eventsourcing Documentation

Release 8.3.0

John Bywater

Mar 14, 2021

Contents

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.

This project is hosted on GitHub.

CHAPTER 1

Contents

1.1 Introduction

1.1.1 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.

That's the basic story, at least. In practice, because a sequence of stored events can be queried in only a limited number of ways, to obtain the views of the state of an application that users need when making command requests, several other design patterns are needed to develop useful software. And several other design patterns are needed to ensure reliability, maintainability, and scalability.

1.1.2 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 event-sourced aggregates in a domain driven design. A variety of schemas and technologies can be used for sequencing and storing events, and this library supports several of these possibilities.

To demonstrate how storing and retrieving domain events can be used effectively as a persistence mechanism in an event-sourced application, this library includes base classes for event-sourced domain entities and applications of different kinds. An domain model developed using these classes will not depend on infrastructure ("onion" archictecture). A style is suggested for event-sourced aggregates: to have command methods which "trigger" domain events. Triggered domain events are used to mutate the state of the aggregate, and then stored. The stored events of an aggregate can be retrieved and used to obtain the current state of the aggregate. The stored events of an application can also be propagated and used to project the state of the application into the materialised views needed by users. Stored events can also be processed further by other applications in the same system, and by other systems.

It is also possible to define an entire application, and indeed 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 effecting exactly the same behaviour.

1.1.3 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 can be "partitioned into a separate lightweight framework".

Whilst some advocate that event sourcing is such a simple thing that you can easily roll-your-own event sourcing framework for each project, in practice it turns out reaching understanding and satisfaction (reliability, scalability, maintainability) is a considerably more complicated and costly undertaking than is disclosed by the simple story that is often told.

This library documents the experience of event sourcing across a broad range of projects. It also functions as an informative "jumping off" point for those who want to create their own framework and "own" all their own code. And it is also used by many as a stable and convenient library that can be used to rapidly develop working software that will be used in production.

1.1.4 Register issues

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

1.2 Support options

It has taken a lot of time and effort to create this library. Similarly, it will take some time and effort to understand and acquire the skills to create a well-designed event-sourced system.

Training workshops and development services are available to support your professional use of this library. Community support is also available.

Please also consider supporting the ongoing development and maintenance of this library by making a regular donation.

1.2.1 Professional support

Design 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 John Bywater via the Slack channel for more information about professional support.

1.2.2 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 John Bywater via the Slack channel for more information about training workshops.

1.2.3 Community support

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

- You can ask questions on the Slack channel.
- You can also register issues and requests on our issue tracker.

1.2.4 Support the project

Please follow the Sponsor button on the GitHub project for options.

1.3 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.

1.3.1 Install library

It is recommended always to install into a virtual environment.

You can use pip to install the library with the 'sqlalchemy' option (other package installers are available).

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

See the *Install documentation* for more information about installing the library.

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

1.3.3 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:'
```

1.3.4 Run application

With the library's application class `SQLAlchemyApplication` instances of the `World` aggregate can persisted in the database identified above using SQLAlchemy.

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__()

    # Retrieve aggregate (replay stored events).
```

(continues on next page)

(continued from previous page)

```
    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.list_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 eventsourcing.application.notificationlog import NotificationLogReader
    reader = NotificationLogReader(app.notification_log)
    notifications = reader.read()
    notification_ids = [n['id'] for n in notifications]
```

(continues on next page)

(continued from previous page)

```
    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 b'dinosaurs' not in item.state
        assert b'trucks' not in item.state
        assert b'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.

1.4 Design

The design of the library follows "layered architecture" in that there are distinct and separate layers for interfaces, application, domain and infrastructure. It also follows the "onion" or "hexagonal" or "clean" architecture, in that the domain layer has no dependencies any other layer. The application layer depends on the domain and infrastructure layers, and the interface layer depends only on the application layer.

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

1.4.1 Onion architecture

An 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).

1.4.2 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.

1.5 Features

1.5.1 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. Runnable with single thread, multiprocessing on a single machine, and in a cluster of machines using the actor model. Parallel pipelines are synchronised with causal dependencies.

1.5.2 Additional features

Versioning - allows model changes to be introduced after an application has been deployed. Both domain events and domain entity classes can be versioned. The recorded state of an older version can be upcast to be compatible with a new version. Stored events and snapshots are upcast from older versions to new versions before the event or entity object is reconstructed.

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.

Hash chaining — Sequences of events can be hash-chained, and the entire sequence of events checked for data integrity. Information lost in transit or on the disk from database corruption can be detected. If the last hash can be independently validated, then so can the entire sequence.

Correlation and causation IDs - Domain events can easily be given correlation and causation IDs, which allows a story to be traced through a system of applications.

Compression - reduces the size of stored domain events and snapshots, usually by around 25% to 50% of the original size. Compression reduces the size of data in the database and decreases transit time across a network.

Application-level encryption — encrypts and decrypts stored events and snapshots, 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). This means data will be encrypted in transit across a network ("on the wire") and at disk level including backups ("at rest"), which is a legal requirement in some jurisdictions when dealing with personally identifiable information (PII) for example the EU's GDPR.

Optimistic concurrency control — ensures a distributed or horizontally scaled application doesn't become inconsistent due to concurrent method execution. Leverages optimistic concurrency controls in adapted database management systems.

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.

1.6 Installation guide

It is recommended always to install into a virtual environment.

You can use pip to install the library from the Python Package Index.

```
$ pip install eventsourcing
```

Other package installers are available.

To avoid installing future incompatible releases, when including the library in a list of project dependencies it is recommended to specify at least some of the current version number. Preferences regarding how and where to specify versions of project dependencies vary.

```
eventsourcing<=8.2.99999
```

As an example, the expression above would install the latest version of v8.2.x series of releases, allowing future bug fixes to be installed with point version number increments, whilst avoiding any potentially destabilising additional features introduced with minor version number increments, and also any backwards incompatible changes introduced with major verison number increments.

1.6.1 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.

SQLAlchemy

If you want to store events using 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.

Django

Similarly, if you want to store events using 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.

Cassandra

If you want to store events using Apache Cassandra, then please install with the 'cassandra' option.

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

Ray

If you want to run a system of applications with Ray, then please install with the 'ray' option.

```
$ pip install eventsourcing[ray]
```

Tests

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

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

Docs

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

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

1.7 Domain model layer

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

1.7.1 Domain events

Domain model events occur when something happens in a domain model, perhaps recording a fact directly from the domain, or more generally registering the results of the work of a command method (perhaps a function of facts from the domain).

The library has a base class for domain model events called *DomainEvent*. Domain events objects can be freely constructed from this class. Attribute values of a domain event object are set directly from constructor keyword arguments.

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

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

Domain events are meant to be immutable. And so the attributes of these domain event objects are read-only: new values cannot be assigned to attributes of existing domain event objects.

```
# 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 and inequality. Instances are equal if they have both the same type and the same attributes.

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

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

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

Publish-subscribe

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

The function *publish()* is used to publish events to subscribed handlers. The argument `event` is required.

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

publish([domain_event])
```

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

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

received_events = []

def receive_events(events):
    received_events.extend(events)

def is_domain_event(events):
    return all(isinstance(e, DomainEvent) for e in events)

subscribe(handler=receive_events, predicate=is_domain_event)

# Publish the domain event.
publish([domain_event])

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

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

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

unsubscribe(handler=receive_events, predicate=is_domain_event)

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

Event library

The library has a small collection of domain event subclasses, such as `EventWithOriginatorID`, `EventWithOriginatorVersion`, `EventWithTimestamp`, `EventWithTimeuuid`, `EventWithHash`, `CreatedEvent`, `AttributeChangedEvent`, and `DiscardedEvent`.

Some classes require particular arguments when constructed. An `originator_id` arg is required for `EventWithOriginatorID` to identify a sequence to which the event belongs. An `originator_version` arg is required for `EventWithOriginatorVersion` to position the events in a sequence.

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

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

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

Some of these classes provide useful defaults for particular attributes, such as the `timestamp` of an *EventWithTimestamp* (a `Decimal` value) and the `event_id` (a version 1 `UUID`) of an *EventWithTimeuuid*.

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

assert isinstance(EventWithTimestamp().timestamp, Decimal)

assert isinstance(EventWithTimeuuid().event_id, UUID)
```

The event classes are useful for their distinct type, for example in subscription predicates.

```
from eventsourcing.domain.model.events import (
    CreatedEvent, AttributeChangedEvent, DiscardedEvent
)

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


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


def is_discarded(event):
    return isinstance(event, DiscardedEvent)


assert is_created(CreatedEvent()) is True
assert is_discarded(CreatedEvent()) is False

assert is_created(DiscardedEvent()) is False
assert is_discarded(DiscardedEvent()) is True

assert is_created(DomainEvent()) is False
assert is_discarded(DomainEvent()) is False
```

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):
    """
    Triggered whenever something happens.
    """
```

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

```
class Job(object):

```

(continues on next page)

(continued from previous page)

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

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

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

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

assert done.timestamp > seen.timestamp
```

Deleting events

The general rule is never to delete events.

However, a perfectly adequate solution to storing and deleting personally identifiable information (for example to comply with data protection regulations such as GDPR) is to record encrypted stored events that are not notifiable (and so won't appear in the notification log of an application, and so won't be propagated) and delete these event records when the information need to be deleted. Each instance attribute could be stored as a separate aggregate, or there could be one aggregate holding all the PII for one individual. Store these events atomically with the events that would otherwise include the events. Consider using UUIDv5 to generated UUIDs for these aggregates.

Use the `get_records()` and `delete_record()` methods of a record manager to delete the records of for an aggregate (see record manager documentation for details).

1.7.2 Domain entities

A domain entity is an object that has an identity which provides a thread of continuity. The attributes of a domain entity can change, directly by assignment, or indirectly by calling a method of the object. But the identity does not change.

The library has a base class for domain entities called *`DomainEntity`*. It has an `id` attribute, because all entities are meant to have a constant ID that provides continuity when other attributes change.

In the example below, a domain entity object is constructed with an ID that is a version 4 UUID.

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

entity_id = uuid4()

entity = DomainEntity(id=entity_id)

assert entity.id == entity_id
```

Entity library

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Naming style

The double leading and trailing underscore naming style, seen above, is used consistently in the library's domain entity and event base classes for attribute and method names, so that developers can begin with a clean namespace. The intention is that the library functionality is included in the application by aliasing these library names with names that work within the project's ubiquitous language.

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

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

Entity events

The library's domain entity classes have domain events defined as inner classes: *Event*, *Created*, *AttributeChanged*, *Discarded*.

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

The domain event class *Event* is inherited by the others. The others also inherit from the corresponding library base classes `Created`, `AttributeChanged`, and `Discarded`.

The domain entity's event class *Event* inherits from the base domain event class *DomainEvent* and from *EventWithOriginatorID* so that all events of *DomainEntity* have an `originator_id` attribute.

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

assert issubclass(DomainEntity.Created, CreatedEvent)
assert issubclass(DomainEntity.AttributeChanged, AttributeChangedEvent)
assert issubclass(DomainEntity.Discarded, DiscardedEvent)

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

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

All events of *DomainEntity* need an `originator_id`. *Created* events also need an `originator_topic`. *AttributeChanged* events also need `name` and `value`.

Events of *VersionedEntity* also need an `originator_version`. Events of *TimestampedEntity* generate a current `timestamp` value, unless one is given.

```
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,
)
```

(continues on next page)

(continued from previous page)

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

All the events have a `__mutate__()` method, which can be used to mutate the state of an entity. This is a convenient way to code the "default" or "self" projection of the entity's sequence of events (the projection of the events into the entity itself).

For example, the `__mutate__()` method of an entity's `Created` event mutates "nothing" to an entity instance. The class that is instantiated is determined by the event's `originator_topic` attribute. Although the `__mutate__()` method of an event normally requires a value to be given for the `obj` argument, it is optional for the method on `Created` events. If a value is provided it must be a callable that returns an entity when called, such as a domain entity class. If a domain entity class is given as the `obj` arg, then the event's `originator_topic` will be ignored for the purposes of determining which class to instantiate.

```
entity = created.__mutate__(None)

assert entity.id == entity_id
```

When a `VersionedEntity` is mutated by one of its domain events, the entity version number is set to the event's `originator_version`.

```
assert entity.__version__ == 0

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

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

Similarly, when a `TimestampedEntity` is mutated by one of its events, the `__last_modified__` attribute of the entity is set to the event's `timestamp` value.

Hash-chained events

The library also has entity class `EntityWithHashchain`. It has event classes that inherit from `EventWithHash`.

```
from eventsourcing.domain.model.entity import EntityWithHashchain
from eventsourcing.domain.model.events import EventWithHash


assert issubclass(EntityWithHashchain.Event, EventWithHash)
assert issubclass(EntityWithHashchain.Created, EventWithHash)
assert issubclass(EntityWithHashchain.AttributeChanged, EventWithHash)
assert issubclass(EntityWithHashchain.Discarded, EventWithHash)
```

All the events of `EntityWithHashchain` use SHA-256 to generate an `event_hash` from the event attribute values when constructed for the first time. Events are chained together by `EntityWithHashchain` by constructing each subsequent event to have an attribute `__previous_hash__` which is the `__event_hash__` of the previous event (stored by the entity on entity's `__head__` attribute).

