrastructure.django.manager.DjangoRecordManager method), 96

delete_record() (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 105

deserialize_section() (eventsourcing.interface.notificationlog.RemoteNotificationLog method), 109

Discarded (class in eventsourcing.domain.model.events), 89

DjangoRecordManager (class in eventsourcing.infrastructure.django.manager), 96

DomainEntity (class in eventsourcing.domain.model.entity), 86

DomainEntity.AttributeChanged (class in eventsourcing.domain.model.entity), 86

DomainEntity.Created (class in eventsourcing.domain.model.entity), 86

DomainEntity.Discarded (class in eventsourcing.domain.model.entity), 86

DomainEntity.Event (class in eventsourcing.domain.model.entity), 86

DomainEvent (class in eventsourcing.domain.model.events), 89

drop_table() (eventsourcing.application.simple.SimpleApplication method), 81

drop_table() (eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore method), 93

drop_table() (eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore method), 104

drop_tables() (eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore method), 93

drop_tables() (eventsourcing.infrastructure.datastore.Datastore method), 95

drop_tables() (eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore method), 104

E

encrypt() (eventsourcing.utils.cipher.aes.AESCipher method), 110

entity_from_snapshot() (in module eventsourcing.infrastructure.snapshotting), 104

EntityIsDiscarded, 112

EntityVersionNotFound, 112

event_id (eventsourcing.domain.model.events.EventWithTimeuuid attribute), 90

event_store (eventsourcing.domain.model.entity.AbstractEntityRepository attribute), 85

event_store (eventsourcing.infrastructure.eventplayer.EventPlayer attribute), 96

event_type (eventsourcing.infrastructure.sequenceditem.StoredEvent attribute), 102

event_type (eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord attribute), 107

EventHandlersNotEmptyError, 89

EventHashError, 112

EventPlayer (class in eventsourcing.infrastructure.eventplayer), 96

EventSourcedRepository (class in eventsourcing.infrastructure.eventsourcedrepository), 97

EventSourcedSnapshotStrategy (class in eventsourcing.infrastructure.snapshotting), 104

eventsourcing.application.base (module), 80

eventsourcing.application.policies (module), 81

eventsourcing.application.simple (module), 81

eventsourcing.domain.model.aggregate (module), 82

eventsourcing.domain.model.array (module), 82

eventsourcing.domain.model.decorators (module), 84

eventsourcing.domain.model.entity (module), 85

eventsourcing.domain.model.events (module), 88

eventsourcing.domain.model.snapshot (module), 90

eventsourcing.domain.model.timebucketedlog (module), 91

eventsourcing.example.application (module), 113

[event sourcing.example.domainmodel \(module\)](#), 113
[event sourcing.example.infrastructure \(module\)](#), 114
[event sourcing.example.interface.flaskapp \(module\)](#), 115
[event sourcing.exceptions \(module\)](#), 111
[event sourcing.infrastructure.base \(module\)](#), 92
[event sourcing.infrastructure.cassandra.datastore \(module\)](#), 93
[event sourcing.infrastructure.cassandra.factory \(module\)](#), 94
[event sourcing.infrastructure.cassandra.manager \(module\)](#), 95
[event sourcing.infrastructure.datastore \(module\)](#), 95
[event sourcing.infrastructure.django.manager \(module\)](#), 96
[event sourcing.infrastructure.eventplayer \(module\)](#), 96
[event sourcing.infrastructure.event sourced repository \(module\)](#), 97
[event sourcing.infrastructure.eventstore \(module\)](#), 97
[event sourcing.infrastructure.integer sequence generators.base Example Application \(module\)](#), 99
[event sourcing.infrastructure.integer sequence generators.redis Example \(module\)](#), 99
[event sourcing.infrastructure.iterators \(module\)](#), 99
[event sourcing.infrastructure.repositories.array \(module\)](#), 100
[event sourcing.infrastructure.repositories.collection_repo \(module\)](#), 101
[event sourcing.infrastructure.repositories.time bucketed log_repo \(module\)](#), 101
[event sourcing.infrastructure.sequenced item \(module\)](#), 101
[event sourcing.infrastructure.sequenced item mapper \(module\)](#), 103
[event sourcing.infrastructure.snapshotting \(module\)](#), 104
[event sourcing.infrastructure.sqlalchemy.datastore \(module\)](#), 104
[event sourcing.infrastructure.sqlalchemy.factory \(module\)](#), 105
[event sourcing.infrastructure.sqlalchemy.manager \(module\)](#), 105
[event sourcing.infrastructure.sqlalchemy.records \(module\)](#), 106
[event sourcing.infrastructure.time bucketed log_reader \(module\)](#), 108
[event sourcing.interface.notificationlog \(module\)](#), 108
[event sourcing.utils.cipher.aes \(module\)](#), 110
[event sourcing.utils.times \(module\)](#), 110
[event sourcing.utils.topic \(module\)](#), 110
[event sourcing.utils.transcoding \(module\)](#), 111
[Event SourcingError](#), 112
[EventStore \(class in event sourcing.infrastructure.eventstore\)](#), 97
[EventWithOriginatorID \(class in event sourcing.domain.model.events\)](#), 89

