

---

# **eventsourcing Documentation**

***Release 7.1.1***

**John Bywater**

**Jun 21, 2019**



---

## Contents

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	What is event sourcing? . . . . .	3
1.2	This library . . . . .	3
1.3	A cohesive mechanism . . . . .	4
1.4	Register issues . . . . .	4
<b>2</b>	<b>Support options</b>	<b>5</b>
2.1	Community support . . . . .	5
2.2	Training workshops . . . . .	5
2.3	Advice and development . . . . .	5
<b>3</b>	<b>Quick start</b>	<b>7</b>
3.1	Install library . . . . .	7
3.2	Define model . . . . .	7
3.3	Configure environment . . . . .	9
3.4	Run application . . . . .	9
<b>4</b>	<b>Design</b>	<b>13</b>
4.1	Layered architecture . . . . .	13
4.2	Domain event store . . . . .	13
<b>5</b>	<b>Features</b>	<b>15</b>
5.1	Core features . . . . .	15
5.2	Additional features . . . . .	15
<b>6</b>	<b>Installation guide</b>	<b>17</b>
6.1	Install options . . . . .	17
<b>7</b>	<b>Domain model layer</b>	<b>19</b>
7.1	Domain events . . . . .	20
7.2	Domain entities . . . . .	23
7.3	Aggregate root . . . . .	30
<b>8</b>	<b>Infrastructure layer</b>	<b>35</b>
8.1	Sequenced item type . . . . .	36
8.2	Sequenced item mapper . . . . .	37
8.3	Record managers . . . . .	41

8.4	Event store . . . . .	47
<b>9</b>	<b>Application layer</b>	<b>53</b>
9.1	Overview . . . . .	53
9.2	Simple application . . . . .	54
9.3	Custom application . . . . .	57
<b>10</b>	<b>Notifications</b>	<b>61</b>
10.1	Three options . . . . .	62
10.2	Application sequence . . . . .	64
10.3	Notification logs . . . . .	70
10.4	Local notification logs . . . . .	70
10.5	Remote notification logs . . . . .	73
10.6	Notification log reader . . . . .	76
<b>11</b>	<b>Projections</b>	<b>79</b>
11.1	Overview . . . . .	79
11.2	Subscribing to events . . . . .	80
11.3	Tracking notification logs . . . . .	80
<b>12</b>	<b>Distributed systems</b>	<b>89</b>
12.1	Overview . . . . .	90
12.2	Process application . . . . .	90
12.3	System of processes . . . . .	92
12.4	Example . . . . .	94
12.5	Integration with APIs . . . . .	105
12.6	Process event pattern . . . . .	106
<b>13</b>	<b>Snapshotting</b>	<b>107</b>
13.1	Domain . . . . .	107
13.2	Application . . . . .	107
13.3	Run the code . . . . .	108
<b>14</b>	<b>Deployment</b>	<b>111</b>
14.1	Application object . . . . .	112
14.2	Database connection . . . . .	114
14.3	Web interfaces . . . . .	115
14.4	Task queues . . . . .	118
<b>15</b>	<b>Stand-alone example</b>	<b>121</b>
15.1	Domain . . . . .	122
15.2	Infrastructure . . . . .	126
15.3	Application . . . . .	129
15.4	Run the code . . . . .	131
<b>16</b>	<b>Background</b>	<b>133</b>
<b>17</b>	<b>Release notes</b>	<b>135</b>
17.1	Version 7.x . . . . .	135
17.2	Version 6.x . . . . .	135
17.3	Version 5.x . . . . .	136
17.4	Version 4.x . . . . .	136
17.5	Version 3.x . . . . .	136
17.6	Version 2.x . . . . .	137
17.7	Version 1.x . . . . .	137

17.8 Version 0.x . . . . .	138
<b>18 Contributing</b>	<b>141</b>
<b>19 Module docs</b>	<b>143</b>
19.1 eventsourcing . . . . .	143
<b>Python Module Index</b>	<b>211</b>
<b>Index</b>	<b>213</b>



A library for event sourcing in Python.





This documentation:

- highlights the *design* and *features* of the library,
- has instructions for *installing* the package,
- describes the *domain model layer*, the *infrastructure layer*, and the *application layer*,
- shows how *notifications* and *projections* can be combined to make a *reliable distributed system*,
- has information about *deployment*, and
- has some *background* information about the project.

## 1.1 What is event sourcing?

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.

## 1.2 This library

This is a library for event sourcing in Python. At its core, this library supports storing and retrieving sequences of items, such as the domain events of aggregates in a event-sourced domain driven design.

To demonstrate how its persistence mechanism can be used effectively, this library documentation has examples of event-sourced applications with event-sourced domain models. The library base classes used in these examples can be conveniently used to create your own applications. A style is suggested for writing event-sourced domain models, and event-sourced aggregates that have command methods which trigger domain events.

Using this library, it is also possible to define an entire distributed system of event-sourced applications independently of infrastructure. That means system behaviours can be rapidly developed whilst running the entire system synchronously in a single thread with a single in-memory database, and then the system can be run asynchronously on a cluster with durable databases, with the system performing exactly the same behaviour.

## 1.3 A cohesive mechanism

Quoting from Eric Evans' book [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.”*

Although the basic event sourcing patterns are quite simple, and can be reproduced in code for each project, the persistence mechanism for event sourced domain driven design appears as a conceptually cohesive mechanism, and so have been “partitioned into a separate lightweight framework”.

## 1.4 Register issues

This project is [hosted on GitHub](#). Please [register any issues, questions, and requests](#) you may have.

Although the library is free to use, it can take some time and effort to understand the library and to acquire sufficient skill to make a working and well-designed event-sourced system. Training workshops and development services are available to support professional users of the library in accomplishing their goals. Community support is also available.

## 2.1 Community support

The library has a growing community that may be able to help.

- You may ask questions on the [Slack channel](#).
- You may also register issues and requests on our [issue tracker](#).

## 2.2 Training workshops

Training workshops are available to help developers more quickly learn how to use the library. Workshop participants will be guided through a series of topics, gradually discovering what the library is capable of doing, and learning how to use the library effectively.

Please contact the [Appropriate Software Foundation](#) for more information about available training workshops.

## 2.3 Advice and development

Advice and development services are available to help managers and developers with the design and development of their event-sourced applications and systems.

- Overall assessment of your existing implementation, with recommendations for improvement.
- Address your specific concerns with how your event-sourced system is built (and running).
- Coaching developers in the use of the library.

- Development of sample applications and systems for guidance or demonstration purposes.
- Development of working applications and systems for production use.

Please contact the [Appropriate Software Foundation](#) for more information about available consulting services.

## CHAPTER 3

---

### 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.

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

### 3.1 Install library

Please use pip to install the library with the ‘sqlalchemy’ option.

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

See the [Install documentation](#) for more information about installing the library.

### 3.2 Define model

Having installed the library, open a Python file in your favourite editor. The code snippets below form a working program. You can either type in the code, or copy and paste. Then run the program.

Firstly let’s define a domain model, by defining an aggregate class. The class `World` defined below is a subclass of `AggregateRoot`. The class `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.what)

```

The most important thing here is to understand that an aggregate command method, such as `make_it_so()` above, should not do some work and then update the attributes of the aggregate with the result of the work, but rather it should trigger events with the results of the work, so that the events can be used to update the state of the aggregate then and, importantly, each time the aggregate is reconstructed from its events.

This aggregate class can be used without any infrastructure. Without infrastructure however, there is no means of recovering the state of the aggregate once the object goes out of scope.

```

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

assert world.ruler == 'gods'

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

# Assign attribute.
world.ruler = 'money'

assert world.history == ('dinosaurs', 'trucks'), world.history
assert world.ruler == 'money'

```

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.

See the [Domain model documentation](#) for more information about developing event-sourced domain models.

## 3.3 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:'
```

## 3.4 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.sqlalchemy import SQLAlchemyApplication
from eventsourcing.exceptions import ConcurrencyError

# Construct simple application (used here as a context manager).
with SQLAlchemyApplication(persist_event_type=World.Event) 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] == 'internet'
    assert world.history[1] == 'trucks'
    assert world.history[0] == 'dinosaurs'

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

(continues on next page)

(continued from previous page)

```
# 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] == 'internet'
assert copy.history[1] == 'trucks'
assert copy.history[0] == 'dinosaurs'

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

# Discard aggregate.
world.__discard__()
world.__save__()

# Discarded aggregate is not found.
assert world.id not in app.repository
try:
    # Repository raises key error.
    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] == '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__

# Project application event notifications.
from event_sourcing.interface.notification_log import NotificationLogReader
reader = NotificationLogReader(app.notification_log)
notifications = reader.read()
```

(continues on next page)



(continued from previous page)

```
notification_ids = [n['id'] for n in notifications]
assert notification_ids == [1, 2, 3, 4, 5, 6]

# 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
```

See the [Application documentation](#) for more information about event-sourced applications, and the [Infrastructure documentation](#) for more information about infrastructure.



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

The library is designed to allow its default functionality to be extended or replaced easily.

### 4.1 Layered architecture

Any interface layer will depend on the application layer.

The library's application layer depends on the domain model and infrastructure layers. An application object has a repository, from which existing aggregates can be retrieved. It may also have policies, such as a persistence policy which stores domain events in an event store. The application's repository shares the event store with the persistence policy, and uses the event store to retrieve events when reconstructing the state of an aggregate.

The library's domain layer contains domain model events and aggregates. Aggregates define a set of domain event classes, and have command methods to trigger new domain events. Nothing in the domain layer depends on anything in the infrastructure layer. These stand-alone library classes are implemented with "double underscore" methods, to keep the normal object namespace free to be used for domain modelling.

The library's infrastructure layer encapsulates infrastructural services required by event sourced applications, in particular by the event store. This layer is the original core of this library (the other layers were originally included merely as reference examples, to demonstrate how to use the infrastructure).

### 4.2 Domain event store

The central object of the infrastructure layer is the event store. The event store object uses a sequenced item mapper and a record manager. Domain events are serialised to (and deserialised from) sequenced items by the sequenced item mapper. The record manager records (and reads) sequenced items in a particular database system.



### 5.1 Core 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.

**Layer 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.

**Notifications and projections** — reliable propagation of application events with pull-based notifications allows the application state to be projected accurately into replicas, indexes, view models, and other applications.

**Process and systems** — scalable event processing with application pipelines. Parallel pipelines are synchronised with causal dependencies. Runnable with single thread, multiprocessing on a single machine, and in a cluster of machines using the actor model.

### 5.2 Additional features

**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.

**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.

**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.

**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).

**Worked examples** — simple example application and systems, with an example entity class, example domain events, and an example database table. Plus lots of examples in the documentation.

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

```
$ pip install eventsourcing
```

## 6.1 Install options

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.

### 6.1.1 SQLAlchemy

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

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

You can use SQLAlchemy with SQLite, in which case you don’t need to install any database driver. Otherwise please also install a [database driver](#) that works with SQLAlchemy and your database system.

```
$ pip install psycopg2-binary # for PostgreSQL
$ pip install pymysql        # for MySQL
```

See the [Infrastructure doc](#) for more information about crafting database connection strings for particular database drivers.

### 6.1.2 Django

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

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

You can use Django with SQLite, in which case you don't need to install any database driver. Otherwise please also install a database driver that works with [Django and your database system](#).

### 6.1.3 Cassandra

If you want to use [Apache Cassandra](#), then please install with the 'cassandra' option.

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

### 6.1.4 Tests

If you want to run the tests, then please install with the 'tests' option.

```
$ pip install eventsourcing[tests]
```

### 6.1.5 Docs

If you want build the docs, then please install with the 'docs' option.

```
$ pip install eventsourcing[docs]
```



---

## Domain model layer

---

The library's domain layer has base classes for domain events and entities. They can be used to develop an event-sourced domain model.

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

## 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 event_sourcing.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)
```

### 7.1.1 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):
```

(continues on next page)

(continued from previous page)

```

    return isinstance(event, DomainEvent)

subscribe(handler=receive_event, predicate=is_domain_event)

# Publish the domain event.
publish(domain_event)

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

```

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

```

from eventsourcing.domain.model.events import unsubscribe

unsubscribe(handler=receive_event, predicate=is_domain_event)

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

```

## 7.1.2 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 event sourcing.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
```

### 7.1.3 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')
```

(continues on next page)

(continued from previous page)

```
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).

## 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
```

### 7.2.1 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)
```

## 7.2.2 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.

## 7.2.3 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,
)

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

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

The 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
```

(continues on next page)

(continued from previous page)

```
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.

## 7.2.4 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
```

## 7.2.5 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

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

## 7.2.6 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:
```

(continues on next page)

(continued from previous page)

```
entity.__change_attribute__('full_name', 'Mr Boots')
except EntityIsDiscarded:
    pass
else:
    raise Exception("Shouldn't get here")
```

## 7.2.7 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.

## 7.2.8 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
```

(continues on next page)

(continued from previous page)

```
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()
```

## 7.2.9 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')
```

## 7.2.10 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(VersionedEntity):

    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(VersionedEntity.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'
```

## 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)
```

(continues on next page)

(continued from previous page)

```
# 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
```

### 7.3.1 Data integrity

Domain events triggered by the library's `AggregateRoot` class are both hashed and also hash-chained together.

The state of each event, including the hash of the previous event, is hashed using SHA-256. The state of each event can be validated as a part of the chain. 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 aggregate root is available as an attribute called `__head__` of the aggregate root.

```
# Entity's head hash is determined exclusively
# by the entire sequence of events and SHA-256.
assert world.__head__ == received_events[0][-1].__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 against an independent record.

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

```
from event sourcing.utils.random import encode_random_bytes

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

(continues on next page)

(continued from previous page)

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

The “genesis hash” used as the previous hash of the first event in a sequence can be set using environment variable `GENESIS_HASH`.

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





---

### Infrastructure layer

---

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*

## 8.1 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 state of the item.

### 8.1.1 SequencedItem namedtuple

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

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

Like in the example above, the library's `SequencedItem` namedtuple has four fields. The `sequence_id` identifies the sequence to 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 `state` holds the state of 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 state is a JSON string in which `foo` is `bar`.

```
from uuid import uuid4

sequence1 = uuid4()

state = '{"foo":"bar","position":0,"sequence_id":{"UUID":"%s"}}' % sequence1.hex

sequenced_item1 = SequencedItem(
    sequence_id=sequence1,
    position=0,
    topic='eventsourcing.domain.model.events#DomainEvent',
```

(continues on next page)

(continued from previous page)

```
state=state,
)
```

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 == 'eventsourcing.domain.model.events#DomainEvent'
assert sequenced_item1.state == state, sequenced_item1.state
```

## 8.1.2 StoredEvent namedtuple

The library provides a sequenced item named tuple called `StoredEvent`. The attributes of the `StoredEvent` namedtuple are `originator_id`, `originator_version`, `topic`, 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 `topic` 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 `state` above.

```
from eventsourcing.infrastructure.sequenceditem import StoredEvent

aggregate1 = uuid4()

stored_event1 = StoredEvent(
    originator_id=aggregate1,
    originator_version=0,
    topic='eventsourcing.domain.model.events#DomainEvent',
    state='{"foo":"bar","originator_version":0,"originator_id":{"UUID":"%s"}}' %
    ↪ aggregate1.hex,
)
assert stored_event1.originator_id == aggregate1
assert stored_event1.originator_version == 0
assert stored_event1.topic == 'eventsourcing.domain.model.events#DomainEvent'
assert stored_event1.state == '{"foo":"bar","originator_version":0,"originator_id":{
    ↪ "UUID":"%s"}}' % aggregate1.hex
```

## 8.2 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 eventsourcing.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 `event_from_item()` can be used to convert sequenced item objects to application-level objects.

```
domain_event = sequenced_item_mapper.event_from_item(sequenced_item1)

assert domain_event.foo == 'bar'
```

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

```
recovered_state = sequenced_item_mapper.item_from_event(domain_event).state
assert recovered_state == sequenced_item1.state, (recovered_state, sequenced_item1.
↪state)
```

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.item_from_event(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.event_from_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 event_sourcing.domain.model.events import Created
from event_sourcing.utils.topic import get_topic, resolve_topic

topic = get_topic(Created)
```

(continues on next page)

(continued from previous page)

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

## 8.2.1 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,
    sequenced_item_class=StoredEvent,
)

domain_event = customized_sequenced_item_mapper.event_from_item(
    StoredEvent(
        originator_id=sequence1,
        originator_version=0,
        topic='eventsourcing.domain.model.events#DomainEvent',
        state='{"foo":{"__set__":["bar","baz"]},"originator_version":0,"originator_id'
        ↪":{"UUID":"%s"}}' % sequence1
        .hex,
    )
)

assert domain_event.foo == set(["bar", "baz"])

sequenced_item = customized_sequenced_item_mapper.item_from_event(domain_event)
assert sequenced_item.state.startswith('{"foo":{"__set__":["ba']
```

## 8.2.2 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 state 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_bytes()` decodes the unicode key string into a sequence of bytes.

```
from eventsourcing.utils.cipher.aes import AESCipher
from eventsourcing.utils.random import encode_random_bytes, decode_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_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.item_from_event(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.
```

(continues on next page)

(continued from previous page)

```
domain_event = ciphered_sequenced_item_mapper.event_from_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.

## 8.3 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 `record()` 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.

### 8.3.1 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,
    application_name=uuid4().hex
)
```

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

```
record_manager.record_sequenced_item(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 `mysql-connector-python-rf` can be used (licenced GPL v2). Please note, I had problems running this driver with Python 2.7 (unicode error when it raises exceptions).

```
$ pip install pymysql-connector-python-rf
```

The uri for MySQL used with this driver would look something like this.

```
mysql+pymysql://username:password@localhost/eventsourcing?charset=utf8mb4&binary_
↳prefix=true
```

Alternatively for MySQL, the Python package `mysqlclient` can be used (also licenced GPL v2). I didn't have problems using this driver with Python 2.7.

```
$ pip install mysqlclient
```

The uri for MySQL used with this driver would look something like this.

```
mysql+mysqldb://username:password@localhost/eventsourcing?charset=utf8mb4&binary_
↳prefix=true
```

Another alternative is `PyMySQL`. It has a BSD licence.

```
$ pip install PyMySQL
```

The uri for MySQL used with this driver would look something like this.

```
mysql+pymysql://username:password@localhost/eventsourcing?charset=utf8mb4&binary_
↳prefix=true
```

## PostgreSQL

For PostgreSQL, the Python package `psycopg2` can be used.

```
$ pip install psycopg2
```

The uri for PostgreSQL used with this driver would look something like this.

```
postgresql+psycopg2://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.

### 8.3.2 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', verbosity=0, interactive=False)
```

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,
    application_name='demo',
)

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

django_record_manager.record_sequenced_item(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.

### 8.3.3 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 was 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 databases indexes supports this purpose well. The transaction isolation level must be at least "read committed", which is true by default for MySQL and PostgreSQL.

Any resulting contention in the record ID will raise an exception so that the query can be retried. The library exception class `RecordConflictError` will be raised.

### 8.3.4 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
```

(continues on next page)

(continued from previous page)

```
cassandra_record_manager.record_sequenced_item(stored_event1)

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

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

### 8.3.5 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 `RecordConflictError` will be raised.

```
from eventsourcing.exceptions import RecordConflictError

# Fail to append an item at the same position in the same sequence as a previous item.
try:
    record_manager.record_sequenced_item(stored_event1)
except RecordConflictError:
    pass
else:
    raise Exception("RecordConflictError 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 an actor framework or [Zookeeper](#) might help.

## 8.4 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 `store()` method can store a domain event in 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 record a sequenced item in the database.

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

```
event_store.store(
    DomainEvent(
        originator_id=aggregatel,
        originator_version=1,
        foo='baz',
    )
)
```

The event store's method `get_domain_events()` is used to get events that have previously been stored. The event store uses the `record_manager` to get the sequenced items from database records, 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'
```

### 8.4.1 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.store(
        DomainEvent(
            originator_id=aggregate1,
            originator_version=1,
            foo='baz',
        )
    )
except ConcurrencyError:
    pass
else:
    raise Exception("ConcurrencyError not raised")
```

This feature depends on the behaviour of the record manager's `append()` method: the event store will raise a `ConcurrencyError` if a `RecordConflictError` 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
```

This feature avoids the sequence of records being corrupted due to concurrent threads operating on the same aggregate. However, the result is that success of appending an event in such circumstances is only probabilistic with respect to concurrency conflicts. Concurrency conflicts can be avoided if all commands for a single aggregate are executed in series, for example by treating each aggregate as an actor within an actor framework, or with locks provided by something like Zookeeper.

## 8.4.2 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,
    application_name=uuid4().hex,
    contiguous_record_ids=True,
)
```

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.

## 8.4.3 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 eventsourcing.infrastructure.sqlalchemy.records import TimestampSequencedRecord
from eventsourcing.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.store(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.



---

### Application layer

---

This section discusses how an *event-sourced domain model* can be combined with *library infrastructure* to make an event sourced application. The normal layered architecture of an enterprise application is followed: an application layer supports an interface layer and depends on both a *domain layer* and an *infrastructure layer*.

- *Overview*
- *Simple application*
  - *Application object*
  - *Event store*
  - *Persistence policy*
  - *Repository*
  - *Notification log*
- *Custom application*
  - *Run the code*
  - *Stored events*
  - *Sequenced items*
  - *Database records*
  - *Close*

### 9.1 Overview

In this library, an application object has an event store which encapsulates infrastructure required to store the state of an application as a sequence of domain events. An application also has a persistence subscriber, a repository, and a

notification log. The event store is used by the repository and notification log to retrieve events, and by the persistence subscriber to store events.

The state of an application is partitioned across a set of event-sourced “aggregates” (domain entities). Each aggregate has a unique ID, and the state of an aggregate is determined by a sequence of domain events. Aggregates implement commands which trigger domain events that mutate the state of their aggregate by augmenting the aggregate’s sequence of events. The repository of an application allows individual aggregates of the application to be retrieved by ID, optionally at a particular version.

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 event sourcing[sqlalchemy]
```

## 9.2 Simple application

The library provides an abstract base class for applications called *SimpleApplication*. This application class is independent of infrastructure, and as such can be used to define applications independently of infrastructure, but also can’t be constructed directly.

For that reason, the example below uses a subclass that depends on SQLAlchemy. The base class is extended by *SQLAlchemyApplication* which uses SQLAlchemy to store and retrieve domain event records.

### 9.2.1 Application object

The SQLAlchemy application class has a `uri` constructor argument, which is an (SQLAlchemy-style) database connection string. The example below uses SQLite with an in-memory database (which is also the default). You can use any valid connection string.

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

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` for PostgreSQL or `pymysql` or even `mysql-connector-python-rf` for MySQL).

```
# SQLite with a file on disk.
sqlite:///tmp/mydatabase

# PostgreSQL with psycopg2.
postgresql+psycopg2://scott:tiger@localhost:5432/mydatabase

# MySQL with pymysql.
mysql+pymysql://scott:tiger@hostname/dbname?charset=utf8mb4&binary_prefix=true

# MySQL with mysql-connector-python-rf.
mysql+sqlconnector://scott:tiger@hostname/dbname
```

In case you were wondering, the `uri` value is used to construct an SQLAlchemy thread-scoped session facade.

Instead of providing a `uri` value, an already existing SQLAlchemy session can be passed in, using constructor argument `session`. For example, a session object provided by a framework integrations such as [Flask-SQLAlchemy](#) could be passed to the application object.

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`. The `persist_event_type` value determines which types of domain event will be persisted by the application. So that different applications can be constructed in the same process, the default value of `persist_event_type` is `None`.

```
from eventsourcing.application.sqlalchemy import SQLAlchemyApplication
from eventsourcing.domain.model.aggregate import AggregateRoot

application = SQLAlchemyApplication(
    uri='sqlite:///memory:',
    cipher_key=cipher_key,
    persist_event_type=AggregateRoot.Event,
)
```

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`.

## 9.2.2 Event store

Once constructed, the application object has an event store, provided by the library's *EventStore* class.

```
from eventsourcing.infrastructure.eventstore import EventStore

assert isinstance(application.event_store, EventStore)
```

## 9.2.3 Persistence policy

The application has a persistence policy, an instance of the library class *PersistencePolicy*. The persistence policy uses the event store.

```
from eventsourcing.application.policies import PersistencePolicy

assert isinstance(application.persistence_policy, PersistencePolicy)
```

The persistence policy will only persist particular types of domain events. The application class attribute `persist_event_type` is used to define which classes of domain events will be persisted by the application's persistence policy.

## 9.2.4 Repository

The application also has a repository, an instance of the library class *EventSourcedRepository*. The repository is generic, and can retrieve all aggregates in an application, regardless of their class. That is, there aren't

different repositories for different types of aggregate in this application. The repository also uses the event store.

```
from eventsourcing.infrastructure.eventourcedrepository import EventSourcedRepository

assert isinstance(application.repository, EventSourcedRepository)
```

The library class `AggregateRoot` can be used directly to create a new aggregate object that is available in the application's repository.

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

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

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

Concurrency errors can be avoided if all commands for a single aggregate are executed in series, for example by treating each aggregate as an actor, within an actor framework.

## 9.2.5 Notification log

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(application.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__()
```

(continues on next page)

(continued from previous page)

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

## 9.3 Custom application

Firstly, a custom aggregate root class called `CustomAggregate` is defined below. It extends the library's `AggregateRoot` entity with event-sourced 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*.

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

The `persist_event_type` value can be set as a class attribute.

```
from eventsourcing.application.sqlalchemy import SQLAlchemyApplication

class MyApplication(SQLAlchemyApplication):
    persist_event_type = AggregateRoot.Event

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

### 9.3.1 Run the code

The custom application object can be constructed.

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

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

```
# Create aggregate using application service, and save it.
aggregate = application.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 application.repository

# Get aggregate using dictionary-like interface.
aggregate = application.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 = application.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__()
aggregate.__save__()

# Check discarded aggregate no longer exists in repository.
assert aggregate.id not in application.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:
    application.repository[aggregate.id]
except KeyError:
    pass
else:
    raise Exception("Shouldn't get here")
```

## 9.3.2 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 = application.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
```

(continues on next page)



(continued from previous page)

```
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)
```

### 9.3.3 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 = application.event_store.record_manager.list_items(aggregate.id)
assert len(items) == 4

assert items[0].originator_id == aggregate.id
assert items[0].topic == '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].topic == '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].topic == '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].topic == 'eventsourcing.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.

### 9.3.4 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.

```
application.event_store.record_manager.record_class
```

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

```
event_records = application.event_store.record_manager.orm_query().all()

assert len(event_records) == 4
```

### 9.3.5 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.
application.close()
```

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

If the domain events of an application can somehow be placed in a sequence, then the sequence of events can be propagated as a sequence of notifications.

- *Three options*
  - *Bounded context controls notifications*
- *Application sequence*
  - *Timestamps*
  - *Integers*
  - *Record managers*
  - *BigArray*
- *Notification logs*
- *Local notification logs*
  - *RecordManagerNotificationLog*
  - *Notification records*
  - *BigArrayNotificationLog*
- *Remote notification logs*
  - *NotificationLogView*
  - *Notification API*
  - *RemoteNotificationLog*
- *Notification log reader*

## 10.1 Three options

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”*

He continues by describing three options. 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. The simple logic of an ascending sequence of integers can allow others to progress along an application’s sequence of events. It is also possible to use timestamps.

The first option implies that each event sourced application functions cohesively also as a messaging service. Assuming that “messaging service” means an AMQP system, it seems impractical in a small library such as this to implement an AMQP broker, something that works with all of the library’s record manager classes. However, perhaps if Kafka could be adapted as a record manager, it could be used both to persist and propagate events.

The second option, which is similar to the first, involves using a separate messaging infrastructure within a two-phase commit, which could be possible, but seems unattractive. At least RabbitMQ can’t participate in a two-phase commit.

The third option is pursued below.

### 10.1.1 Bounded context controls notifications

The third option doesn’t depend on messaging infrastructure, but does involve “notifications”. If the events of an application can be placed in a sequence, the sequence of events can be presented as a sequence of notifications, with notifications being propagated in a standard way. For example, notifications could be presented by a server, and clients could pull notifications they don’t have, in linked sections, perhaps using a standard format and standard protocols.

Pulling new notifications could resume whenever a “prompt” is received via an AMQP system. This way, the client doesn’t have the latency or intensity of a polling interval, the messaging infrastructure doesn’t need to deliver messages in order (which AMQP messaging systems don’t normally provide), and message size is minimised.

If the “substantive” changes enacted by a receiver are stored atomically with the position in the sequence of notifications, the receiver can resume by looking at the latest record it has written, and from that record identify the next notification to consume. This is obvious when considering pure replication of a sequence of events. But in general, by storing in the produced sequence the position of the receiver in the consumed sequence, and using optimistic concurrency control when records are inserted in the produced sequence, at least from the point of view of consumers of the produced sequence, “exactly once” processing can be achieved. Redundant (concurrent) processing of the same sequence would then be possible, which may help to avoid interruptions in processing the application’s events.

If placing all the events in a single sequence is restrictive, then perhaps partitioning the application notifications may offer a scalable approach. For example, if each user has one partition, each notification sequence would contain all events from the aggregates pertaining to one user. So that the various sequences can be discovered, it would be useful to have a sequence in which the creation of each partition is recorded. The application could then be scaled by partition. (Please note, the library doesn’t currently support partitioning the aggregates of an application.)

If partitioning the aggregates of application is restrictive, then it would also be possible to have a notification sequence for each aggregate; in this case, the events would already have been placed in a sequence and so they could be presented

directly as notifications. But as aggregates come and go, the overhead of keeping track of the notification sequences may become restrictive. (Please note, the library doesn't currently support notifications for individual aggregates.)

An alternative to sequencing events individually would be to sequence the lists of events published when saving the pending events of an aggregate (see `AggregateRoot`), so that in cases where it is feasible to place all the commands in a sequence, it would also be feasible to place the resulting lists of events in a sequence, assuming the number of such lists is less than or equal to the number of commands. Sequencing the lists would allow these little units of coherence to be propagated, which may be useful in some cases. (Please note, the library doesn't currently support notifications for lists of events.)

Before continuing with code examples below, we need to setup an event store.

```
from uuid import uuid4

from eventsourcing.application.policies import PersistencePolicy
from eventsourcing.domain.model.entity import DomainEntity
from eventsourcing.infrastructure.eventstore import EventStore
from eventsourcing.infrastructure.repositories.array import BigArrayRepository
from eventsourcing.infrastructure.sequenceditem import StoredEvent
from eventsourcing.infrastructure.sequenceditemmapper import SequencedItemMapper
from eventsourcing.infrastructure.sqlalchemy.datastore import SQLAlchemyDatastore, \
    SQLAlchemySettings
from eventsourcing.infrastructure.sqlalchemy.manager import SQLAlchemyRecordManager
from eventsourcing.infrastructure.sqlalchemy.records import StoredEventRecord

# 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,
    application_name=uuid4().hex,
)

# 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,
    persist_event_type=DomainEntity.Event
)
```

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

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

## 10.2 Application sequence

The fundamental concern here is to propagate the events of an application without events being missed, duplicated, or jumbled.

The events of an application sequence could be sequenced with either timestamps or 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.

Sequencing with integers involves generating a sequence of integers, which is easy to follow, but can limit the rate at which records can be written. Using timestamps allows records to be inserted independently of others, but timestamps can cause uncertainty when following the events of an application.

If an application's domain model involves the library's `AggregateRoot` class, which publishes all pending events together as a list, rather than inserting each event, it would be possible to insert the lists of events into the application sequence as a single entry. This may reduce the number of inserts into the application sequence. The lists could be sequenced by timestamp or integer. Timestamps may allow the greatest write-speed. (This approach currently hasn't been explored any further, but it should be.)

### 10.2.1 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 others.

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 (which randomizes the order of otherwise "simultaneous" events).

These techniques (and others) 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 ordered by timestamp. If you use this class in this way, make sure your clocks are in sync, and query events from the last position until a time in the recent past, in order to implement a jitter buffer.

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.)

## 10.2.2 Integers

To reproduce the application's sequence of events perfectly, without any risk of gaps or duplicates or jumbled items, or race conditions, we can generate and then follow a contiguous sequence of integers. It is also possible to generate and follow a non-contiguous sequence of integers, but the gaps will need to be negotiated, by guessing how long an unusually slow write would take to become visible, since such gaps could be filled in the future.

The library's relational record managers with record classes that have an indexed integer ID column. Record IDs are 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, which depends on the insert statement.

By default, these library record classes have an auto-incrementing ID, which will generate an increasing sequence as records are inserted, but which may have gaps if an insert fails. Optionally, the record managers can also be used to generate contiguous record IDs, with an "insert select from" SQL statement that, as a clause in the insert statement, selects the maximum record ID from the visible table records. Since it is only possible to extend the sequence, the visible record IDs will form a contiguous sequence, which is the easiest thing to follow, because there is no possibility for race conditions where events appear behind the last visible event. The "insert select from" statement will probably be slower than the default "insert values" and the auto-incrementing ID, and only one of many concurrent inserts will be successful. Exceptions from concurrent inserts could be mitigated with retried, and avoided entirely by serialising the inserts with a queue, for example in an actor framework. Although this will smooth over spikes, and unfortunate coincidences will be avoided, the continuous maximum throughput will not be increased, a queue will eventually reach a limit and a different exception will be raised.

Given the rate limit, it could take an application quite a long time to fill up a well provisioned database table. Nevertheless, if the rate of writing or the volume of domain event records in your system inclines you towards partitioning the table of stored events, or if anyway your database works in this way (e.g. Cassandra), then the table would need to be partitioned by sequence ID so that the aggregate performance isn't compromised by having its events distributed across partitions, which means 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.

The `BigArray` can be used to construct both contiguous and non-contiguous integer sequences. As with the record IDs above, if each item is positioned in the next position after the last visible record, then a contiguous sequence is generated, but at the cost of finding the last visible record. However, if a number generator is used, the rate is limited by the rate at which numbers can be issued, but if inserts can fail, then numbers can be lost and the integer sequence will have gaps.

## 10.2.3 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 `get_notifications()` method of the record manager. The record IDs will form a contiguous sequence, suitable for the `RecordManagerNotificationLog`.

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

notifications = record_manager.get_notifications()

assert len(notifications) == 0, notifications

first_entity = VersionedEntity.__create__()

notifications = record_manager.get_notifications(start=0, stop=5)

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

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

## 10.2.4 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 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'
```

(continues on next page)

(continued from previous page)

```
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 event sourcing.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 event sourcing.infrastructure.integersequencegenerators.redisincr import RedisIncr

integers = RedisIncr()
generated = []
for i in integers:
```

(continues on next page)

(continued from previous page)

```
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.

Please note: there is an unimplemented enhancement which would allow this data structure to be modified in a single transaction, because the new non-leaf nodes can be determined from the position of the new leaf node, however currently a less optimal approach is used which attempts to add all non-leaf nodes and carries on in case of conflicts.

## 10.3 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.

## 10.4 Local notification logs

### 10.4.1 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)
```

(continues on next page)

(continued from previous page)

```
# 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 notifications (`section.items`) from `RecordManagerNotificationLog` are Python dicts with keys: `id`, `topic`, `state`, `originator_id`, `originator_version`, and `casual_dependencies`.

A domain event can be obtained from a notification by calling the method `event_from_topic_and_state()` on a sequenced item mapper, passing in the `topic` and `state` values. 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 state. The notification's state is simply the stored event state, so if the record data was encrypted, the notification data will also be encrypted. The sequenced item mapper needs to be configured accordingly.

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.event_from_topic_and_state(
            topic=notification['topic'],
            state=notification['state']
        ) for notification in notifications
    ]

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

from eventsourcing.domain.model.array import ItemAssigned
assert type(domain_events[0]) == VersionedEntity.Created
assert type(domain_events[1]) == ItemAssigned
assert type(domain_events[2]) == ItemAssigned
assert type(domain_events[3]) == ItemAssigned
assert type(domain_events[4]) == ItemAssigned

assert domain_events[0].originator_id == first_entity.id
assert domain_events[1].item == 'item0'
assert domain_events[3].item == 'item1'
assert domain_events[4].item == 'item2'
```

If the notification data was encrypted by the sequenced item mapper, the sequenced 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.