Factory method

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

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


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


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


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

Triggering events

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

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

For example, triggering an `AttributeChangedEvent` 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__
```

Changing attributes

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_events, 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_events, predicate=is_domain_event)
del received_events[:]  # received_events.clear()
```

Discarding entities

The command method `__discard__()` triggers a `Discarded` event, after which the entity is unavailable for further changes.

```
from eventsourcing.exceptions import EntityIsDiscarded

entity.__discard__()

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

Custom entities

The library entity classes can be subclassed.

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

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

Custom attributes

The library function `attribute()` is a decorator that provides a property getter and setter. It will trigger an `AttributeChanged` event when a value is assigned to the property. Simple mutable attributes can be coded as decorated functions without a body (any body is ignored) such as `full_name` 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):
        """
        The full name of the user (an event-sourced attribute).
        """
```

In the code below, after the entity has been created, assigning to `full_name` triggers an `AttributeChanged`. A `Created` event and an `AttributeChanged` event are received by a subscriber.

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

# Publish a Created event.
user = User.__create__(full_name='Mrs Boots')

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

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

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

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

Custom commands

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

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

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


class User(VersionedEntity):

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

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

        # Change private _password attribute.
        self.__change_attribute__('_password', password)

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

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


user = User(id='1', __version__=0)

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

Custom events

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

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

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

```
class World(VersionedEntity):

    def __init__(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        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):
        """Triggered when something happens in the world."""
        def mutate(self, obj):
            obj.history.append(self.what)
```

A new "world" entity can now be created, using the class method `__create__()`. The entity 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] == 'dinosaurs'
assert world.history[1] == 'trucks'
assert world.history[2] == 'internet'
```

Auto-subclassing events

In order to distinguish between events of different entity classes that inherit their events from a common entity base class, it is necessary to subclass the event classes on each of the entity classes.

Without subclassing the domain events of an inherited entity class, the custom entity classes will have exactly the same domain event classes.

```
class Example1(DomainEntity):
    pass


class Example2(DomainEntity):
    pass


assert Example1.Event == Example2.Event
assert Example1.Created  == Example2.Created
assert Example1.Discarded  == Example2.Discarded
assert Example1.AttributeChanged  == Example2.AttributeChanged
```

With subclassing the domain events of an inherited entity class, the custom entity classes will have distinct domain event classes.

```
class Example3(DomainEntity):
    class Event(DomainEntity.Event): pass
```

(continues on next page)

(continued from previous page)

```
    class Created(Event, DomainEntity.Created): pass
    class Discarded(Event, DomainEntity.Discarded): pass
    class AttributeChanged(Event, DomainEntity.AttributeChanged): pass
    class SomethingHappened(Event): pass


class Example4(DomainEntity):
    class Event(DomainEntity.Event): pass
    class Created(Event, DomainEntity.Created): pass
    class Discarded(Event, DomainEntity.Discarded): pass
    class AttributeChanged(Event, DomainEntity.AttributeChanged): pass
    class SomethingHappened(Event): pass


assert Example3.Event != Example4.Event
assert Example3.Created != Example4.Created
assert Example3.Discarded != Example4.Discarded
assert Example3.AttributeChanged != Example4.AttributeChanged
```

Some people will like to make explict the event subclasses. However, some people will find this cumbersome "boilerplate".

To avoid the appearance of "boilerplate", it is possible to achieve exactly the same distinct event subclasses, as above, by decorating the entity class with the `@subclassevents` decorator. In this case, custom events need only to inherit from the base `DomainEvent` class, and will then be subclassed automatically as an `Event` of the custom entity class (which will be defined first, if missing).

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


@subclassevents
class Example5(DomainEntity):
    class SomethingHappened(DomainEvent):
        pass


@subclassevents
class Example6(DomainEntity):
    class SomethingHappened(DomainEvent):
        pass


assert Example5.Event != Example6.Event
assert Example5.Created != Example6.Created
assert Example5.Discarded != Example6.Discarded
assert Example5.AttributeChanged != Example6.AttributeChanged

assert issubclass(Example5.SomethingHappened, Example5.Event)
assert issubclass(Example6.SomethingHappened, Example6.Event)
```

To avoid having to use the decorator on all of the custom entity classes in a model, which may itself start to feel like "boilerplate", it is possible to set `__subclassevents__` on a common custom base entity class.

```
class BaseEntity(DomainEntity):
    __subclassevents__ = True
```

(continues on next page)

(continued from previous page)

```
class Example5(BaseEntity):
    class SomethingHappened(DomainEvent):
        pass


class Example6(BaseEntity):
    class SomethingHappened(DomainEvent):
        pass


assert Example5.Event != Example6.Event
assert Example5.Created != Example6.Created
assert Example5.Discarded != Example6.Discarded
assert Example5.AttributeChanged != Example6.AttributeChanged

assert issubclass(Example5.SomethingHappened, Example5.Event)
assert issubclass(Example6.SomethingHappened, Example6.Event)
```

1.7.3 Aggregate root

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

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

Therefore,

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

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

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

Therefore, to implement the aggregate as a consistency boundary, all the events from a command on an aggregate must be appended to the event store in a single atomic transaction, so that if some of the events resulting from executing a command cannot be stored then none of them will be stored. If all the events from an aggregate are to be written to a database as a single atomic operation, then they must have been published by the entity as a single list.

Base class

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

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


class World(BaseAggregateRoot):
    """
    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(BaseAggregateRoot.Event):
        def mutate(self, obj):
            obj.history.append(self.what)
```

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_events)

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

Events are pending, and will not be published until `__save__()` 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) == 4
```

(continues on next page)

(continued from previous page)

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

Data integrity

The library class *AggregateRootWithHashchainedEvents* extends *BaseAggregateRoot* by also inheriting from *EntityWithHashchain*, so that aggregate events are individually hashed and also hash-chained together. It is "aliased" as *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.what)


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

# Publish pending events.
world.__save__()
```

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.
```

(continues on next page)

(continued from previous page)

```
assert world.__head__ == received_events[-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 eventsourcing.utils.random import encoded_random_bytes

# Keep this safe.
salt = encoded_random_bytes(num_bytes=32)

# 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`.

The class *`AggregateRootWithHashchainedEvents`* can be used when you want to be able to verify aggregates' sequences of events cryptographically (which can be useful even during development to catch programming errors and to avoid doubt that the infrastructure is working properly). However, the class *`BaseAggregateRoot`* is probably faster and can be used whenever you don't actually need to verify the sequence of events cryptographically.

```
# Clean up after running examples.
unsubscribe(handler=receive_events)
del received_events[:]  # received_events.clear()
```

1.7.4 Versioning

The library class *`Upcastable`* supports versioning. This class is inherited by all of the domain event classes in the library, so that custom event classes can be versioned. It is also inherited by the domain entity classes, so that custom entity classes can be versioned (snapshots can be upcast).

Versioning events

As changes are made to an event class, the class attribute `__class_version__` can be incremented through a series of integer values. If the `__class_version__` is a non-zero value, it will be included in the recorded states of all instances of the event class. The original value is `0` and so the first time this attribute is set on a custom event class, the attribute should be set to `1`.

If the event class attribute `__class_version__` has a non-zero value, when the event class method *`__upcast_state__()`* is called, the event class method *`__upcast__()`* will be called successively, once for each version, starting from the version of the stored event state, until the current version is reached.

By default, *`__upcast__()`* raises a `NotImplementedError` exception. And so if the `__class_version__` of a custom event class has a non-zero value, then this method will need to be overridden on the custom event class, and implemented to support upcasting from the original version `0` to version `1`. The next time the event class is changed, the class version number will need to be set to `2`, and the custom `__upcast__()` method amended so that it supports both upcasting from version `0` to version `1` and additionally from version `1` to version `2`. And so on.

```
from copy import copy

# Original version.
class ExampleEvent(DomainEvent):
    pass

# Construct state with original version of the event class.
state_v0 = ExampleEvent(a=1).__dict__
assert state_v0["a"] == 1

# Check version 1 is correctly upcast to version 1.
state_v0_from_v0 = ExampleEvent.__upcast_state__(copy(state_v0))
assert state_v0_from_v0["a"] == 1

# Version 1 (has attribute 'b').
class ExampleEvent(DomainEvent):
    __class_version__ = 1

    @classmethod
    def __upcast__(cls, obj_state, class_version):
        if class_version == 0:
            # Supply default for 'b'.
            obj_state['b'] = 0
        return obj_state

# Construct state with version 1 of the event class.
state_v1 = ExampleEvent(a=1, b=2).__dict__
assert state_v1["a"] == 1
assert state_v1["b"] == 2

# Check original version is correctly upcast to version 1.
state_v1_from_v0 = ExampleEvent.__upcast_state__(copy(state_v0))
assert state_v1_from_v0["a"] == 1
assert state_v1_from_v0["b"] == 0  # gets default value

# Check version 1 is correctly upcast to version 1.
state_v1_from_v1 = ExampleEvent.__upcast_state__(copy(state_v1))
assert state_v1_from_v1["a"] == 1
assert state_v1_from_v1["b"] == 2

# Version 2 (has attribute 'c').
class ExampleEvent(DomainEvent):
    __class_version__ = 2

    @classmethod
    def __upcast__(cls, obj_state, class_version):
        if class_version == 0:
            # Supply default for 'b'.
            obj_state['b'] = 0
        elif class_version == 1:
            # Supply default for 'c'.
            obj_state['c'] = ''
        return obj_state

# Construct state with version 2 of the event class.
state_v2 = ExampleEvent(a=1, b=2, c='c').__dict__
assert state_v2["a"] == 1
```

(continues on next page)

(continued from previous page)

```
assert state_v2["b"] == 2

# Check original version is correctly upcast to version 2.
state_v2_from_v0 = ExampleEvent.__upcast_state__(copy(state_v0))
assert state_v2_from_v0["a"] == 1
assert state_v2_from_v0["b"] == 0  # gets default value
assert state_v2_from_v0["c"] == ''  # gets default value

# Check version 1 is correctly upcast to version 2.
state_v2_from_v1 = ExampleEvent.__upcast_state__(copy(state_v1))
assert state_v2_from_v1["a"] == 1
assert state_v2_from_v1["b"] == 2
assert state_v2_from_v1["c"] == ''  # gets default value

# Check version 2 is correctly upcast to version 2.
state_v2_from_v2 = ExampleEvent.__upcast_state__(copy(state_v2))
assert state_v2_from_v2["a"] == 1
assert state_v2_from_v2["b"] == 2
assert state_v2_from_v2["c"] == 'c'
```

Please refer to the *Upcastable* documentation for more information about versioning events, especially about restrictions involved when providing for forward compatibility, and when you might need to do that.

Versioning entities

When reconstructing domain entities from stored event records, for example when retrieving aggregates from an application repository, the sequenced item mapper calls the library function *reconstruct_object()* which calls the event class method *__upcast_state__()*, as above. This is the only place in the library where *__upcast_state__()* is called.

Care needs to be taken when using both snapshotting and versioning, since differences introduced by newer versions of events, and changes to an entity class since a snapshot was made might not exist in the snapshot, and that might matter.

One option is to delete snapshots created by a previous version of the class. New snapshots will need to be made. Suddenly stopping use of old snapshots, and so replaying all the stored events to create a new snapshot, would briefly degrade performance to the extent it was improved by using snapshots.

Another option is upcasting the snapshotted state. The domain entity classes are also `Upcastable` classes, and so it is possible to override the `__upcast__()` method on the entity class, which will be called when reconstructing an entity from a snapshot. The body of this implementation needs to manipulate state of the snapshot to conform with the state that would be obtained by reconstructing using the upgraded event versions. This can help in simple cases, but there may cases where the correct state cannot be obtained in this way. The class attribute `__class_version__` is used to define the version of the entity class (with integer version numbers 1, 2, etc).

The example below shows a custom domain entity class, which upcasts snapshotted state by adding default values for ‘value’ and ‘units’. This class gestures towards having been defined originally without either attribute. It is supposed that version 1 added the ‘value’ attribute, and the ‘units’ attribute was added in version 2.

A snapshot of the state of an original version of the entity wouldn’t have ‘value’, and so upcasting from the original version to version 1 involves defining ‘value’. A snapshot of the state of version 1 of the entity woud have ‘value’ but wouldn’t have ‘units’, and so upcasting from version 1 to version 2 involves defining ‘units’.

```
class ExampleAggregate(BaseAggregateRoot):
    __class_version__ = 2

```

(continues on next page)

(continued from previous page)

```
    DEFAULT_VALUE = 0
    DEFAULT_UNITS = ""

    def __init__(self, **kwargs):
        self.value = self.DEFAULT_VALUE  # added in version 1
        self.units = self.DEFAULT_UNITS  # added in version 2

    @classmethod
    def __upcast__(cls, obj_state, class_version):
        if class_version == 0:
            # Upcast to version 1.
            obj_state['value'] = cls.DEFAULT_VALUE
        elif class_version == 1:
            # Upcast to version 2.
            obj_state['units'] = cls.DEFAULT_UNITS
        return obj_state
```

Copy and replace

Copy-and-replace is an alternative to upcasting. It is possible to accumulate so many changes that it becomes desirable to replace the old versions of stored events with new versions.

1.8 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.

1.8.1 Sequenced items

Sequenced item types are declared as Python named tuples. The example below is a sequenced item type with four fields.

```
from collections import namedtuple

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

The field names are arbitrary, 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.