[EventWithOriginatorVersion \(class in event sourcing.domain.model.events\)](#), 90
[EventWithTimestamp \(class in event sourcing.domain.model.events\)](#), 90
[EventWithTimeuuid \(class in event sourcing.domain.model.events\)](#), 90
[Example \(class in event sourcing.example.domainmodel\)](#), 113
[Example.AttributeChanged \(class in event sourcing.example.domainmodel\)](#), 113
[Example.Created \(class in event sourcing.example.domainmodel\)](#), 114
[Example.Discarded \(class in event sourcing.example.domainmodel\)](#), 114
[Example.Event \(class in event sourcing.example.domainmodel\)](#), 114
[Example.Heartbeat \(class in event sourcing.example.domainmodel\)](#), 114
[Example.Application \(class in event sourcing.example.application\)](#), 113
[ExampleRepository \(class in event sourcing.example.infrastructure\)](#), 114

F

[filter\(\) \(event sourcing.infrastructure.cassandra.manager.CassandraRecordManager method\)](#), 95
[filter\(\) \(event sourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method\)](#), 106
[foo \(event sourcing.example.domainmodel.Example attribute\)](#), 114
[format_section_id\(\) \(event sourcing.interface.notificationlog.LocalNotificationLog static method\)](#), 108
[from_jsonable\(\) \(event sourcing.utils.transcoding.ObjectJSONDecoder class method\)](#), 111
[from_record\(\) \(event sourcing.infrastructure.base.RelationalRecordManager method\)](#), 93
[from_record\(\) \(event sourcing.infrastructure.cassandra.manager.CassandraRecordManager method\)](#), 95
[from_sequenced_item\(\) \(event sourcing.infrastructure.sequenced item mapper.AbstractSequencedItemMapper method\)](#), 103
[from_sequenced_item\(\) \(event sourcing.infrastructure.sequenced item mapper.SequencedItemMapper method\)](#), 103
[from_topic_and_data\(\) \(event sourcing.infrastructure.sequenced item mapper.SequencedItemMapper method\)](#), 103

G

[get_domain_event\(\)](#) (event sourcing

Index	125
--------------	------------

ing.interface.notificationlog.LocalNotificationLogindex (eventsourcing.domain.model.array.ItemAssigned method), 109 attribute), 84

get_next_position() (eventsourcing.interface.notificationlog.RecordManagerNotificationLogging.example.application), 113

get_or_create() (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.domain.model.timebucketedlog.Timebucketedlog.index.example.application, 113

method), 91

get_records() (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.infrastructure.base.RelationalRecordManager insert_values (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

method), 93

get_records() (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.infrastructure.django.manager.DjangoRecordManager insert_values (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

method), 96

get_records() (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.infrastructure.cassandra.factory.CassandraInfrastructureFactory insert_values (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

method), 106

get_resource() (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.interface.notificationlog.RemoteNotificationLogindex.example.application, 113