## 10.4.2 Notification records

An application could write separate notification records and event records. Having separate notification records allows notifications to be arbitrarily and therefore evenly distributed across a variable set of notification logs.

The number of logs could be governed automatically by a scaling process so the cadence of each notification log is actively controlled to a constant level.

Todo: Merge these paragraphs, remove repetition (params below were moved from projections doc).

When an application has one notification log, any causal ordering between events will be preserved in the log: you won't be informed that something changed without previously being informed that it was created. But if there are many notification logs, then it would be possible to record causal ordering between events: the notifications recorded for the last events that were applied to the aggregates used when triggering new events can be included in the notifications for the new events. This avoids downstream needing to: serialise everything in order to recover order e.g. by merge sorting all logs by timestamp; partitioning the application state; or to ignore causal ordering. For efficiency, prior events that were notified in the same log wouldn't need to be included. So it would make sense for all the events of a particular aggregate to be notified in the same log, but if necessary they could be distributed across different notification logs without downstream processing needing to be incoherent or bottle-necked. To scale data, it might become necessary to fix an aggregate to a notification log, so that many databases can be used with each having the notification records and the event records together (and any upstream notification tracking records) so that atomic transactions for these records are still workable.

If all events in a process are placed in the same notification log sequence, since a notification log will need to be processed in series, the throughput is more or less limited by the rate at which a sequence can be processed by a single thread. To scale throughput, the application event notifications could be distributed into many different notification logs, and a separate operating system process (or thread) could run concurrently for each log. A set of notification logs could be processed by a single thread, that perhaps takes one notification in turn from each log, but with parallel processing, total throughput could be proportional to the number of notification logs used to propagate the domain events of an application.

Causal ordering can be maintained across the logs, so long as each event notification references the notifications for the last event in all the aggregates that were required to trigger the new events. If a notification references a notification in another log, then the processing can wait until that other notification has been processed. Hence notifications do not need to include notifications in the same log, as they will be processed first. On the other hand, if all notifications include such references to other notifications, then a notification log could be processed in parallel: since it is unlikely that each notification in a log depends on its immediate predecessor, wherever a sub-sequence of notifications all depend on notifications that have already been processed, those notifications could perhaps be processed concurrently.

There will be a trade-off between evenly distributing events across the various logs and minimising the number of causal ordering that go across logs. A simple and probably effective rule would be to place all the events of one aggregate in the same log. But it may also help to partition the aggregates of an application by e.g. user, and place the events of all aggregates created by a user in the same notification log, since they are perhaps most likely to be causally related. This mechanism would allow the number of logs to be increased and decreased, with aggregate event notifications switching from one log to another and still be processed coherently.

## 10.4.3 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.

Todo: Fix problem with not being able to write all of big array with one SQL expression, since it involves constructing the non-leaf records. Perhaps could be more precise about predicting which non-leaf records need to be inserted so that we don't walk down from the top each time discovering whether or not a record exists. It's totally predictable, but the code is cautious. But it would be possible to identify all the new records and add them. Still not really possible to use "insert select max", but if each log has it's own process, then IDs can be issued from a generator, initialised from a query, and reused if an insert fails so the sequence is contiguous.

## 10.5 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.

### 10.5.1 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 event sourcing.interface.notificationlog import NotificationLogView
from event sourcing.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 item['topic'] == 'event sourcing.domain.model.entity#VersionedEntity.Created'

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



## 10.5.2 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.

## 10.5.3 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.

## 10.6 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_list()
assert len(all_notifications) == 9
```

(continues on next page)

(continued from previous page)

```
# 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_list()
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_list()
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_list()
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_list()
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.

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

A projection is a function of application state. If the state of an application is event sourced, projections can operate by processing the events of an application. There are lots of different kinds of projection.

- *Overview*
- *Subscribing to events*
- *Tracking notification logs*
  - *Application state replication*
  - *Index of email addresses*

### 11.1 Overview

Projected state can be updated by direct event subscription, or by following notification logs that propagate the events of an application. Notification logs are described in the previous section.

Notifications in a notification log can be pulled. Pulling notifications can be prompted periodically, prompts could also be pushed onto a message bus. Notifications could be pushed onto the message bus, but if order is lost, the projection will need to refer to the notification log.

Projections can track the notifications that have been processed: the position in an upstream notification log can be recorded. The position can be recorded as a value in the projected application state, if the nature of the projection lends itself to be used also to track notifications. Alternatively, the position in the notification log can be recorded with a separate tracking record.

Projections can update more or less anything. To be reliable, the projection must write in the same atomic database transaction all the records that result from processing a notification. Otherwise it is possible to track the position and fail to update the projection, or vice versa.

Since projections can update more or less anything, they can also call command methods on event sourced aggregates. The important thing is for any new domain events that are triggered by the aggregates to be recorded in the

same database transaction as the tracking records. For reliability, if these new domain events are also placed in a notification log, then the tracking record and the domain event records and the notification records can be written in the same database transaction. A projection that operates by calling methods on aggregates and recording events with notifications, can be called an “application process”.

## 11.2 Subscribing to events

The library function `subscribe_to()` can be used to decorate functions, so that the function is called each time a matching event is published by the library’s pub-sub mechanism.

The example below, which is incomplete because the `TodoView` is not defined, suggests that record `TodoView` can be created whenever a `Todo.Created` event is published. Perhaps there’s a `TodoView` table with an index of todo titles, or perhaps it can be joined with a table of users.

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

class Todo(DomainEntity):
    class Created(DomainEntity.Created):
        pass

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

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. Transactions can be used to insert related records, building a model for a view.

It is possible to use the decorator in a downstream application which republishes domain events perhaps published by an original application that uses messaging infrastructure such as an AMQP system. The decorated function would be called synchronously with the republishing of the event, but asynchronously with respect to the original application.

A naive projection might consume events as they are published and update the projection without considering whether the event was a duplicate, or if previous events were missed. The trouble with this approach is that, without further modification, without referring to a fixed sequence and maintaining position in that sequence, there is no way to recover when events have been missed. If the projection somehow fails to update when an event is received, then the event will be lost forever to the projection, and the projection will be forever inconsistent.

Of course, it is possible to follow a fixed sequence of events, for example by tracking notification logs.

## 11.3 Tracking notification logs

If the events of an application are presented as a sequence of notifications, then the events can be projected using a notification reader to pull unseen items.

Getting new items can be triggered by pushing prompts to e.g. an AMQP messaging system, and having the remote components handle the prompts by pulling the new notifications.

To minimise load on the messaging infrastructure, it may be sufficient simply to send an empty message, thereby prompting receivers to pull new notifications. This may reduce latency or avoid excessive polling for updates, but the benefit would be obtained at the cost of additional complexity. Prompts that include the position of the notification in

its sequence would allow a follower to know when it is being prompted about notifications it already pulled, and could then skip the pulling operation.

If an application has partitioned notification logs, they could be consumed concurrently.

If a projection creates a sequence that it appends to at least once for each notification, the position in the notification log can be tracked as part of the projection's sequence. Tracking the position is important when resuming to process the notifications. An example of using the projection's sequence to track the notifications is replication (see example below). However, with this technique, something must be written to the projection's sequence for each notification received. That can be tolerated by writing "null" records that extend the sequence without spoiling the projections, for example in the event sourced index example below, random keys are inserted instead of email addresses to extend the sequence.

An alternative which avoids writing unnecessarily to a projection's sequence is to separate the concerns, and write a tracking record for each notification that is consumed, and then optionally any records created for the projection in response to the notification.

A tracking record can simply have the position of a notification in a log. If the notifications are interpreted as commands, then a command log could function effectively to track the notifications, so long as one command is written for each notification (which might then involve "null" commands). For reliability, the tracking records need to be written in the same atomic database transaction as the projection records.

The library's `Process` class uses tracking records.

### 11.3.1 Application state replication

Using event record notifications, the state of an application can be replicated perfectly. If an application can present its event records as a notification log, then a "replicator" can read the notification log and write copies of the original records into a replica's record manager.

In the example below, the `SimpleApplication` class is used, which has a `RecordManagerNotificationLog` as its `notification_log`. Reading this log, locally or remotely, will yield all the event records persisted by the `SimpleApplication`. The `SimpleApplication` uses a record manager with contiguous record IDs which allows it to be used within a record manager notification log object.

A record manager notification log object represents records as record notifications. With record notifications, the ID of the record in the notification is used to place the notification in its sequence. Therefore the ID of the last replicated record is used to determine the current position in the original application's notification log, which gives "exactly once" processing.

```
from event sourcing.application.sqlalchemy import SQLAlchemyApplication
from event sourcing.exceptions import ConcurrencyError
from event sourcing.domain.model.aggregate import AggregateRoot
from event sourcing.interface.notificationlog import NotificationLogReader, \
    RecordManagerNotificationLog

# Define record replicator.
class RecordReplicator(object):
    def __init__(self, notification_log, record_manager):
        self.reader = NotificationLogReader(notification_log)
        self.manager = record_manager
        # Position reader at max record ID.
        self.reader.seek(self.manager.get_max_record_id() or 0)

    def pull(self):
        for notification in self.reader.read():
```

(continues on next page)

(continued from previous page)

```

        notification_id = notification.pop('id')
        record = self.manager.record_class(
            notification_id=notification_id,
            **notification
        )
        self.manager.write_records([record])

# Construct original application.
original = SQLAlchemyApplication(persist_event_type=AggregateRoot.Event)

# Construct replica application.
replica = SQLAlchemyApplication()

# Construct replicator.
replicator = RecordReplicator(
    notification_log=original.notification_log,
    record_manager=replica.event_store.record_manager
)

# Publish some events.
aggregate1 = AggregateRoot.__create__()
aggregate1.__save__()
aggregate2 = AggregateRoot.__create__()
aggregate2.__save__()
aggregate3 = AggregateRoot.__create__()
aggregate3.__save__()

assert aggregate1.__created_on__ != aggregate2.__created_on__
assert aggregate2.__created_on__ != aggregate3.__created_on__

# Check aggregates not in replica.
assert aggregate1.id in original.repository
assert aggregate1.id not in replica.repository
assert aggregate2.id in original.repository
assert aggregate2.id not in replica.repository
assert aggregate3.id in original.repository
assert aggregate3.id not in replica.repository

# Pull records.
replicator.pull()

# Check aggregates are now in replica.
assert aggregate1.id in replica.repository
assert aggregate2.id in replica.repository
assert aggregate3.id in replica.repository

# Check the aggregate attributes are correct.
assert aggregate1.__created_on__ == replica.repository[aggregate1.id].__created_on__
assert aggregate2.__created_on__ == replica.repository[aggregate2.id].__created_on__
assert aggregate3.__created_on__ == replica.repository[aggregate3.id].__created_on__

# Create another aggregate.
aggregate4 = AggregateRoot.__create__()
aggregate4.__save__()

# Check aggregate exists in the original only.

```

(continues on next page)



(continued from previous page)

```

assert aggregate4.id in original.repository
assert aggregate4.id not in replica.repository

# Resume pulling records.
replicator.pull()

# Check aggregate exists in the replica.
assert aggregate4.id in replica.repository

# Terminate replicator (position in notification sequence is lost).
replicator = None

# Create new replicator.
replicator = RecordReplicator(
    notification_log=original.notification_log,
    record_manager=replica.event_store.record_manager
)

# Create another aggregate.
aggregate5 = AggregateRoot.__create__()
aggregate5.__save__()

# Check aggregate exists in the original only.
assert aggregate5.id in original.repository
assert aggregate5.id not in replica.repository

# Pull after replicator restart.
replicator.pull()

# Check aggregate exists in the replica.
assert aggregate5.id in replica.repository

# Setup event driven pulling. Could prompt remote
# readers with an AMQP system, but to make a simple
# demonstration just subscribe to local events.

@subscribe_to(AggregateRoot.Event)
def prompt_replicator(_):
    replicator.pull()

# Now, create another aggregate.
aggregate6 = AggregateRoot.__create__()
aggregate6.__save__()
assert aggregate6.id in original.repository

# Check aggregate was automatically replicated.
assert aggregate6.id in replica.repository

# Clean up.
original.close()
replica.close()

```

For simplicity in the example, the notification log reader uses a local notification log in the same process as the events originated. Perhaps it would be better to run a replication job away from the application servers, on a node remote from the application servers, away from where the domain events are triggered. A local notification log could be used on a worker-tier node that can connect to the original application's database. It could equally well use a remote notification log without compromising the accuracy of the replication. A remote notification log, with an API service provided by

the application servers, would avoid the original application database connections being shared by countless others. Notification log sections can be cached in the network to avoid loading the application servers with requests from a multitude of followers.

Since the replica application uses optimistic concurrency control for its event records, it isn't possible to corrupt the replica by attempting to write the same record twice. Hence jobs can pull at periodic intervals, and at the same time message queue workers can respond to prompts pushed to AMQP-style messaging infrastructure by the original application, without needing to serialise their access to the replica with locks: if the two jobs happen to collide, one will succeed and the other will encounter a concurrency error exception that can be ignored.

The replica could itself be followed, by using its notification log. Although replicating replicas indefinitely is perhaps pointless, it suggests how notification logs can be potentially be chained with processing being done at each stage.

For example, a sequence of events could be converted into a sequence of commands, and the sequence of commands could be used to update an event sourced index, in an index application. An event that does not affect the projection can be recorded as “noop”, so that the position is maintained. All but the last noop could be deleted from the command log. If the command is committed in the same transaction as the events resulting from the command, then the reliability of the arbitrary projection will be as good as the pure replica. The events resulting from each commands could be many or none, which shows that a sequence of events can be projected equally reliably into a different sequence with a different length.

### 11.3.2 Index of email addresses

This example is similar to the replication example above, in that notifications are tracked with the records of the projected state. In consequence, an index entry is added for each notification received, which means progress can be made along the notification log even when the notification doesn't imply a real entry in the index.

```
import uuid

# Define domain model.
class User(AggregateRoot):
    def __init__(self, *arg, **kwargs):
        super(User, self).__init__(*arg, **kwargs)
        self.email_addresses = {}

    class Event(AggregateRoot.Event):
        pass

    class Created(Event, AggregateRoot.Created):
        pass

    def add_email_address(self, email_address):
        self.__trigger_event__(User.EmailAddressAdded, email_address=email_address)

    class EmailAddressAdded(Event):
        def mutate(self, aggregate):
            email_address = User.EmailAddress(self.email_address)
            aggregate.email_addresses[self.email_address] = email_address

    def verify_email_address(self, email_address):
        self.__trigger_event__(User.EmailAddressVerified, email_address=email_address)

    class EmailAddressVerified(Event):
        def mutate(self, aggregate):
            aggregate.email_addresses[self.email_address].is_verified = True
```

(continues on next page)

(continued from previous page)

```

class EmailAddress(object):
    def __init__(self, email_address):
        self.email_address = email_address
        self.is_verified = False

class IndexItem(AggregateRoot):
    def __init__(self, index_value=None, *args, **kwargs):
        super(IndexItem, self).__init__(*args, **kwargs)
        self.index_value = index_value

class Event(AggregateRoot.Event):
    pass

class Created(Event, AggregateRoot.Created):
    pass

class Updated(Event):
    def mutate(self, aggregate):
        aggregate.index_value = self.index_value

    def update_index(self, index_value):
        self.__trigger_event__(IndexItem.Updated, index_value=index_value)

def uuid_from_url(url):
    return uuid.uuid5(uuid.NAMESPACE_URL, url.encode('utf8') if bytes == str else url)

# Define indexer.
class Indexer(object):
    def __init__(self, notification_log, record_manager, repository):
        self.reader = NotificationLogReader(notification_log)
        self.repository = repository
        self.manager = record_manager
        # Position reader at max record ID.
        # - this can be generalised to get the max ID from many
        #   e.g. big arrays so that many notification logs can
        #   be followed, consuming a group of notification logs
        #   would benefit from using transactions to set records
        #   in a big array per notification log atomically with
        #   inserting the result of combining the notification log
        #   because processing more than one stream would produce
        #   a stream that has a different sequence of record IDs
        #   which couldn't be used directly to position any of the
        #   notification log readers
        # - if producing one stream from many can be as reliable as
        #   replicating a stream, then the unreliability will be
        #   caused by interoperating with systems that just do push,
        #   but the push notifications could be handled by adding
        #   to an application partition sequence, so e.g. all bank
        #   payment responses wouldn't need to go in the same sequence
        #   and therefore be replicated with mostly noops in all application
        #   partitions, or perhaps they could initially go in the same
        #   sequence, and transactions could be used to project that into
        #   many different sequences, in other words splitting the stream
        #   (splitting is different from replicating many times). When splitting
        #   the stream, the splits's record ID couldn't be used to position to_
        ↪ splitter

```

(continues on next page)

(continued from previous page)

```

        # in the consumed notification log, so there would need to be a command
        # log that tracks the consumed sequence whose record IDs can be used to_
    ↪position
        # the splitter in the notification log, with the commands
        # defining how the splits are extended, and everything committed in a_
    ↪transaction
        # so the splits are atomic with the command log
        # Todo: Bring out different projectors: splitter (one-many), combiner (many-
    ↪one), repeater (one-one).
        self.reader.seek(self.manager.get_max_record_id() or 0)

    def pull(self):
        # Project events into commands for the index.
        for notification in self.reader.read():

            # Construct index items.
            # Todo: Be more careful, write record with an ID explicitly,
            # (walk the event down the stack explicitly, and then set the ID)
            # so concurrent processing is safe. Providing the ID also avoids
            # the cost of computing the next record ID.
            # Alternatively, construct, execute, then record index commands in a big_
    ↪array.
            # Could record commands in same transaction as result of commands if_
    ↪commands are not idempotent.
            # Could use compaction to remove all blank items, but never remove the_
    ↪last record.
            if notification['topic'].endswith('User.EmailAddressVerified'):
                event = original.event_store.mapper.event_from_topic_and_state(
                    notification['topic'],
                    notification['state'],
                )
                index_key = uuid_from_url(event.email_address)
                index_value = event.originator_id
                try:
                    index_item = self.repository[index_key]
                    index_item.update_index(index_value)
                except KeyError:
                    index_item = IndexItem.__create__(originator_id=index_key, index_
    ↪value=index_value)
                    index_item.__save__()

        # Construct original application.
        original = SQLAlchemyApplication(persist_event_type=User.Event)

        # Construct index application.
        index = SQLAlchemyApplication(name='indexer', persist_event_type=IndexItem.Event)

        # Setup event driven indexing.
        indexer = Indexer(
            notification_log=original.notification_log,
            record_manager=index.event_store.record_manager,
            repository=index.repository,
        )

    @subscribe_to(User.Event)
    def prompt_indexer(_):

```

(continues on next page)

(continued from previous page)

```

    indexer.pull()

    user1 = User.__create__()
    user1.__save__()
    assert user1.id in original.repository
    assert user1.id not in index.repository

    user1.add_email_address('me@example.com')
    user1.__save__()

    assert not user1.email_addresses['me@example.com'].is_verified

    index_key = uuid_from_url('me@example.com')
    assert index_key not in index.repository

    user1.verify_email_address('me@example.com')
    user1.__save__()

    assert user1.email_addresses['me@example.com'].is_verified

    assert index_key in index.repository
    assert index.repository[index_key].index_value == user1.id

    user1.add_email_address('me@example.com')
    user1.verify_email_address('me@example.com')
    user1.__save__()
    assert index_key in index.repository
    assert index.repository[index_key].index_value == user1.id

    assert uuid_from_url(u'mycat@example.com') not in index.repository

    user1.add_email_address(u'mycat@example.com')
    user1.verify_email_address(u'mycat@example.com')
    user1.__save__()

    assert uuid_from_url(u'mycat@example.com') in index.repository

    assert user1.id in original.repository
    assert user1.id not in index.repository

```

Todo: Projection into a timeline view?

Todo: Projection into snapshots (policy determines when to snapshot)?

Todo: Projection for data analytics?

Todo: Concurrent processing of notification logs, respecting causal relations.

Todo: Order reservation payments, system with many processes and many notification logs.

Todo: Single process with single log.

Todo: Single process with many logs.

Todo: Many processes with one log each.

Todo: Many processes with many logs each (static config).

Todo: Many processes with many logs each (dynamic config).

Todo: Single process with state machine semantics (whatever they are)?

---

## Distributed systems

---

This section discusses how to make a reliable distributed system that is scalable and maintainable.

- *Overview*
- *Process application*
  - *Reliability*
  - *Notification tracking*
  - *Policies*
  - *Atomicity*
- *System of processes*
  - *Infrastructure-independence*
  - *System runners*
  - *Maintainability*
  - *Scalability*
  - *Causal dependencies*
  - *Kahn process networks*
  - *Process managers*
- *Example*
  - *Aggregates*
  - *Commands*
  - *Processes*
  - *System*

– *Runners*

- \* *Single threaded runner*
- \* *Multiprocess runner*
- \* *Single pipeline*
- \* *Multiple pipelines*
- \* *Actor model runner*
- \* *Pool of workers*
- *Integration with APIs*
- *Process event pattern*

## 12.1 Overview

A design for distributed systems is introduced that uses event-sourced applications as building blocks. The earlier design of the *event-sourced application* is extended in the design of the “process application”. Process application classes can be composed into a set of pipeline expressions that together define a system pipeline. This definition of a system can be entirely independent of infrastructure. Such a system can be run in different ways with identical results.

Rather than seeking reliability in mathematics (e.g. *CSP*) or in physics (e.g. *Actor model*), the approach taken here is to seek reliable foundations in engineering empiricism, specifically in the empirical reliability of *counting* and of *ACID database transactions*.

Just as event sourcing “atomised” application state as a set of domain events, similarly the processing of domain events can be atomised as a potentially distributed set of local “process events” in which new domain events may occur. The subjective aim of a process event is “catching up on what happened”.

The processing of domain events is designed to be atomic and successive so that processing can progress in a determinate manner according to infrastructure availability. A provisional description of a design pattern for process events is included at the end of this section.

## 12.2 Process application

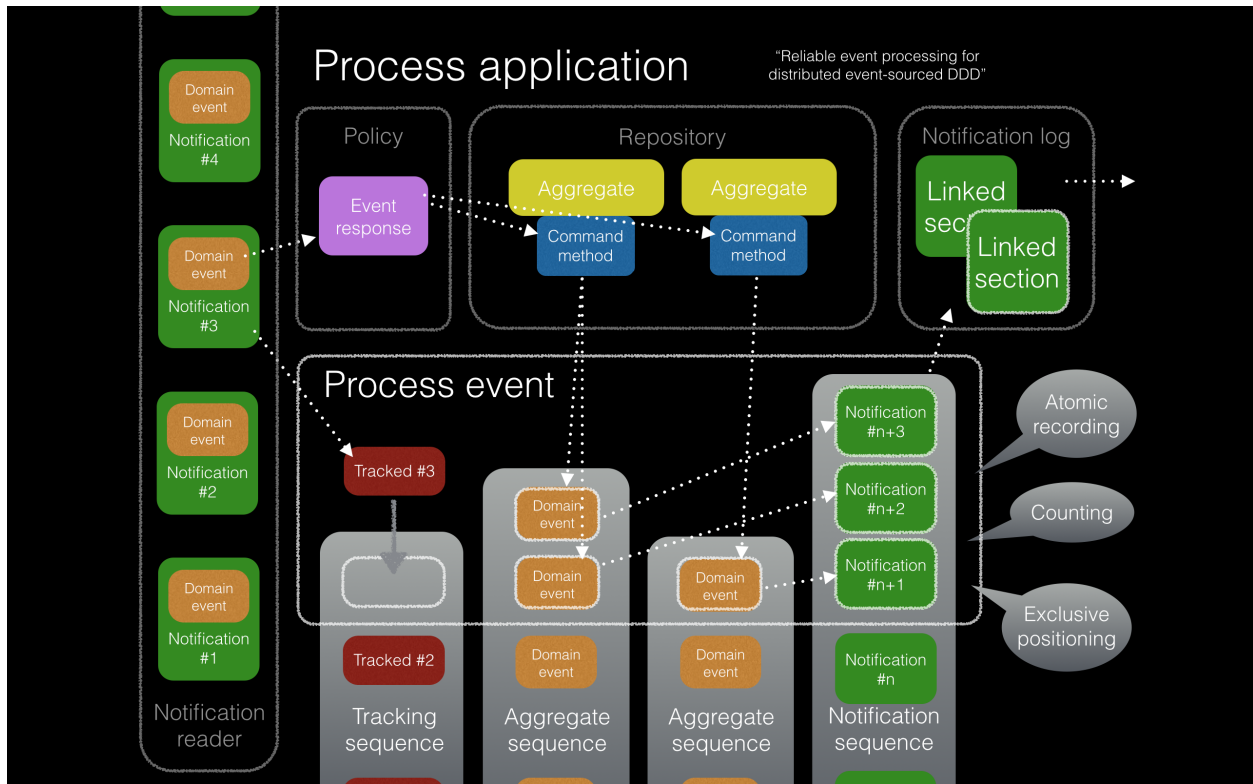
A process application is an event-sourced *projection* into an event-sourced *application*. One process application may “follow” another. A process application projects the state of the applications it follows into its own state. Being an event sourced application, the state of a process application can be obtained using a *notification log reader* to obtain the state as a sequence of domain events. The projection itself is defined by the application’s policy, which defines responses to domain events in terms of domain model operations, causing new domain events to be generated.

The library has a process application class *ProcessApplication*, it functions as a projection into an event-sourced application. It extends *SimpleApplication* by having a notification log reader for each application it follows. It has an application policy which defines how to respond to the domain events it reads from the notification logs of the applications it follows. This process application class implements the process event pattern (notification tracking and new domain event data are stored together atomically).

### 12.2.1 Reliability

Reliability is the most important concern in this section. A process is considered to be reliable if its result is entirely unaffected (except in being delayed) by infrastructure failure such as network partitions or sudden termination of





operating system processes. Infrastructure unreliability may cause processing delays, but disorderly environments shouldn't (at least by default) cause disorderly processing results.

The only trick was remembering that production is determined in general by consumption with recording. In particular, if consumption and recording are reliable, then production is bound to be reliable. As shown below, the reliability of this library's approach to event processing depends only on counting and the atomicity of database transactions, both of which are normally considered reliable.

### 12.2.2 Notification tracking

A process application consumes domain events by *reading event notifications* from its notification log readers. The domain events are retrieved in a reliable order, without race conditions or duplicates or missing items. Each event notification in a notification log has a unique integer ID, and the notification log IDs form a contiguous sequence (counting).

To keep track of its position in the notification log, a process application will create a unique tracking record for each event notification it processes. The tracking records determine how far the process has progressed through the notification log. The tracking records are used to set the position of the notification log reader when the process application is commenced or resumed.

There can only be one tracking record for each event notification. Once the tracking record has been written it can't be written again, in which case neither will any new domain events. Hence, if a domain event notification can be processed at all, then it will be processed exactly once.

### 12.2.3 Policies

A process application will respond to domain events according to its policy. Its policy might do nothing in response to one type of event, and it might call an aggregate command method in response to another type of event. If the

aggregate method generates new domain events, they will be available in its notification log for others to read, just like a normal event-sourced application.

Whatever the policy response, the process application will write one tracking record for each event notification, along with new stored event and notification records, in an atomic database transaction.

### 12.2.4 Atomicity

Just like a ratchet is as strong as its teeth and pawl, a process application is as reliable as the atomicity of its database transactions. If some of the new records from processing a domain event can't be written, then none will be committed. If anything goes wrong before all the records are written, the transaction will abort, and none of the records will be committed. On the other hand, if a some records are committed, then all will be committed, and the process will complete an atomic progression.

The atomicity of the recording and consumption determines the production as atomic: a continuous stream of events is processed in discrete, sequenced, indivisible units. Hence, interruptions can only cause delays.

Whilst the heart of this design is having the event processing proceed atomically so that any completed “process events” are exactly what they should be, of course the “CID” parts of ACID database transactions are also crucial. Especially, it is assumed that any records that have been committed will be available after any so-called “infrastructure failure”. The continuing existence of data that has been successfully committed to a database is beyond the scope of this discussion about reliability. However, the “single point of failure” this may represent is acknowledged.

## 12.3 System of processes

The library class `System` can be used to define a system of process applications, entirely independently of infrastructure. In a system, one process application can follow another. One process can follow two other processes in a slightly more complicated system. A system could be just one process application following itself.

The reliability of the domain event processing allows a reliable “saga” or a “process manager” to be written without restricting or cluttering the application logic with precaution and remediation for infrastructure failures.

### 12.3.1 Infrastructure-independence

A system of process applications can be defined independently of infrastructure so that the same system can be run with different infrastructure at different times. For example, a system of process applications could be developed for use with SQLAlchemy, and later reused in a Django project.

### 12.3.2 System runners

A system of process applications can run in a single thread, with synchronous propagation and processing of events through the system pipeline. A system can also be run with multiple threads or multiple operating system processes, with application state propagated asynchronously in various ways.

An asynchronous pipeline with multi-threading or multi-processing means one event can be processed by each process application at the same time. This is very much like [instruction pipelining](#) in a CPU core.

### 12.3.3 Maintainability

Whilst maintainability is greatly assisted by having an entire system of applications defined independently of infrastructure, it also greatly helps to run such a system synchronously with a single thread. So long as the behaviours

are preserved, running the system without any concurrent threads or processes makes it much easier to develop and maintain the system.

### 12.3.4 Scalability

Especially when using multiple operating system processes, throughput can be increased by breaking longer steps into smaller steps, up but only to a limit provided by the number of steps actually required by the domain. Such “diachronic” parallelism therefore provides limited opportunities for scaling throughput.

A system of process applications can also be run with many parallel instances of the system pipeline. This is very much like the way a multi-core CPU has many cores (a core is a pipeline). This “synchronic” parallelism means that many domain events can be processed by the same process application at the same time. This kind of parallelism allows the system to be scaled, but only to a limit provided by the degree of parallelism inherent in the domain (greatest when there are no causal dependencies between domain events, least when there are maximal causal dependencies between domain events).

### 12.3.5 Causal dependencies

Causal dependencies are needed to synchronise between parallel processing of a sequence of events. This is used in the library when a system is run with multiple pipelines.

Causal dependencies between events can be automatically detected and used to synchronise the processing of parallel pipelines downstream. For example, if an aggregate is created and then updated, the second event is obviously causally dependent on the first (you can’t update something that doesn’t exist). Downstream processing in one pipeline can wait (stall) for a dependency to be processed in another pipeline. This is like a pipeline interlock in a multi-core CPU.

In the process applications, the causal dependencies are automatically inferred by detecting the originator ID and version of aggregates as they are retrieved from the repository. The old notifications are referenced in the first new notification. Downstream can then check all causal dependencies have been processed, using its tracking records.

In case there are many dependencies in the same pipeline, only the newest dependency in each pipeline is included. By default in the library, only dependencies in different pipelines are included. If causal dependencies from all pipelines were included in each notification, each pipeline could be processed in parallel, to an extent limited by the dependencies between the notifications.

### 12.3.6 Kahn process networks

Because a notification log and reader functions effectively as a FIFO, a system of determinate process applications can be recognised as a [Kahn Process Network](#) (KPN).

Kahn Process Networks are determinate systems. If a system of process applications happens to involve processes that are not determinate, or if the processes split and combine or feedback in a random way so that nondeterminacy is introduced by design, the system as a whole will not be determinate, and could be described in more general terms as “dataflow” or “stream processing”.

Whether or not a system of process applications is determinate, the processing will be reliable (results unaffected by infrastructure failures).

High performance or “real time” processing could be obtained by avoiding writing to a durable database and instead running applications with an in-memory database.

## 12.3.7 Process managers

A process application, specifically an aggregate combined with a policy in a process application, could function effectively as a “saga”, or “process manager”, or “workflow manager”. That is, it could effectively control a sequence of steps involving other aggregates in other bounded contexts, steps that might otherwise be controlled with a “long-lived transaction”. It could ‘maintain the state of the sequence and determine the next processing step based on intermediate results’, to quote a phrase from Enterprise Integration Patterns. Exceptional “unhappy path” behaviour can be implemented as part of the logic of the application.

## 12.4 Example

The example below is suggestive of an orders-reservations-payments system. The system automatically processes a new Order by making a Reservation, and then a Payment; facts registered with the Order as they happen.

The system is run: firstly as a single threaded system; then with multiprocessing using a single pipeline; multiprocessing with multiple pipelines; and finally multiple pipelines with the actor model.

The behaviour of the system is entirely defined by the combination of the aggregates and the policies of its process applications. This allows highly maintainable code, code that is easily tested, easily understood, easily changed, easily reconfigured for use with different infrastructure.

Please note, the code presented in the example below works only with the library’s SQLAlchemy and Django infrastructure code. Support for Cassandra is being considered but such applications will probably be simple replications of application state, due to the limited atomicity of Cassandra’s lightweight transactions. For example, Cassandra could be used to archive events written firstly into a relational database. Events could be removed from the relational database before storage limits are encountered. Events missing in the relational database could be sourced from Cassandra.

### 12.4.1 Aggregates

In the domain model below, event-sourced aggregates are defined for orders, reservations, and payments.

An Order can be created. An existing order can be set as reserved, which involves a reservation ID. Having been created and reserved, an order can be set as paid, which involves a payment ID.

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

class Order(AggregateRoot):

    class Event(AggregateRoot.Event):
        pass

    @classmethod
    def create(cls, command_id):
        return cls.__create__(command_id=command_id)

    class Created(Event, AggregateRoot.Created):
        pass

    def __init__(self, command_id=None, **kwargs):
        super(Order, self).__init__(**kwargs)
        self.command_id = command_id
        self.reservation_id = None
        self.payment_id = None
```

(continues on next page)

(continued from previous page)

```
@property
def is_reserved(self):
    return self.reservation_id is not None

def set_is_reserved(self, reservation_id):
    assert not self.is_reserved, "Order {} already reserved.".format(self.id)
    self.__trigger_event__(
        Order.Reserved, reservation_id=reservation_id
    )

class Reserved(Event):
    def mutate(self, order: "Order"):
        order.reservation_id = self.reservation_id

@property
def is_paid(self):
    return self.payment_id is not None

def set_is_paid(self, payment_id):
    assert not self.is_paid, "Order {} already paid.".format(self.id)
    self.__trigger_event__(
        self.Paid, payment_id=payment_id, command_id=self.command_id
    )

class Paid(Event):
    def mutate(self, order: "Order"):
        order.payment_id = self.payment_id
```

A Reservation can be created. A reservation has an order\_id.