SequencedItem

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

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

Like in the example above, the library's *SequencedItem* 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
).encode('utf8')
```

(continues on next page)

(continued from previous page)

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

StoredEvent

The library also provides a sequenced item type called *StoredEvent*. Its attributes are originator_id, originator_version, topic, and state.

The originator_id is perhaps the ID of a domain entity that triggered the event, and is equivalent to sequence_id above. The originator_version could be the version of a domain entity that triggered 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()

state = (
    '{"foo":"bar","originator_version":0,"originator_id":{"UUID":"%s"}}' % aggregate1.
↪hex
).encode('utf8')

stored_event1 = StoredEvent(
    originator_id=aggregate1,
    originator_version=0,
    topic='eventsourcing.domain.model.events#DomainEvent',
    state=state,
)
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 == state
```

1.8.2 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 eventsourcing.domain.model.events import DomainEvent

domain_event1 = DomainEvent(
    originator_id=aggregate1,
    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 == aggregate1
```

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 eventsourcing.domain.model.events import CreatedEvent
from eventsourcing.utils.topic import get_topic, resolve_topic

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

Substitutions

The module `eventsourcing.utils.topic` has a module level `dict` called `substitutions` which can be configured to substitute one topic for another. If an entity or event is moved or renamed, then any stored events that refer to the old position will fail to resolve, unless a mapping from the old topic to the new topic is added to the `substitutions` dict.

```
from eventsourcing.utils.topic import substitutions


substitutions['old_topic'] = 'new_topic'
```

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 shows how to extend the JSON transcoding to support sets. The library now supports encoding and decoding sets, but the example is still demonstrative.

```
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,
```

(continues on next page)

(continued from previous page)

```
    json_decoder_class=CustomObjectJSONDecoder,
    sequenced_item_class=StoredEvent,
)
state = (
    '{"foo":{"__set__":["bar","baz"]},"originator_version":0,"originator_id":{"UUID":"
↪%s"}}' % sequence1.hex
).encode('utf8')
domain_event = customized_sequenced_item_mapper.event_from_item(
    StoredEvent(
        originator_id=sequence1,
        originator_version=0,
        topic='eventsourcing.domain.model.events#DomainEvent',
        state=state,
    )
)
assert domain_event.foo == set(["bar", "baz"])

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

It is also possible to extend the encoder and decoder classes by registering encode and decode functions using function decorators. This is a more convenient way to add support for particular types.

```
from eventsourcing.utils.transcoding import encoder, decoder

@encoder.register(set)
def encode_set(obj):
    return {"__set__": sorted(list(obj))}


@decoder.register("__set__")
def decode_set(d):
    return set(d["__set__"])
```

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 library contains 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
```

(continues on next page)

(continued from previous page)

```
# 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(b'plaintext')
assert ciphertext != b'plaintext'

# Decrypt some ciphertext.
plaintext = cipher.decrypt(ciphertext)
assert plaintext == b'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 b'foo' not in stored_event.state
assert b'bar' not in stored_event.state

# Recover the domain event from the encrypted state.
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.

1.8.3 Record managers

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

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

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

SQLAlchemy

The library class `SQLAlchemyRecordManager` is a record manager for SQLAlchemy.

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 class `IntegerSequencedRecord` 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, if 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 a 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 `SQLAlchemyDatastore` has a thread-scoped session facade set as its a `session` attribute. You may wish to use a different scoped session facade, such as a request-scoped session object provided by a Web framework.

With the database setup, an `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,
```

(continues on next page)

(continued from previous page)

```
    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 `record_items()` method of the record manager.

```
record_manager.record_item(stored_event1)
```

All the previously appended items of a sequence can be retrieved by using the `list_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 insfrastructure defaults to using an in memory SQLite database, which is the fastest way to run the library, and is recommended as a convenience for development.

Django ORM

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

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

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

For the *`DjangoRecordManager`*, the *`IntegerSequencedRecord`* 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_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.

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 being 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.

Cassandra

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

```
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.records`* 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 was as `SQLAlchemyDatastore` and `SQLAlchemySettings` above. Please investigate library class *`CassandraSettings`* for information about configuring away from default settings.

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

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

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

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

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

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

cassandra_record_manager.record_item(stored_event1)

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

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

Sequenced item conflicts

It is a common feature of the record manager classes that it isn't possible successfully to append two items at the same position in the same sequence. If such an attempt is made, a `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_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.

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

```
from eventsourcing.infrastructure.eventstore import EventStore

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

The method *store_events()* can store a domain event in its sequence. The event store uses its `sequenced_item_mapper` to obtain sequenced items (named tuple) from domain events, and it uses its `record_manager` to record sequenced items in the database.

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

```
event_store.store_events([
    DomainEvent(
        originator_id=aggregate1,
        originator_version=1,
        foo='baz',
    )
])
```

The method *list_events()* can be used to get events that have previously been stored. The event store uses its `record_manager` to get the sequenced items from database records, and it uses its `sequenced_item_mapper` to obtain domain events from the sequenced items.

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

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

```
assert len(results) == 2

assert results[0].foo == 'bar'
assert results[1].foo == 'baz'
```

The optional arguments of *list_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.list_events(aggregate1, lte=0)
assert len(result) == 1, result
assert result[0].foo == 'bar'

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

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

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

Optimistic concurrency control

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

```
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_events([
        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 method `record_items`. 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 decorator `retry()` can help code retries on commands.

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

errors = []

@retry(ConcurrencyError, max_attempts=5)
def set_password():
```

(continues on next page)

(continued from previous page)

```
    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.

1.8.5 Infrastructure factory

To help with construction of infrastructure objects, the library has a various infrastructure factory classes. The abstract base class `InfrastructureFactory` defines the common method signatures. The concrete subclass `SQLAlchemyInfrastructureFactory` helps with construction of SQLAlchemy infrastructure. Similarly `DjangoInfrastructureFactory` helps with Django infrastructure and `CassandraInfrastructureFactory` helps with Cassandra.

1.9 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*.

1.9.1 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 eventsourcing[sqlalchemy]
```

1.9.2 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.

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

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

# PostgreSQL with psycopg2.
```

(continues on next page)

(continued from previous page)

```
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 encoded_random_bytes

# Keep this safe (random bytes encoded with Base64).
cipher_key = encoded_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`.

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)
```

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.

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.eventsourcedrepository 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.

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.application.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__()

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

1.9.3 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)
```

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")
```

Stored events

You can list the domain events of an aggregate by using the method *list_events()* of the event store of the application.

```
events = application.event_store.list_events(originator_id=aggregate.id)
assert len(events) == 4

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

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

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

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

Sequenced items

It is also possible to get the sequenced item namedtuples for an aggregate, by using the method *get_items()* of the event store's record manager.

```
items = 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.decode('utf8'), items[0].state
assert b'"timestamp":' in items[0].state

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

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

assert items[3].originator_id == aggregate.id
assert items[3].topic == 'eventsourcing.domain.model.aggregate#AggregateRoot.Discarded
↪'
assert b'"timestamp":' in items[3].state
```

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

Database records

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

```
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, len([r.originator_id for r in event_records])
```

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()
```

1.10 Notifications

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.

1.10.1 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.

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's work is independent of the others', then each could have 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.

If partitioning the aggregates of application is restrictive, perhaps because each user's work is dependent on the others', then it might 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.

Another possibility would be to sequence the lists of events published when saving the pending events of an aggregate (see *BaseAggregateRoot*), 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 for each command in a sequence. Sequencing the lists would allow these little units of coherence to be propagated atomically, which may be useful in some cases. (Please note, the library doesn't currently support atomic notification of collections of events, instead each event is notified individually.)

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,
    event_mapper=sequenced_item_mapper
```

(continues on next page)

(continued from previous page)

```
)

# Set up a persistence policy.
persistence_policy = PersistencePolicy(
    event_store=event_store,
    persist_event_type=DomainEntity.Event
)
```

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

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

1.10.2 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 `BaseAggregateRoot` 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. However since this library uses one table with two indexes for the aggregate and application sequence, perhaps the greatest benefit would be processing these atomically a list of events that have been created atomically. That might avoid projections being in an intermediate state such that a user could view only some of the effects of an action when that would be alarming. This isn't implemented at time of writing.

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 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.

Integers

To reproduce an application's sequence of events perfectly, without any risk of gaps or duplicates or jumbled items, or race conditions, it seems that we need to 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 have 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 can 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.

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 class *RecordManagerNotificationLog* (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 be 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 = list(record_manager.get_notifications())

assert len(notifications) == 0, notifications

first_entity = VersionedEntity.__create__()

notifications = list(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.

BigArray

This is a long section, and can be skipped if you aren't currently trying 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.

```python
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 arger the array size, the less often the base arrays fill up. For a decent array size, the probability of needing to build the tree is very low. And when the tree does need building, it doesn't take very long (and most of it probably already exists).

```
from eventsourcing.exceptions import ConcurrencyError

assert big_array.get_next_position() == 4

big_array[4] = 'item4'
try:
    big_array[4] = 'item4a'
except ConcurrencyError:
    pass
else:
    raise
```

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

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

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

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

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

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

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

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

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

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

integers = RedisIncr()
generated = []
for i in integers:
    generated.append(i)
    if i >= 4:
        break

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

The integer sequence generator can be used when assigning items to the 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.

1.10.3 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.

1.10.4 Local notification logs

RecordManagerNotificationLog

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

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

```
from eventsourcing.application.notificationlog import RecordManagerNotificationLog

# Construct notification log.
```

(continues on next page)

(continued from previous page)

```
notification_log = RecordManagerNotificationLog(record_manager, section_size=5)

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

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

The notifications (`section.items`) from a *`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 the *`RecordManagerNotificationLog`* class.

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 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 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 casual 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 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.

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.application.notificationlog import BigArrayNotificationLog

```

(continues on next page)

(continued from previous page)

```
# 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, 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.

1.10.5 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.

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 a `json_encoder` (for example an instance of the library's *ObjectJSONEncoder* class).

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

```
import json

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

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

section_json = view.present_resource('1,5')

section_dict = ObjectJSONDecoder().decode(section_json.decode('utf8'))

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'] == 'eventsourcing.domain.model.entity#VersionedEntity.Created'

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

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

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

Notification API

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

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

```
def notification_log_wsgi(environ, start_response):

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

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

    # Get serialized section.
    section = view.present_resource(section_id)

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

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

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

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

RemoteNotificationLog

The library class `RemoteNotificationLog` can be used in the same way as the local notification logs above. The difference is that rather than accessing a database using a 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 class `ObjectJSONDecoder`. This encoder matches the default `json_encoder_class` of the library class `NotificationLogView`, which is also the library class `ObjectJSONDecoder`. 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 was as the local notification logs described above, returning section objects for section IDs using the square brackets syntax.

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

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

1.10.6 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 class *NotificationLogReader*, and would have to be added in a subclass or client object.

The class *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.

```python
from eventsourcing.application.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

# 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
```

(continues on next page)

(continued from previous page)

```
# 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()
```

1.11 Projections

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 ways of implementing a projection. This section discusses three approaches: subscribing directly to domain events as are they are published from the domain model of an event sourced application; reading a notification log which presents recorded domain events of an application in a serial order; and tracking a notification log with tracking records that are persisted in the same atomic transaction as the projected state.

1.11.1 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 by 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".

1.11.2 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 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.

1.11.3 Reading event notifications

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 `ProcessApplication` class uses tracking records.

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 events into a replica.

In the example below, the `SimpleApplication` class is used, which has a `RecordManagerNotificationLog` as its `notification_log`. Reading this log will yield all the event records persisted by the `SimpleApplication`.

A record manager notification log object represents stored as event notifications. Each event notification has a notification ID which can be used to position the replicated domain event record in a sequence. Therefore the maximum notification log ID in the replica can be used to determine the current position in the original application's notification log, which gives "exactly once" processing.

```
from eventsourcing.application.notificationlog import NotificationLogReader,␣
↪RecordManagerNotificationLog
from eventsourcing.application.sqlalchemy import SQLAlchemyApplication
from eventsourcing.exceptions import ConcurrencyError
from eventsourcing.domain.model.aggregate import AggregateRoot


# Define the "replicator".
class Replicator(object):
    def __init__(self, original, replica):
        self.reader = NotificationLogReader(original.notification_log)
        self.mapper = replica.event_store.event_mapper
        self.manager = replica.event_store.record_manager
        # Position reader at max record ID.
        self.reader.seek(self.manager.get_max_notification_id() or 0)

    def run(self):
        for event_notification in self.reader.read():
            # Get the notification ID.
            notification_id = event_notification['id']

            # Reconstruct the domain event from the notification.
            domain_event = self.mapper.event_from_notification(event_notification)

            # Serialise the reconstructed domain event to a "sequenced item".
            sequenced_item = self.mapper.item_from_event(domain_event)

            # Convert the sequenced item to a stored event record.
            record = self.manager.to_record(sequenced_item)

            # Set the notification ID on the stored event record object.
            record.notification_id = notification_id

            # Write the stored event record object into the database.
            self.manager.write_records([record])
```

(continues on next page)

(continued from previous page)

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

# Construct replica application.
replica = SQLAlchemyApplication()

# Construct replicator.
replicator = Replicator(original, replica)

# 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.run()

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

# Resume pulling records.
replicator.run()

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

(continues on next page)

(continued from previous page)

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

# Create new replicator.
replicator = Replicator(original, replica)

# 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.run()

# 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.run()

# 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 record conflict 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.

As we will see later on, using "noop" to track position in an upstream sequence might be inefficient, and the tracking purpose be supported directly by using tracking records separately from the recording of new domain events. In fact, that is how the process applications work.