method), 109

get_slice() (eventsourcing.domain.model.array.BigArray IntegerSequencedNoIDRecord (class in eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

method), 84

ing.infrastructure.sqlalchemy.records), 106

get_snapshot() (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.infrastructure.snapshotting.AbstractSnapshotStrategy IntegerSequencedRecord (in module eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

method), 104

ing.infrastructure.sqlalchemy.records), 106

get_snapshot() (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.infrastructure.sqlalchemy.records), 106

ing.infrastructure.snapshotting.EventSourcedSnapshotStrategy IntegerSequencedWithIDRecord (class in eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

method), 104

ing.application.policies.PersistencePolicy IntegerSequencedWithIDRecord (class in eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

method), 81

get_timebucketedlog_reader() (in module eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.infrastructure.timebucketedlog_reader), 108

item (eventsourcing.domain.model.array.ItemAssigned attribute), 84

get_topic() (in module eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.utils.topic), 110

GetEntityEventsThread (class in eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.infrastructure.iterators), 100

iterator_class (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

ing.infrastructure.eventstore.EventStore attribute), 98

H

HeadHashError, 112

hello() (in module eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

HOSTS (eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore.index.example.application), 113

I

id (eventsourcing.domain.model.entity.DomainEntity attribute), 87

id (eventsourcing.example.interface.flaskapp.IntegerSequencedItem attribute), 115

id (eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord attribute), 106

id (eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord attribute), 107

id (eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedWithIDRecord attribute), 107

J

json_dumps() (in module eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

json_loads() (in module eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

L

list_items() (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

LocalNotificationLog (class in eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

Logged (class in eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

M

map_sequenced_with_id_record() (eventsourcing.interface.notificationlog.RemoteNotificationLogindex.example.application), 113

make_timebucket_id() (in module eventsourcing.domain.model.timebucketedlog), 92

message (eventsourcing.domain.model.timebucketedlog.MessageLogged attribute), 91

MessageLogged (class in eventsourcing.domain.model.timebucketedlog), 91

MismatchedOriginatorError, 112

mutate() (eventsourcing.domain.model.events.DomainEvent method), 89

mutate() (eventsourcing.example.domainmodel.Example.Heard attribute), 114

mutate() (eventsourcing.infrastructure.eventplayer.EventPlayer static method), 96

mutator() (in module eventsourcing.domain.model.decorators), 84

MutatorRequiresTypeNotInstance, 112

originator_version (eventsourcing.domain.model.events.EventWithOriginatorVersion attribute), 90

originator_version (eventsourcing.domain.model.snapshot.AbstractSnapshot attribute), 90

originator_version (eventsourcing.infrastructure.sequenceditem.StoredEvent attribute), 103

originator_version (eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord attribute), 107

OriginatorIDError, 112

OriginatorVersionError, 112

other_names (eventsourcing.infrastructure.sequenceditem.SequencedItemFieldNames attribute), 102

N

name (eventsourcing.domain.model.events.AttributeChanged attribute), 88

name (eventsourcing.domain.model.timebucketedlog.Timebucketedlog attribute), 91

next() (eventsourcing.infrastructure.integersequencegenerators.base.AbstractIntegerSequenceGenerator method), 99

next() (eventsourcing.interface.notificationlog.NotificationLogReader attribute), 109

next_bucket_starts() (in module eventsourcing.domain.model.timebucketedlog), 92

NotificationLogReader (class in eventsourcing.interface.notificationlog), 109

NotificationLogView (class in eventsourcing.interface.notificationlog), 109

O

ObjectJSONDecoder (class in eventsourcing.utils.transcoding), 111

ObjectJSONEncoder (class in eventsourcing.utils.transcoding), 111

originator_id (eventsourcing.domain.model.events.EventWithOriginatorID attribute), 89

originator_id (eventsourcing.domain.model.snapshot.AbstractSnapshot attribute), 90

originator_id (eventsourcing.infrastructure.sequenceditem.StoredEvent attribute), 102

originator_id (eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord attribute), 107

originator_topic (eventsourcing.domain.model.entity.DomainEntity.Created attribute), 86