```
class Reservation(AggregateRoot):

    class Event(AggregateRoot.Event):
        pass

    @classmethod
    def create(cls, order_id):
        return cls.__create__(order_id=order_id)

    class Created(Event, AggregateRoot.Created):
        pass

    def __init__(self, order_id, **kwargs):
        super(Reservation, self).__init__(**kwargs)
        self.order_id = order_id
```

Similarly, a Payment can be created. A payment also has an order\_id.

```
class Payment(AggregateRoot):

    class Event(AggregateRoot.Event):
        pass

    @classmethod
    def create(cls, order_id):
        return cls.__create__(order_id=order_id)
```

(continues on next page)

(continued from previous page)

```
class Created(Event, AggregateRoot.Created):
    pass

def __init__(self, order_id, **kwargs):
    super(Payment, self).__init__(**kwargs)
    self.order_id = order_id
```

All the domain event classes are defined explicitly on the aggregate root classes. This is important because the application policies will use the domain event classes to decide how to respond to the events, and if the aggregate classes use the event classes from the base aggregate root class, then one aggregate’s Created event can’t be distinguished from another’s, and the application policy won’t work as expected.

The behaviours of this domain model can be fully tested with simple test cases, without involving any other components.

## 12.4.2 Commands

Commands have been discussed previously as *methods on domain entities*. Here, system commands are introduced, as event sourced aggregates created within a separate “commands application”.

One advantage of having distinct command aggregates is that old commands can be used to check the same application state is generated by a new version of the system.

Another advantage of using a separate commands application is that commands can be introduced into an event processing system without interrupting the event processing of the core process applications. (Treating a process application as a normal application certainly works, but can potentially cause contention writing to the notification log.)

Responses can be collected by creating separate “command response” aggregates in a separate “responses” process application. An alternative approach involves updating the command aggregate, and having the commands application follow a core process application.

In the example below, the command class CreateOrder is defined using the library’s command class, *Command*, which extends the library’s *AggregateRoot* class with a method *done()* and a property *is\_done*.

The CreateOrder class extends the library’s *Command*. A CreateOrder command can be assigned an order ID. Its *order\_id* is initially None.

```
from event sourcing.domain.model.command import Command
from event sourcing.domain.model.decorators import attribute

class CreateOrder(Command):

    class Event(Command.Event):
        pass

    @classmethod
    def create(cls):
        return cls.__create__()

    class Created(Event, Command.Created):
        pass

    @attribute
    def order_id(self):
```

(continues on next page)

(continued from previous page)

```
pass

class AttributeChanged(Event, Command.AttributeChanged):
    pass
```

The `order_id` will eventually be used to keep the ID of an Order aggregate created by the system in response to a `CreateOrder` command being created.

The behaviour of a system command aggregate can be fully tested with simple test cases, without involving any other components.

```
from uuid import uuid4

def test_create_order_command():

    # Create a "create order" command.
    cmd = CreateOrder.create()

    # Check the initial values.
    assert cmd.order_id is None
    assert cmd.is_done is False

    # Assign an order ID.
    order_id = uuid4()
    cmd.order_id = order_id
    assert cmd.order_id == order_id

    # Mark the command as "done".
    cmd.done()
    assert cmd.is_done is True

    # Check the events.
    events = cmd.__batch_pending_events__()
    assert len(events) == 3
    assert isinstance(events[0], CreateOrder.Created)
    assert isinstance(events[1], CreateOrder.AttributeChanged)
    assert isinstance(events[2], CreateOrder.Done)

    # Run the test.
    test_create_order_command()
```

### 12.4.3 Processes

A process application has a policy which defines how events are processed. In the code below, process applications are defined for orders, reservations, and payments.

The Orders process application policy responds to new commands by creating a new Order aggregate. It responds to new reservations by setting an Order as reserved. And it responds to a new Payment, by setting an Order as paid.

```
from eventsourcing.application.process import ProcessApplication
from eventsourcing.application.decorators import applicationpolicy
```

(continues on next page)

(continued from previous page)

```
class Orders(ProcessApplication):

    @applicationpolicy
    def policy(self, repository, event):
        pass

    @policy.register(CreateOrder.Created)
    def _(self, repository, event):
        return self.create_order(command_id=event.originator_id)

    @policy.register(Reservation.Created)
    def _(self, repository, event):
        self._set_order_is_reserved(repository, event)

    @policy.register(Payment.Created)
    def _(self, repository, event):
        self._set_order_is_paid(repository, event)

    @staticmethod
    def create_order(command_id):
        return Order.create(command_id=command_id)

    def _set_order_is_reserved(self, repository, event):
        order = repository[event.order_id]
        assert not order.is_reserved
        order.set_is_reserved(event.originator_id)

    def _set_order_is_paid(self, repository, event):
        order = repository[event.order_id]
        assert not order.is_paid
        order.set_is_paid(event.originator_id)
```

The decorator `@applicationpolicy` is similar to `@singledispatch` from the `functools` core Python package. It isn't magic, it's just a slightly better alternative to an "if-instance-elif-isinstance-..." block.

The Reservations process application responds to an `Order.Created` event by creating a new Reservation aggregate.

```
class Reservations(ProcessApplication):

    @applicationpolicy
    def policy(self, repository, event):
        pass

    @policy.register(Order.Created)
    def _(self, repository, event):
        return self.create_reservation(event.originator_id)

    @staticmethod
    def create_reservation(order_id):
        return Reservation.create(order_id=order_id)
```

The payments process application responds to an `Order.Reserved` event by creating a new Payment.

```
class Payments(ProcessApplication):
```

(continues on next page)



(continued from previous page)

```
@applicationpolicy
def policy(self, repository, event):
    pass

@policy.register(Order.Reserved)
def _(self, repository, event):
    order_id = event.originator_id
    return self.create_payment(order_id)

@staticmethod
def create_payment(order_id):
    return Payment.create(order_id=order_id)
```

Additionally, the library class `CommandProcess` is extended by defining a policy that responds to `Order.Created` events by setting the `order_id` on the command, and to `Order.Paid` events by setting the command as done. It also has a factory method `create_order()` which can be used to create new `Order` aggregates.

```
from event sourcing.application.command import CommandProcess
from event sourcing.domain.model.decorators import retry
from event sourcing.exceptions import OperationalError, RecordConflictError

class Commands(CommandProcess):

    persist_event_type = CreateOrder.Event

    @applicationpolicy
    def policy(self, repository, event):
        pass

    @policy.register(Order.Created)
    def _(self, repository, event):
        cmd = repository[event.command_id]
        cmd.order_id = event.originator_id

    @policy.register(Order.Paid)
    def _(self, repository, event):
        cmd = repository[event.command_id]
        cmd.done()

    @staticmethod
    @retry((OperationalError, RecordConflictError), max_attempts=10, wait=0.01)
    def create_order():
        cmd = CreateOrder.create()
        cmd.__save__()
        return cmd.id
```

The `@retry` decorator overcomes contention when creating new commands whilst also processing domain events from the `Orders` application.

Please note, the `__save__()` method of aggregates must not be called in a process policy, because pending events from both new and changed aggregates will be automatically collected by the process application after its `policy()` method has returned. To be reliable, a process application needs to commit all the event records atomically with a tracking record, and calling `__save__()` will instead commit events in a separate transaction. Policies must return new aggregates to the caller, but do not need to return existing aggregates that have been accessed or changed.

Process policies are just functions, and are easy to test.

In the orders policy test below, an existing order is marked as reserved because a reservation was created. The only complication comes from needing to prepare at least a fake repository and a domain event, given as required arguments when calling the policy in the test. If the policy response depends on already existing aggregates, they will need to be added to the fake repository. A Python dict can function effectively as a fake repository in such tests. It seems simplest to directly use the model domain event classes and aggregate classes in these tests, rather than coding [test doubles](#).

```
def test_orders_policy():

    # Prepare repository with a real Order aggregate.
    order = Order.create(command_id=None)
    repository = {order.id: order}

    # Check order is not reserved.
    assert not order.is_reserved

    # Check order is reserved whenever a reservation is created.
    event = Reservation.Created(originator_id=uuid4(), originator_topic='', order_
→ id=order.id)
    Orders().policy(repository, event)
    assert order.is_reserved

# Run the test.
test_orders_policy()
```

In the payments policy test below, a new payment is created because an order was reserved.

```
def test_payments_policy():

    # Prepare repository with a real Order aggregate.
    order = Order.create(command_id=None)
    repository = {order.id: order}

    # Check payment is created whenever order is reserved.
    event = Order.Reserved(originator_id=order.id, originator_version=1)
    payment = Payments().policy(repository, event)
    assert isinstance(payment, Payment), payment
    assert payment.order_id == order.id

# Run the test.
test_payments_policy()
```

It isn't necessary to return changed aggregates from the policy. The test will already have a reference to the aggregate, since it will have constructed the aggregate before passing it to the policy in the fake repository, so the test will already be in a good position to check that already existing aggregates are changed by the policy as expected. The test gives a repository to the policy, which contains the order aggregate expected by the policy.

## 12.4.4 System

A system of process applications can be defined using one or many pipeline expressions.

The expression `A | A` would have a process application class called `A` following itself. The expression `A | B | C` would have `A` followed by `B` and `B` followed by `C`. This can perhaps be recognised as the “pipes and filters” pattern, where the process applications function effectively as the filters.

In this example, firstly the `Orders` process will follow the `Commands` process so that orders can be created. The

Commands process will follow the Orders process, so that commands can be marked as done when processing is complete.

```
commands_pipeline = Commands | Orders | Commands
```

Similarly, the Orders process and the Reservations process will follow each other. Also the Orders and the Payments process will follow each other.

```
reservations_pipeline = Orders | Reservations | Orders
payments_pipeline = Orders | Payments | Orders
```

The orders-reservations-payments system can be defined using these pipeline expressions.

```
from eventsourcing.application.system import System

system = System(commands_pipeline, reservations_pipeline, payments_pipeline)
```

This is equivalent to a system defined with the following single pipeline expression.

```
pipeline = Commands | Orders | Reservations | Orders | Payments | Orders | Commands

system = System(pipeline)
```

Although a process application class can appear many times in the pipeline expressions, there will only be one instance of each process when the pipeline system is instantiated. Each application can follow one or many applications, and can be followed by one or many applications.

In this system, application state is propagated between process applications through notification logs only. This can perhaps be recognised as the “bounded context” pattern. Each application can access only the aggregates it has created. For example, an Order aggregate created by the Orders process is available in neither the repository of Reservations nor the repository of Payments. If an application could directly use the aggregates of another application, then processing could produce different results at different times, and in consequence the processing wouldn’t be reliable. If necessary, a process application can replicate upstream aggregates within its own state.

## 12.4.5 Runners

The system above has been defined entirely independently of infrastructure. Concrete application infrastructure is introduced by the system runners. A concrete application infrastructure class can be specified when constructing a system runner with a suitable value of `infrastructure_class`. A system runner can be used as a context manager.

```
from eventsourcing.application.popo import PopoApplication
from eventsourcing.application.system import SingleThreadedRunner

with SingleThreadedRunner(system, infrastructure_class=PopoApplication):

    # Do stuff here...
    pass
```

### Single threaded runner

If the `system` object is used with the library class `SingleThreadedRunner`, the process applications will run in a single thread in the current process.

Events will be processed with synchronous handling of prompts, so that policies effectively call each other recursively, according to which applications each is followed by.

In the example below, the `system` object is used directly as a context manager. Using the `system` object in this manner implicitly constructs a *SingleThreadedRunner*. As a special case, by default this runner introduces concrete application infrastructure class *PopoApplication*, which literally uses plain old Python objects to store domain events in memory, and is the fastest concrete application infrastructure class in the library (much faster than in-memory SQLite). It can be used when proper disk-based durability is not required, for example during system development.

```
with system:

    # Create "create order" command.
    cmd_id = system.commands.create_order()

    # Check the command has an order ID and is done.
    cmd = system.commands.repository[cmd_id]
    assert cmd.order_id
    assert cmd.is_done

    # Check the order is reserved and paid.
    order = system.orders.repository[cmd.order_id]
    assert order.is_reserved
    assert order.is_paid

    # Check the reservation exists.
    reservation = system.reservations.repository[order.reservation_id]

    # Check the payment exists.
    payment = system.payments.repository[order.payment_id]
```

Everything happens synchronously, in a single thread, so that by the time `create_order()` has returned, the command has been fully processed by the system.

Running the system with a single thread and an in-memory database is useful when developing and testing a system of process applications, because it runs very quickly and the behaviour is very easy to follow.

## Multiprocess runner

The example below shows the same system of process applications running in different operating system processes, using the library's *MultiprocessRunner* class (which uses Python's multiprocessing library).

Running the system with multiple operating system processes means the different processes are running concurrently, so that as the payment is made for one order, another order might get reserved, whilst a third order is at the same time created.

The code below uses the library's *MultiprocessRunner* class to run the `system`. It will start one operating system process for each process application in the system, which in this example will give a pipeline with four child operating system processes. This example uses SQLAlchemy to access a MySQL database. The concrete infrastructure class is *SQLAlchemyApplication*.

```
from eventsourcing.application.multiprocess import MultiprocessRunner
from eventsourcing.application.sqlalchemy import SQLAlchemyApplication

runner = MultiprocessRunner(
    system=system,
    infrastructure_class=SQLAlchemyApplication,
```

(continues on next page)

(continued from previous page)

```
    setup_tables=True
)
```

The following MySQL database connection string is compatible with SQLAlchemy.

```
import os

os.environ['DB_URI'] = 'mysql+pymysql://{user}:{password}@{host}/eventsourcing?charset=utf8mb4&binary_
↳prefix=true'.format(
    os.getenv('MYSQL_USER', 'root'),
    os.getenv('MYSQL_PASSWORD', ''),
    os.getenv('MYSQL_HOST', '127.0.0.1'),
)
```

The MySQL database needs to be created before running the next bit of code.

```
$ mysql -e "CREATE DATABASE eventsourcing;"
```

## Single pipeline

Since the multi-processing pipeline is asynchronous, let's define a method to check things are eventually done.

```
@retry((AssertionError, KeyError), max_attempts=50, wait=0.1)
def assert_eventually_done(repository, cmd_id):
    """Checks the command is eventually done."""
    assert repository[cmd_id].is_done
```

The multiple operating system processes can be started by using the runner as a context manager.

```
with runner:

    # Create "create order" command.
    cmd_id = runner.commands.create_order()

    # Wait for the processing to complete....
    assert_eventually_done(system.commands.repository, cmd_id)

    # Check the command has an order ID and is done.
    cmd = runner.commands.repository[cmd_id]
    assert cmd.order_id

    # Check the order is reserved and paid.
    order = runner.orders.repository[cmd.order_id]
    assert order.is_reserved
    assert order.is_paid

    # Check the reservation exists.
    reservation = runner.reservations.repository[order.reservation_id]

    # Check the payment exists.
    payment = runner.payments.repository[order.payment_id]
```

## Multiple pipelines

The system can run with many instances of its pipeline. By having more than one instance of the system pipeline, more than one instance of each process application can be instantiated (one for each pipeline). Pipelines are distinguished by integer ID. The `pipeline_ids` are given to the `MultiprocessRunner` class when the runner is constructed.

In this example, there are three pipeline IDs, so there will be three instances of the system pipeline, giving twelve child operating system processes altogether.

```
runner = MultiprocessRunner(
    system=system,
    infrastructure_class=SQLAlchemyApplication,
    setup_tables=True,
    pipeline_ids = [0, 1, 2]
)
```

Fifteen orders will be processed by the system altogether, five in each pipeline.

```
num_orders = 15

with runner:

    # Create new orders.
    command_ids = []
    while len(command_ids) < num_orders:
        for pipeline_id in runner.pipeline_ids:

            # Change the pipeline for the command.
            runner.commands.change_pipeline(pipeline_id)

            # Create a "create new order" command.
            cmd_id = runner.commands.create_order()
            command_ids.append(cmd_id)

    # Check all commands are eventually done.
    assert len(command_ids)
    for command_id in command_ids:
        assert_eventually_done(runner.commands.repository, command_id)
```

It would be possible to run the system with e.g. pipelines 0-7 on one machine, pipelines 8-15 on another machine, and so on. That sort of thing can be expressed in configuration management, for example with [Kubernetes](#).

If cluster scaling is automated, it would be useful for processes to be distributed automatically across the cluster. Actor model seems like one possible foundation for such automation.

## Actor model runner

An Actor model library, in particular the [Thespian Actor Library](#), can also be used to run a multi-pipeline system of process applications.

The example below runs with Thespian's "simple system base". The actors will run by sending messages recursively.

```
from eventsourcing.application.actors import ActorModelRunner

with ActorModelRunner(system=system, setup_tables=True,
                      infrastructure_class=SQLAlchemyApplication,
                      pipeline_ids=[0, 1, 2]) as runner:
```

(continues on next page)

(continued from previous page)

```
# Create new orders.
command_ids = []
while len(command_ids) < num_orders:
    for pipeline_id in runner.pipeline_ids:

        # Change the pipeline for the command.
        runner.commands.change_pipeline(pipeline_id)

        # Create a "create new order" command.
        cmd_id = runner.commands.create_order()
        command_ids.append(cmd_id)

# Check all commands are eventually done.
assert len(command_ids)
for command_id in command_ids:
    assert_eventually_done(runner.commands.repository, command_id)
```

With Thespian, a “system base” other than the default “simple system base” can be started by calling the functions `start_multiproc_tcp_base_system()` or `start_multiproc_queue_base_system()` before starting the system actors.

The base system can be shutdown by calling `shutdown_actor_system()`, which will shutdown any actors that are running in that base system.

With the “multiproc” base systems, the process application system actors will be started in separate operating system processes. After they have been started, they will continue to run until they are shutdown. The system actors can be started by calling `actors.start()`. The actors can be shutdown with `actors.shutdown()`.

If `actors` is used as a context manager, as above, the `start()` method is called when the context manager enters. The `close()` method is called when the context manager exits. By default the `shutdown()` method is not called by `close()`. If `ActorModelRunner` is constructed with `shutdown_on_close=True`, which is `False` by default, then the actors will be shutdown by `close()`, and so also when the context manager exits. Even so, shutting down the system actors will not shutdown a “multiproc” base system.

## Pool of workers

An alternative to having a thread dedicated to every process application for each pipeline, the prompts could be sent to via a queue to a pool of workers, which change pipeline and application according to the prompt. Causal dependencies would be needed for all notifications, which is not the library default. The library does not currently support processing events with a pool of workers.

## 12.5 Integration with APIs

Integration with systems that present a server API or otherwise need to be sent messages (rather than using notification logs), can be integrated by responding to events with a policy that uses a client to call the API or send a message. However, if there is a breakdown during the API call, or before the tracking record is written, then to avoid failing to make the call, it may happen that the call is made twice. If the call is not idempotent, and is not otherwise guarded against duplicate calls, there may be consequences to making the call twice, and so the situation cannot really be described as reliable.

If the server response is asynchronous, any callbacks that the server will make could be handled by calling commands on aggregates. If callbacks might be retried, perhaps because the handler crashes after successfully calling a command but before returning successfully to the caller, unless the callbacks are also tracked (with exclusive tracking records

written atomically with new event and notification records) the aggregate commands will need to be idempotent, or otherwise guarded against duplicate callbacks. Such an integration could be implemented as a separate “push-API adapter” process, and it might be useful to have a generic implementation that can be reused, with documentation describing how to make such an integration reliable, however the library doesn’t currently have any such adapter process classes or documentation.

## 12.6 Process event pattern

*draft*

A set of EVENT SOURCED APPLICATIONS can be composed into a system of applications. Application state can be propagated to other applications. Application state is defined by domain event records that have been committed. Each application has a policy which defines how it responds to the domain events it processes.

Infrastructure may fail at any time. Although committed database transactions are expected to be durable, the operating system processes, the network, and the databases may go down at any time. Depending on the system design, application state may be adversely affected by infrastructure failures.

Therefore...

Use counting to sequence the domain events of an application. Use a unique constraint to make sure only one domain event is recorded for each position. Ensure there are no gaps by calculating the next position from the last recorded position. Also use counting to follow the domain events of an upstream application. Use a tracking record to store the current position in the upstream sequence. Use a unique constraint to make sure tracking can be recorded for each upstream domain event only once.

Use atomic database transactions to record process event atomically. Include the tracking position, the new domain events created by application policy, and their position in the application’s sequence. Use an object class (or other data type) called “ProcessEvent” to keep these data together, so that they can be passed into functions as a single argument.

Then, the distributed system can be considered reliable in the sense that the facts in the database will represent either that a process event occurred or that it didn’t occur, and so application state will be entirely unaffected by infrastructure failures.

Event sourced applications may be implemented with EVENT SOURCED AGGREGATES. To scale the system, use CAUSAL DEPENDENCIES to synchronise parallel pipelines. Use SYSTEM RUNNERS to bind system to infrastructure it needs to run.



# CHAPTER 13

## 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*

### 13.1 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
```

### 13.2 Application

The library class `SnapshottingApplication`, extends `Application` by setting up the application for snapshotting with a snapshot store, a dedicated table for snapshots, and a policy to take snapshots every so many events. A separate table is used for snapshot records.

```
from eventsourcing.application.snapshotting import SnapshottingApplication
from eventsourcing.application.sqlalchemy import SQLAlchemyApplication

class MyApplication(SQLAlchemyApplication, SnapshottingApplication):
    pass
```

## 13.3 Run the code

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

```
with MyApplication(snapshot_period=2, persist_event_type=Example.Event) 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 is not None
    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
```

(continues on next page)

(continued from previous page)

```

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

```



---

## 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*

## 14.1 Application object

In general you want one, and only one, instance of your application class in each process. If your eventsourcing application class has any subscriptions to the internal pub-sub mechanism, for example if has a persistence policy that will persist events whenever they are published, then constructing more than one instance of the application will cause 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 very well.

To make sure there is only one instance of your application class 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.

By the way, it's possible to have more than one application, but in general they need to have different domain event classes for the `persist_event_type` value of each application, so they don't try to persist each other's events. However, in this example only one application is deployed and it is deployed without a value of `persist_event_type` being set, so before this application would actually persist any events, the class of domain events it will persist must be given.

```
from eventsourcing.application.sqlalchemy import SQLAlchemyApplication

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

_application = None

def init_application(persist_event_type=None, **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()

app = 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()
```

### 14.1.1 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

app = get_application()
```

(continues on next page)

(continued from previous page)

```
app = get_application() # same object
app = get_application() # same object

close_application()
```

## 14.2 Database connection

Typically, your event sourcing 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 event sourcing 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 event sourcing 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 event sourcing 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.

### 14.2.1 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 event sourcing application.

### 14.2.2 Cassandra

Cassandra connections can be set up entirely independently of the application object.



## 14.3 Web interfaces

### 14.3.1 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.

### 14.3.2 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__)
application.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
application.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# 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)
```

(continues on next page)

(continued from previous page)

```
# 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).
topic = 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 eventsourcing 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 `eventsourcing.example.interface.flaskapp`, which is tested both stand-alone and with uWSGI. The Flask application method “before\_first\_request” is used to decorate an application object constructor, just before a request is made, so that the module can be imported by the test suite, without immediately constructing the application.

## 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.

### 14.3.3 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).

### 14.3.4 Zope

#### Zope with SQLAlchemy

The `Zope-SQLAlchemy` project serves a similar function to `Flask-SQLAlchemy`.

## 14.4 Task queues

This section contains suggestions about using an eventsourcing application in task queue workers.

### 14.4.1 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_application(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.

### 14.4.2 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.



---

### 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*

## 15.1 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.

### 15.1.1 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](#).



### 15.1.2 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)
```

### 15.1.3 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
```

(continues on next page)

(continued from previous page)

```

    # 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.

### 15.1.4 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.

### 15.1.5 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'
```

## 15.2 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]
```

### 15.2.1 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).
    state = 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
```

(continues on next page)

(continued from previous page)

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

## 15.2.2 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 an 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.

## 15.2.3 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,
```

(continues on next page)

(continued from previous page)

```
mutator_func=mutate
)
```

## 15.2.4 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.store(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'
```

## 15.2.5 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 *state*.

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 *state* 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].state

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].state
```

## 15.3 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.

### 15.3.1 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.store(event)
```

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

### 15.3.2 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.



## 15.4 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
```



# CHAPTER 16

---

## Background

---

### Inspiration:

- [Martin Fowler's article on event sourcing](#)
- [Greg Young's discussions about event sourcing, and Event Store 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:

- [An introduction to event storming by a Steven Lowe](#)
- [An Introduction to Domain Driven Design by Dan Haywood](#)
- [Object-relational impedance mismatch page on Wikipedia](#)
- [From CRUD to Event Sourcing \(Why CRUD is the wrong approach for microservices\) by James Roper](#)
- [Event Sourcing and Stream Processing at Scale by Martin Kleppmann](#)
- [Data Intensive Applications with Martin Kleppmann](#)
- [The Path Towards Simplifying Consistency In Distributed Systems by Caitie McCaffrey](#)
- [Kahn Process Networks page on Wikipedia](#)
- [Don't Let the Internet Dupe You, Event Sourcing is Hard by Chris Kiehl](#)

### Citations:

- [An Evaluation on Using Coarse grained Events in an Event Sourcing Context and its Effects Compared to Fine-grained Events by Brian Ye](#)



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.

If you need help upgrading code and migrating data, please get in touch.

### 17.1 Version 7.x

Version 7.x series brought many incremental improvements across the library.

#### 17.1.1 Version 7.1.1 (released 21 Jun 2019)

Version 7.1.1 adds ‘Support options’ and ‘Contributing’ sections to the documentation.

#### 17.1.2 Version 7.1.0 (released 11 Jun 2019)

Version 7.1.0 brings improved structure to the documentation.

#### 17.1.3 Version 7.0.0 (released 21 Feb 2019)

Version 7.0.0 brought many incremental improvements across the library, especially the ability to define an entire system of process applications independently of infrastructure. Please note, records fields have been renamed.

### 17.2 Version 6.x

Version 6.x series was the first release of the “process and system” code.

### **17.2.1 Version 6.2.0 (released 15 Jul 2018)**

### **17.2.2 Version 6.2.0 (released 26 Jun 2018)**

### **17.2.3 Version 6.1.0 (released 14 Jun 2018)**

### **17.2.4 Version 6.0.0 (released 23 Apr 2018)**

## **17.3 Version 5.x**

Version 5.x series was released after quite a lot of refactoring made things backward-incompatible. Support for Django ORM was added in version 5.0.0.

### **17.3.1 Version 5.1.1 (released 4 Apr 2018)**

### **17.3.2 Version 5.1.0 (released 16 Feb 2018)**

### **17.3.3 Version 5.0.0 (released 24 Jan 2018)**

## **17.4 Version 4.x**

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.

### **17.4.1 Version 4.0.0 (released 11 Dec 2017)**

## **17.5 Version 3.x**

Version 3.x series was a released after quite 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).

### **17.5.1 Version 3.1.0 (released 23 Nov 2017)**

### **17.5.2 Version 3.0.0 (released 25 May 2017)**

## **17.6 Version 2.x**

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.

### **17.6.1 Version 2.1.1 (released 30 Mar 2017)**

### **17.6.2 Version 2.1.0 (released 27 Mar 2017)**

### **17.6.3 Version 2.0.0 (released 27 Mar 2017)**

## **17.7 Version 1.x**

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.

**17.7.1 Version 1.2.1 (released 23 Oct 2016)**

**17.7.2 Version 1.2.0 (released 23 Oct 2016)**

**17.7.3 Version 1.1.0 (released 19 Oct 2016)**

**17.7.4 Version 1.0.10 (released 5 Oct 2016)**

**17.7.5 Version 1.0.9 (released 17 Aug 2016)**

**17.7.6 Version 1.0.8 (released 30 Jul 2016)**

**17.7.7 Version 1.0.7 (released 13 Jul 2016)**

**17.7.8 Version 1.0.6 (released 7 Jul 2016)**

**17.7.9 Version 1.0.5 (released 1 Jul 2016)**

**17.7.10 Version 1.0.4 (released 30 Jun 2016)**

**17.7.11 Version 1.0.3 (released 30 Jun 2016)**

**17.7.12 Version 1.0.2 (released 8 Jun 2016)**

**17.7.13 Version 1.0.1 (released 7 Jun 2016)**

## **17.8 Version 0.x**

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.



- 17.8.1 Version 0.9.4 (released 11 Feb 2016)
- 17.8.2 Version 0.9.3 (released 1 Dec 2015)
- 17.8.3 Version 0.9.2 (released 1 Dec 2015)
- 17.8.4 Version 0.9.1 (released 10 Nov 2015)
- 17.8.5 Version 0.9.0 (released 14 Sep 2015)
- 17.8.6 Version 0.8.4 (released 14 Sep 2015)
- 17.8.7 Version 0.8.3 (released 5 Sep 2015)
- 17.8.8 Version 0.8.2 (released 5 Sep 2015)
- 17.8.9 Version 0.8.1 (released 4 Sep 2015)
- 17.8.10 Version 0.8.0 (released 29 Aug 2015)
- 17.8.11 Version 0.7.0 (released 29 Aug 2015)
- 17.8.12 Version 0.6.0 (released 28 Aug 2015)
- 17.8.13 Version 0.5.0 (released 28 Aug 2015)
- 17.8.14 Version 0.4.0 (released 28 Aug 2015)
- 17.8.15 Version 0.3.0 (released 28 Aug 2015)
- 17.8.16 Version 0.2.0 (released 27 Aug 2015)
- 17.8.17 Version 0.1.0 (released 27 Aug 2015)
- 17.8.18 Version 0.0.1 (released 27 Aug 2015)



# CHAPTER 18

---

## Contributing

---

This library depends on its community. As interest keeps growing, we always need more people to help others. As soon as you learn the library, you can contribute in many ways.

- Join the [Slack channel](#) and answer questions. The library has a growing audience, and we really want to maintain a friendly and helpful atmosphere.
- Blog and tweet about the library. If you would like others to see your blog or tweet, send a message about it to the Slack channel.
- Contribute to open-source projects that use this library, write some documentation, or release your own event sourcing project.

If you think using the library is a lot of fun, wait until you start working on it. We're passionate about helping users of the library make the jump to contributing members of the community, so there are several ways you can help the library's development:

- Report bugs on our [issue tracker](#).
- Join the [Slack channel](#) and share your ideas for how to improve the library. We're always open to suggestions.
- Submit patches or pull requests for new and/or fixed behavior.
- Improve the documentation or write unit tests.



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

- `genindex`
- `modindex`

## 19.1 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.

### 19.1.1 application

The application layer brings together the domain and infrastructure layers.

#### actors

```
class eventsourcing.application.actors.ActorModelRunner (system:      eventsourc-  
                                                         ing.application.system.System,  
                                                         pipeline_ids,          sys-  
                                                         tem_actor_name='system',  
                                                         shut-  
                                                         down_on_close=False,  
                                                         **kwargs)
```

Bases: *eventsourcing.application.system.SystemRunner*

Uses actor model framework to run a system of process applications.

**actor\_system**

**close()**

Stops all the actors running a system of process applications.

```

    forward_prompt(prompt)

    static is_prompt(event)

    shutdown()

    start()
        Starts all the actors to run a system of process applications.

class eventsourcing.application.actors.PipelineActor
    Bases: thespian.actors.Actor

    forward_prompt(msg)

    init_pipeline(msg)

    receiveMessage(msg, sender)
        Main entry point handling a request received by this Actor. Runs without interruption and may access
        locals to this Actor (only) without concern that these locals will be modified externally.

class eventsourcing.application.actors.PipelineInitRequest(process_classes, in-
                                                         frastructure_class,
                                                         system_followings,
                                                         pipeline_id)

    Bases: object

class eventsourcing.application.actors.ProcessInitRequest(process_application_class,
                                                         infrastructure_class,
                                                         pipeline_id, up-
                                                         stream_application_names,
                                                         downstream_actors,
                                                         pipeline_actor)

    Bases: object

class eventsourcing.application.actors.ProcessMaster
    Bases: thespian.actors.Actor

    consume_prompt(prompt)

    handle_slave_run_response()

    init_process(msg)

    receiveMessage(msg, sender)
        Main entry point handling a request received by this Actor. Runs without interruption and may access
        locals to this Actor (only) without concern that these locals will be modified externally.

    run_slave()

class eventsourcing.application.actors.ProcessSlave
    Bases: thespian.actors.Actor

    close()

    init_process(msg)

    is_my_prompt(prompt)

    receiveMessage(msg, sender)
        Main entry point handling a request received by this Actor. Runs without interruption and may access
        locals to this Actor (only) without concern that these locals will be modified externally.

    run_process(msg)

    send_prompt(prompt)

```

```

class eventsourcing.application.actors.SlaveRunRequest (last_prompts, master)
    Bases: object

class eventsourcing.application.actors.SlaveRunResponse
    Bases: object

class eventsourcing.application.actors.SystemActor
    Bases: thespian.actors.Actor

    init_pipelines (msg)

    receiveMessage (msg, sender)
        Main entry point handling a request received by this Actor. Runs without interruption and may access
        locals to this Actor (only) without concern that these locals will be modified externally.

class eventsourcing.application.actors.SystemInitRequest (process_classes, in-
                                                         frastructure_class,
                                                         system_followings,
                                                         pipeline_ids)

    Bases: object

class eventsourcing.application.actors.SystemInitResponse (pipeline_actors)
    Bases: object

eventsourcing.application.actors.shutdown_actor_system()

eventsourcing.application.actors.start_actor_system(system_base=None,
                                                    logcfg={'formatters': {'nor-
                                                         mal': {'format': '%(levelname)-
                                                         8s %(message)s'}}, 'handlers':
                                                         {}, 'loggers': {}, 'version': 1})

eventsourcing.application.actors.start_multiproc_tcp_base_system()

```

## command

```

class eventsourcing.application.command.CommandProcess (name=None, policy=None,
                                                         setup_table=False,
                                                         **kwargs)

    Bases: eventsourcing.application.process.ProcessApplication

    persist_event_type
        alias of eventsourcing.domain.model.command.Command.Event

```

## django

```
class eventsourcing.application.django.DjangoApplication (name="",      persis-
                                     tence_policy=None, per-
                                     sist_event_type=None,
                                     cipher_key=None,    se-
                                     quenced_item_class=None,
                                     se-
                                     quenced_item_mapper_class=None,
                                     record_manager_class=None,
                                     stored_event_record_class=None,
                                     snap-
                                     shot_record_class=None,
                                     setup_table=True,
                                     contigu-
                                     ous_record_ids=True,
                                     pipeline_id=0,
                                     json_encoder_class=None,
                                     json_decoder_class=None,
                                     notifica-
                                     tion_log_section_size=None)
```

Bases: *eventsourcing.application.simple.ApplicationWithConcreteInfrastructure*

### **record\_manager\_class**

alias of *eventsourcing.infrastructure.django.manager.DjangoRecordManager*

### **classmethod reset\_connection\_after\_forking()**

Resets database connection after forking.