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

    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
```

(continues on next page)

(continued from previous page)

```
    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, original, index):
        self.reader = NotificationLogReader(original.notification_log)
        self.repository = index.repository
        self.mapper = index.event_store.event_mapper
        self.manager = index.event_store.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 used to project that into
        #   many different sequences, in order words splitting the stream
        #   (splitting is different from replicating many time). When splitting
        #   the stream, the splits's record ID couldn't be used to position to␣
↪splitter
        #   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
```

(continues on next page)

(continued from previous page)

```
        # Todo: Bring out different projectors: splitter (one-many), combiner (many-
↪one), repeater (one-one).
        self.reader.seek(self.manager.get_max_notification_id() or 0)

    def run(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 explicity, 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 = self.mapper.event_from_notification(notification)
                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(original, index)

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

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
```

(continues on next page)

(continued from previous page)

```
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
```

1.11.4 Reliable projections

To make projections reliable with respect to sudden restarts, make sure to record the position in the notification log of the last processed domain event notification along with the state of the projection that results from processing that domain event, all in the same atomic transaction. When restarting, the tracking records can be used to position the notification reader in the notification log sequence.

Looking ahead to the next section on *distributed systems*, the `save_orm_obj()` method of the `WrappedRepository` object, which is passed into the `policy()` function of a *`ProcessApplication`* object as the `repository` argument, can be used to persist custom ORM records atomically with tracking information. Instead of calling, for example `obj.save()` on a Django ORM object, which would tend to record the state of the ORM in a separate transaction from the tracking information of the “process event”, making the state of the projection vulnerable to sudden restarts, the ORM obj can be included in the “process event” by passing it as an argument to `save_orm_obj()`. Similarly, the method `delete_orm_obj()` can be used to delete custom ORM object records.

This code example shows how it can be done with SQLAlchemy.

```
from eventsourcing.application.process import ProcessApplication
from eventsourcing.application.decorators import applicationpolicy
from eventsourcing.domain.model.aggregate import AggregateRoot
from eventsourcing.infrastructure.sqlalchemy.records import Base
from sqlalchemy import Column, Text
from sqlalchemy_utils import UUIDType


# Define a custom ORM object for "reliable projection" records.
class ProjectionRecord(Base):
    __tablename__ = "projections"
```

(continues on next page)

(continued from previous page)

```
    # Projection ID.
    projection_id = Column(UUIDType(), primary_key=True)

    # Some values.
    state = Column(Text())


# Define a process application to do "reliable projections".
class ProjectionApplication(SQLAlchemyApplication, ProcessApplication):

    # This is just so we can __save__() AggregateRoot events.
    persist_event_type = AggregateRoot.Event

    # Define process application policy, to create a projection
    # record whenever an aggregate is created, and to delete the
    # projection record whever an aggregate is discarded.
    @applicationpolicy
    def policy(self, repository, event):
        """Do nothing by default."""

    @policy.register(AggregateRoot.Created)
    def _(self, repository, event):
        projection_record = ProjectionRecord(projection_id=event.originator_id)
        repository.save_orm_obj(projection_record)

    @policy.register(AggregateRoot.Discarded)
    def _(self, repository, event):
        projection_record = self.session.query(ProjectionRecord).get(event.originator_
↪id)
        repository.delete_orm_obj(projection_record)


# Start the process application.
with ProjectionApplication(setup_table=True) as app:

    # Setup the projections table.
    app.datastore.setup_table(ProjectionRecord)

    # Configure the process application to follow itself.
    app.follow(app.name, app.notification_log)

    # Create a new aggregate.
    aggregate = AggregateRoot.__create__()
    aggregate.__save__()
    assert aggregate.id in app.repository

    # Check the projection record does not exist.
    assert app.session.query(ProjectionRecord).get(aggregate.id) is None

    # Run the process application policy (creates projection record).
    app.run()

    # Check the project record has been created.
    assert app.session.query(ProjectionRecord).get(aggregate.id) is not None

    # Discard the aggregate.
```

(continues on next page)

(continued from previous page)

```
    aggregate.__discard__()
    aggregate.__save__()

    # Run the process application policy (deletes projection record).
    app.run()

    # Check the projection record has been deleted.
    assert app.session.query(ProjectionRecord).get(aggregate.id) is None
```

This code example shows how it can be done with Django.

```
import os

import django
from django.db import connections, models
from django.core.management import call_command
from eventsourcing.application.process import ProcessApplication
from eventsourcing.application.decorators import applicationpolicy
from eventsourcing.domain.model.aggregate import AggregateRoot
from eventsourcing.application.django import DjangoApplication

os.environ['DJANGO_SETTINGS_MODULE'] = 'eventsourcing.tests.djangoproject.
↪djangoproject.settings'
django.setup()


call_command("migrate", verbosity=0, interactive=False)


# Define a custom ORM object for "reliable projection" records.
class ProjectionRecord(models.Model):
    projection_id = models.UUIDField()
    state = models.TextField()

    class Meta:
        db_table = "projections"
        app_label = "projections"
        managed = False

# Setup the projections table.
with connections["default"].schema_editor() as schema_editor:
    try:
        schema_editor.delete_model(ProjectionRecord)
    except django.db.utils.ProgrammingError:
        pass
with connections["default"].schema_editor() as schema_editor:
    schema_editor.create_model(ProjectionRecord)


# Define a process application to do "reliable projections".
class ProjectionApplication(DjangoApplication, ProcessApplication):

    # This is just so we can __save__() AggregateRoot events.
    persist_event_type = AggregateRoot.Event

    # Define process application policy, to create a projection
    # record whenever an aggregate is created, and to delete the
```

(continues on next page)

(continued from previous page)

```
    # projection record whever an aggregate is discarded.
    @applicationpolicy
    def policy(self, repository, event):
        """Do nothing by default."""

    @policy.register(AggregateRoot.Created)
    def _(self, repository, event):
        projection_record = ProjectionRecord(projection_id=event.originator_id)
        repository.save_orm_obj(projection_record)

    @policy.register(AggregateRoot.Discarded)
    def _(self, repository, event):
        projection_record = ProjectionRecord.objects.get(projection_id=event.
↪originator_id)
        repository.delete_orm_obj(projection_record)


# Start the process application.
with ProjectionApplication() as app:

    # Configure the process application to follow itself.
    app.follow(app.name, app.notification_log)

    # Create a new aggregate.
    aggregate = AggregateRoot.__create__()
    aggregate.__save__()
    assert aggregate.id in app.repository

    # Check the projection record does not exist.
    assert not list(ProjectionRecord.objects.filter(projection_id=aggregate.id))

    # Run the process application policy (creates projection record).
    app.run()

    # Check the project record has been created.
    assert list(ProjectionRecord.objects.filter(projection_id=aggregate.id))

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

    # Run the process application policy (deletes projection record).
    app.run()

    # Check the projection record has been deleted.
    assert not list(ProjectionRecord.objects.filter(projection_id=aggregate.id))
```

Please note, `save_orm_obj()` doesn't immediately update the database, but instead appends the ORM object to a list of ORM objects that will be passed down the stack and recorded within a single transaction along with other factors of the process event after the policy function has returned. This method can be called many times within a single policy function, but only needs to be called once per object. All changes made within a policy function to an ORM object that has been so included within a process event will be persisted after the policy function returns, regardless of whether the changes were made before or after calling `save_orm_obj()`. Support for writing custom ORM objects as a factor of an atomic "process events" is implemented in the library's `SQLAlchemy` and `Django` infrastructure.

Using custom ORM objects of a particular kind (ie. Django or SQLAlchemy) in a system of process applications renders that system dependent on particular infrastructure, and so it won't be possible using this technique to define an entire system of process applications independently of infrastructure. It wouldn't be so very hard to develop some non-event sourced projection model classes that are independent of infrastructure, but the library doesn't include any support for that at the moment. Nevertheless, it is still possible to run with different system runners (ie single-threaded, multiprocessing, etc.).

1.12 Distributed systems

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

1.12.1 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.

1.12.2 Process application

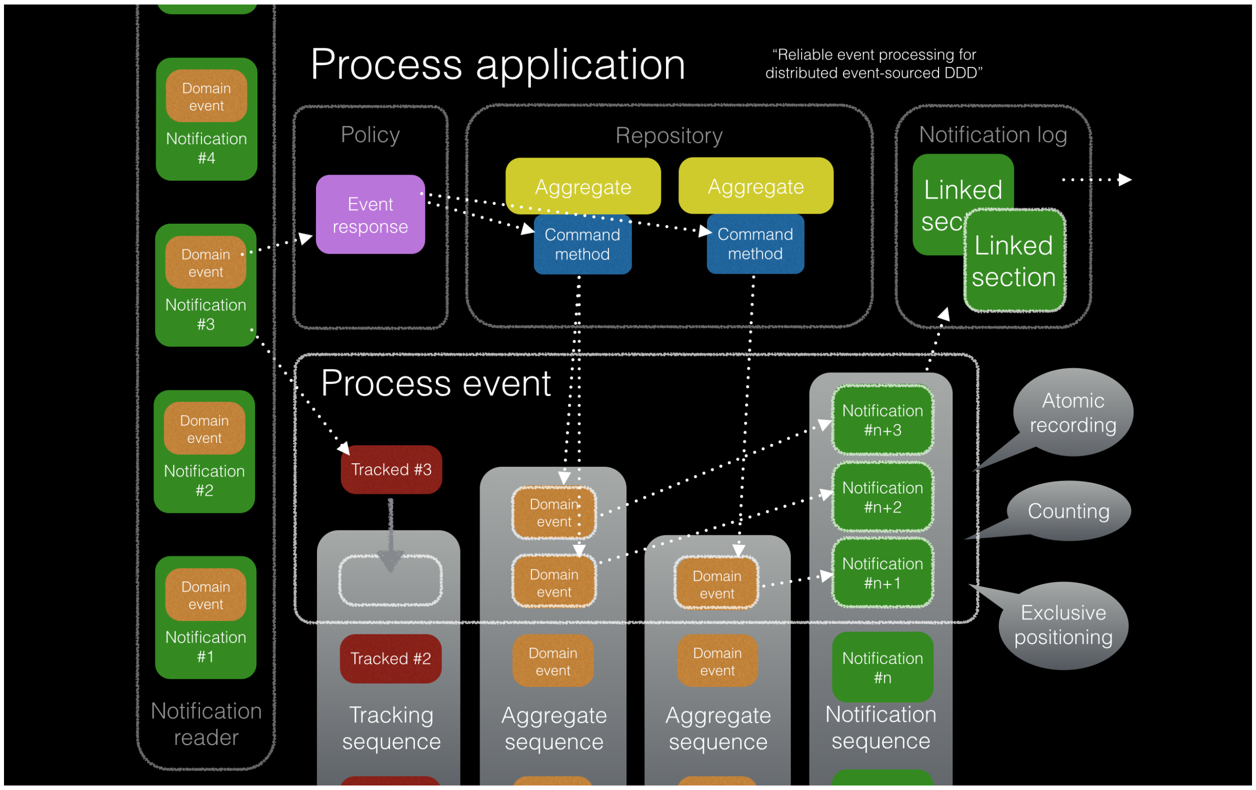
A process application is an event-sourced *projection* into an event-sourced *application*. One process application may "follow" another. A process application projects the state of the applications it follows into its own state. Being an event sourced application, the state of a process application 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).

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.



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.

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.

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.

1.12.3 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.

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.

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.

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.

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).

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.

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.

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 cause a sequence of steps involving other aggregates in other applications, 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. These terms ("saga", "process manager", etc.) can be used here as names for things we already have, but they add nothing in particular, since any reliable behaviour can be coded with combinations of events sourced aggregates and application policies. Exceptional "unhappy path" behaviour can be implemented as part of the logic of the application.

1.12.4 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 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 that is easily tested, easily understood, easily changed, and easily reconfigured for use with different infrastructure.

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

Please note, this example system is designed to exhibit a range of capabilities of the library and is not necessarily an example of good system design. In particular, whilst having "branches" (where one application is followed by more than one other application) does not introduce indeterminacy in the final system state, having "joins" (where one application follows more than one other) does so. Perhaps a better generic template for most domains is more simply to have a commands process followed by a core application for the domain, that is followed by a reporting application (`Commands | Core | Reporting`).

Please also note, the code presented in the example below works with the library's SQLAlchemy infrastructure code, and it can work with the library's 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.

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.

```python
from eventsourcing.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

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

(continues on next page)

(continued from previous page)

```
    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)

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

(continues on next page)

(continued from previous page)

```
    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.

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 eventsourcing.domain.model.command import Command
from eventsourcing.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):
        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()
```

Processes

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

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


class Orders(ProcessApplication):

    @applicationpolicy
    def policy(self, repository, event):
        """Do nothing by default."""
```

(continues on next page)

(continued from previous page)

```
    @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):
        """Do nothing by default."""

    @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):

    @applicationpolicy
    def policy(self, repository, event):
        """Do nothing by default."""

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

(continues on next page)

(continued from previous page)

```
        return self._create_payment(order_id)

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

A separate "commands application" is defined below. It extends the library class *CommandProcess*.

It has a factory method `create_order()` which can be used to create and save new `Order` aggregates.

The library class *CommandProcess* extends *ProcessApplication* and so is also a *SimpleApplication*. It and has its `persist_event_type` set to the *Event* supertype for domain events of *Command* aggregates, so that by default the domain events of a command aggregate will be persisted when a command aggregate is "saved".

The `Commands` class below also defines a policy that responds both to `Order.Created` events by setting the `order_id` on the command, and to `Order.Paid` events by setting the command as done.

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


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

    @applicationpolicy
    def policy(self, repository, event):
        """Do nothing by default."""

    @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()
```

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.

Apply policy to generated events

It is possible to set the `apply_policy_to_generated_events` class attribute of `ProcessApplication` to a `True` value. In this case, the policy will be applied to events that are generated by the policy, and all of them will be saved within the same atomic "process event". This can avoid having a process application follow itself in a system.