P

PersistencePolicy (class in eventsourcing.application.policies), 81

PORT (eventsourcing.infrastructure.cassandra.datastore.CassandraSettings attribute), 94

position (eventsourcing.example.interface.flaskapp.IntegerSequencedItem attribute), 115

position (eventsourcing.infrastructure.sequenceditem.SequencedItem attribute), 102

position (eventsourcing.infrastructure.sequenceditem.SequencedItemFieldNames attribute), 102

position (eventsourcing.infrastructure.sqlalchemy.records.IntegerSequenced attribute), 106

position (eventsourcing.infrastructure.sqlalchemy.records.IntegerSequenced attribute), 106

position (eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord attribute), 107

position (eventsourcing.infrastructure.sqlalchemy.records.TimestampSequenced attribute), 107

position (eventsourcing.infrastructure.sqlalchemy.records.TimestampSequenced attribute), 108

present_section() (eventsourcing.interface.notificationlog.NotificationLogView method), 109

previous_bucket_starts() (in module eventsourcing.domain.model.timebucketedlog), 92

ProgrammingError, 112

PROTOCOL_VERSION (eventsourcing.infrastructure.cassandra.datastore.CassandraSettings attribute), 94

publish() (in module eventsourcing.domain.model.events), 90

Q

QualnameABC (class in eventsourcing.domain.model.events), 90

[QualnameABCMeta](#) (class in `event sourcing.domain.model.events`), [90](#)
[query](#) (`event sourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager` attribute), [106](#)
R
[raise_after_integrity_error\(\)](#) (`event sourcing.infrastructure.base.RelationalRecordManager` method), [93](#)
[raise_index_error\(\)](#) (`event sourcing.infrastructure.base.AbstractRecordManager` method), [92](#)
[raise_record_id_conflict\(\)](#) (`event sourcing.infrastructure.base.RelationalRecordManager` static method), [93](#)
[raise_sequenced_item_conflict\(\)](#) (`event sourcing.infrastructure.base.AbstractRecordManager` method), [92](#)
[random\(\)](#) (in module `event sourcing.domain.model.decorators`), [85](#)
[read\(\)](#) (`event sourcing.interface.notificationlog.NotificationLogReader` method), [109](#)
[reconstruct_object\(\)](#) (in module `event sourcing.infrastructure.sequenceditemmapper`), [103](#)
[record_manager_class](#) (`event sourcing.infrastructure.cassandra.factory.CassandraInfrastructureFactory` attribute), [94](#)
[record_manager_class](#) (`event sourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory` attribute), [105](#)
[record_table_name](#) (`event sourcing.infrastructure.base.RelationalRecordManager` attribute), [93](#)
[record_table_name](#) (`event sourcing.infrastructure.django.manager.DjangoRecordManager` attribute), [96](#)
[record_table_name](#) (`event sourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager` attribute), [106](#)
[RecordIDConflict](#), [112](#)
[RecordManagerNotificationLog](#) (class in `event sourcing.interface.notificationlog`), [109](#)
[RedisIncr](#) (class in `event sourcing.infrastructure.integersequencegenerators.redisincr`), [99](#)
[RelationalRecordManager](#) (class in `event sourcing.infrastructure.base`), [92](#)
[RemoteNotificationLog](#) (class in `event sourcing.interface.notificationlog`), [109](#)
[replay_entity\(\)](#) (`event sourcing.infrastructure.event sourced repository.EventSourcedRepository` method), [97](#)
[replay_events\(\)](#) (`event sourcing.infrastructure.eventplayer.EventPlayer` method), [96](#)
[REPLICATION_FACTOR](#) (`event sourcing.infrastructure.cassandra.datastore.CassandraSettings` attribute), [94](#)
[RepositoryKeyError](#), [112](#)
[resolve_attr\(\)](#) (in module `event sourcing.utils.topic`), [111](#)
[resolve_topic\(\)](#) (in module `event sourcing.utils.topic`), [111](#)
[retry\(\)](#) (in module `event sourcing.domain.model.decorators`), [85](#)
[run\(\)](#) (`event sourcing.infrastructure.iterators.GetEntityEventsThread` method), [100](#)
S
[Section](#) (class in `event sourcing.interface.notificationlog`), [109](#)
[seek\(\)](#) (`event sourcing.interface.notificationlog.NotificationLogReader` method), [109](#)
[sequence_id](#) (`event sourcing.example.interface.flaskapp.IntegerSequencedItem` attribute), [115](#)
[sequence_id](#) (`event sourcing.infrastructure.sequenceditem.SequencedItem` attribute), [102](#)
[sequence_id](#) (`event sourcing.infrastructure.sequenceditem.SequencedItemFieldNames` attribute), [102](#)
[sequence_id](#) (`event sourcing.infrastructure.sqlalchemy.records.IntegerSequencedNoIDRecord` attribute), [106](#)
[sequence_id](#) (`event sourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord` attribute), [106](#)
[sequence_id](#) (`event sourcing.infrastructure.sqlalchemy.records.SnapshotRecord` attribute), [107](#)
[sequence_id](#) (`event sourcing.infrastructure.sqlalchemy.records.TimestampSequencedNoIDRecord` attribute), [107](#)
[sequence_id](#) (`event sourcing.infrastructure.sqlalchemy.records.TimestampSequencedWithIDRecord` attribute), [108](#)
[SequencedItem](#) (class in `event sourcing.infrastructure.sequenceditem`), [101](#)
[SequencedItemConflict](#), [112](#)
[SequencedItemFieldNames](#) (class in `event sourcing.infrastructure.sequenceditem`), [102](#)
[SequencedItemIterator](#) (class in `event sourcing.infrastructure.iterators`), [100](#)
[SequencedItemMapper](#) (class in `event sourcing.infrastructure.sequenceditemmapper`), [103](#)