### **snapshot\_record\_class**

alias of *eventsourcing.infrastructure.django.models.EntitySnapshotRecord*

### **stored\_event\_record\_class**

alias of *eventsourcing.infrastructure.django.models.StoredEventRecord*

## multiprocess

```
class eventsourcing.application.multiprocess.MultiprocessRunner (system:
                                     eventsourc-
                                     ing.application.system.System,
                                     pipeline_ids=(0,
                                     ),
                                     poll_interval=None,
                                     setup_tables=False,
                                     sleep_for_setup_tables=0,
                                     *args,
                                     **kwargs)
```

Bases: *eventsourcing.application.system.SystemRunner*

**broadcast\_prompt** (prompt)

**close** ()

**static is\_prompt** (event)

**start** ()



```
class eventsourcing.application.multiprocess.OperatingSystemProcess (application_process_class,
                                                                    infras-
                                                                    truc-
                                                                    ture_class,
                                                                    up-
                                                                    stream_names,
                                                                    pipeline_id=0,
                                                                    poll_interval=5,
                                                                    setup_tables=False,
                                                                    in-
                                                                    box=None,
                                                                    out-
                                                                    box=None,
                                                                    *args,
                                                                    **kwargs)

    Bases: multiprocessing.context.Process

    broadcast_prompt (prompt)

    static is_prompt (event)

    loop_on_prompts ()

    run ()
        Method to be run in sub-process; can be overridden in sub-class

    run_process (prompt=None)
```

## pipeline

```
class eventsourcing.application.pipeline.Pipeable
    Bases: object

class eventsourcing.application.pipeline.PipeableMetaclass
    Bases: abc.ABCMeta

class eventsourcing.application.pipeline.PipelineExpression (left, right)
    Bases: object
```

## policies

```
class eventsourcing.application.policies.PersistencePolicy (event_store,      per-
                                                                    sist_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,      snap-
                                                                    shot_store,      per-
                                                                    sist_event_type=<class
                                                                    'eventsourc-
                                                                    ing.domain.model.events.EventWithOriginato
                                                                    period=2)
```

Bases: `object`

**close()**

**condition** (*event*)

**take\_snapshot** (*event*)

## popo

```
class eventsourcing.application.popo.PopoApplication (name="",
                                                    persistence_policy=None,
                                                    persistence_event_type=None,
                                                    cipher_key=None,
                                                    sequenced_item_class=None,
                                                    sequenced_item_mapper_class=None,
                                                    record_manager_class=None,
                                                    stored_event_record_class=None,
                                                    snapshot_record_class=None,
                                                    setup_table=True,
                                                    contiguous_record_ids=True,
                                                    pipeline_id=0,
                                                    json_encoder_class=None,
                                                    json_decoder_class=None,
                                                    notification_log_section_size=None)
```

Bases: `eventsourcing.application.simple.ApplicationWithConcreteInfrastructure`

**infrastructure\_factory\_class**

alias of `eventsourcing.infrastructure.popo.factory.PopoInfrastructureFactory`

**sequenced\_item\_mapper\_class**

alias of `eventsourcing.infrastructure.popo.mapper.SequencedItemMapperForPopo`

**stored\_event\_record\_class**

alias of `eventsourcing.infrastructure.popo.records.StoredEventRecord`

## process

```
class eventsourcing.application.process.ProcessApplication (name=None,
                                                            policy=None,
                                                            setup_table=False,
                                                            **kwargs)
```

Bases: `eventsourcing.application.pipeline.Pipeable`, `eventsourcing.application.simple.SimpleApplication`

**call\_policy** (*event*)

**close()**

**collect\_pending\_events** (*aggregates*)

**construct\_event\_records** (*pending\_events*, *causal\_dependencies=None*)

**construct\_tracking\_kwargs** (*notification\_id*, *upstream\_application\_name*)

**del\_notification\_generator** (*upstream\_name*)

```

drop_table()

follow(upstream_application_name, notification_log)

get_event_from_notification(notification)

get_notification_generator(upstream_name, advance_by)

static policy(repository, event)
    Empty method, can be overridden in subclasses to implement concrete policy.

process_upstream_event(event, notification_id, upstream_name)

publish_prompt(event=None)
    Publishes prompt for a given event.

    Used to prompt downstream process application when an event is published by this application's model,
    which can happen when application command methods, rather than the process policy, are called.

    Wraps exceptions with PromptFailed, to avoid application policy exceptions being seen directly in other
    applications when running synchronously in single thread.

read_reader(upstream_name, advance_by=None)

record_process_event(process_event)

run(prompt=None, advance_by=None)

set_notification_ids = False

set_reader_position_from_tracking_records(upstream_name)

setup_table()

take_snapshots(new_events)

use_causal_dependencies = False

class eventsourcing.application.process.ProcessApplicationWithSnapshotting(snapshot_period=None,
                                                                           snap-
                                                                           shot_record_class=None,
                                                                           **kwargs)

    Bases: eventsourcing.application.snapshotting.SnapshottingApplication,
            eventsourcing.application.process.ProcessApplication

    take_snapshots(new_events)

class eventsourcing.application.process.ProcessEvent(new_events, tracking_kargs=None,
                                                    causal_dependencies=None)

    Bases: object

class eventsourcing.application.process.Prompt(process_name, pipeline_id)

    Bases: object

class eventsourcing.application.process.RepositoryWrapper(repository)

    Bases: object

```

## simple

```
class eventsourcing.application.simple.ApplicationWithConcreteInfrastructure (name="",
                                                                    per-
                                                                    sis-
                                                                    tence_policy=None,
                                                                    per-
                                                                    sist_event_type=None,
                                                                    ci-
                                                                    pher_key=None,
                                                                    se-
                                                                    quenced_item_class=None,
                                                                    se-
                                                                    quenced_item_mapper_class=None,
                                                                    record_manager_class=None,
                                                                    stored_event_record_class=None,
                                                                    snap-
                                                                    shot_record_class=None,
                                                                    setup_table=True,
                                                                    con-
                                                                    tigu-
                                                                    ous_record_ids=True,
                                                                    pipeline_id=0,
                                                                    json_encoder_class=None,
                                                                    json_decoder_class=None,
                                                                    no-
                                                                    ti-
                                                                    fi-
                                                                    ca-
                                                                    tion_log_section_size=None)
```

Bases: `eventsourcing.application.simple.SimpleApplication`

Subclasses have actual infrastructure.

```
class eventsourcing.application.simple.SimpleApplication (name="",      persis-
                                                                    tence_policy=None, per-
                                                                    sist_event_type=None,
                                                                    cipher_key=None,    se-
                                                                    quenced_item_class=None,
                                                                    se-
                                                                    quenced_item_mapper_class=None,
                                                                    record_manager_class=None,
                                                                    stored_event_record_class=None,
                                                                    snap-
                                                                    shot_record_class=None,
                                                                    setup_table=True,
                                                                    contigu-
                                                                    ous_record_ids=True,
                                                                    pipeline_id=0,
                                                                    json_encoder_class=None,
                                                                    json_decoder_class=None,
                                                                    notifica-
                                                                    tion_log_section_size=None)
```

Bases: `abc.ABC`

Base class for event sourced applications.

Constructs infrastructure objects such as the repository and event store, and also the notification log which presents the application state as a sequence of events.

Needs actual infrastructure classes.

```

classmethod bind (*bases, **kwargs)
change_pipeline (pipeline_id)
close ()
construct_cipher (cipher_key)
construct_datastore ()
construct_event_store ()
construct_infrastructure (*args, **kwargs)
construct_infrastructure_factory (*args, **kwargs)

    Return type InfrastructureFactory
construct_notification_log ()
construct_persistence_policy ()
construct_repository (**kwargs)
datastore
drop_table ()
event_store
event_store_class
    alias of eventsourcing.infrastructure.eventstore.EventStore
infrastructure_factory_class
    alias of eventsourcing.infrastructure.factory.InfrastructureFactory
is_constructed_with_session = False
json_decoder_class = None
json_encoder_class = None
classmethod mixin (*bases)
notification_log_section_size = None
persist_event_type = None
record_manager_class = None
repository
repository_class
    alias of eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository
classmethod reset_connection_after_forking ()
sequenced_item_class = None
sequenced_item_mapper_class = None
setup_table ()
snapshot_record_class = None

```

```
stored_event_record_class = None
use_cache = False
```

## snapshotting

```
class eventsourcing.application.snapshotting.SnapshottingApplication (snapshot_period=None,
                                                                    snap-
                                                                    shot_record_class=None,
                                                                    **kwargs)
```

Bases: *eventsourcing.application.simple.SimpleApplication*

```
close()
construct_event_store()
construct_persistence_policy()
construct_repository (**kwargs)
drop_table()
setup_table()
snapshot_period = 2
```

## sqlalchemy

```
class eventsourcing.application.sqlalchemy.SQLAlchemyApplication (uri=None,
                                                                ses-
                                                                sion=None,
                                                                **kwargs)
```

Bases: *eventsourcing.application.simple.ApplicationWithConcreteInfrastructure*

```
construct_infrastructure (*args, **kwargs)
infrastructure_factory_class
    alias      of      eventsourcing.infrastructure.sqlalchemy.factory.
                        SQLAlchemyInfrastructureFactory
is_constructed_with_session = True
snapshot_record_class
    alias      of      eventsourcing.infrastructure.sqlalchemy.records.
                        EntitySnapshotRecord
stored_event_record_class
    alias of eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord
```

## system

```
class eventsourcing.application.system.BarrierControlledApplicationThread(process:
                                                                    eventsourc-
                                                                    ing.application.process.I
                                                                    fetch_barrier:
                                                                    thread-
                                                                    ing.Barrier,
                                                                    ex-
                                                                    e-
                                                                    cute_barrier:
                                                                    thread-
                                                                    ing.Barrier,
                                                                    stop_event:
                                                                    thread-
                                                                    ing.Event)
```

Bases: `threading.Thread`

**abort** ()

**run** ()

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

```
class eventsourcing.application.system.BarrierControllingClockThread(normal_speed,
                                                                    scale_factor,
                                                                    tick_interval,
                                                                    fetch_barrier:
                                                                    thread-
                                                                    ing.Barrier,
                                                                    exe-
                                                                    cute_barrier:
                                                                    thread-
                                                                    ing.Barrier,
                                                                    stop_event:
                                                                    thread-
                                                                    ing.Event,
                                                                    is_verbose=False)
```

Bases: `eventsourcing.application.system.ClockThread`

**actual\_clock\_speed**

**run** ()

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

```
class eventsourcing.application.system.ClockThread (*args, **kwargs)
```

Bases: `threading.Thread`

**call\_commands** ()

**call\_in\_future** (*cmd, ticks\_delay*)

```
class eventsourcing.application.system.InProcessRunner (system:      eventsourc-
                                                    ing.application.system.System,
                                                    infrastructure_class=None,
                                                    setup_tables=False)
```

Bases: *eventsourcing.application.system.SystemRunner*

Runs a system in the current process, either in the current thread, or with one thread for each process in the system.

```
close()
```

```
handle_prompt (prompt)
```

```
start()
```

```
class eventsourcing.application.system.MultiThreadedRunner (system:  eventsourc-
                                                    ing.application.system.System,
                                                    poll_interval=None,
                                                    clock_speed=None)
```

Bases: *eventsourcing.application.system.InProcessRunner*

Runs a system with a thread for each process.

```
broadcast_prompt (prompt)
```

```
close()
```

```
handle_prompt (prompt)
```

```
start()
```

```
start_clock()
```

```
class eventsourcing.application.system.ProcessRunningClockThread (normal_speed,
                                                                    scale_factor,
                                                                    stop_event:
                                                                    thread-
                                                                    ing.Event,
                                                                    is_verbose=False,
                                                                    seen_prompt_events=None,
                                                                    pro-
                                                                    cesses=None)
```

Bases: *eventsourcing.application.system.ClockThread*

```
actual_clock_speed
```

```
run()
```

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

```
class eventsourcing.application.system.PromptOutbox
```

Bases: object

Has a collection of downstream prompt inboxes.

```
put (prompt)
```

Puts prompt in each downstream inbox (an actual queue).



```
class eventsourcing.application.system.PromptQueuedApplicationThread (process:
                                                                    eventsourc-
                                                                    ing.application.process.Process,
                                                                    poll_interval=5,
                                                                    in-
                                                                    box=None,
                                                                    out-
                                                                    box=None,
                                                                    clock_event=None)
```

Bases: `threading.Thread`

Application thread which uses queues of prompts.

It loops on an “inbox” queue of prompts, and adds its prompts to an “outbox” queue.

**loop\_on\_prompts** ()

**run** ()

Method representing the thread’s activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

**run\_process** (*prompt=None*)

```
class eventsourcing.application.system.SingleThreadedRunner (system: eventsourc-
                                                                    ing.application.system.System,
                                                                    infrastruc-
                                                                    ture_class=<class
                                                                    'eventsourc-
                                                                    ing.application.popo.PopoApplication'>,
                                                                    *args, **kwargs)
```

Bases: `eventsourcing.application.system.InProcessRunner`

Runs a system in the current thread.

**handle\_prompt** (*prompt*)

**run\_followers** (*prompt*)

First caller adds a prompt to queue and runs followers until there are no more pending prompts.

Subsequent callers just add a prompt to the queue, avoiding recursion.

```
class eventsourcing.application.system.SteppingMultiThreadedRunner (*args,
                                                                    **kwargs)
```

Bases: `eventsourcing.application.system.SteppingRunner`

Has a clock thread, and a thread for each application process in the system. The clock thread loops until stopped, waiting for a barrier, after sleeping for remaining tick interval timer. Application threads loop until stopped, waiting for the same barrier. Then, after all threads are waiting at the barrier, the barrier is lifted. The clock thread proceeds by sleeping for the clock tick interval. The application threads proceed by getting new notifications and processing all of them.

There are actually two barriers, so that each application thread waits before getting notifications, and then waits for all processes to complete getting notification before processing the notifications through the application policy. This avoids events created by a process application “bleeding” into the notifications of another process application in the same clock cycle.

Todo: Receive prompts, but set an event for the prompting process, to avoid unnecessary runs.

Allow commands to be scheduled at future clock tick number, and execute when reached.

```

    close()
    handle_prompt(prompt)
    start()
class eventsourcing.application.system.SteppingRunner(normal_speed=1,
                                                    scale_factor=1,
                                                    is_verbose=False,    *args,
                                                    **kwargs)
    Bases: eventsourcing.application.system.InProcessRunner
    call_in_future(cmd, ticks_delay)
class eventsourcing.application.system.SteppingSingleThreadedRunner(*args,
                                                                    **kwargs)
    Bases: eventsourcing.application.system.SteppingRunner
    close()
    handle_prompt(prompt)
    start()
class eventsourcing.application.system.System(*pipeline_exprs, **kwargs)
    Bases: object
    __init__(*pipeline_exprs, **kwargs)
        Initialises a “process network” system object.

        Parameters pipeline_exprs – Pipeline expressions involving process application classes.

        Each pipeline expression of process classes shows directly which process follows which other process in
        the system.

        For example, the pipeline expression (A | B | C) shows that B follows A, and C follows B.

        The pipeline expression (A | A) shows that A follows A.

        The pipeline expression (A | B | A) shows that B follows A, and A follows B.

        The pipeline expressions ((A | B | A), (A | C | A)) are equivalent to (A | B | A | C | A).

    construct_app(process_class, infrastructure_class=None, **kwargs)
    is_prompt(event)
class eventsourcing.application.system.SystemRunner(system: eventsourcing.application.system.System,
                                                    infrastructure_class=None,
                                                    setup_tables=False)
    Bases: abc.ABC
    close()
    start()

```

### 19.1.2 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 (*args, **kwargs)
    Bases: eventsourcing.domain.model.aggregate.AggregateRootWithHashchainedEvents

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

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

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

class Event (**kwargs)
    Bases: eventsourcing.domain.model.aggregate.Event
    Supertype for aggregate events.
    __module__ = 'eventsourcing.domain.model.aggregate'
__module__ = 'eventsourcing.domain.model.aggregate'

class eventsourcing.domain.model.aggregate.AggregateRootWithHashchainedEvents (*args,
                                                                                   **kwargs)
    Bases: eventsourcing.domain.model.entity.EntityWithHashchain, eventsourcing.
    domain.model.aggregate.BaseAggregateRoot

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

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

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

Published when an AggregateRoot is discarded.

```
__module__ = 'event sourcing.domain.model.aggregate'
```

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

Bases: `event sourcing.domain.model.entity.Event`, `event sourcing.domain.model.aggregate.Event`

Supertype for aggregate events.

```
__module__ = 'event sourcing.domain.model.aggregate'
```

```
__module__ = 'event sourcing.domain.model.aggregate'
```

```
class event sourcing.domain.model.aggregate.BaseAggregateRoot (**kwargs)
```

Bases: `event sourcing.domain.model.entity.TimestampedVersionedEntity`

Root entity for an aggregate in a domain driven design.

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

Bases: `event sourcing.domain.model.aggregate.Event`, `event sourcing.domain.model.entity.AttributeChanged`

Published when an AggregateRoot is changed.

```
__module__ = 'event sourcing.domain.model.aggregate'
```

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

Bases: `event sourcing.domain.model.aggregate.Event`, `event sourcing.domain.model.entity.Created`

Published when an AggregateRoot is created.

```
__module__ = 'event sourcing.domain.model.aggregate'
```

```
class Discarded (originator_version, **kwargs)
```

Bases: `event sourcing.domain.model.aggregate.Event`, `event sourcing.domain.model.entity.Discarded`

Published when an AggregateRoot is discarded.

```
__module__ = 'event sourcing.domain.model.aggregate'
```

```
class Event (originator_version, **kwargs)
```

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

Supertype for aggregate events.

```
__module__ = 'event sourcing.domain.model.aggregate'
```

```
__batch_pending_events__ ()
```

```
__init__ (**kwargs)
```

Initialize self. See `help(type(self))` for accurate signature.

```
__module__ = 'event sourcing.domain.model.aggregate'
```

```
__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.
```

```
__init__ (array_size=10000, *args, **kwargs)
    Initialize self. See help(type(self)) for accurate signature.
```

```
__module__ = 'eventsourcing.domain.model.array'
```

```
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.
```

```
__module__ = 'eventsourcing.domain.model.array'
```

```
subrepo
    Sub-sequence repository.
```

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

```
__dict__ = mappingproxy({'__module__': 'eventsourcing.domain.model.array', '__init__':
```

```
__eq__ (other)
    Return self==value.
```

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

```
__hash__ = None
```

```
__init__ (array_id, repo)
    Initialize self. See help(type(self)) for accurate signature.
```

```
__len__ ()
    Returns length of array.
```

```
__module__ = 'eventsourcing.domain.model.array'
```

```
__ne__ (other)
    Return self!=value.
```

```
__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.

```
__weakref__
    list of weak references to the object (if defined)
```

```
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 event sourcing.domain.model.array.BigArray(array_id, repo)
    Bases: event sourcing.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.

```

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

__init__(array_id, repo)
    Initialize self. See help(type(self)) for accurate signature.

__len__()
    Returns length of array.

__module__ = 'event sourcing.domain.model.array'

__setitem__(position, 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.

```

**calc\_parent** (*i, j, h*)  
Returns get\_big\_array and end of span of parent sequence that contains given child.

**calc\_required\_height** (*n, size*)

**create\_array\_id** (*i, j*)

**get\_item** (*position*)

**get\_last\_array** ()  
Returns last array in compound.

**Return type** CompoundSequenceReader

**get\_last\_item\_and\_next\_position** ()

**get\_slice** (*start, stop*)

**class** eventsourcing.domain.model.array.**ItemAssigned** (*item, index, \*args, \*\*kwargs*)  
Bases: eventsourcing.domain.model.entity.Event

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

**\_\_init\_\_** (*item, index, \*args, \*\*kwargs*)  
Initialises event attribute values directly from constructor kwargs.

**\_\_module\_\_** = 'eventsourcing.domain.model.array'

**index**

**item**

## collection

Collections.

**class** eventsourcing.domain.model.collection.**AbstractCollectionRepository**  
Bases: *eventsourcing.domain.model.entity.AbstractEntityRepository*

**\_\_module\_\_** = 'eventsourcing.domain.model.collection'

**class** eventsourcing.domain.model.collection.**Collection** (*\*\*kwargs*)  
Bases: *eventsourcing.domain.model.entity.TimestampedVersionedEntity*

**class** **Created** (*originator\_version=0, \*\*kwargs*)  
Bases: eventsourcing.domain.model.collection.Event, eventsourcing.domain.model.entity.Created

Published when collection is created.

**\_\_module\_\_** = 'eventsourcing.domain.model.collection'

**class** **Discarded** (*originator\_version, \*\*kwargs*)  
Bases: eventsourcing.domain.model.collection.Event, eventsourcing.domain.model.entity.Discarded

Published when collection is discarded.

**\_\_module\_\_** = 'eventsourcing.domain.model.collection'

**class** **Event** (*originator\_version, \*\*kwargs*)  
Bases: eventsourcing.domain.model.entity.Event

Supertype for events of collection entities.

**\_\_module\_\_** = 'eventsourcing.domain.model.collection'

```

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

    __module__ = 'eventsourcing.domain.model.collection'

    item

class ItemAdded(originator_version, **kwargs)
    Bases: eventsourcing.domain.model.collection.EventWithItem

    __module__ = 'eventsourcing.domain.model.collection'

    __mutate__(obj)
        Update obj with values from self.

class ItemRemoved(originator_version, **kwargs)
    Bases: eventsourcing.domain.model.collection.EventWithItem

    __module__ = 'eventsourcing.domain.model.collection'

    __mutate__(obj)
        Update obj with values from self.

__init__(**kwargs)
    Initialize self. See help(type(self)) for accurate signature.

__iter__()

__module__ = 'eventsourcing.domain.model.collection'

add_item(item)

items

remove_item(item)

```

```
eventsourcing.domain.model.collection.register_new_collection(collection_id=None)
```

## command

Commands as aggregates.

```

class eventsourcing.domain.model.command.Command(**kwargs)
    Bases: eventsourcing.domain.model.aggregate.AggregateRoot

    class AttributeChanged(**kwargs)
        Bases: eventsourcing.domain.model.command.Event, eventsourcing.domain.model.aggregate.AttributeChanged

        __module__ = 'eventsourcing.domain.model.command'

    class Created(**kwargs)
        Bases: eventsourcing.domain.model.command.Event, eventsourcing.domain.model.aggregate.Created

        __module__ = 'eventsourcing.domain.model.command'

    class Discarded(**kwargs)
        Bases: eventsourcing.domain.model.command.Event, eventsourcing.domain.model.aggregate.Discarded

        __module__ = 'eventsourcing.domain.model.command'

    class Done(**kwargs)
        Bases: eventsourcing.domain.model.command.Event

```



```
__module__ = 'eventsourcing.domain.model.command'
```

**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

```
class Event (**kwargs)
    Bases: eventsourcing.domain.model.aggregate.Event

    __module__ = 'eventsourcing.domain.model.command'

    __init__ (**kwargs)
        Initialize self. See help(type(self)) for accurate signature.

    __module__ = 'eventsourcing.domain.model.command'

    done ()

    is_done
```

## decorator

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

`eventsourcing.domain.model.decorators.attribute` (*getter*)

When used as a method decorator, returns a property object with the method as the getter and a setter defined to call instance method `change_attribute()`, which publishes an `AttributeChanged` event.

`eventsourcing.domain.model.decorators.mutator` (*arg=None*)

Structures mutator functions by allowing handlers to be registered for different types of event. When the decorated function is called with an initial value and an event, it will call the handler that has been registered for that type of event.

It works like `singledispatch`, which it uses. The difference is that when the decorated function is called, this decorator dispatches according to the type of last call arg, which fits better with `reduce()`. The builtin Python function `reduce()` is used by the library to replay a sequence of events against an initial state. If a mutator function is given to `reduce()`, along with a list of events and an initializer, `reduce()` will call the mutator function once for each event in the list, but the initializer will be the first value, and the event will be the last argument, and we want to dispatch according to the type of the event. It happens that `singledispatch` is coded to switch on the type of the first argument, which makes it unsuitable for structuring a mutator function without the modifications introduced here.

The other aspect introduced by this decorator function is the option to set the type of the handled entity in the decorator. When an entity is replayed from scratch, in other words when all its events are replayed, the initial state is `None`. The handler which handles the first event in the sequence will probably construct an object instance. It is possible to write the type into the handler, but that makes the entity more difficult to subclass because you will also need to write a handler for it. If the decorator is invoked with the type, when the initial value passed as a call arg to the mutator function is `None`, the handler will instead receive the type of the entity, which it can use to construct the entity object.

```
class Entity(object):
    class Created(object):
```

(continues on next page)

(continued from previous page)

```

    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())

```

`event sourcing.domain.model.decorators.random()` → x in the interval [0, 1).

`event sourcing.domain.model.decorators.retry(exc=<class 'Exception'>, max_attempts=1, wait=0, stall=0, verbose=False)`

`event sourcing.domain.model.decorators.subscribe_to(*event_classes)`

Decorator for making a custom event handler function subscribe to a certain class of event.

The decorated function will be called once for each matching event that is published, and will be given one argument, the event, when it is called. If events are published in lists, for example the `AggregateRoot` publishes a list of pending events when its `__save__()` method is called, then the decorated function will be called once for each event that is an instance of the given `event_class`.

Please note, this decorator isn't suitable for use with object class methods. The decorator receives in Python 3 an unbound function, and defines a handler which it subscribes that calls the decorated function for each matching event. However the method isn't called on the object, so the object instance is never available in the decorator, so the decorator can't call a normal object method because it doesn't have a value for 'self'.

`event_class`: type used to match published events, an event matches if it is an instance of this type

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** `event sourcing.domain.model.entity.AbstractEntityRepository`

Bases: `event sourcing.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.

`__module__ = 'event sourcing.domain.model.entity'`

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

    __dict__ = mappingproxy({'__module__': 'eventsourcing.domain.model.entity', '__dict__'
    __module__ = 'eventsourcing.domain.model.entity'

    __weakref__
        list of weak references to the object (if defined)

class eventsourcing.domain.model.entity.DomainEntity (id)
    Bases: object

    Base class for domain entities.

    class AttributeChanged (originator_id, **kwargs)
        Bases: eventsourcing.domain.model.entity.Event, eventsourcing.domain.
        model.events.AttributeChanged

        Published when a DomainEntity is discarded.

        __module__ = 'eventsourcing.domain.model.entity'

        __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

    class Created (originator_topic, **kwargs)
        Bases: eventsourcing.domain.model.entity.Event, eventsourcing.domain.
        model.events.Created

        Published when an entity is created.

        __entity_kwargs__

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

        __module__ = 'eventsourcing.domain.model.entity'

        __mutate__ (entity_class=None)
            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

        originator_topic

    class Discarded (originator_id, **kwargs)
        Bases: eventsourcing.domain.model.events.Discarded, eventsourcing.domain.
        model.entity.Event

        Published when a DomainEntity is discarded.

        __module__ = 'eventsourcing.domain.model.entity'

```

```

    __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

class Event (originator_id, **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.

    __module__ = 'event sourcing.domain.model.entity'

    __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

    __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.

    classmethod __create__ (originator_id=None, event_class=None, **kwargs)

    __dict__ = mappingproxy({'__module__': 'event sourcing.domain.model.entity', '__doc__'
    __discard__ ()
        Discards self, by triggering a Discarded event.

    __eq__ (other)
        Return self==value.

    __hash__ = None

    __init__ (id)
        Initialize self. See help(type(self)) for accurate signature.

    __module__ = 'event sourcing.domain.model.entity'

    __ne__ (other)
        Return self!=value.

    __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.

```

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**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.**EntityWithHashchain**(\*args, \*\*kwargs)

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

**class** **Created**(\*\*kwargs)

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

**\_\_entity\_kwargs\_\_**

**\_\_module\_\_** = 'eventsourcing.domain.model.entity'

**\_\_mutate\_\_**(entity\_class=None)

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

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

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

**\_\_module\_\_** = 'eventsourcing.domain.model.entity'

**\_\_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

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

Bases: *eventsourcing.domain.model.events.EventWithHash*, eventsourcing.domain.model.entity.Event

Supertype for events of domain entities.

**\_\_check\_obj\_\_**(obj)

Extends superclass method by checking the **\_\_previous\_hash\_\_** of this event matches the **\_\_head\_\_** hash of the entity obj.

**\_\_module\_\_** = 'eventsourcing.domain.model.entity'

**\_\_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

**classmethod** **\_\_create\_\_**(\*args, \*\*kwargs)

**\_\_genesis\_hash\_\_** = ''

**\_\_init\_\_**(\*args, \*\*kwargs)

Initialize self. See help(type(self)) for accurate signature.

```

__module__ = 'event sourcing.domain.model.entity'

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

class event sourcing.domain.model.entity.TimestampedEntity(__created_on__=None,
                                                         **kwargs)
    Bases: event sourcing.domain.model.entity.DomainEntity

    class AttributeChanged(originator_id, **kwargs)
        Bases: event sourcing.domain.model.entity.Event, event sourcing.domain.
            model.entity.AttributeChanged

        Published when a TimestampedEntity is changed.

        __module__ = 'event sourcing.domain.model.entity'

    class Created(originator_topic, **kwargs)
        Bases: event sourcing.domain.model.entity.Created, event sourcing.domain.
            model.entity.Event

        Published when a TimestampedEntity is created.

        __entity_kwargs__

        __module__ = 'event sourcing.domain.model.entity'

    class Discarded(originator_id, **kwargs)
        Bases: event sourcing.domain.model.entity.Event, event sourcing.domain.
            model.entity.Discarded

        Published when a TimestampedEntity is discarded.

        __module__ = 'event sourcing.domain.model.entity'

    class Event(originator_id, **kwargs)
        Bases: event sourcing.domain.model.entity.Event, event sourcing.domain.
            model.events.EventWithTimestamp

        Supertype for events of timestamped entities.

        __module__ = 'event sourcing.domain.model.entity'

        __mutate__(obj)
            Update obj with values from self.

        __created_on__

        __init__(__created_on__=None, **kwargs)
            Initialize self. See help(type(self)) for accurate signature.

        __last_modified__

        __module__ = 'event sourcing.domain.model.entity'

class event sourcing.domain.model.entity.TimestampedVersionedEntity(__created_on__=None,
                                                                    **kwargs)
    Bases: event sourcing.domain.model.entity.TimestampedEntity, event sourcing.
        domain.model.entity.VersionedEntity

    class AttributeChanged(originator_version, **kwargs)
        Bases: event sourcing.domain.model.entity.Event, event sourcing.domain.
            model.entity.AttributeChanged, event sourcing.domain.model.entity.
            AttributeChanged

        Published when a TimestampedVersionedEntity is created.

```

```

    __module__ = 'eventsourcing.domain.model.entity'

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.

    __module__ = 'eventsourcing.domain.model.entity'

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.

    __module__ = 'eventsourcing.domain.model.entity'

class Event(originator_version, **kwargs)
    Bases: eventsourcing.domain.model.entity.Event, eventsourcing.domain.model.entity.Event
    Supertype for events of timestamped, versioned entities.

    __module__ = 'eventsourcing.domain.model.entity'
__module__ = 'eventsourcing.domain.model.entity'

class eventsourcing.domain.model.entity.TimeuuidedEntity(event_id, **kwargs)
    Bases: eventsourcing.domain.model.entity.DomainEntity
    __created_on__
    __init__(event_id, **kwargs)
        Initialize self. See help(type(self)) for accurate signature.
    __last_modified__
    __module__ = 'eventsourcing.domain.model.entity'

class eventsourcing.domain.model.entity.TimeuuidedVersionedEntity(event_id, **kwargs)
    Bases: eventsourcing.domain.model.entity.TimeuuidedEntity, eventsourcing.domain.model.entity.VersionedEntity
    __module__ = 'eventsourcing.domain.model.entity'

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.

    __module__ = 'eventsourcing.domain.model.entity'

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

    __entity_kwargs__

```

```

__init__(originator_version=0, **kwargs)
    Initialises event attribute values directly from constructor kwargs.

__module__ = 'event sourcing.domain.model.entity'

class Discarded(originator_version, **kwargs)
    Bases: event sourcing.domain.model.entity.Event, event sourcing.domain.
    model.entity.Discarded

    Published when a VersionedEntity is discarded.

    __module__ = 'event sourcing.domain.model.entity'

class Event(originator_version, **kwargs)
    Bases: event sourcing.domain.model.events.EventWithOriginatorVersion,
    event sourcing.domain.model.entity.Event

    Supertype for events of versioned entities.

    __check_obj__(obj)
        Extends superclass method by checking the event's originator version follows (1 +) this entity's ver-
        sion.

    __module__ = 'event sourcing.domain.model.entity'

    __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

    __init__(__version__=None, **kwargs)
        Initialize self. See help(type(self)) for accurate signature.

    __module__ = 'event sourcing.domain.model.entity'

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

        The event carries the version number that the originator will have when the originator is mutated with
        this event. (The event's originator version isn't the version of the originator that triggered the event. The
        Created event has version 0, and so a newly created instance is at version 0. The second event has originator
        version 1, and so will the originator when the second event has been applied.)

    __version__

```

## events

Base classes for domain events of different kinds.

```

class event sourcing.domain.model.events.AttributeChanged(**kwargs)
    Bases: event sourcing.domain.model.events.DomainEvent

    Can be published when an attribute of an entity is created.