```
class ReflexiveApplication(ProcessApplication):
    apply_policy_to_generated_events = True
```

This will have no effect unless the policy is written to respond to the types of domain events that generated by parts of the policy that respond to the types of domain events generated by the applications that this process application is following.

System

A system of process applications can be defined using the library `System` object class, with one or many "pipeline expressions", each involving process application classes "linked" in a "pipeline" with Python's bitwise OR operator `|`.

For example, the pipeline expression `A | A` would have process application class `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. (The library's process application class uses a metaclass to support this, and although I'm normally averse to "extending the language", this seems to add a certain distinctiveness to the expression of a system.).

In the system defined below, the `Orders` process follows the `Commands` process, and the `Commands` process follows the `Orders` process, so that each will receive the events that its policy has been defined to process. Similarly, `Orders` and `Reservations` follow each other, and also `Orders` and `Payments` follow each other.

```
from eventsourcing.system.definition import System

system = System(
    Commands | Orders | Commands,
    Orders | Reservations | Orders,
    Orders | Payments | Orders
)
```

This system can alternatively be defined with a single pipeline expression, which expresses exactly the same set of relationships between the process applications.

```
system = System(
    Commands | Orders | Reservations | Orders | Payments | Orders | Commands
)
```

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.

Application state is propagated between process applications through notification logs only. 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 might not be reliable. If necessary, a process application can replicate upstream state within its own state.

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.system.runner 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`*, which uses the infrastructure class *`PopoApplication`* by default. This infrastructure class uses "plain old Python objects" to store domain events in memory, implementing atomic transactions and uniqueness constraints like SQLAlchemy and Django infrastructure classes, and is the fastest concrete application infrastructure class in the library (much faster than in-memory SQLite database, for example). This infrastructure can be used when proper disk-based durability is not required, for example during system development.

```
with system as runner:

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

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

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

    # Check the reservation exists.
    reservations = runner.get(Reservations)
    reservation = reservations.repository[order.reservation_id]

    # Check the payment exists.
    payments = runner.get(Payments)
    payment = payments.repository[order.payment_id]
```

Using the single-threaded runner means that 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 is useful when developing and testing a system of process applications, because it runs very quickly and the behaviour is very easy to follow.

Multi-threaded runner

Todo: More about the *`MultiThreadedRunner`*.

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

runner = MultiprocessRunner(
    system=system,
    infrastructure_class=SQLAlchemyApplication,
    setup_tables=True
)
```

The following MySQL database connection string is compatible with SQLAlchemy.

```
import os

os.environ['DB_URI'] = 'mysql+pymysql://{}:{}@{}/eventsourcing?charset=utf8mb4&binary_
↪prefix=true'.format(
    os.getenv('MYSQL_USER', 'eventsourcing'),
    os.getenv('MYSQL_PASSWORD', 'eventsourcing'),
    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=60, wait=0.5)
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.
    commands = runner.get(Commands)
    cmd_id = commands.create_order()

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

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

(continues on next page)

(continued from previous page)

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

    # Check the reservation exists.
    reservations = runner.get(Reservations)
    reservation = reservations.repository[order.reservation_id]

    # Check the payment exists.
    payments = runner.get(Payments)
    payment = 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 = []
    commands = runner.get(Commands)
    while len(command_ids) < num_orders:
        for pipeline_id in runner.pipeline_ids:

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

            # Create a "create new order" command.
            cmd_id = 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(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.

Thespian actor model runner

The Thespian Actor Library, can also be used to run a multi-pipeline system of process applications.

The library's *ThespianRunner* is a system runner that uses the Thespian actor model system.

The example below runs with Thespian's "simple system base".

```
from eventsourcing.system.thespian import ThespianRunner

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

with runner:

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

            # Change the pipeline for the command.

            commands.change_pipeline(pipeline_id)

            # Create a "create new order" command.
            cmd_id = 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(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 `runner` 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 *ThespianRunner* is constructed with `shutdown_on_close=True`, which is `False` by default, then the actors will be shutdown when the runner `close()` method is called (which happens when the runner is used as a context manager, and the context manager exits). Even so, shutting down the system actors will not shutdown a "multiproc" base system. Please refer to the Thespian documentation for more information.

Ray actor model runner

Ray can also be used to run a multi-pipeline system of process applications.

The library's *RayRunner* is a system runner that uses Ray's actor model system,.

```
from eventsourcing.system.ray import RayRunner

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

Please refer to the test for more information about using this class.

Pure gRPC runner

Inspired by Ray's use of gRPC, the library offers a pure gRPC runner.

The library's *GrpcRunner* is a system runner that uses gRPC to run a system of process applications.

```
from eventsourcing.system.grpc.runner import GrpcRunner

runner = GrpcRunner(
    system=system,
    infrastructure_class=SQLAlchemyApplication,
    setup_tables=True,
)
```

Please refer to the test for more information about using this class.

1.12.5 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.

1.12.6 Process event pattern

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 by 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.

1.13 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.

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

1.13.2 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
```

1.13.3 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
```

(continues on next page)

(continued from previous page)

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

    # Get historical snapshots.
    snapshot = app.snapshot_strategy.get_snapshot(entity.id, lte=2)
    assert snapshot.state['___version__'] == 1  # one behind
    assert snapshot.state['_foo'] == 'bar2'

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

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

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

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

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

1.14 Deployment

This section gives an overview of the concerns that arise when using an eventsourcing application in Web applications and task queue workers. There are many combinations of frameworks, databases, and process models. The complicated aspect is setting up the database configuration to work well with the framework. Your event sourcing application can be constructed just after the database is configured, and before requests are handled.

Please note, unlike the code snippets in the other examples, the snippets of code in this section are merely suggestive, and do not form a complete working program. For a working example using Flask and SQLAlchemy, please refer to the library module `eventsourcing.example.interface.flaskapp`, which is tested both stand-alone and with uWSGI.

1.14.1 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.

```python
from eventsourcing.application.sqlalchemy import SQLAlchemyApplication


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


```

(continues on next page)

(continued from previous page)

```
_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()
```

Lazy initialization

An alternative to having separate "init" and "get" functions is having one "get" function that does lazy initialization of the application object when first requested. With lazy initialization, the getter will first check if the object it needs to return has been constructed, and will then return the object. If the object hasn't been constructed, before returning the object it will construct the object. So you could use a lock around the construction of the object, to make sure it only happens once. After the lock is obtained and before the object is constructed, it is recommended to check again that the object wasn't constructed by another thread before the lock was acquired.

```python
import threading

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

close_application()
```

1.14.2 Database connection

Typically, your eventsourcing application object will be constructed after its database connection has been configured, and before any requests are handled. Views or tasks can then safely use the already constructed application object.

If your eventsourcing application depends on receiving a database session object when it is constructed, for example if you are using the SQLAlchemy classes in this library, then you will need to create a correctly scoped session object first and use it to construct the application object.

On the other hand, if your eventsourcing application does not depend on receiving a database session object when it is constructed, for example if you are using the Cassandra classes in this library, then you may construct the application object before configuring the database connection - just be careful not to use the application object before the database connection is configured otherwise your queries just won't work.

Setting up connections to databases is out of scope of the eventsourcing application classes, and should be set up in a "normal" way. The documentation for your Web or worker framework may describe when to set up database connections, and your database documentation may also have some suggestions. It is recommended to make use of any hooks or decorators or signals intended for the purpose of setting up the database connection also to construct the application once for the process. See below for some suggestions.

SQLAlchemy

SQLAlchemy has very good documentation about constructing sessions. If you are an SQLAlchemy user, it is well worth reading the documentation about sessions in full. Here's a small quote:

> *Some web frameworks include infrastructure to assist in the task of aligning the lifespan of a Session with that of a web request. This includes products such as Flask-SQLAlchemy for usage in conjunction with the Flask web framework, and Zope-SQLAlchemy, typically used with the Pyramid framework. SQLAlchemy recommends that these products be used as available.*
>
> *In those situations where the integration libraries are not provided or are insufficient, SQLAlchemy includes its own "helper" class known as scoped_session. A tutorial on the usage of this object is at*

Contextual/Thread-local Sessions. It provides both a quick way to associate a Session with the current thread, as well as patterns to associate Session objects with other kinds of scopes.

The important thing is to use a scoped session, and it is better to have the session scoped to the request or task, rather than the thread, but scoping to the thread is ok.

As soon as you have a scoped session object, you can construct your eventsourcing application.

Cassandra

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

1.14.3 Web interfaces

uWSGI

If you are running uWSGI in prefork mode, and not using a Web application framework, please note that uWSGI has a postfork decorator which may help.

Your "wsgi.py" file can have a module-level function decorated with the `@postfork` decorator that initialises your eventsourcing application for the Web application process after child workers have been forked.

```
from uwsgidecorators import postfork

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

Other decorators are available.

Flask

Flask with SQLAlchemy

If you wish to use eventsourcing with Flask and SQLAlchemy, then you may wish to use Flask-SQLAlchemy. You just need to define your record class(es) using the model classes from that library, and then use it instead of the library classes in your eventsourcing application object, along with the session object it provides.

The docs snippet below shows that it can work simply to construct the eventsourcing application in the same place as the Flask application object.

The Flask-SQLAlchemy class *SQLAlchemy* is used to set up a session object that is scoped to the request.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy_utils.types.uuid import UUIDType


# Construct Flask application.
application = Flask(__name__)
application.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
application.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

(continues on next page)

(continued from previous page)

```
# Construct Flask-SQLAlchemy object.
db = SQLAlchemy(application)

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

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

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

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

    # Topic of the item (e.g. path to domain event class).
    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():
```

(continues on next page)

(continued from previous page)

```
    global session, prepared
    session = Cluster().connect()
    prepared = session.prepare("SELECT release_version FROM system.local WHERE key=?")

app = Flask(__name__)

@app.route('/')
def server_version():
    row = session.execute(prepared, ('local',))[0]
    return row.release_version
```

The Flask-Cassandra project serves a similar function to Flask-SQLAlchemy.

Django

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

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

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

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

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

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

Django ORM

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

The independent project djangoevents by Applause is a Django app that provides a more integrated approach to event sourcing in a Django project. It also uses the Django ORM to store events. Using djangoevents is well documented in its README file. It adds some nice enhancements to the capabilities of this library, and shows how various components can be extended or replaced. Please note, the djangoevents project currently works with a much older version of this library which isn't recommended for new projects.

Django with Cassandra

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

It's also possible to use this library directly with Django and Cassandra. You just need to configure the connection and initialise the application before handling requests in a way that is correct for your configuration (which is what Django-Cassandra tries to make easy).

Zope

Zope with SQLAlchemy

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

1.14.4 Task queues

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

Celery

Celery has a worker_process_init signal decorator, which may be appropriate if you are running Celery workers in prefork mode. Other decorators are available.

Your Celery tasks or config module can have a module-level function decorated with the `@worker-process-init` decorator that initialises your eventsourcing application for the Celery worker process.

```
from celery.signals import worker_process_init

@worker_process_init.connect
def init_process(sender=None, conf=None, **kwargs):
    # Set up database connection.
    database = {}
    # Construct eventsourcing application.
    init_application()
```

As an alternative, it may work to use decorator `@task_prerun` with a getter that supports lazy initialization.

```
from celery.signals import task_prerun
@task_prerun.connect
def init_process(*args, **kwargs):
    get_appliation(lazy_init=True)
```

Once the application has been safely initialized once in the process, your Celery tasks can use function `get_application()` to complete their work. Of course, you could just call a getter with lazy initialization from the tasks.

```
from celery import Celery

app = Celery()
```

(continues on next page)

(continued from previous page)

```
# Use Celery app to route the task to the worker.
@app.task
def hello_world():
    # Use eventsourcing app to complete the task.
    app = get_application()
    return "Hello World, {}".format(id(app))
```

Again, the most important thing is configuring the database, and making things work across all modes of execution, including your test suite.

Redis Queue

Redis queue workers are quite similar to Celery workers. You can call `get_application()` from within a job function. To fit with the style in the RQ documentation, you could perhaps use your eventsourcing application as a context manager, just like the Redis connection example.

1.15 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.

1.15.1 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.

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`.

```python
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 *`CreatedEvent`*, *`AttributeChangedEvent`*, and *`DiscardedEvent`*.

Publish-subscribe

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

```
subscribers = []

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


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


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

Domain entity

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

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

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

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

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

    @property
    def __version__(self):
        return self.___version__

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

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

        # Construct an 'AttributeChanged' event object.
        event = AttributeChanged(
            originator_id=self.id,
            originator_version=self.__version__,
            name='foo',
```

(continues on next page)

(continued from previous page)

```
            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.

```
import uuid

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])

    # Return the new entity.
    return entity
```

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

Mutator function

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

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

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

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

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

For the sake of simplicity in this example, an if-else block is used to structure the mutator function. The library has a function decorator *mutator()* that allows a default mutator function to register handlers for different types of event, much like singledispatch.

Run the code

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

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

# Subscribe to receive published events.
subscribe(lambda e: received_events.extend(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')

```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

1.15.2 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]
```

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

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

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

Event store

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

The event store then uses a record manager to persist the sequenced items into a particular database management system. The record manager uses 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,
)

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

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

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

Entity repository

It is common to retrieve entities from a repository. An event sourced repository for the `example` entity class can be constructed directly using library class *EventSourcedRepository*.

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

```
from eventsourcing.infrastructure.eventsourcedrepository import EventSourcedRepository

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

Run the code

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

```python
# Put each received event into the event store.
event_store.store_events(received_events)

# Check the events exist in the event store.
stored_events = event_store.list_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.

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

Sequenced items

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

```python
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 b'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 b'baz' in sequenced_items[1].state
```

1.15.3 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.