[session \(eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore attribute\), 104](#)
[setup_cipher\(\) \(eventsourcing.application.simple.SimpleApplication method\), 81](#)
[setup_connection\(\) \(eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore method\), 93](#)
[setup_connection\(\) \(eventsourcing.infrastructure.datastore.Datastore method\), 95](#)
[setup_connection\(\) \(eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore method\), 104](#)
[setup_datastore\(\) \(eventsourcing.application.simple.SimpleApplication method\), 81](#)
[setup_event_store\(\) \(eventsourcing.application.simple.SimpleApplication method\), 81](#)
[setup_event_store\(\) \(eventsourcing.application.simple.SnapshottingApplication method\), 81](#)
[setup_persistence_policy\(\) \(eventsourcing.application.simple.SimpleApplication method\), 81](#)
[setup_persistence_policy\(\) \(eventsourcing.application.simple.SnapshottingApplication method\), 81](#)
[setup_repository\(\) \(eventsourcing.application.simple.SimpleApplication method\), 81](#)
[setup_repository\(\) \(eventsourcing.application.simple.SnapshottingApplication method\), 81](#)
[setup_table\(\) \(eventsourcing.application.simple.SimpleApplication method\), 81](#)
[setup_table\(\) \(eventsourcing.application.simple.SnapshottingApplication method\), 82](#)
[setup_table\(\) \(eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore method\), 104](#)
[setup_tables\(\) \(eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore method\), 94](#)
[setup_tables\(\) \(eventsourcing.infrastructure.datastore.Datastore method\), 95](#)
[setup_tables\(\) \(eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore method\), 104](#)
[SimpleApplication \(class in eventsourcing.application.simple\), 81](#)
[SimpleIntegerSequenceGenerator \(class in eventsourcing.infrastructure.integersequencegenerators.base\), 99](#)
[Snapshot \(class in eventsourcing.domain.model.snapshot\), 91](#)
[SnapshotRecord \(class in eventsourcing.infrastructure.cassandra.factory.CassandraInfrastructureFactory\), 94](#)
[SnapshotRecord \(class in eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory\), 105](#)
[SnapshotRecord \(class in eventsourcing.infrastructure.sqlalchemy.records\), 106](#)
[SnapshottingApplication \(class in eventsourcing.application.simple\), 81](#)
[SnapshottingPolicy \(class in eventsourcing.application.policies\), 81](#)
[SQLAlchemyDatastore \(class in eventsourcing.infrastructure.sqlalchemy.datastore\), 104](#)
[SQLAlchemyInfrastructureFactory \(class in eventsourcing.infrastructure.sqlalchemy.factory\), 105](#)
[SQLAlchemyRecordManager \(class in eventsourcing.infrastructure.sqlalchemy.manager\), 105](#)
[SQLAlchemySettings \(class in eventsourcing.infrastructure.sqlalchemy.datastore\), 105](#)
[start_new_timebucketedlog\(\) \(in module eventsourcing.domain.model.timebucketedlog\), 92](#)
[start_thread\(\) \(eventsourcing.infrastructure.iterators.ThreadedSequencedItemIterator method\), 100](#)
[started_on \(eventsourcing.domain.model.timebucketedlog.Timebucketedlog attribute\), 91](#)
[state \(eventsourcing.domain.model.snapshot.AbstractSnapshot attribute\), 90](#)
[state \(eventsourcing.domain.model.snapshot.Snapshot attribute\), 91](#)
[state \(eventsourcing.infrastructure.sequenceditem.StoredEvent attribute\), 103](#)
[state \(eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord attribute\), 107](#)
[store_event\(\) \(eventsourcing.application.policies.PersistencePolicy method\), 81](#)
[StoredEvent \(class in eventsourcing.infrastructure.sequenceditem\), 102](#)
[StoredEventRecord \(class in eventsourcing.infrastructure.sqlalchemy.records\), 107](#)
[subrepo \(eventsourcing.domain.model.array.AbstractBigArrayRepository attribute\), 83](#)
[subrepo \(eventsourcing.infrastructure.repositories.array.BigArrayRepository attribute\), 83](#)