    __module__ = 'event sourcing.domain.model.events'

    name

    value

```



**class** eventsourcing.domain.model.events.Created(\*\*kwargs)

Bases: *eventsourcing.domain.model.events.DomainEvent*

Can be published when an entity is created.

**\_\_module\_\_** = 'eventsourcing.domain.model.events'

**class** eventsourcing.domain.model.events.Discarded(\*\*kwargs)

Bases: *eventsourcing.domain.model.events.DomainEvent*

Published when something is discarded.

**\_\_module\_\_** = 'eventsourcing.domain.model.events'

**class** eventsourcing.domain.model.events.DomainEvent(\*\*kwargs)

Bases: object

Base class for domain events.

Implements methods to make instances read-only, comparable for equality, have recognisable representations, and hashable.

**\_\_dict\_\_** = *mappingproxy*({'\_\_module\_\_': 'eventsourcing.domain.model.events', '\_\_doc\_\_'

**\_\_eq\_\_** (*other*)

Tests for equality of two event objects.

**\_\_hash\_\_** ()

Computes a Python integer hash for an event, using its event hash string.

Supports equality and inequality comparisons.

**classmethod** **\_\_hash\_object\_\_** (*obj*)

**\_\_init\_\_** (\*\*kwargs)

Initialises event attribute values directly from constructor kwargs.

**\_\_json\_encoder\_class\_\_**

alias of *eventsourcing.utils.transcoding.ObjectJSONEncoder*

**\_\_module\_\_** = 'eventsourcing.domain.model.events'

**\_\_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.

**\_\_notifiable\_\_** = **True**

**\_\_repr\_\_** ()

Returns string representing the type and attribute values of the event.

**\_\_setattr\_\_** (*key*, *value*)

Inhibits event attributes from being updated by assignment.

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**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** `event sourcing.domain.model.events.EventHandlersNotEmptyError`

Bases: `Exception`

`__module__` = `'event sourcing.domain.model.events'`

`__weakref__`

list of weak references to the object (if defined)

**class** `event sourcing.domain.model.events.EventWithHash` (*\*\*kwargs*)

Bases: `event sourcing.domain.model.events.DomainEvent`

Base class for domain events.

Implements methods to make instances read-only, comparable for equality, have recognisable representations, and hashable.

`__check_hash__` ()

`__event_hash__`

`__hash__` ()

Computes a Python integer hash for an event, using its event hash string.

Supports equality and inequality comparisons.

`__init__` (*\*\*kwargs*)

Initialises event attribute values directly from constructor *kwargs*.

`__module__` = `'event sourcing.domain.model.events'`

`__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

**class** `event sourcing.domain.model.events.EventWithOriginatorID` (*originator\_id*, *\*\*kwargs*)

Bases: `event sourcing.domain.model.events.DomainEvent`

`__init__` (*originator\_id*, *\*\*kwargs*)

Initialises event attribute values directly from constructor *kwargs*.

`__module__` = `'event sourcing.domain.model.events'`

*originator\_id*

**class** `event sourcing.domain.model.events.EventWithOriginatorVersion` (*originator\_version*, *\*\*kwargs*)

Bases: `event sourcing.domain.model.events.DomainEvent`

For events that have an originator version number.

```

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

__module__ = 'eventsourcing.domain.model.events'
originator_version

class eventsourcing.domain.model.events.EventWithTimestamp(timestamp=None,
                                                            **kwargs)
    Bases: eventsourcing.domain.model.events.DomainEvent
    For events that have a timestamp value.

    __init__(timestamp=None, **kwargs)
        Initialises event attribute values directly from constructor kwargs.

    __module__ = 'eventsourcing.domain.model.events'
    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.

    __init__(event_id=None, **kwargs)
        Initialises event attribute values directly from constructor kwargs.

    __module__ = 'eventsourcing.domain.model.events'
    event_id

class eventsourcing.domain.model.events.Logged(**kwargs)
    Bases: eventsourcing.domain.model.events.DomainEvent
    Published when something is logged.

    __module__ = 'eventsourcing.domain.model.events'

eventsourcing.domain.model.events.assert_event_handlers_empty()
eventsourcing.domain.model.events.clear_event_handlers()
eventsourcing.domain.model.events.create_timestequence_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: abc.ABC

    __abstractmethods__ = frozenset({'originator_id', 'originator_version', 'topic', 'state'})
    __dict__ = mappingproxy({'__module__': 'eventsourcing.domain.model.snapshot', 'topic'})
    __module__ = 'eventsourcing.domain.model.snapshot'
    __weakref__
        list of weak references to the object (if defined)

```

**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

__abstractmethods__ = frozenset()

__init__ (originator_id, originator_version, topic, state)
    Initialises event attribute values directly from constructor kwargs.

__module__ = 'eventsourcing.domain.model.snapshot'

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

__init__ (message, originator_id)
    Initialises event attribute values directly from constructor kwargs.

__module__ = 'eventsourcing.domain.model.timebucketedlog'

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

    __module__ = 'eventsourcing.domain.model.timebucketedlog'
```

```

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

    Supertype for events of time-bucketed log.

    __module__ = 'eventsourcing.domain.model.timebucketedlog'

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

    __module__ = 'eventsourcing.domain.model.timebucketedlog'

__init__(name, bucket_size=None, **kwargs)
    Initialize self. See help(type(self)) for accurate signature.

__module__ = 'eventsourcing.domain.model.timebucketedlog'

bucket_size

log_message(message)

name

started_on

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

    __module__ = 'eventsourcing.domain.model.timebucketedlog'

    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)

```

### 19.1.3 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.ACIDRecordManager (record_class,
                                                         sequenced_item_class=<class
                                                         'eventsourcing.infrastructure.sequenceditem.SequencedItem
                                                         contiguous_record_ids=False,
                                                         application_name=None,
                                                         pipeline_id=0)
```

Bases: `eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager`

ACID record managers can write tracking records and event records in an atomic transaction, needed for atomic processing in process applications.

```
get_max_record_id ()
    Return maximum notification ID in pipeline.

get_max_tracking_record_id (upstream_application_name)
    Return maximum tracking record ID for notification from upstream application in pipeline.

get_pipeline_and_notification_id (sequence_id, position)
    Returns pipeline ID and notification ID for event at given position in given sequence.

has_tracking_record (upstream_application_name, pipeline_id, notification_id)
    True if tracking record exists for notification from upstream in pipeline.

tracking_record_class = None

tracking_record_field_names = ['application_name', 'upstream_application_name', 'pipeline_id']

write_records (records, tracking_kwargs=None)
    Writes tracking, event and notification records for a process event.
```

```
class eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager (record_class,
                                                         sequenced_item_class=<class
                                                         'eventsourcing.infrastructure.sequenceditem.SequencedItem
                                                         contiguous_record_ids=False,
                                                         application_name=None,
                                                         pipeline_id=0)
```

Bases: `abc.ABC`

```
all_sequence_ids ()
    Returns all sequence IDs.

clone (application_name, pipeline_id, **kwargs)

delete_record (record)
    Removes permanently given record from the table.

from_record (record)
    Constructs and returns a sequenced item object, from given ORM object.

get_field_kwargs (item)
```

**get\_item** (*sequence\_id*, *position*)  
 Gets 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*)  
 Returns sequenced item generator.

**get\_notifications** (*start=None*, *stop=None*, \**args*, \*\**kwargs*)  
 Returns records sequenced by notification ID, from application, for pipeline, in given range.  
 Args 'start' and 'stop' are positions in a zero-based integer sequence.

**get\_record** (*sequence\_id*, *position*)  
 Gets record at position in sequence.

**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.

**list\_items** (\**args*, \*\**kwargs*)  
 Returns list of sequenced items.

**list\_sequence\_ids** ()

**raise\_index\_error** (*position*)

**raise\_operational\_error** (*e*)

**raise\_record\_integrity\_error** (*e*)

**raise\_sequenced\_item\_conflict** ()

**record\_sequenced\_item** (*sequenced\_item*)  
 Writes sequenced item into the datastore.

**record\_sequenced\_items** (*sequenced\_item\_or\_items*)  
 Writes sequenced item(s) into the datastore.

**to\_record** (*sequenced\_item*)  
 Constructs a record object from given sequenced item object.

**class** eventsourcing.infrastructure.base.**SQLRecordManager** (\**args*, \*\**kwargs*)  
 Bases: *eventsourcing.infrastructure.base.ACIDRecordManager*

This is has code common to (extracted from) the SQLAlchemy and Django record managers.

This makes the subclasses harder to read and probably more brittle. So it might be better to inline this with the subclasses, so that each looks more like normal Django or SQLAlchemy code. Also, the record manager test cases don't cover the notification log and tracking record functionality needed by ProcessApplication, and should so that other record managers can more easily be developed.

**\_prepare\_insert** (*tmpl*, *record\_class*, *field\_names*, *placeholder\_for\_id=False*)  
 Compile SQL statement with placeholders for bind parameters.

**get\_record\_table\_name** (*record\_class*)  
 Returns table name - used in raw queries.  
 Return type `str`

**insert\_select\_max**  
 SQL statement that inserts records with contiguous IDs, by selecting max ID from indexed table records.

**insert\_tracking\_record**  
 SQL statement that inserts tracking records.

#### **insert\_values**

SQL statement that inserts records without ID.

#### **record\_sequenced\_items** (*sequenced\_item\_or\_items*)

Writes sequenced item(s) into the datastore.

#### **to\_records** (*sequenced\_item\_or\_items*)

### **cassandra**

Classes for event sourcing with Apache Cassandra.

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

    Bases: eventsourcing.infrastructure.datastore.Datastore

    close_connection ()
        Drops connection to a datastore.

    drop_table (*_)

    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.

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
ap-
pli-
ca-
tion_nam
pipeline_
```

Bases: eventsourcing.infrastructure.factory.InfrastructureFactory

**integer\_sequenced\_record\_class**

alias of `eventsourcing.infrastructure.cassandra.records.IntegerSequencedRecord`

**record\_manager\_class**

alias of `eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager`

**snapshot\_record\_class**

alias of `eventsourcing.infrastructure.cassandra.records.SnapshotRecord`

**timestamp\_sequenced\_record\_class**

alias of `eventsourcing.infrastructure.cassandra.records.TimestampSequencedRecord`

```
class eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager (record_class,
se-
quenced_item_class=
'eventsourc-
ing.infrastructure.seq
con-
tigu-
ous_record_ids=False
ap-
pli-
ca-
tion_name=None,
pipeline_id=0)
```

Bases: `eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager`

**all\_sequence\_ids()**

Returns all sequence IDs.

**delete\_record(record)**

Removes permanently given record from the table.

```

filter (**kwargs)

get_notifications (start=None, stop=None, *args, **kwargs)
    Not implemented.

get_record (sequence_id, position)
    Gets record at position in sequence.

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.

record_sequenced_items (sequenced_item_or_items)
    Writes sequenced item(s) into the datastore.

class eventsourcing.infrastructure.cassandra.records.IntegerSequencedRecord (**values)
    Bases: cassandra.cqlengine.models.Model

    Stores integer-sequenced items in Cassandra.

    exception DoesNotExist
        Bases: cassandra.cqlengine.models.DoesNotExist

    exception MultipleObjectsReturned
        Bases: cassandra.cqlengine.models.MultipleObjectsReturned

    pk = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    position = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    sequence_id = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    state = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    topic = <cassandra.cqlengine.models.ColumnQueryEvaluator object>

class eventsourcing.infrastructure.cassandra.records.SnapshotRecord (**values)
    Bases: cassandra.cqlengine.models.Model

    Stores snapshots in Cassandra.

    exception DoesNotExist
        Bases: cassandra.cqlengine.models.DoesNotExist

    exception MultipleObjectsReturned
        Bases: cassandra.cqlengine.models.MultipleObjectsReturned

    pk = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    position = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    sequence_id = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    state = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    topic = <cassandra.cqlengine.models.ColumnQueryEvaluator object>

class eventsourcing.infrastructure.cassandra.records.StoredEventRecord (**values)
    Bases: cassandra.cqlengine.models.Model

    Stores integer-sequenced items in Cassandra.

    exception DoesNotExist
        Bases: cassandra.cqlengine.models.DoesNotExist

    exception MultipleObjectsReturned
        Bases: cassandra.cqlengine.models.MultipleObjectsReturned

```

```

    originator_id = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    originator_version = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    pk = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    state = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    topic = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
class eventsourcing.infrastructure.cassandra.records.TimestampSequencedRecord (**values)
    Bases: cassandra.cqlengine.models.Model
    Stores timestamp-sequenced items in Cassandra.

    exception DoesNotExist
        Bases: cassandra.cqlengine.models.DoesNotExist

    exception MultipleObjectsReturned
        Bases: cassandra.cqlengine.models.MultipleObjectsReturned

    pk = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    position = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    sequence_id = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    state = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    topic = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
class eventsourcing.infrastructure.cassandra.records.TimeuuidSequencedRecord (**values)
    Bases: cassandra.cqlengine.models.Model
    Stores timeuuid-sequenced items in Cassandra.

    exception DoesNotExist
        Bases: cassandra.cqlengine.models.DoesNotExist

    exception MultipleObjectsReturned
        Bases: cassandra.cqlengine.models.MultipleObjectsReturned

    pk = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    position = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    sequence_id = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    state = <cassandra.cqlengine.models.ColumnQueryEvaluator object>
    topic = <cassandra.cqlengine.models.ColumnQueryEvaluator object>

```

## datastore

Base classes for concrete datastore classes.

```

class eventsourcing.infrastructure.datastore.Datastore (settings)
    Bases: abc.ABC

    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.factory.DjangoInfrastructureFactory` (*record\_manager\_class=Sequence-  
sequenced\_item\_class=None, in-  
te-  
ger\_sequenced\_record\_class=None, times-  
stamp\_sequenced\_record\_class=None, snapshot\_record\_class=None, con-  
tinuous\_record\_ids=False, ap-  
plication\_name=None, pipeline\_id=0*)

Bases: `eventsourcing.infrastructure.factory.InfrastructureFactory`

**integer\_sequenced\_record\_class**  
alias of `eventsourcing.infrastructure.django.models.IntegerSequencedRecord`

**record\_manager\_class**  
alias of `eventsourcing.infrastructure.django.manager.DjangoRecordManager`

**snapshot\_record\_class**  
alias of `eventsourcing.infrastructure.django.models.SnapshotRecord`

**timestamp\_sequenced\_record\_class**  
alias of `eventsourcing.infrastructure.django.models.TimestampSequencedRecord`

```

class eventsourcing.infrastructure.django.manager.DjangoRecordManager(*args,
                                                                    **kwargs)
    Bases: eventsourcing.infrastructure.base.SQLRecordManager

    _prepare_insert(tmpl, record_class, field_names, placeholder_for_id=False)
        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_sequence_ids()
        Returns all sequence IDs.

    delete_record(record)
        Permanently removes record from table.

    get_max_record_id()
        Return maximum notification ID in pipeline.

    get_max_tracking_record_id(upstream_application_name)
        Return maximum tracking record ID for notification from upstream application in pipeline.

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

    get_record(sequence_id, position)
        Gets record at position in sequence.

    get_record_table_name(record_class)
        Returns table name from SQLAlchemy record class.

    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.

    has_tracking_record(upstream_application_name, pipeline_id, notification_id)
        True if tracking record exists for notification from upstream in pipeline.

    tracking_record_class
        alias of eventsourcing.infrastructure.django.models.NotificationTrackingRecord

    write_records(records, tracking_kwargs=None)
        Writes tracking, event and notification records for a process event.

class eventsourcing.infrastructure.django.models.EntitySnapshotRecord(uid,
                                                                    appli-
                                                                    ca-
                                                                    tion_name,
                                                                    orig-
                                                                    ina-
                                                                    tor_id,
                                                                    orig-
                                                                    ina-
                                                                    tor_version,
                                                                    topic,
                                                                    state)
    Bases: django.db.models.base.Model

    exception DoesNotExist
        Bases: django.core.exceptions.ObjectDoesNotExist

```

**exception MultipleObjectsReturned**

Bases: `django.core.exceptions.MultipleObjectsReturned`

**application\_name**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**objects** = `<django.db.models.manager.Manager object>`

**originator\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**originator\_version**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**state**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**topic**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**uid**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

```
class eventsourcing.infrastructure.django.models.IntegerSequencedRecord (id,
                                                                    se-
                                                                    quence_id,
                                                                    po-
                                                                    si-
                                                                    tion,
                                                                    topic,
                                                                    state)
```

Bases: `django.db.models.base.Model`

**exception DoesNotExist**

Bases: `django.core.exceptions.ObjectDoesNotExist`

**exception MultipleObjectsReturned**

Bases: `django.core.exceptions.MultipleObjectsReturned`

**id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**objects** = `<django.db.models.manager.Manager object>`

**position**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**sequence\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**state**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**topic**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

```
class eventsourcing.infrastructure.django.models.NotificationTrackingRecord(uid,
                                                                    ap-
                                                                    pli-
                                                                    ca-
                                                                    tion_name,
                                                                    up-
                                                                    stream_application_n
                                                                    pipeline_id,
                                                                    no-
                                                                    ti-
                                                                    fi-
                                                                    ca-
                                                                    tion_id)
```

Bases: django.db.models.base.Model

**exception DoesNotExist**

Bases: django.core.exceptions.ObjectDoesNotExist

**exception MultipleObjectsReturned**

Bases: django.core.exceptions.MultipleObjectsReturned

**application\_name**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**notification\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**objects** = <django.db.models.manager.Manager object>

**pipeline\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**uid**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**upstream\_application\_name**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

```
class eventsourcing.infrastructure.django.models.SnapshotRecord(uid,          se-
                                                                    quence_id,
                                                                    position, topic,
                                                                    state)
```

Bases: django.db.models.base.Model

**exception DoesNotExist**

Bases: django.core.exceptions.ObjectDoesNotExist

**exception MultipleObjectsReturned**

Bases: django.core.exceptions.MultipleObjectsReturned

**objects** = <django.db.models.manager.Manager object>

**position**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**sequence\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**state**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**topic**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**uid**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

```
class eventsourcing.infrastructure.django.models.StoredEventRecord(uid,
                                                                    applica-
                                                                    tion_name,
                                                                    origina-
                                                                    tor_id,
                                                                    origina-
                                                                    tor_version,
                                                                    pipeline_id,
                                                                    notifica-
                                                                    tion_id,
                                                                    topic,
                                                                    state,
                                                                    causal_dependencies)
```

Bases: django.db.models.base.Model

**exception DoesNotExist**

Bases: django.core.exceptions.ObjectDoesNotExist

**exception MultipleObjectsReturned**

Bases: django.core.exceptions.MultipleObjectsReturned

**application\_name**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**causal\_dependencies**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**notification\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**objects** = <django.db.models.manager.Manager object>

**originator\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**originator\_version**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is



executed.

**pipeline\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**state**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**topic**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**uid**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

```
class eventsourcing.infrastructure.django.models.TimestampSequencedRecord(id,
                                                                    se-
                                                                    quence_id,
                                                                    po-
                                                                    si-
                                                                    tion,
                                                                    topic,
                                                                    state)
```

Bases: `django.db.models.base.Model`

**exception DoesNotExist**

Bases: `django.core.exceptions.ObjectDoesNotExist`

**exception MultipleObjectsReturned**

Bases: `django.core.exceptions.MultipleObjectsReturned`

**id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**objects** = `<django.db.models.manager.Manager object>`

**position**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**sequence\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**state**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**topic**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`eventsourcing.infrastructure.django.utils.close_django_connection()`

`eventsourcing.infrastructure.django.utils.setup_django()`

## 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)

    project_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=None,
                                                    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_and_project_events (entity_id, gt=None, gte=None, lt=None, lte=None, limit=None, ini-
                           tial_state=None, query_descending=False)
        Reconstitutes requested domain entity from domain events found in event store.

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

    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: abc.ABC

    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.
```

**get\_domain\_event** (*originator\_id, position*)

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.

**store** (*domain\_event\_or\_events*)

Put domain event in event store for later retrieval.

**class** eventsourcing.infrastructure.eventstore.**EventStore** (*record\_manager, sequenced\_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** ()

Yields all domain events in the event store.

**get\_domain\_event** (*originator\_id, position*)

Gets a domain event from the sequence identified by *originator\_id* at position *eq*.

#### Parameters

- **originator\_id** – ID of a sequence of events
- **position** – 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

**item\_from\_event** (*domain\_event\_or\_events*)

Maps domain event to sequenced item namedtuple.

**Parameters** **domain\_event\_or\_events** – application-level object (or list)

**Returns** namedtuple: sequence item namedtuple (or list)

**iterator\_class**

alias of `event_sourcing.infrastructure.iterators.SequencedItemIterator`

**store** (*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

## integersequencegenerators

Different ways of generating sequences of integers.

**class** `event_sourcing.infrastructure.integersequencegenerators.base.AbstractIntegerSequenceGenerator`

Bases: object

**\_\_next\_\_** ()

Returns the next item in the container.

**class** `event_sourcing.infrastructure.integersequencegenerators.base.SimpleIntegerSequenceGenerator`

Bases: `event_sourcing.infrastructure.integersequencegenerators.base.AbstractIntegerSequenceGenerator`

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

Bases: `event_sourcing.infrastructure.integersequencegenerators.base.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: `abc.ABC`

**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,
                                                                    sequence_id,
                                                                    gt=None,
                                                                    gte=None,
                                                                    lt=None,
                                                                    lte=None,
                                                                    page_size=None,
                                                                    is_ascending=True,
                                                                    *args,
                                                                    **kwargs)
```

Bases: `threading.Thread`

**run()**  
Method representing the thread’s activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

```
class eventsourcing.infrastructure.iterators.SequencedItemIterator (record_manager,
                                                                    sequence_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.event sourced repository.EventSourcedRepository*

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

Bases: *eventsourcing.domain.model.array.AbstractBigArrayRepository*,  
*eventsourcing.infrastructure.event sourced repository.EventSourcedRepository*

### subrepo

Sub-sequence repository.

### subrepo\_class

alias of *ArrayRepository*

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

Bases: *eventsourcing.infrastructure.event sourced repository.EventSourcedRepository*,  
*eventsourcing.domain.model.collection.AbstractCollectionRepository*

Event sourced repository for the Collection domain model entity.

```
class eventsourcing.infrastructure.repositories.timebucketedlog_repo.TimebucketedlogRepo (ev
```

Bases: `eventsourcing.infrastructure.event sourced repository. 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,
                                                                state)
```

Bases: tuple

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

**static** `__new__(_cls, sequence_id, position, topic, state)`  
Create new instance of SequencedItem(sequence\_id, position, topic, state)

`__repr__()`  
Return a nicely formatted representation string

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

**classmethod** `__make(iterable)`  
Make a new SequencedItem object from a sequence or iterable

`__replace(**kws)`  
Return a new SequencedItem object replacing specified fields with new values

**position**  
Alias for field number 1

**sequence\_id**  
Alias for field number 0

**state**  
Alias for field number 3

**topic**  
Alias for field number 2

```
class eventsourcing.infrastructure.sequenceditem.SequencedItemFieldNames (sequenced_item_class)
```

Bases: object

**other\_names**

**position**

**sequence\_id**

**state**

**topic**

**class** `event sourcing.infrastructure.sequenceditem.StoredEvent` (*originator\_id*,  
*originator\_version*,  
*topic*, *state*)

Bases: `tuple`

**\_\_getnewargs\_\_** ()

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

**static** **\_\_new\_\_** (*\_cls*, *originator\_id*, *originator\_version*, *topic*, *state*)

Create new instance of `StoredEvent`(*originator\_id*, *originator\_version*, *topic*, *state*)

**\_\_repr\_\_** ()

Return a nicely formatted representation string

**\_asdict** ()

Return a new `OrderedDict` which maps field names to their values.

**classmethod** **\_\_make** (*iterable*)

Make a new `StoredEvent` object from a sequence or iterable

**\_\_replace** (\*\**kws*)

Return a new `StoredEvent` object replacing specified fields with new values

**originator\_id**

Alias for field number 0

**originator\_version**

Alias for field number 1

**state**

Alias for field number 3

**topic**

Alias for field number 2

## sequenceditemmapper

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

**class** `event sourcing.infrastructure.sequenceditemmapper.AbstractSequencedItemMapper`

Bases: `abc.ABC`

**event\_from\_item** (*sequenced\_item*)

Constructs and returns a domain event for given sequenced item.

**item\_from\_event** (*domain\_event*)

Constructs and returns a sequenced item for given domain event.



```
class eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper (sequenced_item_class=  
    'eventsourc-  
    ing.infrastructure.sequenceditemmapper.SequencedItemMapper',  
    sequence_id_attr_name='sequence_id',  
    position_attr_name='position',  
    projection_attr_name=None,  
    json_encoder_class=None,  
    json_decoder_class=None,  
    ci_encoder=None,  
    ci_decoder=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)

**event\_from\_item** (sequenced\_item)  
Reconstructs domain event from stored event topic and event attrs. Used in the event store when getting domain events.

**event\_from\_topic\_and\_state** (topic, state)

**get\_event\_class\_and\_attrs** (topic, state)

**get\_item\_topic\_and\_state** (domain\_event\_class, event\_attrs)

**item\_from\_event** (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 eventsourcing.infrastructure.snapshotting.AbstractSnapshotStrategy
```

Bases: `abc.ABC`

```
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 eventsourcing.infrastructure.snapshotting.EventSourcedSnapshotStrategy (snapshot_store)
```

Bases: `eventsourcing.infrastructure.snapshotting.AbstractSnapshotStrategy`

Snapshot strategy that uses an event sourced snapshot.

**get\_snapshot** (*entity\_id*, *lt=None*, *lte=None*)

Gets the last snapshot for entity, optionally until a particular version number.

**Return type** *Snapshot*

**take\_snapshot** (*entity\_id*, *entity*, *last\_event\_version*)

Creates a Snapshot from the given state, and appends it to the snapshot store.

**Return type** *Snapshot*

`eventsourcing.infrastructure.snapshotting.entity_from_snapshot` (*snapshot*)

Reconstructs domain entity from given snapshot.

## sqlalchemy

Classes for event sourcing with SQLAlchemy.

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

Bases: `eventsourcing.infrastructure.datastore.Datastore`

**close\_connection** ()

Drops connection to a datastore.

**drop\_table** (*table*)

**drop\_tables** ()

Drops tables used to store events.

**is\_sqlite** ()

**session**

**setup\_connection** ()

Sets up a connection to a datastore.

**setup\_table** (*table*)

**setup\_tables** (*tables=None*)

Sets up tables used to store events.

**truncate\_tables** ()

Truncates tables used to store events.

```
class eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemySettings (uri=None,
                                                                    pool_size=None)
```

Bases: `eventsourcing.infrastructure.datastore.DatastoreSettings`

```
class eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory (session=None,
                                                                    uri=None,
                                                                    pool_size=None,
                                                                    *args,
                                                                    **kwargs)
```

Bases: `eventsourcing.infrastructure.factory.InfrastructureFactory`

```

construct_datastore()

construct_record_manager(**kwargs)

integer_sequenced_record_class
    alias of eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord

record_manager_class
    alias of eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager

snapshot_record_class
    alias of eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord

timestamp_sequenced_record_class
    alias of eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedNoIDRecord

eventsourcing.infrastructure.sqlalchemy.factory.construct_sqlalchemy_eventstore(session,
se-
quenced_item_c
se-
quence_id_attr
po-
si-
tion_attr_name
json_encoder_c
json_decoder_c
ci-
pher=None,
record_class=N
con-
tigu-
ous_record_ids
ap-
pli-
ca-
tion_name=Non
pipeline_id=0)

class eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager(session,
*args,
**kwargs)

Bases: eventsourcing.infrastructure.base.SQLRecordManager

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

all_sequence_ids()
    Returns all sequence IDs.

clone(**kwargs)

delete_record(record)
    Permanently removes record from table.

filter_by(**kwargs)

```

**get\_max\_record\_id()**

Return maximum notification ID in pipeline.

**get\_max\_tracking\_record\_id**(*upstream\_application\_name*)

Return maximum tracking record ID for notification from upstream application in pipeline.

**get\_notifications**(*start=None, stop=None, \*args, \*\*kwargs*)

Returns records sequenced by notification ID, from application, for pipeline, in given range.

Args 'start' and 'stop' are positions in a zero-based integer sequence.

**get\_record**(*sequence\_id, position*)

Gets record at position in sequence.

**get\_record\_table\_name**(*record\_class*)

Returns table name - used in raw queries.

**Return type** str

**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.

**has\_tracking\_record**(*upstream\_application\_name, pipeline\_id, notification\_id*)

True if tracking record exists for notification from upstream in pipeline.

**orm\_query**()

**tracking\_record\_class**

alias of `eventsourcing.infrastructure.sqlalchemy.records.NotificationTrackingRecord`

**write\_records**(*records, tracking\_kwargs=None*)

Writes tracking, event and notification records for a process event.

**class** eventsourcing.infrastructure.sqlalchemy.records.**EntitySnapshotRecord**(*\*\*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.

**application\_name**

**originator\_id**

**originator\_version**

**state**

**topic**

**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.

**position**

**sequence\_id**

**state**

**topic**

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

alias of *eventsourcing.infrastructure.sqlalchemy.records.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.

**id**

**position**

**sequence\_id**

**state**

**topic**

**class** eventsourcing.infrastructure.sqlalchemy.records.**NotificationTrackingRecord** (\*\*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.

**application\_name**

**notification\_id**

**pipeline\_id**

**upstream\_application\_name**

**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.

**position**

**sequence\_id**

**state**

**topic**

**class** `event sourcing.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.

**application\_name**

**causal\_dependencies**

**notification\_id**

**originator\_id**

**originator\_version**

**pipeline\_id**

**state**

**topic**

**class** `event sourcing.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.

**position**

**sequence\_id**

**state**

**topic**

`event sourcing.infrastructure.sqlalchemy.records.TimestampSequencedRecord`  
 alias of `event sourcing.infrastructure.sqlalchemy.records.TimestampSequencedNoIDRecord`

**class** `event sourcing.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.

**id**

**position**

**sequence\_id**

**state**

**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*

## 19.1.4 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: abc.ABC

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)
    Returns items for section.
```

**Return type** list

```
get_next_position ()
    Returns items for section.
```

**Return type** int

```
class eventsourcing.interface.notificationlog.LocalNotificationLog (section_size=None)
```

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** `event sourcing.interface.notificationlog.NotificationLogReader` (*notification\_log*,  
*use\_direct\_query\_if\_available=F*)

Bases: `abc.ABC`

**read** (*advance\_by=None*)

**read\_items** (*stop\_index=None*, *advance\_by=None*)

**read\_list** (*advance\_by=None*)

**seek** (*position*)

**class** `event sourcing.interface.notificationlog.NotificationLogView` (*notification\_log*,  
*json\_encoder\_class=None*)

Bases: `object`

**present\_section** (*section\_id*)

**class** `event sourcing.interface.notificationlog.RecordManagerNotificationLog` (*record\_manager*,  
*sec-*  
*tion\_size*)

Bases: `event sourcing.interface.notificationlog.LocalNotificationLog`

**get\_items** (*start*, *stop*, *next\_position=None*)

Returns items for section.

**Return type** list

**get\_next\_position** ()

Next unoccupied position in zero-based sequence.

Since the notification IDs are one-based, the next position is the current max notification ID. If there are no records, the max notification ID will be None, and the next position is zero.

**class** `event sourcing.interface.notificationlog.RemoteNotificationLog` (*base\_url*,  
*json\_decoder\_class=None*)

Bases: `event sourcing.interface.notificationlog.AbstractNotificationLog`

**deserialize\_section** (*section\_json*)

**get\_json** (*section\_id*)

**get\_resource** (*doc\_url*)

**make\_notification\_log\_url** (*section\_id*)

**class** `event sourcing.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.



### 19.1.5 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 naive UTC datetime from decimal UNIX timestamps such as `time.time()`.

**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

`event sourcing.utils.topic.get_topic(domain_class)`

Returns a string describing a class.

**Args:** domain\_class: A class.

**Returns:** A string describing the class.

`event sourcing.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.

`event sourcing.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** `event sourcing.utils.transcoding.ObjectJSONDecoder(object_hook=None, **kwargs)`

Bases: `json.decoder.JSONDecoder`

**classmethod** `from_jsonable(d)`

**class** `event sourcing.utils.transcoding.ObjectJSONEncoder(sort_keys=True, *args, **kwargs)`

Bases: `json.encoder.JSONEncoder`

**default** (*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

**iterencode** (*o*, *\_one\_shot=False*)

Encode the given object and yield each string representation as available.

For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

eventsourcing.utils.transcoding.**json\_dumps** (*obj*, *cls=None*)

eventsourcing.utils.transcoding.**json\_loads** (*s*, *cls=None*)

## 19.1.6 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.**CausalDependencyFailed**  
 Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when a causal dependency fails (after its tracking record not found).

**exception** eventsourcing.exceptions.**ConcurrencyError**  
 Bases: `eventsourcing.exceptions.RecordConflictError`

Raised when a record conflict is due to concurrency.

**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 is damaged (hash doesn't match data)

**exception** eventsourcing.exceptions.**DatasourceSettingsError**  
 Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when an error is detected in settings for a datasource.

**exception** eventsourcing.exceptions.**EntityIsDiscarded**  
 Bases: `AssertionError`

Raised when access to a recently discarded entity object is attempted.

**exception** eventsourcing.exceptions.**EntityVersionNotFound**  
 Bases: `eventsourcing.exceptions.EventSourcingError`

Raise when accessing an entity version that does not exist.

**exception** eventsourcing.exceptions.**EventHashError**  
 Bases: `eventsourcing.exceptions.DataIntegrityError`

Raised when an event's seal hash doesn't match the hash of the state of the event.

**exception** eventsourcing.exceptions.**EventRecordNotFound**  
 Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when an event record is not found.

**exception** eventsourcing.exceptions.**EventSourcingError**  
 Bases: `Exception`

Base eventsourcing exception.

**exception** `eventsourcing.exceptions.HeadHashError`

Bases: `eventsourcing.exceptions.DataIntegrityError`, `eventsourcing.exceptions.MismatchedOriginatorError`

Raised when applying an event with hash different from aggregate head.

**exception** `eventsourcing.exceptions.MismatchedOriginatorError`

Bases: `eventsourcing.exceptions.ConsistencyError`

Raised when applying an event to an inappropriate object.

**exception** `eventsourcing.exceptions.MutatorRequiresTypeNotInstance`

Bases: `eventsourcing.exceptions.ConsistencyError`

Raised when mutator function received a class rather than an entity.

**exception** `eventsourcing.exceptions.OperationalError`

Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when an operational error is encountered.

**exception** `eventsourcing.exceptions.OriginatorIDError`

Bases: `eventsourcing.exceptions.MismatchedOriginatorError`

Raised when applying an event to the wrong entity or aggregate.

**exception** `eventsourcing.exceptions.OriginatorVersionError`

Bases: `eventsourcing.exceptions.MismatchedOriginatorError`

Raised when applying an event to the wrong version of an entity or aggregate.

**exception** `eventsourcing.exceptions.ProgrammingError`

Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when programming errors are encountered.

**exception** `eventsourcing.exceptions.PromptFailed`

Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when prompt fails.

**exception** `eventsourcing.exceptions.RecordConflictError`

Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when database raises an integrity error.

**exception** `eventsourcing.exceptions.RepositoryKeyError`

Bases: `KeyError`, `eventsourcing.exceptions.EventSourcingError`

Raised when using entity repository's dictionary like interface to get an entity that does not exist.

**exception** `eventsourcing.exceptions.TimeSequenceError`

Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when a time sequence error occurs e.g. trying to save a timestamp that already exists.

**exception** `eventsourcing.exceptions.TopicResolutionError`

Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when unable to resolve a topic to a Python class.

**exception** `eventsourcing.exceptions.TrackingRecordNotFound`

Bases: `eventsourcing.exceptions.EventSourcingError`

Raised when a tracking record is not found.

## 19.1.7 example

A simple, unit-tested, event sourced application.