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.

```python
class PersistencePolicy(object):
    def __init__(self, event_store):
        self.event_store = event_store
        subscribe(self.store_events)

    def close(self):
```

(continues on next page)

(continued from previous page)

```
        unsubscribe(self.store_events)

    def store_events(self, events):
        self.event_store.store_events(events)
```

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

Application object

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

```
class ExampleApplication(object):
    def __init__(self, session):
        # Construct event store.
        self.event_store = EventStore(
            record_manager=SQLAlchemyRecordManager(
                record_class=IntegerSequencedRecord,
                session=session,
            ),
            event_mapper=SequencedItemMapper(
                sequence_id_attr_name='originator_id',
                position_attr_name='originator_version'
            )
        )
        # Construct persistence policy.
        self.persistence_policy = PersistencePolicy(
            event_store=self.event_store
        )
        # Construct example repository.
        self.example_repository = EventSourcedRepository(
            event_store=self.event_store,
            mutator_func=mutate
        )

    def __enter__(self):
        return self

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

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

1.15.4 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:
```

(continues on next page)

(continued from previous page)

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

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

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

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

1.16 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 Simplyfying 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

1.17 Release notes

It is the aim of the project that releases with the same major version number are backwards compatible, within the scope of the documented examples. New major versions indicate backwards incompatible changes have been introduced since the previous major version. New minor version indicate new functionality has been added, or existing functionality extended. New point version indicates existing code or documentation has been improved in a way that neither breaks backwards compatibility nor extends the functionality of the library.

1.17.1 Version 8.x

Version 8.x series brings more efficient storage, static type hinting, improved transcoding, event and entity versioning, and integration with Axon Server (specialist event store) and Ray. Code for defining and running systems of application, previously in the "application" package, has been moved to a new "system" package.

Version 8.3.0 (released 9 January 2020)

Added gRPC runner. Improved Django record manager, so that it supports setting notification log IDs in the application like the SQLAlchemy record manager (this optionally avoids use of the "insert select max" statement and thereby makes it possible to exclude domain events from the notification log at the risk of non-gapless notification log sequences). Also improved documentation.

Version 8.2.5 (released 22 Dec 2020)

Increased versions of dependencies on requests, Django, Celery, PyMySQL.

Version 8.2.4 (released 12 Nov 2020)

Fixed issue with using Oracle database, where a trailing semicolon in an SQL statement caused the "invalid character" error (ORA-00911).

Version 8.2.3 (released 19 May 2020)

Improved interactions with process applications in RayRunner so that they have the same style as interactions with process applications in other runners. This makes the RayRunner more interchangeable with the other runners, so that system client code can be written to work with any runner.

Version 8.2.2 (released 16 May 2020)

Improved documentation. Updated dockerization for local development. Added Makefile, to setup development environment, to build and run docker containers, to run the test suite, to format the code, and to build the docs. Reformatted the code.

Version 8.2.1 (released 11 March 2020)

Improved documentation.

Version 8.2.0 (released 10 March 2020)

Added optional versioning of domain events and entities, so that domain events and entity snapshots can be versioned and old versions of state can be upcast to new versions.

Added optional correlation and causation IDs for domain events, so that a story can be traced through a system of applications.

Added AxonApplication and AxonRecordManager so that Axon Server can be used as an event store by event sourced applications.

Added RayRunner, which allows a system of applications to be run with the Ray framework.

Version 8.1.0 (released 11 January 2020)

Improved documentation. Improved transcoding (e.g. tuples are encoded as tuples also within other collections). Added event hash method name to event attributes, so that event hashes created with old version of event hashing can still be checked. Simplified repository base classes (removed "event player" class).

Version 8.0.0 (released 7 December 2019)

The storage of event state has been changed from strings to bytes. This is definitely a backwards incompatible change. Previously state bytes were encoded with base64 before being saved as strings, which adds 33% to the size of each stored state. Compression of event state is now an option, independently of encryption, and compression is now configurable (defaults to zlib module, other compressors can be used). Attention will need to be paid to one of two alternatives. One alternative is to migrate your stored events (the state field), either from being stored as plaintext strings to being stored as plaintext bytes (you need to encode as utf-8), or from being stored as ciphertext bytes encoded with base64 decoded as utf-8 to being stored as ciphertext bytes (you need to encode as utf-8 and decode base64). The other alternative is to carry on using the same database schema, define custom stored event record classes in your project (copied from the previous version of the library), and extend the record manager to convert the bytes to strings and back. A later version of this library may bring support for one or both of these options, so if this change presents a challenge, please hold off from upgrading, and discuss your situation with the project developer(s). There is nothing wrong with the previous version, and you can continue to use it.

Other backwards incompatible changes involve renaming a number of methods, and moving classes and also modules (for example, the system modules have been moved from the applications package to a separate package). Please see the commit log for all the details.

This version also brings improved and expanded transcoding, additional type annotations, automatic subclassing on domain entities of domain events (not enabled by default), an option to apply the policy of a process application to all events that are generated by its policy when an event notification is processed (continues until all successively generated events have been processed, with all generated events stored in the same atomic process event, as if all generated events were generated in a single policy function).

Please note, the transcoding now supports the encoding of tuples, and named tuples, as tuples. Previously tuples were encoded by the JSON transcoding as lists, and so tuples became lists, which is the default behaviour on the core json package. So if you have code that depends on the transcoder converting tuples to lists, then attention will have to paid to the fact that tuples will now be encoded and returned as tuples. However, any existing stored events generated with an earlier version of this library will continue to be returned as lists, since they were encoded as lists not tuples.

Please note, the system runner class was changed to keep references to constructed process application classes in the runner object, rather than the system object. If you have code that accesses the process applications as attributes on the system object, then attention will need to be paid to accessing the process applications by class on the runner object.

1.17.2 Version 7.x

Version 7.x series refined the "process and system" code.

Version 7.2.4 (released 9 Oct 2019)

Version 7.2.4 fixed an issue in running the test suite.

Version 7.2.3 (released 9 Oct 2019)

Version 7.2.3 fixed a bug in MultiThreadedRunner.

Version 7.2.2 (released 6 Oct 2019)

Version 7.2.2 has improved documentation for "reliable projections".

Version 7.2.1 (released 6 Oct 2019)

Version 7.2.1 has improved support for "reliable projections", which allows custom records to be deleted (previously only create and update was supported). The documentation for "reliable projections" was improved. The previous code snippet, which was merely suggestive, was replaced by a working example.

Version 7.2.0 (released 1 Oct 2019)

Version 7.2.0 has support for "reliable projections" into custom ORM objects that can be coded as process application policies.

Also a few issues were resolved: avoiding importing Django models from library when custom models are being used to store events prevents model conflicts; fixed multiprocess runner to work when an application is not being followed by another; process applications now reflect off the sequenced item tuple when reading notifications so that custom field names are used.

Version 7.1.6 (released 2 Aug 2019)

Version 7.1.6 fixed an issue with the notification log reader. The notification log reader was sometimes using a "fast path" to get all the notifications without paging through the notification log using the linked sections. However, when there were too many notification, this failed to work. A few adjustments were made to fix the performance and robustness and configurability of the notification log reading functionality.

Version 7.1.5 (released 26 Jul 2019)

Version 7.1.5 improved the library documentation with better links to module reference pages. The versions of dependencies were also updated, so that all versions of dependencies are the current stable versions of the package distributions on PyPI. In particular, requests was updated to a version that fixes a security vulnerability.

Version 7.1.4 (released 10 Jul 2019)

Version 7.1.4 improved the library documentation.

Version 7.1.3 (released 4 Jul 2019)

Version 7.1.3 improved the domain model layer documentation.

Version 7.1.2 (released 26 Jun 2019)

Version 7.1.2 fixed method 'construct_app()' on class 'System' to set 'setup_table' on its process applications using the system's value of 'setup_tables'. Also updated version of dependency of SQLAlchemy-Utils.

Version 7.1.1 (released 21 Jun 2019)

Version 7.1.1 added 'Support options' and 'Contributing' sections to the documentation.

Version 7.1.0 (released 11 Jun 2019)

Version 7.1.0 improved structure to the documentation.

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.

1.17.3 Version 6.x

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

Version 6.2.0 (released 15 Jul 2018)

Version 6.2.0 (released 26 Jun 2018)

Version 6.1.0 (released 14 Jun 2018)

Version 6.0.0 (released 23 Apr 2018)

1.17.4 Version 5.x

Version 5.x added support for Django ORM. It was released as a new major version after quite a lot of refactoring made things backward-incompatible.

Version 5.1.1 (released 4 Apr 2018)

Version 5.1.0 (released 16 Feb 2018)

Version 5.0.0 (released 24 Jan 2018)

Support for Django ORM was added in version 5.0.0.

1.17.5 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.

Version 4.0.0 (released 11 Dec 2017)

1.17.6 Version 3.x

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

Version 3.1.0 (released 23 Nov 2017)

Version 3.0.0 (released 25 May 2017)

1.17.7 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.

Version 2.1.1 (released 30 Mar 2017)

Version 2.1.0 (released 27 Mar 2017)

Version 2.0.0 (released 27 Mar 2017)

1.17.8 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.

Version 1.2.1 (released 23 Oct 2016)

Version 1.2.0 (released 23 Oct 2016)

Version 1.1.0 (released 19 Oct 2016)

Version 1.0.10 (released 5 Oct 2016)

Version 1.0.9 (released 17 Aug 2016)

Version 1.0.8 (released 30 Jul 2016)

Version 1.0.7 (released 13 Jul 2016)

Version 1.0.6 (released 7 Jul 2016)

Version 1.0.5 (released 1 Jul 2016)

Version 1.0.4 (released 30 Jun 2016)

Version 1.0.3 (released 30 Jun 2016)

Version 1.0.2 (released 8 Jun 2016)

Version 1.0.1 (released 7 Jun 2016)

1.17.9 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.

Version 0.9.4 (released 11 Feb 2016)

Version 0.9.3 (released 1 Dec 2015)

Version 0.9.2 (released 1 Dec 2015)

Version 0.9.1 (released 10 Nov 2015)

Version 0.9.0 (released 14 Sep 2015)

Version 0.8.4 (released 14 Sep 2015)

Version 0.8.3 (released 5 Sep 2015)

Version 0.8.2 (released 5 Sep 2015)

Version 0.8.1 (released 4 Sep 2015)

Version 0.8.0 (released 29 Aug 2015)

Version 0.7.0 (released 29 Aug 2015)

Version 0.6.0 (released 28 Aug 2015)

Version 0.5.0 (released 28 Aug 2015)

Version 0.4.0 (released 28 Aug 2015)

Version 0.3.0 (released 28 Aug 2015)

Version 0.2.0 (released 27 Aug 2015)

Version 0.1.0 (released 27 Aug 2015)

Version 0.0.1 (released 27 Aug 2015)

1.18 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.

1.18.1 Community development

- Join the Slack channel and answer questions. The library has a growing audience. Help to create and 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 improve the unit test suite.

1.18.2 Making changes

To make changes to the library, you will want to set up a local environment. To get set up, fork the repository on GitHub, clone your fork using Git, and then checkout the `develop` branch.

Create a virtual Python environment, install Python dependencies, install and start the databases that are used by the test suite. and then run the tests. The library test suite depends on several databases. It's much easier to run databases in Docker containers, but it's slightly faster to run databases without containers.

Once you have the tests running, you can make changes, run the tests again, push changes to your fork, and then maybe create a pull request to the project's develop branch.

This library has a *Makefile* to help with development. You can read more about the GNU make utility in this article. There are commands to install Python dependencies into a virtual Python environment, to build containers for the databases, to start and stop databases, to run the test suite, to build the docs, to reformat code, and for static type checking and linting. The actual code of the commands described below can be easily found in `Makefile` at the root of the project repository.

Virtual Python environment

This project runs on Python 3.6+, and you need to have it installed on your system. The recommended way for all platforms is to use the official download page. But also, you may use pyenv.

You can use virtualenv to create a virtual Python environment. Choose for your flavour:

- virtualenv

- virtualenvwrapper
- pyenv-virtualenv for pyenv

For example, you can create and activate a virtual Python environment using `virtualenv` directly:

```
$ virtualenv ~/.virtualenvs/eventsourcing-py3
$ source ~/.virtualenvs/eventsourcing-py3/bin/activate
```

Inside your activated virtualenv, use the following command to install all project dependencies required for contribution:

```
$ make install
```

Git blame (optional)

Setup `git` to ignore specific revs with `blame`.

This project is old, and several times in its history a massive changes were performed. One such change is moving towards use of `isort` and `flake8` and `black`. While these changes are inevitable, they clutter the history, especially if you use `git blame` or _Annotate_ option in PyCharm. But in newer versions of git (>= 2.23), this can be mitigated: new options –ignore-rev and –ignore-revs-file were added. There is a file in this repository called `.git-blame-ignore-revs` which contains all such major reformattings. In order to pick it up by `git blame` and PyCharm, add a special config line:

```
$ git config --local blame.ignoreRevsFile .git-blame-ignore-revs
```

More info can be found here.

Run databases with Docker

You can run the databases in Docker containers.

To pull docker images:

```
$ make docker-pull
```

To build docker images:

```
$ make docker-build
```

To up and keep running containers in detached mode:

```
$ make docker-up
```

To stop the containers:

```
$ make docker-stop
```

To tear down the containers removing volumes and "orphans":

```
$ make docker-down
```

To attach to the latest containers output:

```
$ make docker-logs
```

All of the commands using predefined "COMPOSE_FILE" and "COMPOSE_PROJECT_NAME" to keep your containers in a more organized and straightforward way.