attribute), 101

subrepo_class (eventsourcing.infrastructure.repositories.array.BigArrayRepository attribute), 101

subscribe() (in module eventsourcing.domain.model.events), 90

subscribe_to() (in module eventsourcing.domain.model.decorators), 85

T

take_snapshot() (eventsourcing.application.policies.SnapshottingPolicy method), 81

take_snapshot() (eventsourcing.domain.model.entity.AbstractEntityRepository method), 86

take_snapshot() (eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository method), 97

take_snapshot() (eventsourcing.infrastructure.snapshotting.AbstractSnapshotStrategy method), 104

take_snapshot() (eventsourcing.infrastructure.snapshotting.EventSourcedSnapshotStrategy method), 104

ThreadedSequencedItemIterator (class in eventsourcing.infrastructure.iterators), 100

Timebucketedlog (class in eventsourcing.domain.model.timebucketedlog), 91

Timebucketedlog.BucketSizeChanged (class in eventsourcing.domain.model.timebucketedlog), 91

Timebucketedlog.Event (class in eventsourcing.domain.model.timebucketedlog), 91

Timebucketedlog.Started (class in eventsourcing.domain.model.timebucketedlog), 91

TimebucketedlogReader (class in eventsourcing.infrastructure.timebucketedlog_reader), 108

TimebucketedlogRepo (class in eventsourcing.infrastructure.repositories.timebucketedlog_repo), 101

TimebucketedlogRepository (class in eventsourcing.domain.model.timebucketedlog), 91

TimeSequenceError, 113

timestamp (eventsourcing.domain.model.events.EventWithTimestamp attribute), 90

timestamp_from_datetime() (in module eventsourcing.domain.model.timebucketedlog), 92

timestamp_long_from_uuid() (in module eventsourcing.utils.times), 110

timestamp_sequenced_record_class (eventsourcing.infrastructure.cassandra.factory.CassandraInfrastructureFactory attribute), 95

timestamp_sequenced_record_class (eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory attribute), 105

TimestampedEntity (class in eventsourcing.domain.model.entity), 87

TimestampedEntity.AttributeChanged (class in eventsourcing.domain.model.entity), 87