### application

```
class eventsourcing.example.application.ApplicationWithEventStores (entity_record_manager=None,
                                                                    log_record_manager=None,
                                                                    snap-
                                                                    shot_record_manager=None,
                                                                    ci-
                                                                    pher=None,
                                                                    se-
                                                                    quenced_item_mapper_class=<class
                                                                    'eventsourc-
                                                                    ing.infrastructure.sequenceditemm
```

Bases: `abc.ABC`

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` (sequenced\_item\_mapper\_class, event\_sequence\_id\_attr, event\_position\_attr, record\_manager, cipher=None)

`construct_sequenced_item_mapper` (sequenced\_item\_mapper\_class, sequenced\_item\_class, event\_sequence\_id\_attr, event\_position\_attr, cipher=None, json\_encoder\_class=<class 'eventsourcing.utils.transcoding.ObjectJSONEncoder'>, json\_decoder\_class=<class 'eventsourcing.utils.transcoding.ObjectJSONDecoder'>)

```
class eventsourcing.example.application.ApplicationWithPersistencePolicies (**kwargs)
    Bases: eventsourcing.example.application.ApplicationWithEventStores
```

`close()`

`construct_entity_persistence_policy()`

`construct_log_persistence_policy()`

`construct_snapshot_persistence_policy()`

```
class eventsourcing.example.application.ExampleApplication (**kwargs)
    Bases: eventsourcing.example.application.ApplicationWithPersistencePolicies
```

Example event sourced application with entity factory and repository.

`create_new_example` (foo=", a=", b=")  
Entity object factory.

```
eventsourcing.example.application.close_example_application()
    Shuts down single global instance of application.
```

To be called when tearing down, perhaps between tests, in order to allow a subsequent call to `init_example_application()`.

```
eventsourcing.example.application.construct_example_application (**kwargs)
    Application object factory.
```

`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.EntityWithHashchain`, `event sourcing.domain.model.entity.TimestampedVersionedEntity`

An example event sourced domain model entity.

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

Bases: `event sourcing.example.domainmodel.Event`, `event sourcing.domain.model.entity.AttributeChanged`

Published when an Example is created.

**class** `Created(**kwargs)`

Bases: `event sourcing.example.domainmodel.Event`, `event sourcing.domain.model.entity.Created`, `event sourcing.domain.model.entity.Created`

Published when an Example is created.

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

Bases: `event sourcing.example.domainmodel.Event`, `event sourcing.domain.model.entity.Discarded`

Published when an Example is discarded.

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

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

Supertype for events of example entities.

**class** `Heartbeat(**kwargs)`

Bases: `event sourcing.example.domainmodel.Event`, `event sourcing.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\_sourced\_repository.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.

**id**

**position**

**sequence\_id**

**state**

**topic**

```
eventsourcing.example.interface.flaskapp.hello()
```

```
eventsourcing.example.interface.flaskapp.init_example_application_with_sqlalchemy()
```





### e

eventsourcing.application.actors, 143  
eventsourcing.application.command, 145  
eventsourcing.application.django, 146  
eventsourcing.application.multiprocess, 146  
eventsourcing.application.pipeline, 147  
eventsourcing.application.policies, 147  
eventsourcing.application.popo, 148  
eventsourcing.application.process, 148  
eventsourcing.application.simple, 150  
eventsourcing.application.snapshotting, 152  
eventsourcing.application.sqlalchemy, 152  
eventsourcing.application.system, 153  
eventsourcing.domain.model.aggregate, 156  
eventsourcing.domain.model.array, 159  
eventsourcing.domain.model.collection, 161  
eventsourcing.domain.model.command, 162  
eventsourcing.domain.model.decorators, 163  
eventsourcing.domain.model.entity, 164  
eventsourcing.domain.model.events, 170  
eventsourcing.domain.model.snapshot, 173  
eventsourcing.domain.model.timebucketedlog, 174  
eventsourcing.example.application, 207  
eventsourcing.example.domainmodel, 208  
eventsourcing.example.infrastructure, 209  
eventsourcing.example.interface.flaskapp, 209  
eventsourcing.exceptions, 205  
eventsourcing.infrastructure.base, 175  
eventsourcing.infrastructure.cassandra.datastore, 178  
eventsourcing.infrastructure.cassandra.factory, 179  
eventsourcing.infrastructure.cassandra.manager, 179  
eventsourcing.infrastructure.cassandra.records, 180  
eventsourcing.infrastructure.datastore, 181  
eventsourcing.infrastructure.django.factory, 182  
eventsourcing.infrastructure.django.manager, 182  
eventsourcing.infrastructure.django.models, 183  
eventsourcing.infrastructure.django.utils, 187  
eventsourcing.infrastructure.eventplayer, 188  
eventsourcing.infrastructure.eventsourcedrepository, 188  
eventsourcing.infrastructure.eventstore, 188  
eventsourcing.infrastructure.integersequencegenerator, 190  
eventsourcing.infrastructure.integersequencegenerator, 190  
eventsourcing.infrastructure.iterators, 190  
eventsourcing.infrastructure.repositories.array, 192  
eventsourcing.infrastructure.repositories.collection, 192  
eventsourcing.infrastructure.repositories.timebucketedlog, 192  
eventsourcing.infrastructure.sequenceditem, 193  
eventsourcing.infrastructure.sequenceditemmapper, 194  
eventsourcing.infrastructure.snapshotting, 195

eventsourcing.infrastructure.sqlalchemy.datastore,  
    [196](#)  
eventsourcing.infrastructure.sqlalchemy.factory,  
    [196](#)  
eventsourcing.infrastructure.sqlalchemy.manager,  
    [197](#)  
eventsourcing.infrastructure.sqlalchemy.records,  
    [198](#)  
eventsourcing.infrastructure.timebucketedlog\_reader,  
    [201](#)  
eventsourcing.interface.notificationlog,  
    [201](#)  
eventsourcing.utils.cipher.aes, [203](#)  
eventsourcing.utils.times, [203](#)  
eventsourcing.utils.topic, [204](#)  
eventsourcing.utils.transcoding, [204](#)

## Symbols

<code>__abstractmethods__</code>	( <i>eventsourcing.domain.model.snapshot.AbstractSnapshot attribute</i> ), 173	<code>__created_on__</code>	( <i>eventsourcing.domain.model.entity.TimestampedEntity attribute</i> ), 168
<code>__abstractmethods__</code>	( <i>eventsourcing.domain.model.snapshot.Snapshot attribute</i> ), 174	<code>__created_on__</code>	( <i>eventsourcing.domain.model.entity.TimeuuidedEntity attribute</i> ), 169
<code>__assert_not_discarded__()</code>	( <i>eventsourcing.domain.model.entity.DomainEntity method</i> ), 166	<code>__dict__</code>	( <i>eventsourcing.domain.model.array.Array attribute</i> ), 159
<code>__batch_pending_events__()</code>	( <i>eventsourcing.domain.model.aggregate.BaseAggregateRoot method</i> ), 158	<code>__dict__</code>	( <i>eventsourcing.domain.model.entity.AbstractEventPlayer attribute</i> ), 165
<code>__change_attribute__()</code>	( <i>eventsourcing.domain.model.entity.DomainEntity method</i> ), 166	<code>__dict__</code>	( <i>eventsourcing.domain.model.entity.DomainEntity attribute</i> ), 166
<code>__check_hash__()</code>	( <i>eventsourcing.domain.model.events.EventWithHash method</i> ), 172	<code>__dict__</code>	( <i>eventsourcing.domain.model.events.DomainEvent attribute</i> ), 171
<code>__check_obj__()</code>	( <i>eventsourcing.domain.model.entity.DomainEntity.Event method</i> ), 166	<code>__dict__</code>	( <i>eventsourcing.domain.model.snapshot.AbstractSnapshot attribute</i> ), 173
<code>__check_obj__()</code>	( <i>eventsourcing.domain.model.entity.EntityWithHashchain.Event method</i> ), 167	<code>__discard__()</code>	( <i>eventsourcing.domain.model.entity.DomainEntity method</i> ), 166
<code>__check_obj__()</code>	( <i>eventsourcing.domain.model.entity.VersionedEntity.Event method</i> ), 170	<code>__entity_kwargs__</code>	( <i>eventsourcing.domain.model.entity.DomainEntity.Created attribute</i> ), 165
<code>__contains__()</code>	( <i>eventsourcing.domain.model.entity.AbstractEntityRepository method</i> ), 164	<code>__entity_kwargs__</code>	( <i>eventsourcing.domain.model.entity.EntityWithHashchain.Created attribute</i> ), 167
<code>__contains__()</code>	( <i>eventsourcing.infrastructure.eventsourcedepository.EventSourcedRepository method</i> ), 188	<code>__entity_kwargs__</code>	( <i>eventsourcing.domain.model.entity.TimestampedEntity.Created attribute</i> ), 168
<code>__create__()</code>	( <i>eventsourcing.domain.model.entity.DomainEntity class method</i> ), 166	<code>__entity_kwargs__</code>	( <i>eventsourcing.domain.model.entity.VersionedEntity.Created attribute</i> ), 169
<code>__create__()</code>	( <i>eventsourcing.domain.model.entity.EntityWithHashchain class method</i> ), 167	<code>__eq__()</code>	( <i>eventsourcing.domain.model.array.Array method</i> ), 159
		<code>__eq__()</code>	( <i>eventsourcing.domain.model.entity.DomainEntity class method</i> ), 167

<code>__eq__()</code> (eventsourcing.domain.model.events.DomainEvent method), 171	<code>__init__()</code> (eventsourcing.domain.model.aggregate.BaseAggregateRoot method), 158
<code>__event_hash__</code> (eventsourcing.domain.model.events.EventWithHash attribute), 172	<code>__init__()</code> (eventsourcing.domain.model.array.AbstractArrayRepository method), 159
<code>__genesis_hash__</code> (eventsourcing.domain.model.entity.EntityWithHashchain attribute), 167	<code>__init__()</code> (eventsourcing.domain.model.array.Array method), 159
<code>__getitem__()</code> (eventsourcing.domain.model.array.AbstractArrayRepository method), 159	<code>__init__()</code> (eventsourcing.domain.model.array.BigArray method), 160
<code>__getitem__()</code> (eventsourcing.domain.model.array.AbstractBigArrayRepository method), 159	<code>__init__()</code> (eventsourcing.domain.model.array.ItemAssigned method), 161
<code>__getitem__()</code> (eventsourcing.domain.model.array.Array method), 159	<code>__init__()</code> (eventsourcing.domain.model.collection.Collection method), 162
<code>__getitem__()</code> (eventsourcing.domain.model.array.BigArray method), 160	<code>__init__()</code> (eventsourcing.domain.model.command.Command method), 163
<code>__getitem__()</code> (eventsourcing.domain.model.entity.AbstractEntityRepository method), 164	<code>__init__()</code> (eventsourcing.domain.model.entity.DomainEntity method), 166
<code>__getitem__()</code> (eventsourcing.domain.model.entity.EntityWithHashchain method), 167	<code>__init__()</code> (eventsourcing.domain.model.entity.DomainEntity.Created method), 165
<code>__getitem__()</code> (eventsourcing.domain.model.entity.EntityWithHashchain method), 167	<code>__init__()</code> (eventsourcing.domain.model.entity.EntityWithHashchain method), 167
<code>__getitem__()</code> (eventsourcing.interface.notificationlog.AbstractNotificationLog method), 201	<code>__init__()</code> (eventsourcing.domain.model.entity.TimestampedEntity method), 168
<code>__getnewargs__()</code> (eventsourcing.infrastructure.sequenceditem.SequencedItem method), 193	<code>__init__()</code> (eventsourcing.domain.model.entity.TimeuuidedEntity method), 169
<code>__getnewargs__()</code> (eventsourcing.infrastructure.sequenceditem.StoredEvent method), 194	<code>__init__()</code> (eventsourcing.domain.model.entity.VersionedEntity method), 170
<code>__hash__</code> (eventsourcing.domain.model.array.Array attribute), 159	<code>__init__()</code> (eventsourcing.domain.model.entity.VersionedEntity.Created method), 169
<code>__hash__</code> (eventsourcing.domain.model.entity.DomainEntity attribute), 166	<code>__init__()</code> (eventsourcing.domain.model.events.DomainEvent method), 171
<code>__hash__()</code> (eventsourcing.domain.model.events.DomainEvent method), 171	<code>__init__()</code> (eventsourcing.domain.model.events.EventWithHash method), 172
<code>__hash__()</code> (eventsourcing.domain.model.events.EventWithHash method), 172	<code>__init__()</code> (eventsourcing.domain.model.events.EventWithOriginatorID method), 172
<code>__hash_object__()</code> (eventsourcing.domain.model.events.DomainEvent class method), 171	<code>__init__()</code> (eventsourcing.domain.model.events.EventWithOriginatorVersion method), 172
<code>__init__()</code> (eventsourcing.application.system.System method), 156	<code>__init__()</code> (eventsourcing.domain.model.events.EventWithTimestamp method), 172
<code>__init__()</code> (eventsourcing.domain.model.aggregate.BaseAggregateRoot method), 158	

<a href="#">method</a> ), 173		<a href="#">method</a> ), 192	
<a href="#">__init__()</a> (eventsourcing.domain.model.events.EventWithTimeuuid <a href="#">method</a> ), 173		<a href="#">__json_encoder_class__</a> (eventsourcing.domain.model.events.DomainEvent <a href="#">attribute</a> ), 171	
<a href="#">__init__()</a> (eventsourcing.domain.model.snapshot.Snapshot <a href="#">method</a> ), 174		<a href="#">__last_modified__</a> (eventsourcing.domain.model.entity.TimestampedEntity <a href="#">attribute</a> ), 168	
<a href="#">__init__()</a> (eventsourcing.domain.model.timebucketedlog.MessageLogged <a href="#">method</a> ), 174		<a href="#">__last_modified__</a> (eventsourcing.domain.model.entity.TimeuuidedEntity <a href="#">attribute</a> ), 169	
<a href="#">__init__()</a> (eventsourcing.domain.model.timebucketedlog.Timebucketedlog <a href="#">method</a> ), 175		<a href="#">__len__()</a> (eventsourcing.domain.model.array.Array <a href="#">method</a> ), 159	
<a href="#">__init__()</a> (eventsourcing.example.interface.flaskapp.IntegerSequencedItem <a href="#">method</a> ), 209		<a href="#">__len__()</a> (eventsourcing.domain.model.array.BigArray <a href="#">method</a> ), 160	
<a href="#">__init__()</a> (eventsourcing.infrastructure.eventstore.EventStore <a href="#">method</a> ), 189		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRoot <a href="#">attribute</a> ), 157	
<a href="#">__init__()</a> (eventsourcing.infrastructure.sqlalchemy.records.EntitySnapshotRecord <a href="#">attribute</a> ), 198		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRoot.AttributeChanged <a href="#">attribute</a> ), 157	
<a href="#">__init__()</a> (eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedNotificationRecord <a href="#">method</a> ), 198		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRoot.Created <a href="#">attribute</a> ), 157	
<a href="#">__init__()</a> (eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithIdRecord <a href="#">method</a> ), 199		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRoot.Discarded <a href="#">attribute</a> ), 157	
<a href="#">__init__()</a> (eventsourcing.infrastructure.sqlalchemy.records.NotificationTrackingRecord <a href="#">method</a> ), 199		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRoot.Event <a href="#">attribute</a> ), 157	
<a href="#">__init__()</a> (eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord <a href="#">method</a> ), 199		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRootWithHashchainedEvent <a href="#">attribute</a> ), 158	
<a href="#">__init__()</a> (eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord <a href="#">method</a> ), 200		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRootWithHashchainedEvent <a href="#">attribute</a> ), 157	
<a href="#">__init__()</a> (eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedNotificationRecord <a href="#">method</a> ), 200		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRootWithHashchainedEvent <a href="#">attribute</a> ), 157	
<a href="#">__init__()</a> (eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedWithIdRecord <a href="#">method</a> ), 200		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRootWithHashchainedEvent <a href="#">attribute</a> ), 158	
<a href="#">__init__()</a> (eventsourcing.utils.cipher.aes.AESCipher <a href="#">method</a> ), 203		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.AggregateRootWithHashchainedEvent <a href="#">attribute</a> ), 158	
<a href="#">__iter__()</a> (eventsourcing.domain.model.collection.Collection <a href="#">method</a> ), 162		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.BaseAggregateRoot <a href="#">attribute</a> ), 158	
<a href="#">__iter__()</a> (eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator <a href="#">method</a> ), 191		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.BaseAggregateRoot.AttributeChanged <a href="#">attribute</a> ), 158	
<a href="#">__iter__()</a> (eventsourcing.infrastructure.iterators.SequencedItemIterator <a href="#">method</a> ), 191		<a href="#">__module__</a> (eventsourcing.domain.model.aggregate.BaseAggregateRoot.Created <a href="#">attribute</a> ), 158	

<code>__module__</code>	(eventsourcing.domain.model.aggregate.BaseAggregateRoot.Discarded attribute), 158	<code>__module__</code>	(eventsourcing.domain.model.command.Command.Discarded attribute), 162
<code>__module__</code>	(eventsourcing.domain.model.aggregate.BaseAggregateRoot.Event attribute), 158	<code>__module__</code>	(eventsourcing.domain.model.command.Command.Done attribute), 162
<code>__module__</code>	(eventsourcing.domain.model.array.AbstractArrayRepository attribute), 159	<code>__module__</code>	(eventsourcing.domain.model.command.Command.Event attribute), 163
<code>__module__</code>	(eventsourcing.domain.model.array.AbstractBigArrayRepository attribute), 159	<code>__module__</code>	(eventsourcing.domain.model.entity.AbstractEntityRepository attribute), 164
<code>__module__</code>	(eventsourcing.domain.model.array.Array attribute), 159	<code>__module__</code>	(eventsourcing.domain.model.entity.AbstractEventPlayer attribute), 165
<code>__module__</code>	(eventsourcing.domain.model.array.BigArray attribute), 160	<code>__module__</code>	(eventsourcing.domain.model.entity.DomainEntity attribute), 166
<code>__module__</code>	(eventsourcing.domain.model.array.ItemAssigned attribute), 161	<code>__module__</code>	(eventsourcing.domain.model.entity.DomainEntity.AttributeChanged attribute), 165
<code>__module__</code>	(eventsourcing.domain.model.collection.AbstractCollectionRepository attribute), 161	<code>__module__</code>	(eventsourcing.domain.model.entity.DomainEntity.Created attribute), 165
<code>__module__</code>	(eventsourcing.domain.model.collection.Collection attribute), 162	<code>__module__</code>	(eventsourcing.domain.model.entity.DomainEntity.Discarded attribute), 165
<code>__module__</code>	(eventsourcing.domain.model.collection.Collection.Created attribute), 161	<code>__module__</code>	(eventsourcing.domain.model.entity.DomainEntity.Event attribute), 166
<code>__module__</code>	(eventsourcing.domain.model.collection.Collection.Discarded attribute), 161	<code>__module__</code>	(eventsourcing.domain.model.entity.EntityWithHashchain attribute), 167
<code>__module__</code>	(eventsourcing.domain.model.collection.Collection.Event attribute), 161	<code>__module__</code>	(eventsourcing.domain.model.entity.EntityWithHashchain.Created attribute), 167
<code>__module__</code>	(eventsourcing.domain.model.collection.Collection.EventWithItem attribute), 162	<code>__module__</code>	(eventsourcing.domain.model.entity.EntityWithHashchain.Discarded attribute), 167
<code>__module__</code>	(eventsourcing.domain.model.collection.Collection.ItemAdded attribute), 162	<code>__module__</code>	(eventsourcing.domain.model.entity.EntityWithHashchain.Event attribute), 167
<code>__module__</code>	(eventsourcing.domain.model.collection.Collection.ItemRemoved attribute), 162	<code>__module__</code>	(eventsourcing.domain.model.entity.TimestampedEntity attribute), 168
<code>__module__</code>	(eventsourcing.domain.model.command.Command attribute), 163	<code>__module__</code>	(eventsourcing.domain.model.entity.TimestampedEntity.AttributeChanged attribute), 168
<code>__module__</code>	(eventsourcing.domain.model.command.Command.AttributeChanged attribute), 162	<code>__module__</code>	(eventsourcing.domain.model.entity.TimestampedEntity.Created attribute), 168
<code>__module__</code>	(eventsourcing.domain.model.command.Command.Created attribute), 162	<code>__module__</code>	(eventsourcing.domain.model.entity.TimestampedEntity.Discarded attribute), 168
<code>__module__</code>	(eventsourcing-	<code>__module__</code>	(eventsourcing-

<code>ing.domain.model.entity.TimestampedEntity.Event</code> <code>attribute</code> ), 168	<code>ing.domain.model.events.EventWithHash</code> <code>attribute</code> ), 172
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.TimestampedVersionedEntity</code> <code>attribute</code> ), 169	<code>__module__</code> (eventsourc- <code>ing.domain.model.events.EventWithOriginatorID</code> <code>attribute</code> ), 172
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.TimestampedVersionedEntity.AttributeChanged</code> <code>attribute</code> ), 168	<code>__module__</code> (eventsourc- <code>ing.domain.model.events.EventWithOriginatorVersion</code> <code>attribute</code> ), 173
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.TimestampedVersionedEntity.Created</code> <code>attribute</code> ), 169	<code>__module__</code> (eventsourc- <code>ing.domain.model.events.EventWithTimestamp</code> <code>attribute</code> ), 173
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.TimestampedVersionedEntity.Discarded</code> <code>attribute</code> ), 169	<code>__module__</code> (eventsourc- <code>ing.domain.model.events.EventWithTimeuuid</code> <code>attribute</code> ), 173
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.TimestampedVersionedEntity.Eventing</code> <code>attribute</code> ), 169	<code>__module__</code> (eventsourc- <code>ing.domain.model.events.Logged</code> <code>attribute</code> ), 173
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.TimeuuidedEntity</code> <code>attribute</code> ), 169	<code>__module__</code> (eventsourc- <code>ing.domain.model.snapshot.AbstractSnapshot</code> <code>attribute</code> ), 173
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.TimeuuidedVersionedEntity</code> <code>attribute</code> ), 169	<code>__module__</code> (eventsourc- <code>ing.domain.model.snapshot.Snapshot</code> <code>attribute</code> ), 174
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.VersionedEntity</code> <code>attribute</code> ), 170	<code>__module__</code> (eventsourc- <code>ing.domain.model.timebucketedlog.MessageLogged</code> <code>attribute</code> ), 174
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.VersionedEntity.AttributeChanged</code> <code>attribute</code> ), 169	<code>__module__</code> (eventsourc- <code>ing.domain.model.timebucketedlog.Timebucketedlog</code> <code>attribute</code> ), 175
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.VersionedEntity.Created</code> <code>attribute</code> ), 170	<code>__module__</code> (eventsourc- <code>ing.domain.model.timebucketedlog.Timebucketedlog.BucketSizeC</code> <code>attribute</code> ), 174
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.VersionedEntity.Discarded</code> <code>attribute</code> ), 170	<code>__module__</code> (eventsourc- <code>ing.domain.model.timebucketedlog.Timebucketedlog.Event</code> <code>attribute</code> ), 175
<code>__module__</code> (eventsourc- <code>ing.domain.model.entity.VersionedEntity.Event</code> <code>attribute</code> ), 170	<code>__module__</code> (eventsourc- <code>ing.domain.model.timebucketedlog.Timebucketedlog.Started</code> <code>attribute</code> ), 175
<code>__module__</code> (eventsourc- <code>ing.domain.model.events.AttributeChanged</code> <code>attribute</code> ), 170	<code>__module__</code> (eventsourc- <code>ing.domain.model.timebucketedlog.TimebucketedlogRepository</code> <code>attribute</code> ), 175
<code>__module__</code> (eventsourc- <code>ing.domain.model.events.Created</code> <code>attribute</code> ), 171	<code>__mutate__()</code> (eventsourc- <code>ing.domain.model.collection.Collection.ItemAdded</code> <code>method</code> ), 162
<code>__module__</code> (eventsourc- <code>ing.domain.model.events.Discarded</code> <code>attribute</code> ), 171	<code>__mutate__()</code> (eventsourc- <code>ing.domain.model.collection.Collection.ItemRemoved</code> <code>method</code> ), 162
<code>__module__</code> (eventsourc- <code>ing.domain.model.events.DomainEvent</code> <code>attribute</code> ), 171	<code>__mutate__()</code> (eventsourc- <code>ing.domain.model.entity.DomainEntity.AttributeChanged</code> <code>method</code> ), 165
<code>__module__</code> (eventsourc- <code>ing.domain.model.events.EventHandlersNotEmptyError</code> <code>attribute</code> ), 172	<code>__mutate__()</code> (eventsourc- <code>ing.domain.model.entity.DomainEntity.Created</code> <code>method</code> ), 165
<code>__module__</code> (eventsourc- <code>ing.domain.model.events.EventHandlersNotEmptyError</code> <code>attribute</code> ), 172	<code>__mutate__()</code> (eventsourc- <code>ing.domain.model.events.EventHandlersNotEmptyError</code> <code>attribute</code> ), 172



<code>ing.domain.model.entity.DomainEntity.Discarded</code>	<code>method), 166</code>	<code>__repr__()</code>	<code>(eventsourc-</code>
<code>method), 165</code>		<code>ing.domain.model.events.DomainEvent</code>	
<code>__mutate__()</code>	<code>(eventsourc-</code>	<code>method), 171</code>	
<code>ing.domain.model.entity.DomainEntity.Event</code>		<code>__repr__()</code>	<code>(eventsourc-</code>
<code>method), 166</code>		<code>ing.infrastructure.sequenceditem.SequencedItem</code>	
<code>__mutate__()</code>	<code>(eventsourc-</code>	<code>method), 193</code>	
<code>ing.domain.model.entity.EntityWithHashchain.Created</code>		<code>__repr__()</code>	<code>(eventsourc-</code>
<code>method), 167</code>		<code>ing.infrastructure.sequenceditem.StoredEvent</code>	
<code>__mutate__()</code>	<code>(eventsourc-</code>	<code>method), 194</code>	
<code>ing.domain.model.entity.EntityWithHashchain.Discarded</code>		<code>__save__()</code>	<code>(eventsourc-</code>
<code>method), 167</code>		<code>ing.domain.model.aggregate.BaseAggregateRoot</code>	
<code>__mutate__()</code>	<code>(eventsourc-</code>	<code>method), 158</code>	
<code>ing.domain.model.entity.EntityWithHashchain.Event</code>		<code>__setattr__()</code>	<code>(eventsourc-</code>
<code>method), 167</code>		<code>ing.domain.model.events.DomainEvent</code>	
<code>__mutate__()</code>	<code>(eventsourc-</code>	<code>method), 171</code>	
<code>ing.domain.model.entity.TimestampedEntity.Event</code>		<code>__setitem__()</code>	<code>(eventsourc-</code>
<code>method), 168</code>		<code>ing.domain.model.array.Array</code>	<code>method),</code>
<code>__mutate__()</code>	<code>(eventsourc-</code>	<code>159</code>	
<code>ing.domain.model.entity.VersionedEntity.Event</code>		<code>__setitem__()</code>	<code>(eventsourc-</code>
<code>method), 170</code>		<code>ing.domain.model.array.BigArray</code>	<code>method),</code>
<code>__mutate__()</code>	<code>(eventsourc-</code>	<code>160</code>	
<code>ing.domain.model.events.DomainEvent</code>		<code>__trigger_event__()</code>	<code>(eventsourc-</code>
<code>method), 171</code>		<code>ing.domain.model.entity.DomainEntity</code>	
<code>__mutate__()</code>	<code>(eventsourc-</code>	<code>method), 166</code>	
<code>ing.domain.model.events.EventWithHash</code>		<code>__trigger_event__()</code>	<code>(eventsourc-</code>
<code>method), 172</code>		<code>ing.domain.model.entity.EntityWithHashchain</code>	
<code>__ne__()</code>	<code>(eventsourcing.domain.model.array.Array</code>	<code>method), 168</code>	
<code>method), 159</code>		<code>__trigger_event__()</code>	<code>(eventsourc-</code>
<code>__ne__()</code>	<code>(eventsourc-</code>	<code>ing.domain.model.entity.VersionedEntity</code>	
<code>ing.domain.model.entity.DomainEntity</code>		<code>method), 170</code>	
<code>method), 166</code>		<code>__version__</code>	<code>(eventsourc-</code>
<code>__ne__()</code>	<code>(eventsourc-</code>	<code>ing.domain.model.entity.VersionedEntity</code>	
<code>ing.domain.model.events.DomainEvent</code>		<code>attribute), 170</code>	
<code>method), 171</code>		<code>__weakref__</code>	<code>(eventsourc-</code>
<code>__new__()</code>	<code>(eventsourc-</code>	<code>ing.domain.model.array.Array</code>	<code>attribute),</code>
<code>ing.infrastructure.sequenceditem.SequencedItem</code>		<code>159</code>	
<code>static method), 193</code>		<code>__weakref__</code>	<code>(eventsourc-</code>
<code>__new__()</code>	<code>(eventsourc-</code>	<code>ing.domain.model.entity.AbstractEventPlayer</code>	
<code>ing.infrastructure.sequenceditem.StoredEvent</code>		<code>attribute), 165</code>	
<code>static method), 194</code>		<code>__weakref__</code>	<code>(eventsourc-</code>
<code>__next__()</code>	<code>(eventsourc-</code>	<code>ing.domain.model.entity.DomainEntity</code>	
<code>ing.infrastructure.integersequencegenerators.base.AbstractIntegerSequenceGenerator</code>		<code>attribute), 166</code>	
<code>method), 190</code>		<code>__weakref__</code>	<code>(eventsourc-</code>
<code>__notifiable__</code>	<code>(eventsourc-</code>	<code>ing.domain.model.events.DomainEvent</code>	<code>at-</code>
<code>tribute), 171</code>		<code>tribute), 171</code>	
<code>__publish__()</code>	<code>(eventsourc-</code>	<code>__weakref__</code>	<code>(eventsourc-</code>
<code>ing.domain.model.aggregate.BaseAggregateRoot</code>		<code>ing.domain.model.events.EventHandlersNotEmptyError</code>	
<code>method), 158</code>		<code>attribute), 172</code>	
<code>__publish__()</code>	<code>(eventsourc-</code>	<code>__weakref__</code>	<code>(eventsourc-</code>
<code>ing.domain.model.entity.DomainEntity</code>		<code>ing.domain.model.snapshot.AbstractSnapshot</code>	
<code>method), 166</code>		<code>attribute), 173</code>	
<code>__publish_to_subscribers__()</code>	<code>(eventsourc-</code>	<code>__asdict__()</code>	<code>(eventsourc-</code>
<code>ing.domain.model.entity.DomainEntity</code>		<code>ing.infrastructure.sequenceditem.SequencedItem</code>	



method), 193

`_asdict()` (eventsourcing.infrastructure.sequenceditem.StoredEvent method), 194

`_inc_page_counter()` (eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator method), 191

`_inc_query_counter()` (eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator method), 191

`_make()` (eventsourcing.infrastructure.sequenceditem.SequencedItem class method), 193

`_make()` (eventsourcing.infrastructure.sequenceditem.StoredEvent class method), 194

`_prepare_insert()` (eventsourcing.infrastructure.base.SQLRecordManager method), 177

`_prepare_insert()` (eventsourcing.infrastructure.django.manager.DjangoRecordManager method), 183

`_prepare_insert()` (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 197

`_replace()` (eventsourcing.infrastructure.sequenceditem.SequencedItem method), 193

`_replace()` (eventsourcing.infrastructure.sequenceditem.StoredEvent method), 194

**A**

`a` (eventsourcing.example.domainmodel.Example attribute), 208

`abort()` (eventsourcing.application.system.BarrierControlledApplicationThread method), 153

`AbstractArrayRepository` (class in eventsourcing.domain.model.array), 159

`AbstractBigArrayRepository` (class in eventsourcing.domain.model.array), 159

`AbstractCollectionRepository` (class in eventsourcing.domain.model.collection), 161

`AbstractEntityRepository` (class in eventsourcing.domain.model.entity), 164

`AbstractEventPlayer` (class in eventsourcing.domain.model.entity), 165

`AbstractEventStore` (class in eventsourcing.infrastructure.eventstore), 188

`AbstractExampleRepository` (class in eventsourcing.example.domainmodel), 208

`AbstractIntegerSequenceGenerator` (class in eventsourcing.infrastructure.integersequencegenerators.base), 190

`AbstractNotificationLog` (class in eventsourcing.interface.notificationlog), 201

`AbstractSequencedItemIterator` (class in eventsourcing.infrastructure.iterators), 190

`AbstractSequencedItemMapper` (class in eventsourcing.infrastructure.sequenceditemmapper), 194

`AbstractSequencedItemRecordManager` (class in eventsourcing.infrastructure.base), 176

`AbstractSnapshot` (class in eventsourcing.domain.model.snapshot), 173

`AbstractSnapshotStrategy` (class in eventsourcing.infrastructure.snapshotting), 195

`ACIDRecordManager` (class in eventsourcing.infrastructure.base), 175

`actor_system` (eventsourcing.application.actors.ActorModelRunner attribute), 143

`ActorModelRunner` (class in eventsourcing.application.actors), 143

`actor_speed` (eventsourcing.application.system.BarrierControllingClockThread attribute), 153

`actual_clock_speed` (eventsourcing.application.system.ProcessRunningClockThread attribute), 154

`add_item()` (eventsourcing.domain.model.collection.Collection method), 162

`AESCipher` (class in eventsourcing.utils.cipher.aes), 203

`AggregateRoot` (class in eventsourcing.domain.model.aggregate), 157

`AggregateRoot.AttributeChanged` (class in eventsourcing.domain.model.aggregate), 157

`AggregateRoot.Created` (class in eventsourcing.domain.model.aggregate), 157

`AggregateRoot.Discarded` (class in eventsourcing.domain.model.aggregate), 157

`AggregateRoot.Event` (class in eventsourcing.domain.model.aggregate), 157

`AggregateRootWithHashchainedEvents` (class in eventsourcing.domain.model.aggregate), 157

`AggregateRootWithHashchainedEvents.AttributeChanged` (class in eventsourcing.domain.model.aggregate), 157

`AggregateRootWithHashchainedEvents.Created` (class in eventsourcing.domain.model.aggregate), 157

`AggregateRootWithHashchainedEvents.Discarded` (class in eventsourcing.domain.model.aggregate), 157

*ing.domain.model.aggregate*), 157

`AggregateRootWithHashchainedEvents.Eventattribute()` (in module *eventsourcing.domain.model.decorators*), 163

*ing.domain.model.aggregate*), 158

`all_domain_events()` (*eventsourcing.infrastructure.eventstore.AbstractEventStore* method), 188

`all_domain_events()` (*eventsourcing.infrastructure.eventstore.EventStore* method), 189

`all_sequence_ids()` (*eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager* method), 176

`all_sequence_ids()` (*eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager* method), 179

`all_sequence_ids()` (*eventsourcing.infrastructure.django.manager.DjangoRecordManager* method), 183

`all_sequence_ids()` (*eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager* method), 197

`append()` (*eventsourcing.domain.model.array.Array* method), 159

`application_name` (*eventsourcing.infrastructure.django.models.EntitySnapshotRecord* attribute), 184

`application_name` (*eventsourcing.infrastructure.django.models.NotificationTrackingRecord* attribute), 185

`application_name` (*eventsourcing.infrastructure.django.models.StoredEventRecord* attribute), 186

`application_name` (*eventsourcing.infrastructure.sqlalchemy.records.EntitySnapshotRecord* attribute), 198

`application_name` (*eventsourcing.infrastructure.sqlalchemy.records.NotificationTrackingRecord* attribute), 199