COMPOSE_FILE is used by *docker-compose* utility to pick development services configuration. The valid format of this value is: `dev/docker-compose.yaml`.

COMPOSE_PROJECT_NAME sets the project name. This value used to prepend the containers on startup along with the service name. `eventsourcing` is a great default value for it.

Run databases on macOS

If you happen to be using a Mac, you can install the databases directly on macOS using the following commands:

```
$ brew install mysql
$ brew install posgresql
$ brew install redis
$ brew install cassandra
$ ./dev/download_axon_server.sh
```

To start the databases, you can run:

```
$ make brew-services-start
```

To stop the services, you can run:

```
$ make brew-services-stop
```

Before running the tests for the first time, create a database in MySQL, and configure user access:

```
$ mysql -u root
mysql> CREATE DATABASE EVENTSOURCING;
mysql> CREATE USER 'eventsourcing'@'localhost' IDENTIFIED BY 'eventsourcing';
mysql> GRANT ALL PRIVILEGES ON eventsourcing.* TO 'eventsourcing'@'localhost';
```

You will also need to create a database in PostgreSQL:

```
$ createdb eventsourcing
```

Run tests

Ensure that you've set up your development environment (see *Virtual Python environment*) and and required services are up and running (see *Run databases with Docker*, or *Run databases on macOS*).

Running tests from an IDE such as PyCharm allows easy navigation to code files.

You can run the full test suite using `make test`:

```
$ make test
```

You can skip the slower tests when running the test suite with `make quick-test`:

```
$ make quick-test
```

Note: To re-run tests, sometimes it requires `make docker-down` for a fresh start. At the moment, Axon Server sometimes doesn't return everything that is expected when listing all the events of an application. But restarting Axon Server seems to clear this up.

Building documentation

This project is using Sphinx documentation builder tool. Run this command to compile documentation into static HTML files at `./docs/_build/html`:

```
$ make docs
```

Linting your code

For now, linting your changes is completely optional - we do not have any checks on CI for it.

Run isort to check imports sorting:

```
$ make lint-isort
```

We are using Black as a tool for style guide enforcement:

```
$ make lint-black
```

We are using Flake8 (and it's Flake8 BugBear plugin) to check the code for PEP8 compatibility:

```
$ make lint-flake8
```

Mypy is a static type checker for Python 3 and Python 2.7. Run mypy to check code for accurate typing annotations:

```
$ make lint-mypy
```

Dockerfilelint is an `npm` module that analyzes a Dockerfile and looks for common traps, mistakes and helps enforce best practices:

```
$ make lint-dockerfile
```

. . . and finally, to run all the checks from above, use:

```
$ make lint
```

Automatic formatting

To apply automatic formatting by using isort and Black, run:

```
$ make fmt
```

Note: In order to keep your Pull Request clean, please, do not apply it for all project but your specific changes. The project is now well formatted, but static typing and and strict compliance with PEP8 is still a working in progress. If you want to help improve the type hints and formatting, please do so in a dedicated PR so things aren't mixed with other changes (it's just easier to review this way).

1.19 Module docs

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

- genindex
- modindex

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

1.19.1 domain.model

The domain model package contains classes and functions that can help develop an event sourced domain model.

events

Base classes for domain events of different kinds.

class `eventsourcing.domain.model.events.`**`DomainEvent`**(***kwargs*)

Bases: *`eventsourcing.domain.model.versioning.Upcastable`*, *`eventsourcing.whitehead.ActualOccasion`*, `typing.Generic`

Base class for domain model events.

Implements methods to make instances read-only, comparable for equality in Python, and have recognisable representations.Custom To make domain events hashable, this class also implements a method to create a cryptographic hash of the state of the event.

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

`__repr__`() → str
Creates a string representing the type and attribute values of the event.

Return type str

`__mutate__`(*obj: Optional[TEntity]*) → Optional[TEntity]
Updates 'obj' with values from 'self'.

Calls the 'mutate()' method.

Can be extended, but subclasses must call super and return an object to their caller.

Parameters **`obj`** – object (normally a domain entity) to be mutated

Returns mutated object

`mutate`(*obj: TEntity*) → None
Updates ("mutates") given 'obj'.

Intended to be overridden by subclasses, as the most concise way of coding a default projection of the event (for example into the state of a domain entity).

The advantage of implementing a default projection using this method rather than __mutate__ is that you don't need to call super or return a value.

Parameters **`obj`** – domain entity to be mutated

`__setattr__`(*key: Any*, *value: Any*) → None
Inhibits event attributes from being updated by assignment.

`__eq__`(*other: object*) → bool
Tests for equality of two event objects.

Return type bool

`__ne__`(*other: object*) → bool
Negates the equality test.

Return type bool

`__hash__`() → int
Computes a Python integer hash for an event.

Supports Python equality and inequality comparisons.

Returns Python integer hash

Return type int

`classmethod __hash_object_v2__`(*obj: dict*) → str
Calculates SHA-256 hash of JSON encoded 'obj'.

Parameters **`obj`** – Object to be hashed.

Returns SHA-256 as hexadecimal string.

Return type str

`classmethod __hash_object_v1__`(*obj: dict*) → str
Calculates SHA-256 hash of JSON encoded 'obj'.

Parameters **`obj`** – Object to be hashed.

Returns SHA-256 as hexadecimal string.

Return type str

`class` `eventsourcing.domain.model.events.`**`EventWithHash`**(***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

Base class for domain events with a cryptographic event hash.

Extends DomainEvent by setting a cryptographic event hash when the event is originated, and checking the event hash whenever its default projection mutates an object.

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

__event_hash__
Returns SHA-256 hash of the original state of the event.

Returns SHA-256 as hexadecimal string.

Return type str

__hash__() → int
Computes a Python integer hash for an event, using its pre-computed event hash.

Supports Python equality and inequality comparisons only.

Returns Python integer hash

Return type int

__mutate__(*obj: Optional[TEntity]*) → Optional[TEntity]
Updates '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

__check_hash__() → None
Raises EventHashError, unless self.__event_hash__ can be derived from the current state of the event object.

class eventsourcing.domain.model.events.**EventWithOriginatorID**(*originator_id: uuid.UUID*, ***kwargs*)
Bases: *eventsourcing.domain.model.events.DomainEvent*

For events that have an originator ID.

__init__(*originator_id: uuid.UUID*, ***kwargs*)
Initialises event attribute values directly from constructor kwargs.

originator_id
Originator ID is the identity of the object that originated this event.

Returns A UUID representing the identity of the originator.

Return type UUID

class eventsourcing.domain.model.events.**EventWithTimestamp**(*timestamp: Optional[decimal.Decimal] = None*, ***kwargs*)
Bases: *eventsourcing.domain.model.events.DomainEvent*

For events that have a timestamp value.

__init__(*timestamp: Optional[decimal.Decimal] = None*, ***kwargs*)
Initialises event attribute values directly from constructor kwargs.

timestamp
A UNIX timestamp as a Decimal object.

class eventsourcing.domain.model.events.**EventWithOriginatorVersion**(*originator_version: int*, ***kwargs*)
Bases: *eventsourcing.domain.model.events.DomainEvent*

For events that have an originator version number.

`__init__`(*originator_version: int*, ***kwargs*)
Initialises event attribute values directly from constructor kwargs.

`originator_version`
Originator version is the version of the object that originated this event.

Returns A integer representing the version of the originator.

class `eventsourcing.domain.model.events.`**`EventWithTimeuuid`**(*event_id: Optional[uuid.UUID] = None*, ***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

For events that have an UUIDv1 event ID.

`__init__`(*event_id: Optional[uuid.UUID] = None*, ***kwargs*)
Initialises event attribute values directly from constructor kwargs.

class `eventsourcing.domain.model.events.`**`CreatedEvent`**(***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

Happens when something is created.

class `eventsourcing.domain.model.events.`**`AttributeChangedEvent`**(***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

Happens when the value of an attribute changes.

class `eventsourcing.domain.model.events.`**`DiscardedEvent`**(***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

Happens when something is discarded.

class `eventsourcing.domain.model.events.`**`LoggedEvent`**(***kwargs*)
Bases: *`eventsourcing.domain.model.events.DomainEvent`*

Happens when something is logged.

`eventsourcing.domain.model.events.`**`subscribe`**(*handler: Callable[[Sequence[TEvent]], None]*, *predicate: Optional[Callable[[Sequence[TEvent]], bool]] = None*) → None
Adds 'handler' to list of event handlers to be called if 'predicate' is satisfied.

If predicate is None, the handler will be called whenever an event is published.

Parameters

- **`handler`** (*`callable`*) – Will be called when an event is published.
- **`predicate`** (*`callable`*) – Conditions whether the handler will be called.

`eventsourcing.domain.model.events.`**`unsubscribe`**(*handler: Callable[[Sequence[TEvent]], None]*, *predicate: Optional[Callable[[Sequence[TEvent]], bool]] = None*) → None
Removes 'handler' from list of event handlers to be called if 'predicate' is satisfied.

Parameters

- **`handler`** (*`callable`*) – Previously subscribed handler.
- **`predicate`** (*`callable`*) – Previously subscribed predicate.

`eventsourcing.domain.model.events.`**`publish`**(*events: Sequence[TEvent]*) → None

Published given 'event' by calling subscribed event handlers with the given 'event', except those with predicates that are not satisfied by the event.

Handlers are called in the order they are subscribed.

Parameters **`events`** (`DomainEvent`) – Domain event to be published.

`exception` `eventsourcing.domain.model.events.`**`EventHandlersNotEmptyError`**

Bases: `Exception`

`eventsourcing.domain.model.events.`**`assert_event_handlers_empty`**() → None

Raises EventHandlersNotEmptyError, unless there are no event handlers subscribed.

`eventsourcing.domain.model.events.`**`clear_event_handlers`**() → None

Removes all previously subscribed event handlers.

`class` `eventsourcing.domain.model.events.`**`AbstractSnapshot`**

Bases: *`eventsourcing.whitehead.ActualOccasion`*

`topic`

Path to the class of the snapshotted entity.

`state`

State of the snapshotted entity.

`originator_id`

ID of the snapshotted entity.

`originator_version`

Version of the last event applied to the entity.

`__mutate__`(*obj: Optional[TEntity]*) → Optional[TEntity]

Reconstructs the snapshotted entity.

entity

Base classes for domain model entities.

`class` `eventsourcing.domain.model.entity.`**`MetaDomainEntity`**(*name: str*, **args*, ***kwargs*)

Bases: `abc.ABCMeta`

`class` `eventsourcing.domain.model.entity.`**`DomainEntity`**(*id: uuid.UUID*)

Bases: *`eventsourcing.domain.model.versioning.Upcastable`*, *`eventsourcing.whitehead.EnduringObject`*

Supertype for domain model entity.

`class Event`(*originator_id: uuid.UUID*, ***kwargs*)

Bases: *`eventsourcing.domain.model.events.EventWithOriginatorID`*

Supertype for events of domain model entities.

`__check_obj__`(*obj: TDomainEntity*) → None

Checks state of obj before mutating.

Parameters **`obj`** – Domain entity to be checked.

Raises ***`OriginatorIDError`*** – if the originator_id is mismatched

`classmethod __create__`(*originator_id: Optional[uuid.UUID] = None*, *event_class: Optional[Type[DomainEntity.Created[TDomainEntity]]] = None*, ***kwargs*) → TDomainEntity

Creates a new domain entity.

Constructs a "created" event, constructs the entity object from the event, publishes the "created" event, and returns the new domain entity object.

Parameters

- **`DomainEntity`** (`cls`) – Class of domain event
- **`originator_id`** – ID of the new domain entity (defaults to `uuid4()`).
- **`event_class`** – Domain event class to be used for the "created" event.
- **`kwargs`** – Other named attribute values of the "created" event.

Returns New domain entity object.

Return type *DomainEntity*

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

Bases: *eventsourcing.domain.model.events.CreatedEvent*, `eventsourcing.domain.model.entity.Event`

Triggered when an entity is created.

originator_topic

Topic (a string) representing the class of the originating domain entity.

Return type str

__mutate__(*obj: Optional[TDomainEntity]*) → Optional[TDomainEntity]

Constructs object from an entity class, which is obtained by resolving the originator topic, unless it is given as method argument `entity_class`.

Parameters **`entity_class`** – Class of domain entity to be constructed.

id

The immutable ID of the domain entity.

This value is set using the `originator_id` of the "created" event constructed by `__create__()`.

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

This attribute has the normal "public" format for a Python object attribute name, because by definition all domain entities have an ID.

__change_attribute__(*name: str*, *value: Any*, ***kwargs*) → None

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

class AttributeChanged(*originator_id: uuid.UUID*, ***kwargs*)

Bases: `eventsourcing.domain.model.entity.Event`, *eventsourcing.domain.model.events.AttributeChangedEvent*

Triggered when a named attribute is assigned a new value.

__mutate__(*obj: Optional[TDomainEntity]*) → Optional[TDomainEntity]

Updates 'obj' with values from 'self'.

Calls the 'mutate()' method.

Can be extended, but subclasses must call super and return an object to their caller.

Parameters **`obj`** – object (normally a domain entity) to be mutated

Returns mutated object

__discard__(***kwargs*) → None

Discards self, by triggering a Discarded event.

class Discarded(*originator_id: uuid.UUID*, ***kwargs*)

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

Triggered when a DomainEntity is discarded.

__mutate__(*obj: Optional[TDomainEntity]*) → Optional[TDomainEntity]

Updates 'obj' with values from 'self'.

Calls the 'mutate()' method.

Can be extended, but subclasses must call super and return an object to their caller.