TimestampedEntity.Created (class in eventsourcing.domain.model.entity), 87

TimestampedEntity.Discarded (class in eventsourcing.domain.model.entity), 87

TimestampedEntity.Event (class in eventsourcing.domain.model.entity), 87

TimestampedVersionedEntity (class in eventsourcing.domain.model.entity), 87

TimestampedVersionedEntity.AttributeChanged (class in eventsourcing.domain.model.entity), 87

TimestampedVersionedEntity.Created (class in eventsourcing.domain.model.entity), 87

TimestampedVersionedEntity.Discarded (class in eventsourcing.domain.model.entity), 87

TimestampedVersionedEntity.Event (class in eventsourcing.domain.model.entity), 87

TimestampSequencedNoIDRecord (class in eventsourcing.infrastructure.sqlalchemy.records), 107

TimestampSequencedRecord (in module eventsourcing.infrastructure.sqlalchemy.records), 107

TimestampSequencedWithIDRecord (class in eventsourcing.infrastructure.sqlalchemy.records), 107

TimeuuidEntity (class in eventsourcing.domain.model.entity), 88

TimeuuidVersionedEntity (class in eventsourcing.domain.model.entity), 88

to_record() (eventsourcing.infrastructure.base.RelationalRecordManager method), 93

to_record() (eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager method), 95

to_sequenced_item() (eventsourcing.infrastructure.eventstore.EventStore method), 99

to_sequenced_item() (eventsourcing.infrastructure.sequenceditemmapper.AbstractSequencedItemMapper method), 103

to_sequenced_item() (eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper method), 103

topic (eventsourcing.domain.model.snapshot.AbstractSnapshot attribute), 91

topic (eventsourcing.domain.model.snapshot.Snapshot attribute), 91

topic (eventsourcing.example.interface.flaskapp.IntegerSequencedItem attribute), 91

[attribute](#)), [115](#)
[topic \(eventsourcing.infrastructure.sequenceditem.SequencedItem](#)
[attribute](#)), [102](#)
[topic \(eventsourcing.infrastructure.sequenceditem.SequencedItemFieldNames](#)
[attribute](#)), [102](#)
[topic \(eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedNoIDRecord](#)
[attribute](#)), [106](#)
[topic \(eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord](#)
[attribute](#)), [106](#)
[topic \(eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord](#)
[attribute](#)), [107](#)
[topic \(eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedNoIDRecord](#)
[attribute](#)), [107](#)
[topic \(eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedWithIDRecord](#)
[attribute](#)), [108](#)
[TopicResolutionError](#), [113](#)
[truncate_tables\(\)](#) ([eventsourc-](#)
[ing.infrastructure.cassandra.datastore.CassandraDatastore](#)
[method](#)), [94](#)
[truncate_tables\(\)](#) ([eventsourc-](#)
[ing.infrastructure.datastore.Datastore](#) [method](#)),
[95](#)
[truncate_tables\(\)](#) ([eventsourc-](#)
[ing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore](#)
[method](#)), [105](#)

U

[unsubscribe\(\)](#) ([in](#) [module](#) [eventsourc-](#)
[ing.domain.model.events](#)), [90](#)

V

[value \(eventsourcing.domain.model.events.AttributeChanged](#)
[attribute](#)), [88](#)
[VersionedEntity](#) ([class](#) [in](#) [eventsourc-](#)
[ing.domain.model.entity](#)), [88](#)
[VersionedEntity.AttributeChanged](#) ([class](#) [in](#) [eventsourc-](#)
[ing.domain.model.entity](#)), [88](#)
[VersionedEntity.Created](#) ([class](#) [in](#) [eventsourc-](#)
[ing.domain.model.entity](#)), [88](#)
[VersionedEntity.Discarded](#) ([class](#) [in](#) [eventsourc-](#)
[ing.domain.model.entity](#)), [88](#)
[VersionedEntity.Event](#) ([class](#) [in](#) [eventsourc-](#)
[ing.domain.model.entity](#)), [88](#)

W

[write_records\(\)](#) ([eventsourc-](#)
[ing.infrastructure.base.RelationalRecordManager](#)
[method](#)), [93](#)