`application_name` (*eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord* attribute), 200

`ApplicationWithConcreteInfrastructure` (class in *eventsourcing.application.simple*), 150

`ApplicationWithEventStores` (class in *eventsourcing.example.application*), 207

`ApplicationWithPersistencePolicies` (class in *eventsourcing.example.application*), 207

`Array` (class in *eventsourcing.domain.model.array*), 159

`ArrayIndexError`, 205

`ArrayRepository` (class in *eventsourcing.infrastructure.repositories.array*), 192

`assert_event_handlers_empty()` (in module *eventsourcing.domain.model.events*), 173

`AttributeChanged` (class in *eventsourcing.domain.model.events*), 170

## B

`b` (*eventsourcing.example.domainmodel.Example* attribute), 208

`BarrierControlledApplicationThread` (class in *eventsourcing.application.system*), 153

`BarrierControllingClockThread` (class in *eventsourcing.application.system*), 153

`BaseAggregateRoot` (class in *eventsourcing.domain.model.aggregate*), 158

`BaseAggregateRoot.AttributeChanged` (class in *eventsourcing.domain.model.aggregate*), 158

`BaseAggregateRoot.Created` (class in *eventsourcing.domain.model.aggregate*), 158

`BaseAggregateRoot.Discarded` (class in *eventsourcing.domain.model.aggregate*), 158

`BaseAggregateRoot.Event` (class in *eventsourcing.domain.model.aggregate*), 158

`beat_heart()` (*eventsourcing.example.domainmodel.Example* method), 208

`BigArray` (class in *eventsourcing.domain.model.array*), 160

`BigArrayNotificationLog` (class in *eventsourcing.interface.notificationlog*), 201

`BigArrayRepository` (class in *eventsourcing.infrastructure.repositories.array*), 192

`bind()` (*eventsourcing.application.simple.SimpleApplication* class method), 151

`broadcast_prompt()` (*eventsourcing.application.multiprocess.MultiprocessRunner* method), 146

`broadcast_prompt()` (*eventsourcing.application.multiprocess.OperatingSystemProcess* method), 147

`broadcast_prompt()` (*eventsourcing.application.system.MultiThreadedRunner* method), 154

`bucket_duration()` (in module *eventsourcing.domain.model.timebucketedlog*), 175

`bucket_size` (*eventsourcing.domain.model.timebucketedlog.Timebucketedlog* attribute), 175

`bucket_starts()` (in module *eventsourcing.domain.model.timebucketedlog*), 175

## C

`calc_parent()` (*eventsourcing*

`ing.domain.model.array.BigArray` method), 160

`calc_required_height()` (`eventsourcing.domain.model.array.BigArray` method), 161

`call_commands()` (`eventsourcing.application.system.ClockThread` method), 153

`call_in_future()` (`eventsourcing.application.system.ClockThread` method), 153

`call_in_future()` (`eventsourcing.application.system.SteppingRunner` method), 156

`call_policy()` (`eventsourcing.application.process.ProcessApplication` method), 148

`CassandraDatastore` (class in `eventsourcing.infrastructure.cassandra.datastore`), 178

`CassandraInfrastructureFactory` (class in `eventsourcing.infrastructure.cassandra.factory`), 179

`CassandraRecordManager` (class in `eventsourcing.infrastructure.cassandra.manager`), 179

`CassandraSettings` (class in `eventsourcing.infrastructure.cassandra.datastore`), 178

`causal_dependencies` (`eventsourcing.infrastructure.django.models.StoredEventRecord` attribute), 186

`causal_dependencies` (`eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord` attribute), 200

`CausalDependencyFailed`, 205

`change_pipeline()` (`eventsourcing.application.simple.SimpleApplication` method), 151

`clear_event_handlers()` (in module `eventsourcing.domain.model.events`), 173

`ClockThread` (class in `eventsourcing.application.system`), 153

`clone()` (`eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager` method), 176

`clone()` (`eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager` method), 197

`close()` (`eventsourcing.application.actors.ActorModelRunner` method), 143

`close()` (`eventsourcing.application.actors.ProcessSlave` method), 144

`close()` (`eventsourcing.application.multiprocess.MultiprocessRunner` method), 146

`close()` (`eventsourcing.application.policies.PersistencePolicy` method), 147

`close()` (`eventsourcing.application.policies.SnapshottingPolicy` method), 148

`close()` (`eventsourcing.application.process.ProcessApplication` method), 148

`close()` (`eventsourcing.application.simple.SimpleApplication` method), 151

`close()` (`eventsourcing.application.snapshotting.SnapshottingApplication` method), 152

`close()` (`eventsourcing.application.system.InProcessRunner` method), 154

`close()` (`eventsourcing.application.system.MultiThreadedRunner` method), 154

`close()` (`eventsourcing.application.system.SteppingMultiThreadedRunner` method), 155

`close()` (`eventsourcing.application.system.SteppingSingleThreadedRunner` method), 156

`close()` (`eventsourcing.application.system.SystemRunner` method), 156

`close()` (`eventsourcing.example.application.ApplicationWithEventStores` method), 207

`close()` (`eventsourcing.example.application.ApplicationWithPersistencePolicies` method), 207

`close_connection()` (`eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore` method), 178

`close_connection()` (`eventsourcing.infrastructure.datastore.Datastore` method), 181

`close_connection()` (`eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore` method), 196

`close_django_connection()` (in module `eventsourcing.infrastructure.django.utils`), 187

`close_example_application()` (in module `eventsourcing.example.application`), 207

`collect_pending_events()` (`eventsourcing.application.process.ProcessApplication` method), 148

[Collection](#) (class in [eventsourcing.domain.model.collection](#)), 161  
[Collection.Created](#) (class in [eventsourcing.domain.model.collection](#)), 161  
[Collection.Discarded](#) (class in [eventsourcing.domain.model.collection](#)), 161  
[Collection.Event](#) (class in [eventsourcing.domain.model.collection](#)), 161  
[Collection.EventWithItem](#) (class in [eventsourcing.domain.model.collection](#)), 161  
[Collection.ItemAdded](#) (class in [eventsourcing.domain.model.collection](#)), 162  
[Collection.ItemRemoved](#) (class in [eventsourcing.domain.model.collection](#)), 162  
[CollectionRepository](#) (class in [eventsourcing.infrastructure.repositories.collection\\_repo](#)), 192  
[Command](#) (class in [eventsourcing.domain.model.command](#)), 162  
[Command.AttributeChanged](#) (class in [eventsourcing.domain.model.command](#)), 162  
[Command.Created](#) (class in [eventsourcing.domain.model.command](#)), 162  
[Command.Discarded](#) (class in [eventsourcing.domain.model.command](#)), 162  
[Command.Done](#) (class in [eventsourcing.domain.model.command](#)), 162  
[Command.Event](#) (class in [eventsourcing.domain.model.command](#)), 163  
[CommandProcess](#) (class in [eventsourcing.application.command](#)), 145  
[ConcurrencyError](#), 205  
[condition\(\)](#) ([eventsourcing.application.policies.SnapshottingPolicy](#) method), 148  
[CONSISTENCY\\_LEVEL](#) ([eventsourcing.infrastructure.cassandra.datastore.CassandraSettings](#) attribute), 178  
[ConsistencyError](#), 205  
[construct\\_app\(\)](#) ([eventsourcing.application.system.System](#) method), 156  
[construct\\_cipher\(\)](#) ([eventsourcing.application.simple.SimpleApplication](#) method), 151  
[construct\\_datastore\(\)](#) ([eventsourcing.application.simple.SimpleApplication](#) method), 151  
[construct\\_datastore\(\)](#) ([eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory](#) method), 196  
[construct\\_entity\\_persistence\\_policy\(\)](#) ([eventsourcing.example.application.ApplicationWithPersistencePolicy](#) method), 207  
[construct\\_event\\_records\(\)](#) ([eventsourcing.application.process.ProcessApplication](#) method), 148  
[construct\\_event\\_store\(\)](#) ([eventsourcing.application.simple.SimpleApplication](#) method), 151  
[construct\\_event\\_store\(\)](#) ([eventsourcing.application.snapshotting.SnapshottingApplication](#) method), 152  
[construct\\_event\\_store\(\)](#) ([eventsourcing.example.application.ApplicationWithEventStores](#) method), 207  
[construct\\_example\\_application\(\)](#) (in module [eventsourcing.example.application](#)), 207  
[construct\\_infrastructure\(\)](#) ([eventsourcing.application.simple.SimpleApplication](#) method), 151  
[construct\\_infrastructure\(\)](#) ([eventsourcing.application.sqlalchemy.SQLAlchemyApplication](#) method), 152  
[construct\\_infrastructure\\_factory\(\)](#) ([eventsourcing.application.simple.SimpleApplication](#) method), 151  
[construct\\_item\\_args\(\)](#) ([eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper](#) method), 195  
[construct\\_log\\_persistence\\_policy\(\)](#) ([eventsourcing.example.application.ApplicationWithPersistencePolicy](#) method), 207  
[construct\\_notification\\_log\(\)](#) ([eventsourcing.application.simple.SimpleApplication](#) method), 151  
[construct\\_persistence\\_policy\(\)](#) ([eventsourcing.application.simple.SimpleApplication](#) method), 151  
[construct\\_persistence\\_policy\(\)](#) ([eventsourcing.application.snapshotting.SnapshottingApplication](#) method), 152  
[construct\\_record\\_manager\(\)](#) ([eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory](#) method), 197  
[construct\\_repository\(\)](#) ([eventsourcing.application.simple.SimpleApplication](#) method), 151  
[construct\\_repository\(\)](#) ([eventsourcing.application.snapshotting.SnapshottingApplication](#) method), 152  
[construct\\_sequenced\\_item\(\)](#) ([eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper](#) method), 195  
[construct\\_sequenced\\_item\\_mapper\(\)](#) ([eventsourcing.example.application.ApplicationWithEventStores](#) method), 207  
[construct\\_snapshot\\_persistence\\_policy\(\)](#) ([eventsourcing.example.application.ApplicationWithPersistencePolicy](#) method), 207



method), 207  
 construct\_sqlalchemy\_eventstore() (in module *eventsourcing.infrastructure.sqlalchemy.factory*), 197  
 construct\_tracking\_kwargs() (*eventsourcing.application.process.ProcessApplication* method), 148  
 consume\_prompt() (*eventsourcing.application.actors.ProcessMaster* method), 144  
 count\_heartbeats() (*eventsourcing.example.domainmodel.Example* method), 208  
 create\_array\_id() (*eventsourcing.domain.model.array.BigArray* method), 161  
 create\_new\_example() (*eventsourcing.example.application.ExampleApplication* method), 207  
 create\_new\_example() (in module *eventsourcing.example.domainmodel*), 209  
 create\_timesequence\_event\_id() (in module *eventsourcing.domain.model.events*), 173  
 Created (class in *eventsourcing.domain.model.events*), 170

## D

DataIntegrityError, 205  
 DataSourceSettingsError, 205  
 Datastore (class in *eventsourcing.infrastructure.datastore*), 181  
 datastore (*eventsourcing.application.simple.SimpleApplication* attribute), 151  
 DatastoreConnectionError, 182  
 DatastoreError, 182  
 DatastoreSettings (class in *eventsourcing.infrastructure.datastore*), 182  
 DatastoreTableError, 182  
 datetime\_from\_timestamp() (in module *eventsourcing.utils.times*), 203  
 decimaltimestamp() (in module *eventsourcing.utils.times*), 203  
 decimaltimestamp\_from\_uuid() (in module *eventsourcing.utils.times*), 203  
 decrypt() (*eventsourcing.utils.cipher.aes.AESCipher* method), 203  
 default() (*eventsourcing.utils.transcoding.ObjectJSONEncoder* method), 204  
 DEFAULT\_KEYSPACE (*eventsourcing.infrastructure.cassandra.datastore.CassandraSettings* attribute), 178

DEFAULT\_PAGE\_SIZE (*eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator* attribute), 191  
 del\_notification\_generator() (*eventsourcing.application.process.ProcessApplication* method), 148  
 delete\_record() (*eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager* method), 176  
 delete\_record() (*eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager* method), 179  
 delete\_record() (*eventsourcing.infrastructure.django.manager.DjangoRecordManager* method), 183  
 delete\_record() (*eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager* method), 197  
 deserialize\_section() (*eventsourcing.interface.notificationlog.RemoteNotificationLog* method), 202  
 Discarded (class in *eventsourcing.domain.model.events*), 171  
 DjangoApplication (class in *eventsourcing.application.django*), 146  
 DjangoInfrastructureFactory (class in *eventsourcing.infrastructure.django.factory*), 182  
 DjangoRecordManager (class in *eventsourcing.infrastructure.django.manager*), 182  
 DomainEntity (class in *eventsourcing.domain.model.entity*), 165  
 DomainEntity.AttributeChanged (class in *eventsourcing.domain.model.entity*), 165  
 DomainEntity.Created (class in *eventsourcing.domain.model.entity*), 165  
 DomainEntity.Discarded (class in *eventsourcing.domain.model.entity*), 165  
 DomainEntity.Event (class in *eventsourcing.domain.model.entity*), 166  
 DomainEvent (class in *eventsourcing.domain.model.events*), 171  
 done() (*eventsourcing.domain.model.command.Command* method), 163  
 drop\_table() (*eventsourcing.application.process.ProcessApplication* method), 148  
 drop\_table() (*eventsourcing.application.simple.SimpleApplication* method), 151  
 drop\_table() (*eventsourcing.application.snapshotting.SnapshottingApplication* method), 152  
 drop\_table() (*eventsourcing*

[ing.infrastructure.cassandra.datastore.CassandraDatastore attribute](#)), 164  
[method](#)), 178  
[drop\\_table\(\)](#) ([eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore attribute](#)), 188  
[method](#)), 196  
[drop\\_tables\(\)](#) ([eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore attribute](#)), 151  
[method](#)), 178  
[drop\\_tables\(\)](#) ([eventsourcing.infrastructure.datastore.Datastore method](#)), 181  
[drop\\_tables\(\)](#) ([eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore method](#)), 196  
**E**  
[encrypt\(\)](#) ([eventsourcing.utils.cipher.aes.AESCipher method](#)), 203  
[entity\\_from\\_snapshot\(\)](#) ([in module eventsourcing.infrastructure.snapshotting](#)), 196  
[EntityIsDiscarded](#), 205  
[EntitySnapshotRecord](#) ([class in eventsourcing.infrastructure.django.models](#)), 183  
[EntitySnapshotRecord](#) ([class in eventsourcing.infrastructure.sqlalchemy.records](#)), 198  
[EntitySnapshotRecord.DoesNotExist](#), 183  
[EntitySnapshotRecord.MultipleObjectsReturned](#), 183  
[EntityVersionNotFound](#), 205  
[EntityWithHashchain](#) ([class in eventsourcing.domain.model.entity](#)), 167  
[EntityWithHashchain.Created](#) ([class in eventsourcing.domain.model.entity](#)), 167  
[EntityWithHashchain.Discarded](#) ([class in eventsourcing.domain.model.entity](#)), 167  
[EntityWithHashchain.Event](#) ([class in eventsourcing.domain.model.entity](#)), 167  
[event\\_from\\_item\(\)](#) ([eventsourcing.infrastructure.sequenceditemmapper.AbstractSequencedItemMapper method](#)), 194  
[event\\_from\\_item\(\)](#) ([eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper method](#)), 195  
[event\\_from\\_topic\\_and\\_state\(\)](#) ([eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper method](#)), 195  
[event\\_id](#) ([eventsourcing.domain.model.events.EventWithTimeuuid attribute](#)), 173  
[event\\_store](#) ([eventsourcing.application.simple.SimpleApplication attribute](#)), 151  
[event\\_store](#) ([eventsourcing.domain.model.entity.AbstractEntityRepository](#)
[ing.infrastructure.cassandra.datastore.CassandraDatastore attribute](#)), 164  
[event\\_store](#) ([eventsourcing.infrastructure.eventplayer.EventPlayer](#)
[ing.infrastructure.cassandra.datastore.CassandraDatastore attribute](#)), 188  
[event\\_store\\_class](#) ([eventsourcing.application.simple.SimpleApplication](#)
[ing.infrastructure.cassandra.datastore.CassandraDatastore attribute](#)), 151  
[EventHandlersNotEmptyError](#), 172  
[EventHashError](#), 205  
[EventPlayer](#) ([class in eventsourcing.infrastructure.eventplayer](#)), 188  
[EventRecordNotFound](#), 205  
[EventSourcedRepository](#) ([class in eventsourcing.infrastructure.eventsourcedrepository](#)), 188  
[EventSourcedSnapshotStrategy](#) ([class in eventsourcing.infrastructure.snapshotting](#)), 195  
[eventsourcing.application.actors](#) ([module](#)), 143  
[eventsourcing.application.command](#) ([module](#)), 145  
[eventsourcing.application.django](#) ([module](#)), 146  
[eventsourcing.application.multiprocess](#) ([module](#)), 146  
[eventsourcing.application.pipeline](#) ([module](#)), 147  
[eventsourcing.application.policies](#) ([module](#)), 147  
[eventsourcing.application.popo](#) ([module](#)), 148  
[eventsourcing.application.process](#) ([module](#)), 148  
[eventsourcing.application.simple](#) ([module](#)), 150  
[eventsourcing.application.snapshotting](#) ([module](#)), 152  
[eventsourcing.application.sqlalchemy](#) ([module](#)), 152  
[eventsourcing.application.system](#) ([module](#)), 153  
[eventsourcing.domain.model.aggregate](#) ([module](#)), 156  
[eventsourcing.domain.model.array](#) ([module](#)), 159  
[eventsourcing.domain.model.collection](#) ([module](#)), 161  
[eventsourcing.domain.model.command](#) ([module](#)), 162  
[eventsourcing.domain.model.decorators](#) ([module](#)), 163  
[eventsourcing.domain.model.entity](#) ([module](#)), 164  
[eventsourcing.domain.model.events](#) ([module](#)), 170

[eventsourcing.domain.model.snapshot \(module\), 173](#)  
[eventsourcing.domain.model.timebucketedlog \(module\), 174](#)  
[eventsourcing.example.application \(module\), 207](#)  
[eventsourcing.example.domainmodel \(module\), 208](#)  
[eventsourcing.example.infrastructure \(module\), 209](#)  
[eventsourcing.example.interface.flaskapp \(module\), 209](#)  
[eventsourcing.exceptions \(module\), 205](#)  
[eventsourcing.infrastructure.base \(module\), 175](#)  
[eventsourcing.infrastructure.cassandra.datastore \(module\), 178](#)  
[eventsourcing.infrastructure.cassandra.eventsourcing \(module\), 179](#)  
[eventsourcing.infrastructure.cassandra.manager \(module\), 179](#)  
[eventsourcing.infrastructure.cassandra.records \(module\), 180](#)  
[eventsourcing.infrastructure.datastore \(module\), 181](#)  
[eventsourcing.infrastructure.django.factory \(module\), 182](#)  
[eventsourcing.infrastructure.django.manager \(module\), 182](#)  
[eventsourcing.infrastructure.django.models \(module\), 183](#)  
[eventsourcing.infrastructure.django.utils \(module\), 187](#)  
[eventsourcing.infrastructure.eventplayer \(module\), 188](#)  
[eventsourcing.infrastructure.eventsourcedepository \(module\), 188](#)  
[eventsourcing.infrastructure.eventstore \(module\), 188](#)  
[eventsourcing.infrastructure.integersequencegenerator \(module\), 190](#)  
[eventsourcing.infrastructure.integersequencegenerator \(module\), 190](#)  
[eventsourcing.infrastructure.iterators \(module\), 190](#)  
[eventsourcing.infrastructure.repositories.array \(module\), 192](#)  
[eventsourcing.infrastructure.repositories.collection \(module\), 192](#)  
[eventsourcing.infrastructure.repositories.timebucketedlog \(module\), 192](#)  
[eventsourcing.infrastructure.sequenceditem \(module\), 193](#)  
[eventsourcing.infrastructure.sequenceditemmapper \(module\), 193](#)  
[eventsourcing.infrastructure.snapshotting \(module\), 195](#)  
[eventsourcing.infrastructure.sqlalchemy.datastore \(module\), 196](#)  
[eventsourcing.infrastructure.sqlalchemy.factory \(module\), 196](#)  
[eventsourcing.infrastructure.sqlalchemy.manager \(module\), 197](#)  
[eventsourcing.infrastructure.sqlalchemy.records \(module\), 198](#)  
[eventsourcing.infrastructure.timebucketedlog\\_reader \(module\), 201](#)  
[eventsourcing.interface.notificationlog \(module\), 201](#)  
[eventsourcing.utils.cipher.aes \(module\), 203](#)  
[eventsourcing.utils.times \(module\), 203](#)  
[eventsourcing.utils.topic \(module\), 204](#)  
[eventsourcing.utils.transcoding \(module\), 204](#)  
[EventsourcingError, 205](#)  
[EventStore \(class in eventsourcing.infrastructure.eventstore\), 189](#)  
[EventWithHash \(class in eventsourcing.domain.model.events\), 172](#)  
[EventWithOriginatorID \(class in eventsourcing.domain.model.events\), 172](#)  
[EventWithOriginatorVersion \(class in eventsourcing.domain.model.events\), 172](#)  
[EventWithTimestamp \(class in eventsourcing.domain.model.events\), 173](#)  
[EventWithTimeuuid \(class in eventsourcing.domain.model.events\), 173](#)  
[Example \(class in eventsourcing.example.domainmodel\), 208](#)  
[Example.AttributeChanged \(class in eventsourcing.example.domainmodel\), 208](#)  
[Example.Created \(class in eventsourcing.example.domainmodel\), 208](#)  
[Example.Discarded \(class in eventsourcing.example.domainmodel\), 208](#)  
[Example.Event \(class in eventsourcing.example.domainmodel\), 208](#)  
[Example.Heartbeat \(class in eventsourcing.example.domainmodel\), 208](#)  
[ExampleApplication \(class in eventsourcing.example.application\), 207](#)  
[ExampleRepository \(class in eventsourcing.example.infrastructure\), 209](#)

## F

[filter\(\)](#)

(eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager)

method), 179

filter\_by() (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager application() (in module eventsourcing.example.application), 207

follow() (eventsourcing.application.process.ProcessApplication method), 149

foo (eventsourcing.example.domainmodel.Example attribute), 208

format\_section\_id() (eventsourcing.interface.notificationlog.LocalNotificationLog static method), 201

forward\_prompt() (eventsourcing.application.actors.ActorModelRunner method), 144

forward\_prompt() (eventsourcing.application.actors.PipelineActor method), 144

from\_jsonable() (eventsourcing.utils.transcoding.ObjectJSONDecoder class method), 204

from\_record() (eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager method), 176

**G**

get\_and\_project\_events() (eventsourcing.infrastructure.eventsourcedrepositary.EventSourcedRepository method), 188

get\_domain\_event() (eventsourcing.infrastructure.eventstore.AbstractEventStore method), 188

get\_domain\_event() (eventsourcing.infrastructure.eventstore.EventStore method), 189

get\_domain\_events() (eventsourcing.infrastructure.eventstore.AbstractEventStore method), 189

get\_domain\_events() (eventsourcing.infrastructure.eventstore.EventStore method), 189

get\_entity() (eventsourcing.domain.model.entity.AbstractEntityRepository method), 164

get\_entity() (eventsourcing.infrastructure.eventsourcedrepositary.EventSourcedRepository method), 188

get\_event\_class\_and\_attrs() (eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper method), 195

get\_event\_from\_notification() (eventsourcing.application.process.ProcessApplication method), 149

get\_events() (eventsourcing.infrastructure.timebucketedlog\_reader.TimebucketedlogReader method), 201

get\_field\_kwargs() (eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager method), 176

get\_item() (eventsourcing.domain.model.array.BigArray method), 161

get\_item() (eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager method), 176

get\_item\_assigned() (eventsourcing.domain.model.array.Array method), 160

get\_item\_topic\_and\_state() (eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper method), 195

get\_items() (eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager method), 177

get\_items() (eventsourcing.interface.notificationlog.BigArrayNotificationLog method), 201

get\_items() (eventsourcing.interface.notificationlog.LocalNotificationLog method), 202

get\_items() (eventsourcing.interface.notificationlog.RecordManagerNotificationLog method), 202

get\_items\_assigned() (eventsourcing.domain.model.array.Array method), 160

get\_json() (eventsourcing.interface.notificationlog.RemoteNotificationLog method), 202

get\_last\_array() (eventsourcing.domain.model.array.BigArray method), 161

get\_last\_item\_and\_next\_position() (eventsourcing.domain.model.array.Array method), 160

get\_last\_item\_and\_next\_position() (eventsourcing.domain.model.array.BigArray method), 161

get\_max\_record\_id() (eventsourcing.infrastructure.base.ACIDRecordManager method), 176

get\_max\_record\_id() (eventsourcing.infrastructure.django.manager.DjangoRecordManager method), 183

get\_max\_record\_id() (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 176



method), 197

get\_max\_tracking\_record\_id() (eventsourcing.infrastructure.base.ACIDRecordManager method), 176

get\_max\_tracking\_record\_id() (eventsourcing.infrastructure.django.manager.DjangoRecordManager method), 183

get\_max\_tracking\_record\_id() (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 198

get\_messages() (eventsourcing.infrastructure.timebucketedlog\_reader.Timebucketedlog\_reader method), 201

get\_most\_recent\_event() (eventsourcing.infrastructure.eventstore.AbstractEventStore method), 189

get\_most\_recent\_event() (eventsourcing.infrastructure.eventstore.EventStore method), 189

get\_next\_position() (eventsourcing.domain.model.array.Array method), 160

get\_next\_position() (eventsourcing.interface.notificationlog.BigArrayNotificationLog method), 201

get\_next\_position() (eventsourcing.interface.notificationlog.LocalNotificationLog method), 202

get\_next\_position() (eventsourcing.interface.notificationlog.RecordManagerNotificationLogging method), 202

get\_notification\_generator() (eventsourcing.application.process.ProcessApplication method), 149

get\_notifications() (eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager method), 177

get\_notifications() (eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager method), 180

get\_notifications() (eventsourcing.infrastructure.django.manager.DjangoRecordManager method), 183

get\_notifications() (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 198

get\_or\_create() (eventsourcing.domain.model.timebucketedlog.Timebucketedlog\_repository method), 175

get\_pipeline\_and\_notification\_id() (eventsourcing.infrastructure.base.ACIDRecordManager method), 176

get\_record() (eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager method), 180

get\_record() (eventsourcing.infrastructure.django.manager.DjangoRecordManager method), 183

get\_record() (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 198

get\_record\_table\_name() (eventsourcing.infrastructure.base.SQLRecordManager method), 177

get\_record\_table\_name() (eventsourcing.infrastructure.django.manager.DjangoRecordManager method), 183

get\_record\_table\_name() (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 198

get\_records() (eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager method), 177

get\_records() (eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager method), 180

get\_records() (eventsourcing.infrastructure.django.manager.DjangoRecordManager method), 183

get\_records() (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 198

get\_resource() (eventsourcing.interface.notificationlog.RemoteNotificationLog method), 202

get\_slice() (eventsourcing.domain.model.array.BigArray method), 161

get\_snapshot() (eventsourcing.infrastructure.snapshotting.AbstractSnapshotStrategy method), 195

get\_snapshot() (eventsourcing.infrastructure.snapshotting.EventSourcedSnapshotStrategy method), 195

get\_timebucketedlog\_reader() (eventsourcing.infrastructure.timebucketedlog\_reader module eventsourcing.infrastructure.timebucketedlog\_reader), 201

Repository() (in module eventsourcing.utils.topic), 204

GetEntityEventsThread (class in eventsourcing.infrastructure.iterators), 191

get\_record() (eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager method), 176

H

handle\_prompt() (eventsourcing-

`ing.application.system.InProcessRunner` method), 154

`handle_prompt()` (`eventsourcing.application.system.MultiThreadedRunner` method), 154

`handle_prompt()` (`eventsourcing.application.system.SingleThreadedRunner` method), 155

`handle_prompt()` (`eventsourcing.application.system.SteppingMultiThreadedRunner` method), 156

`handle_prompt()` (`eventsourcing.application.system.SteppingSingleThreadedRunner` method), 156

`handle_slave_run_response()` (`eventsourcing.application.actors.ProcessMaster` method), 144

`has_tracking_record()` (`eventsourcing.infrastructure.base.ACIDRecordManager` method), 176

`has_tracking_record()` (`eventsourcing.infrastructure.django.manager.DjangoRecordManager` method), 183

`has_tracking_record()` (`eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager` method), 198

`HeadHashError`, 206

`hello()` (in module `eventsourcing.example.interface.flaskapp`), 209

`HOSTS` (`eventsourcing.infrastructure.cassandra.datastore.CassandraSettings` attribute), 178

**I**

`id` (`eventsourcing.domain.model.entity.DomainEntity` attribute), 167

`id` (`eventsourcing.example.interface.flaskapp.IntegerSequencedItem` attribute), 209

`id` (`eventsourcing.infrastructure.django.models.IntegerSequencedRecord` attribute), 184

`id` (`eventsourcing.infrastructure.django.models.TimestampSequencedRecord` attribute), 187

`id` (`eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedRecord` attribute), 199

`id` (`eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedRecord` attribute), 200

`index` (`eventsourcing.domain.model.array.ItemAssigned` attribute), 161

`infrastructure_factory_class` (`eventsourcing.application.popo.PopoApplication` attribute), 148

`infrastructure_factory_class` (`eventsourcing.application.simple.SimpleApplication` attribute), 151

`infrastructure_factory_class` (`eventsourcing.application.sqlalchemy.SQLAlchemyApplication` attribute), 152

`init_example_application()` (in module `eventsourcing.example.application`), 208

`init_example_application_with_sqlalchemy()` (in module `eventsourcing.example.interface.flaskapp`), 209

`init_pipeline()` (`eventsourcing.application.actors.PipelineActor` method), 144

`init_pipelines()` (`eventsourcing.application.actors.SystemActor` method), 145

`init_process()` (`eventsourcing.application.actors.ProcessMaster` method), 144

`init_process()` (`eventsourcing.application.actors.ProcessSlave` method), 144

`InProcessRunner` (class in `eventsourcing.application.system`), 153

`insert_select_max` (`eventsourcing.infrastructure.base.SQLRecordManager` method), 177

`insert_tracking_record` (`eventsourcing.infrastructure.base.SQLRecordManager` attribute), 177

`insert_values` (`eventsourcing.infrastructure.base.SQLRecordManager` attribute), 177

`integer_sequenced_record_class` (`eventsourcing.infrastructure.cassandra.factory.CassandraInfrastructureFactory` attribute), 179

`integer_sequenced_record_class` (`eventsourcing.infrastructure.django.factory.DjangoInfrastructureFactory` attribute), 182

`integer_sequenced_record_class` (`eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory` attribute), 197

`IntegerSequencedItem` (class in `eventsourcing.example.interface.flaskapp`), 209

`IntegerSequencedNoIDRecord` (class in `eventsourcing.infrastructure.sqlalchemy.records`), 198

`IntegerSequencedRecord` (class in `eventsourcing.infrastructure.cassandra.records`), 180

`IntegerSequencedRecord` (class in `eventsourcing.infrastructure.django.models`), 184

`IntegerSequencedRecord` (in module `eventsourcing.infrastructure.sqlalchemy.records`), 199

`IntegerSequencedRecord.DoesNotExist`, 180, 184

`IntegerSequencedRecord.MultipleObjectsReturned`,

- 180, 184
- `IntegerSequencedWithIDRecord` (class in `eventsourcing.infrastructure.sqlalchemy.records`), 199
- `is_constructed_with_session` (`eventsourcing.application.simple.SimpleApplication` attribute), 151
- `is_constructed_with_session` (`eventsourcing.application.sqlalchemy.SQLAlchemyApplication` attribute), 152
- `is_done` (`eventsourcing.domain.model.command.Command` attribute), 163
- `is_event()` (`eventsourcing.application.policies.PersistencePolicy` method), 147
- `is_my_prompt()` (`eventsourcing.application.actors.ProcessSlave` method), 144
- `is_prompt()` (`eventsourcing.application.actors.ActorModelRunner` static method), 144
- `is_prompt()` (`eventsourcing.application.multiprocess.MultiprocessRunner` static method), 146
- `is_prompt()` (`eventsourcing.application.multiprocess.OperatingSystemProcess` static method), 147
- `is_prompt()` (`eventsourcing.application.system.System` method), 156
- `is_sqlite()` (`eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore` method), 196
- `item` (`eventsourcing.domain.model.array.ItemAssigned` attribute), 161
- `item` (`eventsourcing.domain.model.collection.Collection.EventWithItem` attribute), 162
- `item_from_event()` (`eventsourcing.infrastructure.eventstore.EventStore` method), 190
- `item_from_event()` (`eventsourcing.infrastructure.sequenceditemmapper.AbstractSequencedItemMapper` method), 194
- `item_from_event()` (`eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper` method), 195
- `ItemAssigned` (class in `eventsourcing.domain.model.array`), 161
- `items` (`eventsourcing.domain.model.collection.Collection` attribute), 162
- `iterator_class` (`eventsourcing.infrastructure.eventstore.EventStore` attribute), 190
- `iterencode()` (`eventsourcing.utils.transcoding.ObjectJSONEncoder` method), 204
- ## J
- `json_decoder_class` (`eventsourcing.application.simple.SimpleApplication` attribute), 151
- `json_dumps()` (in module `eventsourcing.utils.transcoding`), 205
- `json_encoder_class` (`eventsourcing.application.simple.SimpleApplication` attribute), 151
- `json_loads()` (in module `eventsourcing.utils.transcoding`), 205
- ## L
- `list_items()` (`eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager` method), 177
- `list_sequence_ids()` (`eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager` method), 177
- `LocalNotificationLog` (class in `eventsourcing.interface.notificationlog`), 201
- `log_message()` (`eventsourcing.domain.model.timebucketedlog.Timebucketedlog` method), 175
- `Logged` (class in `eventsourcing.domain.model.events`), 173
- `loop_on_prompts()` (`eventsourcing.application.multiprocess.OperatingSystemProcess` method), 147
- `loop_on_prompts()` (`eventsourcing.application.system.PromptQueuedApplicationThread` method), 155
- ## M
- `make_notification_log_url()` (`eventsourcing.interface.notificationlog.RemoteNotificationLog` method), 202
- `make_timebucket_id()` (in module `eventsourcing.domain.model.timebucketedlog`), 175
- `message` (`eventsourcing.domain.model.timebucketedlog.MessageLogged` attribute), 174
- `MessageLogged` (class in `eventsourcing.domain.model.timebucketedlog`), 174
- `MismatchedOriginatorError`, 206
- `mixin()` (`eventsourcing.application.simple.SimpleApplication` class method), 151
- `MultiprocessRunner` (class in `eventsourcing.application.multiprocess`), 146

MultiThreadedRunner (class in *event sourcing.application.system*), 154

mutate() (event sourcing.domain.model.command.Command.Done method), 163

mutate() (event sourcing.domain.model.events.DomainEvent method), 171

mutate() (event sourcing.example.domainmodel.Example.Heartbeat method), 208

mutate() (event sourcing.infrastructure.eventplayer.EventPlayer static method), 188

mutator() (in module *event sourcing.domain.model.decorators*), 163

MutatorRequiresTypeNotInstance, 206

## N

name (event sourcing.domain.model.events.AttributeChanged attribute), 170

name (event sourcing.domain.model.timebucketedlog.Timebucketedlog attribute), 175

next\_bucket\_starts() (in module *event sourcing.domain.model.timebucketedlog*), 175

notification\_id (event sourcing.infrastructure.django.models.NotificationTrackingRecord attribute), 185

notification\_id (event sourcing.infrastructure.django.models.StoredEventRecord attribute), 186

notification\_id (event sourcing.infrastructure.sqlalchemy.records.NotificationTrackingRecord attribute), 199

notification\_id (event sourcing.infrastructure.sqlalchemy.records.StoredEventRecord attribute), 200

notification\_log\_section\_size (event sourcing.application.simple.SimpleApplication attribute), 151

NotificationLogReader (class in *event sourcing.interface.notificationlog*), 202

NotificationLogView (class in *event sourcing.interface.notificationlog*), 202

NotificationTrackingRecord (class in *event sourcing.infrastructure.django.models*), 185

NotificationTrackingRecord (class in *event sourcing.infrastructure.sqlalchemy.records*), 199

NotificationTrackingRecord.DoesNotExist, 185

NotificationTrackingRecord.MultipleObjectsReturned, 185

## O

ObjectJSONDecoder (class in *event sourcing.utils.transcoding*), 204

ObjectJSONEncoder (class in *event sourcing.utils.transcoding*), 204

objects (event sourcing.infrastructure.django.models.EntitySnapshotRecord attribute), 184

objects (event sourcing.infrastructure.django.models.IntegerSequencedRecord attribute), 184

objects (event sourcing.infrastructure.django.models.NotificationTrackingRecord attribute), 185

objects (event sourcing.infrastructure.django.models.SnapshotRecord attribute), 185

objects (event sourcing.infrastructure.django.models.StoredEventRecord attribute), 186

objects (event sourcing.infrastructure.django.models.TimestampSequencedRecord attribute), 187

OperatingSystemProcess (class in *event sourcing.application.multiprocess*), 146

OperationalError, 206

originator\_id (event sourcing.domain.model.events.EventWithOriginatorID attribute), 172

originator\_id (event sourcing.domain.model.snapshot.AbstractSnapshot attribute), 174

originator\_id (event sourcing.infrastructure.cassandra.records.StoredEventRecord attribute), 180

originator\_id (event sourcing.infrastructure.django.models.EntitySnapshotRecord attribute), 184

originator\_id (event sourcing.infrastructure.django.models.StoredEventRecord attribute), 186

originator\_id (event sourcing.infrastructure.sequenceditem.StoredEvent attribute), 194

originator\_id (event sourcing.infrastructure.sqlalchemy.records.EntitySnapshotRecord attribute), 198

originator\_id (event sourcing.infrastructure.sqlalchemy.records.StoredEventRecord attribute), 200

originator\_topic (event sourcing.domain.model.entity.DomainEntity.Created attribute), 165

originator\_version (event sourcing-



[ing.domain.model.events.EventWithOriginatorVersion](#) (eventsourcing.attribute), 173  
[originator\\_version](#) (eventsourcing.ing.domain.model.snapshot.AbstractSnapshot attribute), 174  
[originator\\_version](#) (eventsourcing.ing.infrastructure.cassandra.records.StoredEventRecord attribute), 181  
[originator\\_version](#) (eventsourcing.ing.infrastructure.django.models.EntitySnapshotRecord attribute), 184  
[originator\\_version](#) (eventsourcing.ing.infrastructure.django.models.StoredEventRecord attribute), 186  
[originator\\_version](#) (eventsourcing.ing.infrastructure.sequenceditem.StoredEvent attribute), 194  
[originator\\_version](#) (eventsourcing.ing.infrastructure.sqlalchemy.records.EntitySnapshotRecord attribute), 198  
[originator\\_version](#) (eventsourcing.ing.infrastructure.sqlalchemy.records.StoredEventRecord attribute), 200  
[OriginatorIDError](#), 206  
[OriginatorVersionError](#), 206  
[orm\\_query\(\)](#) (eventsourcing.ing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager method), 198  
[other\\_names](#) (eventsourcing.ing.infrastructure.sequenceditem.SequencedItemFieldNames attribute), 193  
**P**  
[persist\\_event\\_type](#) (eventsourcing.ing.application.command.CommandProcess attribute), 145  
[persist\\_event\\_type](#) (eventsourcing.ing.application.simple.SimpleApplication attribute), 151  
[PersistencePolicy](#) (class in eventsourcing.ing.application.policies), 147  
[Pipeable](#) (class in eventsourcing.ing.application.pipeline), 147  
[PipeableMetaclass](#) (class in eventsourcing.ing.application.pipeline), 147  
[pipeline\\_id](#) (eventsourcing.ing.infrastructure.django.models.NotificationTrackingRecord attribute), 185  
[pipeline\\_id](#) (eventsourcing.ing.infrastructure.django.models.StoredEventRecord attribute), 187  
[pipeline\\_id](#) (eventsourcing.ing.infrastructure.sqlalchemy.records.NotificationTrackingRecord attribute), 199  
[pipeline\\_id](#) (eventsourcing.ing.infrastructure.sqlalchemy.records.StoredEventRecord attribute), 200  
[PipelineActor](#) (class in eventsourcing.ing.application.actors), 144  
[PipelineExpression](#) (class in eventsourcing.ing.application.pipeline), 147  
[PipelineInitRequest](#) (class in eventsourcing.ing.application.actors), 144  
[pk](#) (eventsourcing.ing.infrastructure.cassandra.records.IntegerSequencedRecord attribute), 180  
[pk](#) (eventsourcing.ing.infrastructure.cassandra.records.SnapshotRecord attribute), 180  
[pk](#) (eventsourcing.ing.infrastructure.cassandra.records.StoredEventRecord attribute), 181  
[pk](#) (eventsourcing.ing.infrastructure.cassandra.records.TimestampSequencedRecord attribute), 181  
[pk](#) (eventsourcing.ing.infrastructure.cassandra.records.TimeuuidSequencedRecord attribute), 181  
[policy\(\)](#) (eventsourcing.ing.application.process.ProcessApplication static method), 149  
[PopoApplication](#) (class in eventsourcing.ing.application.popo), 148  
[PORT](#) (eventsourcing.ing.infrastructure.cassandra.datastore.CassandraSettings attribute), 178  
[RecordManager](#) (eventsourcing.ing.example.interface.flaskapp.IntegerSequencedItem attribute), 209  
[FieldNames](#) (eventsourcing.ing.infrastructure.cassandra.records.IntegerSequencedRecord attribute), 180  
[position](#) (eventsourcing.ing.infrastructure.cassandra.records.SnapshotRecord attribute), 180  
[position](#) (eventsourcing.ing.infrastructure.cassandra.records.TimestampSequencedRecord attribute), 181  
[position](#) (eventsourcing.ing.infrastructure.cassandra.records.TimeuuidSequencedRecord attribute), 181  
[position](#) (eventsourcing.ing.infrastructure.django.models.IntegerSequencedRecord attribute), 184  
[position](#) (eventsourcing.ing.infrastructure.django.models.SnapshotRecord attribute), 185  
[position](#) (eventsourcing.ing.infrastructure.django.models.TimestampSequencedRecord attribute), 187  
[position](#) (eventsourcing.ing.infrastructure.sequenceditem.SequencedItem attribute), 193  
[position](#) (eventsourcing.ing.infrastructure.sqlalchemy.records.NotificationTrackingRecord attribute), 199

[ing.infrastructure.sequenceditem.SequencedItemFieldManager \(in module eventsourcing.domain.model.events\), 173](#)  
[ing.infrastructure.sqlalchemy.records.IntegerSequencedNoIDRecord \(in module eventsourcing.application.process.ProcessApplication\), 149](#)  
[ing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord \(in module eventsourcing.application.system.PromptOutbox\), 154](#)  
[ing.infrastructure.sqlalchemy.records.SnapshotRecord \(in module eventsourcing.application.system.SnapshotRecordManager\), 199](#)  
[ing.infrastructure.sqlalchemy.records.TimestampSequencedNoIDRecord \(in module eventsourcing.application.system.TimestampRecordManager\), 200](#)  
[ing.infrastructure.sqlalchemy.records.TimestampSequencedWithIDRecord \(in module eventsourcing.application.system.TimestampRecordManager\), 200](#)  
[ing.interface.notificationlog.NotificationLogView \(in module eventsourcing.interface.notificationlog\), 202](#)  
[previous\\_bucket\\_starts \(in module eventsourcing.domain.model.timebucketedlog\), 175](#)  
[process\\_upstream\\_event \(in module eventsourcing.application.process.ProcessApplication\), 149](#)  
[ProcessApplication \(class in module eventsourcing.application.process\), 148](#)  
[ProcessApplicationWithSnapshotting \(class in module eventsourcing.application.process\), 149](#)  
[ProcessEvent \(class in module eventsourcing.application.process\), 149](#)  
[ProcessInitRequest \(class in module eventsourcing.application.actors\), 144](#)  
[ProcessMaster \(class in module eventsourcing.application.actors\), 144](#)  
[ProcessRunningClockThread \(class in module eventsourcing.application.system\), 154](#)  
[ProcessSlave \(class in module eventsourcing.application.actors\), 144](#)  
[ProgrammingError, 206](#)  
[project\\_events \(in module eventsourcing.infrastructure.eventplayer.EventPlayer\), 188](#)  
[Prompt \(class in module eventsourcing.application.process\), 149](#)  
[PromptFailed, 206](#)  
[PromptOutbox \(class in module eventsourcing.application.system\), 154](#)  
[PromptQueuedApplicationThread \(class in module eventsourcing.application.system\), 154](#)  
[PROTOCOL\\_VERSION \(in module eventsourcing.infrastructure.cassandra.datastore.CassandraSettings\), 178](#)  
[raise\\_index\\_error \(in module eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager\), 177](#)  
[raise\\_no\\_id\\_record\\_error \(in module eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager\), 177](#)  
[raise\\_with\\_id\\_record\\_integrity\\_error \(in module eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager\), 177](#)  
[raise\\_sequenced\\_item\\_conflict \(in module eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager\), 177](#)  
[random \(in module eventsourcing.domain.model.decorators\), 164](#)  
[read \(in module eventsourcing.interface.notificationlog.NotificationLogReader\), 202](#)  
[read\\_items \(in module eventsourcing.interface.notificationlog.NotificationLogReader\), 202](#)  
[read\\_list \(in module eventsourcing.interface.notificationlog.NotificationLogReader\), 202](#)  
[read\\_reader \(in module eventsourcing.application.process.ProcessApplication\), 149](#)  
[receiveMessage \(in module eventsourcing.application.actors.PipelineActor\), 144](#)  
[receiveMessage \(in module eventsourcing.application.actors.ProcessMaster\), 144](#)  
[receiveMessage \(in module eventsourcing.application.actors.ProcessSlave\), 144](#)  
[receiveMessage \(in module eventsourcing.application.actors.SystemActor\), 145](#)  
[reconstruct\\_object \(in module eventsourcing.infrastructure.sequenceditemmapper\), 195](#)  
[record\\_manager\\_class \(in module eventsourcing.application.django.DjangoApplication\), 146](#)  
[record\\_manager\\_class \(in module eventsourcing.application.simple.SimpleApplication\), 151](#)

record\_manager\_class (eventsourcing.infrastructure.cassandra.factory.CassandraInfrastructureFactoryCustomMethod), 151  
 attribute), 179  
 resolve\_attr() (in module eventsourcing.utils.topic), 204  
 record\_manager\_class (eventsourcing.infrastructure.django.factory.DjangoInfrastructureFactoryCustomMethod), 182  
 attribute), 182  
 record\_manager\_class (eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactoryCustomMethod), 197  
 attribute), 197  
 run() (eventsourcing.application.multiprocess.OperatingSystemProcess method), 147  
 record\_process\_event() (eventsourcing.application.process.ProcessApplication method), 149  
 run() (eventsourcing.application.process.ProcessApplication method), 149  
 record\_sequenced\_item() (eventsourcing.infrastructure.base.AbstractSequencedItemRecordManagerCustomMethod), 177  
 method), 177  
 run() (eventsourcing.application.system.BarrierControllingClockThread method), 153  
 record\_sequenced\_items() (eventsourcing.infrastructure.base.AbstractSequencedItemRecordManagerCustomMethod), 177  
 method), 177  
 run() (eventsourcing.application.system.ProcessRunningClockThread method), 154  
 record\_sequenced\_items() (eventsourcing.infrastructure.base.SQLRecordManager method), 178  
 method), 178  
 run() (eventsourcing.application.system.PromptQueuedApplicationThread method), 155  
 record\_sequenced\_items() (eventsourcing.infrastructure.cassandra.manager.CassandraRecordManagerCustomMethod), 180  
 method), 180  
 run() (eventsourcing.infrastructure.iterators.GetEntityEventsThread method), 191  
 record\_sequenced\_items() (eventsourcing.infrastructure.cassandra.manager.CassandraRecordManagerCustomMethod), 180  
 method), 180  
 run() (eventsourcing.application.system.SingleThreadedRunner method), 155  
 RecordConflictError, 206  
 RecordManagerNotificationLog (class in eventsourcing.interface.notificationlog), 202  
 run\_process() (eventsourcing.application.actors.ProcessSlave method), 144  
 RedisIncr (class in eventsourcing.infrastructure.integersequencegenerators.redisincr), 190  
 run\_process() (eventsourcing.application.multiprocess.OperatingSystemProcess method), 147  
 register\_new\_collection() (in module eventsourcing.domain.model.collection), 162  
 run\_process() (eventsourcing.application.system.PromptQueuedApplicationThread method), 155  
 RemoteNotificationLog (class in eventsourcing.interface.notificationlog), 202  
 run\_slave() (eventsourcing.application.actors.ProcessMaster method), 144  
 remove\_item() (eventsourcing.domain.model.collection.Collection method), 162  
 REPLICATION\_FACTOR (eventsourcing.infrastructure.cassandra.datastore.CassandraSettings attribute), 179  
 repository (eventsourcing.application.simple.SimpleApplication attribute), 151  
 repository\_class (eventsourcing.application.simple.SimpleApplication attribute), 151  
 RepositoryKeyError, 206  
 RepositoryWrapper (class in eventsourcing.application.process), 149  
 reset\_connection\_after\_forking() (eventsourcing.application.django.DjangoApplication class method), 146  
 reset\_connection\_after\_forking() (eventsourcing.application.django.DjangoApplication class method), 146  
 sequence\_id (eventsourcing.example.interface.flaskapp.IntegerSequencedItem attribute), 209  
 sequence\_id (eventsourcing.infrastructure.cassandra.records.IntegerSequencedRecord attribute), 180  
 sequence\_id (eventsourcing.infrastructure.cassandra.records.SnapshotRecord attribute), 180

## S

attribute), 180	194
sequence_id (eventsource-ing.infrastructure.cassandra.records.TimestampSequencedRecord attribute), 181	ing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore attribute), 196
sequence_id (eventsource-ing.infrastructure.cassandra.records.TimestampSequencedRecord attribute), 181	set_notification_ids (eventsource-ing.application.process.ProcessApplication attribute), 149
sequence_id (eventsource-ing.infrastructure.django.models.IntegerSequencedRecord attribute), 184	set_reader_position_from_tracking_records() (eventsource-ing.application.process.ProcessApplication method), 149
sequence_id (eventsource-ing.infrastructure.django.models.SnapshotRecord attribute), 186	setup_connection() (eventsource-ing.infrastructure.cassandra.datastore.CassandraDatastore method), 178
sequence_id (eventsource-ing.infrastructure.django.models.TimestampSequencedRecord attribute), 187	setup_connection() (eventsource-ing.infrastructure.datastore.Datastore method), 181
sequence_id (eventsource-ing.infrastructure.sequenceditem.SequencedItem attribute), 193	setup_connection() (eventsource-ing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore method), 196
sequence_id (eventsource-ing.infrastructure.sequenceditem.SequencedItemFieldNameAttribute attribute), 193	setup_django() (in module eventsource-ing.infrastructure.django.utils), 187
sequence_id (eventsource-ing.infrastructure.sqlalchemy.records.IntegerSequencedNoIDRecord attribute), 199	setup_table() (eventsource-ing.application.process.ProcessApplication method), 149
sequence_id (eventsource-ing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord attribute), 199	setup_table() (eventsource-ing.application.simple.SimpleApplication method), 151
sequence_id (eventsource-ing.infrastructure.sqlalchemy.records.SnapshotRecord attribute), 199	setup_table() (eventsource-ing.application.snapshotting.SnapshottingApplication method), 152
sequence_id (eventsource-ing.infrastructure.sqlalchemy.records.TimestampSequencedNoIDRecord attribute), 200	setup_table() (eventsource-ing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore method), 196
sequence_id (eventsource-ing.infrastructure.sqlalchemy.records.TimestampSequencedWithIDRecord attribute), 200	setup_tables() (eventsource-ing.infrastructure.cassandra.datastore.CassandraDatastore method), 178
sequenced_item_class (eventsource-ing.application.simple.SimpleApplication attribute), 151	setup_tables() (eventsource-ing.infrastructure.datastore.Datastore method), 182
sequenced_item_mapper_class (eventsource-ing.application.popo.PopoApplication attribute), 148	setup_tables() (eventsource-ing.infrastructure.sqlalchemy.datastore.SQLAlchemyDatastore method), 196
sequenced_item_mapper_class (eventsource-ing.application.simple.SimpleApplication attribute), 151	shutdown() (eventsource-ing.application.actors.ActorModelRunner method), 144
SequencedItem (class in eventsource-ing.infrastructure.sequenceditem), 193	shutdown_actor_system() (in module eventsource-ing.application.actors), 145
SequencedItemFieldNames (class in eventsource-ing.infrastructure.sequenceditem), 193	SimpleApplication (class in eventsource-ing.application.simple), 150
SequencedItemIterator (class in eventsource-ing.infrastructure.iterators), 191	SimpleIntegerSequenceGenerator (class in eventsource-ing.infrastructure.integersequencegenerators.base), 190
SequencedItemMapper (class in eventsource-ing.infrastructure.sequenceditemmapper),	SingleThreadedRunner (class in eventsource-



- [ing.application.system\), 155](#)
- [SlaveRunRequest \(class in eventsourcing.application.actors\), 144](#)
- [SlaveRunResponse \(class in eventsourcing.application.actors\), 145](#)
- [Snapshot \(class in eventsourcing.domain.model.snapshot\), 174](#)
- [snapshot\\_period \(eventsourcing.application.snapshotting.SnapshottingApplication attribute\), 152](#)
- [snapshot\\_record\\_class \(eventsourcing.application.django.DjangoApplication attribute\), 146](#)
- [snapshot\\_record\\_class \(eventsourcing.application.simple.SimpleApplication attribute\), 151](#)
- [snapshot\\_record\\_class \(eventsourcing.application.sqlalchemy.SQLAlchemyApplication attribute\), 152](#)
- [snapshot\\_record\\_class \(eventsourcing.infrastructure.cassandra.factory.CassandraInfrastructureFactory attribute\), 179](#)
- [snapshot\\_record\\_class \(eventsourcing.infrastructure.django.factory.DjangoInfrastructureFactory attribute\), 182](#)
- [snapshot\\_record\\_class \(eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory attribute\), 197](#)
- [SnapshotRecord \(class in eventsourcing.infrastructure.cassandra.records\), 180](#)
- [SnapshotRecord \(class in eventsourcing.infrastructure.django.models\), 185](#)
- [SnapshotRecord \(class in eventsourcing.infrastructure.sqlalchemy.records\), 199](#)
- [SnapshotRecord.DoesNotExist, 180, 185](#)
- [SnapshotRecord.MultipleObjectsReturned, 180, 185](#)
- [SnapshottingApplication \(class in eventsourcing.application.snapshotting\), 152](#)
- [SnapshottingPolicy \(class in eventsourcing.application.policies\), 147](#)
- [SQLAlchemyApplication \(class in eventsourcing.application.sqlalchemy\), 152](#)
- [SQLAlchemyDatastore \(class in eventsourcing.infrastructure.sqlalchemy.datastore\), 196](#)
- [SQLAlchemyInfrastructureFactory \(class in eventsourcing.infrastructure.sqlalchemy.factory\), 196](#)
- [SQLAlchemyRecordManager \(class in eventsourcing.infrastructure.sqlalchemy.manager\), 197](#)
- [SQLAlchemySettings \(class in eventsourcing.infrastructure.sqlalchemy.datastore\), 196](#)
- [SQLRecordManager \(class in eventsourcing.infrastructure.cassandra.factory\), 177](#)
- [start \(\) \(eventsourcing.application.actors.ActorModelRunner method\), 144](#)
- [start \(\) \(eventsourcing.application.multiprocess.MultiprocessRunner method\), 146](#)
- [start \(\) \(eventsourcing.application.system.InProcessRunner method\), 154](#)
- [start \(\) \(eventsourcing.application.system.MultiThreadedRunner method\), 154](#)
- [start \(\) \(eventsourcing.application.system.SteppingMultiThreadedRunner method\), 156](#)
- [start \(\) \(eventsourcing.application.system.SteppingSingleThreadedRunner method\), 156](#)
- [start \(\) \(eventsourcing.application.system.SystemRunner method\), 156](#)
- [start\\_actor\\_system \(\) \(in module eventsourcing.application.actors\), 145](#)
- [start\\_clock \(\) \(eventsourcing.application.system.MultiThreadedRunner method\), 154](#)
- [start\\_multiproc\\_tcp\\_base\\_system \(\) \(in module eventsourcing.application.actors\), 145](#)
- [start\\_new\\_timebucketedlog \(\) \(in module eventsourcing.domain.model.timebucketedlog\), 175](#)
- [start\\_thread \(\) \(eventsourcing.infrastructure.iterators.ThreadedSequencedItemIterator method\), 192](#)
- [started\\_on \(eventsourcing.domain.model.timebucketedlog.Timebucketedlog attribute\), 175](#)
- [state \(eventsourcing.domain.model.snapshot.AbstractSnapshot attribute\), 174](#)
- [state \(eventsourcing.domain.model.snapshot.Snapshot attribute\), 174](#)
- [state \(eventsourcing.example.interface.flaskapp.IntegerSequencedItem attribute\), 209](#)
- [state \(eventsourcing.infrastructure.cassandra.records.IntegerSequencedRecord attribute\), 180](#)
- [state \(eventsourcing.infrastructure.cassandra.records.SnapshotRecord attribute\), 180](#)
- [state \(eventsourcing.infrastructure.cassandra.records.StoredEventRecord attribute\), 181](#)
- [state \(eventsourcing.infrastructure.cassandra.records.TimestampSequencedRecord attribute\), 181](#)
- [state \(eventsourcing.infrastructure.cassandra.records.TimeuuidSequencedRecord attribute\), 181](#)

[state \(eventsourcing.infrastructure.django.models.EntitySnapshotRecord attribute\), 184](#)  
[state \(eventsourcing.infrastructure.django.models.IntegerSequencedItem attribute\), 184](#)  
[state \(eventsourcing.infrastructure.django.models.SnapshotRecord attribute\), 186](#)  
[state \(eventsourcing.infrastructure.django.models.StoredEventRecord attribute\), 187](#)  
[state \(eventsourcing.infrastructure.django.models.TimestampSequencedItem attribute\), 187](#)  
[state \(eventsourcing.infrastructure.sequenceditem.SequencedItem attribute\), 193](#)  
[state \(eventsourcing.infrastructure.sequenceditem.SequencedItemFieldNames attribute\), 193](#)  
[state \(eventsourcing.infrastructure.sequenceditem.StoredEvent attribute\), 194](#)  
[state \(eventsourcing.infrastructure.sqlalchemy.records.EntitySnapshotRecord attribute\), 198](#)  
[state \(eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedItem attribute\), 199](#)  
[state \(eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord attribute\), 199](#)  
[state \(eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord attribute\), 199](#)  
[state \(eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord attribute\), 200](#)  
[state \(eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedItem attribute\), 200](#)  
[state \(eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedWithIDRecord attribute\), 200](#)  
[SteppingMultiThreadedRunner \(class in eventsourcing.application.system\), 155](#)  
[SteppingRunner \(class in eventsourcing.application.system\), 156](#)  
[SteppingSingleThreadedRunner \(class in eventsourcing.application.system\), 156](#)  
[store \(\) \(eventsourcing.infrastructure.eventstore.AbstractEventStore method\), 189](#)  
[store \(\) \(eventsourcing.infrastructure.eventstore.EventStore method\), 190](#)  
[store\\_event \(\) \(eventsourcing.application.policies.PersistencePolicy method\), 147](#)  
[stored\\_event\\_record\\_class \(eventsourcing.application.django.DjangoApplication attribute\), 146](#)  
[stored\\_event\\_record\\_class \(eventsourcing.application.popo.PopoApplication attribute\), 148](#)  
[stored\\_event\\_record\\_class \(eventsourcing.application.simple.SimpleApplication attribute\), 151](#)  
[stored\\_event\\_record\\_class \(eventsourcing.application.sqlalchemy.SQLAlchemyApplication attribute\), 152](#)  
[StoredEvent \(class in eventsourcing.infrastructure.sequenceditem\), 194](#)  
[StoredEventRecord \(class in eventsourcing.infrastructure.cassandra.records\), 180](#)  
[StoredEventRecord \(class in eventsourcing.infrastructure.django.models\), 186](#)  
[StoredEventRecord \(class in eventsourcing.infrastructure.sqlalchemy.records\), 200](#)  
[StoredEventRecord.DoesNotExist, 180, 186](#)  
[StoredEventRecord.MultipleObjectsReturned, 180, 186](#)  
[StoredEventRepo \(eventsourcing.domain.model.array.AbstractBigArrayRepository attribute\), 159](#)  
[subrepo \(eventsourcing.repositories.array.BigArrayRepository attribute\), 192](#)  
[subscribe \(\) \(in module eventsourcing.domain.model.events\), 173](#)  
[subscribe\\_to \(\) \(in module eventsourcing.domain.model.events\), 164](#)  
[System \(class in eventsourcing.application.system\), 156](#)  
[SystemInitRequest \(class in eventsourcing.application.actors\), 145](#)  
[SystemInitResponse \(class in eventsourcing.application.actors\), 145](#)  
[SystemRunner \(class in eventsourcing.application.system\), 156](#)  
**T**  
[take\\_snapshot \(\) \(eventsourcing.application.policies.SnapshottingPolicy method\), 148](#)  
[take\\_snapshot \(\) \(eventsourcing.domain.model.entity.AbstractEntityRepository method\), 165](#)  
[take\\_snapshot \(\) \(eventsourcing.infrastructure.event sourced repository.EventSourcedRepository method\), 188](#)  
[take\\_snapshot \(\) \(eventsourcing.infrastructure.snapshotting.AbstractSnapshotStrategy method\), 195](#)  
[take\\_snapshot \(\) \(eventsourcing.infrastructure.snapshotting.EventSourcedSnapshotStrategy method\), 196](#)

- `take_snapshots()` (*eventsourcing.application.process.ProcessApplication* method), 149
- `take_snapshots()` (*eventsourcing.application.process.ProcessApplicationWithSnapshotting* method), 149
- `ThreadedSequencedItemIterator` (class in *eventsourcing.infrastructure.iterators*), 192
- `Timebucketedlog` (class in *eventsourcing.domain.model.timebucketedlog*), 174
- `Timebucketedlog.BucketSizeChanged` (class in *eventsourcing.domain.model.timebucketedlog*), 174
- `Timebucketedlog.Event` (class in *eventsourcing.domain.model.timebucketedlog*), 174
- `Timebucketedlog.Started` (class in *eventsourcing.domain.model.timebucketedlog*), 175
- `TimebucketedlogReader` (class in *eventsourcing.infrastructure.timebucketedlog\_reader*), 201
- `TimebucketedlogRepo` (class in *eventsourcing.infrastructure.repositories.timebucketedlog\_repo*), 192
- `TimebucketedlogRepository` (class in *eventsourcing.domain.model.timebucketedlog*), 175
- `TimeSequenceError`, 206
- `timestamp` (*eventsourcing.domain.model.events.EventWithTimestamp* attribute), 173
- `timestamp_from_datetime()` (in module *eventsourcing.domain.model.timebucketedlog*), 175
- `timestamp_long_from_uuid()` (in module *eventsourcing.utils.times*), 203
- `timestamp_sequenced_record_class` (*eventsourcing.infrastructure.cassandra.factory.CassandraInfrastructureFactory* attribute), 179
- `timestamp_sequenced_record_class` (*eventsourcing.infrastructure.django.factory.DjangoInfrastructureFactory* attribute), 182
- `timestamp_sequenced_record_class` (*eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory* attribute), 197
- `TimestampedEntity` (class in *eventsourcing.domain.model.entity*), 168
- `TimestampedEntity.AttributeChanged` (class in *eventsourcing.domain.model.entity*), 168
- `TimestampedEntity.Created` (class in *eventsourcing.domain.model.entity*), 168
- `TimestampedEntity.Discarded` (class in *eventsourcing.domain.model.entity*), 168
- `TimestampedEntity.Event` (class in *eventsourcing.domain.model.entity*), 168
- `TimestampedVersionedEntity` (class in *eventsourcing.domain.model.entity*), 168
- `TimestampedVersionedEntity.AttributeChanged` (class in *eventsourcing.domain.model.entity*), 168
- `TimestampedVersionedEntity.Created` (class in *eventsourcing.domain.model.entity*), 169
- `TimestampedVersionedEntity.Discarded` (class in *eventsourcing.domain.model.entity*), 169
- `TimestampedVersionedEntity.Event` (class in *eventsourcing.domain.model.entity*), 169
- `TimestampSequencedNoIDRecord` (class in *eventsourcing.infrastructure.sqlalchemy.records*), 200
- `TimestampSequencedRecord` (class in *eventsourcing.infrastructure.cassandra.records*), 181
- `TimestampSequencedRecord` (class in *eventsourcing.infrastructure.django.models*), 187
- `TimestampSequencedRecord` (in module *eventsourcing.infrastructure.sqlalchemy.records*), 200
- `TimestampSequencedRecord.DoesNotExist`, 181, 187
- `TimestampSequencedRecord.MultipleObjectsReturned`, 181, 187
- `TimestampSequencedWithIDRecord` (class in *eventsourcing.infrastructure.sqlalchemy.records*), 200
- `TimeuuidedEntity` (class in *eventsourcing.domain.model.entity*), 169
- `TimeuuidedVersionedEntity` (class in *eventsourcing.domain.model.entity*), 169
- `TimeuuidSequencedRecord` (class in *eventsourcing.infrastructure.cassandra.records*), 181
- `TimeuuidSequencedRecord.DoesNotExist`, 181
- `TimeuuidSequencedRecord.MultipleObjectsReturned`, 181
- `TopicInfrastructureFactory` (*eventsourcing.infrastructure.base.AbstractSequencedItemRecordManager* method), 177
- `TopicInfrastructureFactory` (*eventsourcing.infrastructure.base.SQLRecordManager* method), 178
- `topic` (*eventsourcing.domain.model.snapshot.AbstractSnapshot* attribute), 174
- `topic` (*eventsourcing.domain.model.snapshot.Snapshot* attribute), 174
- `topic` (*eventsourcing.example.interface.flaskapp.IntegerSequencedItem* attribute), 209
- `topic` (*eventsourcing.infrastructure.cassandra.records.IntegerSequencedRecord* attribute), 180
- `topic` (*eventsourcing.infrastructure.cassandra.records.SnapshotRecord* attribute), 180

[attribute](#)), 180  
[truncate\\_tables\(\)](#) (eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore method), 178  
[topic](#) (eventsourcing.infrastructure.cassandra.records.StoredEventRecord attribute), 181  
[topic](#) (eventsourcing.infrastructure.cassandra.records.TimestampSequencedRecord attribute), 181  
[topic](#) (eventsourcing.infrastructure.cassandra.records.UUIDSequencedRecord attribute), 181  
[topic](#) (eventsourcing.infrastructure.django.models.EntitySnapshotRecord attribute), 184  
[topic](#) (eventsourcing.infrastructure.django.models.IntegerSequencedRecord attribute), 184  
[topic](#) (eventsourcing.infrastructure.django.models.NotificationTrackingRecord attribute), 185  
[topic](#) (eventsourcing.infrastructure.django.models.SnapshotRecord attribute), 186  
[topic](#) (eventsourcing.infrastructure.django.models.StoredEventRecord attribute), 186  
[topic](#) (eventsourcing.infrastructure.django.models.TimestampSequencedRecord attribute), 187  
[topic](#) (eventsourcing.infrastructure.sequenceditem.SequencedItem attribute), 193  
[topic](#) (eventsourcing.infrastructure.sequenceditem.SequencedItemFieldNames attribute), 193  
[topic](#) (eventsourcing.infrastructure.sequenceditem.StoredEventRecord attribute), 194  
[topic](#) (eventsourcing.infrastructure.sqlalchemy.records.EntitySnapshotRecord attribute), 198  
[topic](#) (eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedRecord attribute), 199  
[topic](#) (eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord attribute), 199  
[topic](#) (eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord attribute), 199  
[topic](#) (eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord attribute), 200  
[topic](#) (eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedRecord attribute), 200  
[topic](#) (eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedWithIDRecord attribute), 201  
[TopicResolutionError](#), 206  
[tracking\\_record\\_class](#) (eventsourcing.infrastructure.base.ACIDRecordManager attribute), 176  
[tracking\\_record\\_class](#) (eventsourcing.infrastructure.django.manager.DjangoRecordManager attribute), 183  
[tracking\\_record\\_class](#) (eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager attribute), 198  
[tracking\\_record\\_field\\_names](#) (eventsourcing.infrastructure.base.ACIDRecordManager attribute), 176  
[TrackingRecordNotFound](#), 206  
[truncate\\_tables\(\)](#) (eventsourcing.infrastructure.cassandra.datastore.CassandraDatastore method), 178  
[truncate\\_tables\(\)](#) (eventsourcing.infrastructure.datastore.Datastore method), 182  
[VersionedEntity](#) (class in eventsourcing.domain.model.entity), 169  
[VersionedEntity.AttributeChanged](#) (class in eventsourcing.domain.model.entity), 169  
[VersionedEntity.Created](#) (class in eventsourcing.domain.model.entity), 169  
[VersionedEntity.Discarded](#) (class in eventsourcing.domain.model.entity), 170  
[VersionedEntity.Event](#) (class in eventsourcing.domain.model.entity), 170  
[write\\_records\(\)](#) (eventsourcing.infrastructure.base.ACIDRecordManager method), 176  
[write\\_records\(\)](#) (eventsourcing.infrastructure.django.manager.DjangoRecordManager method), 183

`write_records()` (*eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager*  
*method*), [198](#)