Parameters **obj** – object (normally a domain entity) to be mutated

Returns mutated object

__assert_not_discarded__() → None

Asserts that this entity has not been discarded.

Raises EntityIsDiscarded exception if entity has been discarded already.

__trigger_event__(*event_class: Type[TDomainEvent], **kwargs*) → None

Constructs, applies, and publishes a domain event.

__mutate__(*event: TDomainEvent*) → None

Mutates this entity with the given event.

This method calls on the event object to mutate this entity, because the mutation behaviour of different types of events was usefully factored onto the event classes, and the event mutate() method is the most convenient way to defined behaviour in domain models.

However, as an alternative to implementing the mutate() method on domain model events, this method can be extended with a method that is capable of mutating an entity for all the domain event classes introduced by the entity class.

Similarly, this method can be overridden entirely in subclasses, so long as all of the mutation behaviour is implemented in the mutator function, including the mutation behaviour of the events defined on the library event classes that would no longer be invoked.

However, if the entity class defines a mutator function, or if a separate mutator function is used, then it must be involved in the event sourced repository used to replay events, which by default knows nothing about the domain entity class. In practice, this means having a repository for each kind of entity, rather than the application just having one repository, with each repository having a mutator function that can project the entity events into an entity.

__publish__(*event: Sequence[TDomainEvent]*) → None

Publishes given event for subscribers in the application.

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

__publish_to_subscribers__(*events: Sequence[TDomainEvent]*) → None

Actually dispatches given event to publish-subscribe mechanism.

Parameters **events** – list of domain events

__eq__(*other: object*) → bool

Return self==value.

__ne__(*other: object*) → bool

Return self!=value.

class eventsourcing.domain.model.entity.**EntityWithHashchain**(**args*, ***kwargs*)

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

class Event(***kwargs*)

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

Supertype for events of domain entities.

__mutate__(*obj: Optional[TEntityWithHashchain]*) → Optional[TEntityWithHashchain]

Updates '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

__check_obj__(*obj: TEntityWithHashchain*) → None

Extends superclass method by checking the __previous_hash__ of this event matches the __head__ hash of the entity obj.

class Created(***kwargs*)

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

class AttributeChanged(***kwargs*)

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

class Discarded(***kwargs*)

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

__mutate__(*obj: Optional[TEntityWithHashchain]*) → Optional[TEntityWithHashchain]

Updates '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__(*originator_id: Optional[uuid.UUID] = None, event_class: Optional[Type[DomainEntity.Created[TEntityWithHashchain]]] = None, **kwargs*) → TEntityWithHashchain

Creates a new domain entity.

Constructs a "created" event, constructs the entity object from the event, publishes the "created" event, and returns the new domain entity object.

Parameters

- **DomainEntity** (*cls*) – Class of domain event
- **originator_id** – ID of the new domain entity (defaults to `uuid4()`).
- **event_class** – Domain event class to be used for the "created" event.
- **kwargs** – Other named attribute values of the "created" event.

Returns New domain entity object.

Return type *DomainEntity*

__trigger_event__(*event_class: Type[TDomainEvent], **kwargs*) → None

Constructs, applies, and publishes a domain event.

class eventsourcing.domain.model.entity.**VersionedEntity**(*__version__: int, **kwargs*)

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

`__trigger_event__`(*event_class: Type[TDomainEvent], **kwargs*) → None
Increments the version number when an event is triggered.

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 before the event was triggered, but represents the result of the work of incrementing the version, which is then set in the event as normal. The Created event has version 0, and 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.

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

Supertype for events of versioned entities.

`__mutate__`(*obj: Optional[TVersionedEntity]*) → Optional[TVersionedEntity]
Updates 'obj' with values from 'self'.

Calls the 'mutate()' method.

Can be extended, but subclasses must call super and return an object to their caller.
Parameters **`obj`** – object (normally a domain entity) to be mutated
Returns mutated object

`__check_obj__`(*obj: TVersionedEntity*) → None
Extends superclass method by checking the event's originator version follows (1 +) this entity's version.

class Created(*originator_version: int = 0, *args, **kwargs*)
Bases: `eventsourcing.domain.model.entity.Created`, `eventsourcing.domain.model.entity.Event`

Published when a VersionedEntity is created.

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

Published when a VersionedEntity is changed.

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

Published when a VersionedEntity is discarded.

class `eventsourcing.domain.model.entity.`**EntityWithECC**(*, *event_id, correlation_id, causation_id, **kwargs*)
Bases: *`eventsourcing.domain.model.entity.DomainEntity`*

Entity whose events have event ID, correlation ID, and causation ID.

class Event (*, *processed_event=None, application_name, **kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`

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

class AttributeChanged(*, *processed_event=None, application_name, **kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.AttributeChanged`

class Discarded(**, processed_event=None, application_name, **kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, `eventsourcing.domain.model.entity.Discarded`

class `eventsourcing.domain.model.entity.`**TimestampedEntity**(*__created_on__: decimal.Decimal, **kwargs*)
Bases: *`eventsourcing.domain.model.entity.DomainEntity`*

class Event(*originator_id: uuid.UUID, **kwargs*)
Bases: `eventsourcing.domain.model.entity.Event`, *`eventsourcing.domain.model.events.EventWithTimestamp`*

Supertype for events of timestamped entities.

__mutate__(*obj: Optional[TTimestampedEntity]*) → Optional[TTimestampedEntity]
Updates 'obj' with values from self.

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

Published when a TimestampedEntity is created.

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

Published when a TimestampedEntity is changed.

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

Published when a TimestampedEntity is discarded.

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

class `eventsourcing.domain.model.entity.`**TimestampedVersionedEntity**(*__created_on__: decimal.Decimal, **kwargs*)
Bases: *`eventsourcing.domain.model.entity.TimestampedEntity`*, *`eventsourcing.domain.model.entity.VersionedEntity`*

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

Supertype for events of timestamped, versioned entities.

class Created(*originator_version: int = 0, *args, **kwargs*)
Bases: `eventsourcing.domain.model.entity.Created`, `eventsourcing.domain.model.entity.Created`, `eventsourcing.domain.model.entity.Event`

Published when a TimestampedVersionedEntity is created.

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

Published when a TimestampedVersionedEntity is created.

class Discarded(*originator_version: int*, ***kwargs*)

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

Published when a TimestampedVersionedEntity is discarded.

class `eventsourcing.domain.model.entity.`**TimeuuidedVersionedEntity**(*event_id: uuid.UUID*, ***kwargs*)

Bases: *`eventsourcing.domain.model.entity.TimeuuidedEntity`*, *`eventsourcing.domain.model.entity.VersionedEntity`*

aggregate

Base classes for aggregates in a domain driven design.

class `eventsourcing.domain.model.aggregate.`**BaseAggregateRoot**(***kwargs*)

Bases: *`eventsourcing.domain.model.entity.TimestampedVersionedEntity`*, `typing.Generic`

Root entity for an aggregate in a domain driven design.

class Event(*originator_version: int*, ***kwargs*)

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

Supertype for base aggregate root events.

class Created(*originator_version: int = 0*, **args*, ***kwargs*)

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

Triggered when an aggregate root is created.

class AttributeChanged(*originator_version: int*, ***kwargs*)

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

Triggered when an aggregate root attribute is changed.

class Discarded(*originator_version: int*, ***kwargs*)

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

Triggered when an aggregate root is discarded.

__init__(***kwargs*) → None

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

__publish__(*event: Sequence[TDomainEvent]*) → None

Defers publishing event(s) to subscribers, by adding event to internal collection of pending events.

__save__() → None

Publishes all pending events to subscribers.

class `eventsourcing.domain.model.aggregate.`**AggregateRootWithHashchainedEvents**(**args*, ***kwargs*)

Bases: *`eventsourcing.domain.model.entity.EntityWithHashchain`*, *`eventsourcing.domain.model.aggregate.BaseAggregateRoot`*

Extends aggregate root base class with hash-chained events.

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

Supertype for aggregate events.

class Created(*originator_version: int = 0*, **args*, ***kwargs*)
Bases: `eventsourcing.domain.model.entity.Created`, `eventsourcing.domain.model.aggregate.Created`, `eventsourcing.domain.model.aggregate.Event`

Triggered when an aggregate root is created.

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

Triggered when an aggregate root attribute is changed.

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

Triggered when an aggregate root is discarded.

class `eventsourcing.domain.model.aggregate.`**AggregateRoot**(**args*, ***kwargs*)
Bases: *`eventsourcing.domain.model.aggregate.AggregateRootWithHashchainedEvents`*

Original name for aggregate root base class with hash-chained events.

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

Supertype for aggregate events.

class Created(*originator_version: int = 0*, **args*, ***kwargs*)
Bases: `eventsourcing.domain.model.aggregate.Event`, `eventsourcing.domain.model.aggregate.Created`

Triggered when an aggregate root is created.

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

Triggered when an aggregate root attribute is changed.

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

Triggered when an aggregate root is discarded.

command

Commands as aggregates.

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

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

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

class Created(*originator_version: int = 0*, **args*, ***kwargs*)
Bases: `eventsourcing.domain.model.command.Event`, `eventsourcing.domain.model.aggregate.Created`

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

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

class Done(*originator_version: int*, ***kwargs*)
Bases: `eventsourcing.domain.model.command.Event`

mutate(*obj: eventsourcing.domain.model.command.Command*) → None
Updates ("mutates") given 'obj'.

Intended to be overridden by subclasses, as the most concise way of coding a default projection of the event (for example into the state of a domain entity).

The advantage of implementing a default projection using this method rather than __mutate__ is that you don't need to call super or return a value.

Parameters obj – domain entity to be mutated

decorators

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

`eventsourcing.domain.model.decorators.`**subscribe_to**(**args*) → Callable
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'.

Parameters 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()
```

`eventsourcing.domain.model.decorators.`**mutator**(*arg: Optional[Callable] = None*) → Callable
Structures mutator functions by allowing handlers to be registered for different types of event. When the deco-

rated function is called with an initial value and an event, it will call the handler that has been registered for that type of event.

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

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

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

@mutator(Entity)
def mutate(initial, event):
    raise NotImplementedError(type(event))

@mutate.register(Entity.Created)
def _(initial, event):
    return initial(**event.__dict__)

entity = mutate(None, Entity.Created())
```

`eventsourcing.domain.model.decorators.`**`attribute`**(*getter: Callable*) → property

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.`**`retry`**(*exc: Union[Type[Exception], Sequence[Type[Exception]]] = <class 'Exception'>, max_attempts: int = 1, wait: float = 0, stall: float = 0, verbose: bool = False*) → Callable

Retry decorator.

Parameters

- **`exc`** – List of exceptions that will cause the call to be retried if raised.
- **`max_attempts`** – Maximum number of attempts to try.
- **`wait`** – Amount of time to wait before retrying after an exception.
- **`stall`** – Amount of time to wait before the first attempt.
- **`verbose`** – If True, prints a message to STDOUT when retries occur.

Returns Returns the value returned by decorated function.

`eventsourcing.domain.model.decorators.`**`subclassevents`**(*cls: type*) → type

Decorator that avoids “boilerplate” subclassing of domain events.

For example, with this decorator you can do this:

```
@subclassevents
class Example(AggregateRoot):
    class SomethingHappened(DomainEvent): pass
```

rather than this:

```
class Example(AggregateRoot):
    class Event(AggregateRoot.Event): pass
    class Created(Event, AggregateRoot.Created): pass
    class Discarded(Event, AggregateRoot.Discarded): pass
    class AttributeChanged(Event, AggregateRoot.AttributeChanged): pass
    class SomethingHappened(Event): pass
```

You can apply this to a tree of domain event classes by defining the base class with attribute 'subclassevents = True'.

snapshot

Snapshotting is implemented in the domain layer as an event.

class eventsourcing.domain.model.snapshot.**Snapshot**(*originator_id: uuid.UUID, originator_version: int, topic: str, state: Optional[Dict[KT, VT]]*)

Bases: *eventsourcing.domain.model.events.EventWithTimestamp*, *eventsourcing.domain.model.events.EventWithOriginatorVersion*, *eventsourcing.domain.model.events.EventWithOriginatorID*, *eventsourcing.domain.model.events.AbstractSnapshot*

__init__(*originator_id: uuid.UUID, originator_version: int, topic: str, state: Optional[Dict[KT, VT]]*)

Initialises event attribute values directly from constructor kwargs.

topic

Path to the class of the snapshotted entity.

state

State of the snapshotted entity.

__mutate__(*obj: Optional[TEntity]*) → Optional[TEntity]

Updates 'obj' with values from 'self'.

Calls the 'mutate()' method.

Can be extended, but subclasses must call super and return an object to their caller.

Parameters **obj** – object (normally a domain entity) to be mutated

Returns mutated object

versioning

Support for upcasting the state of older version of domain events is implemented in base class *Upcastable*.

class eventsourcing.domain.model.versioning.**Upcastable**

Bases: *eventsourcing.whitehead.Event*

For things that are upcastable.

http://code.fed.wiki.org/view/wyatt-software/view/remote-database-schema-migration

"I was sure that we could not get the schema for WyCash Plus right on the first try. I was familiar with Smaltalk-80's object migration mechanisms. I designed a version that could serve us in a commercial software distribution environment.

"I chose to version each class independently and write that as a sequential integer in the serialized versions. Objects would be mutated to the current version on read. We supported all versions we ever had forever.

"I recorded mutation vectors for each version to the present. These could add, remove and reorder fields within an object. One-off mutation methods handled the rare case where the vectors were not enough description.

"We shipped migrations to our customers with each release to be performed on their own machines when needed without any intervention."

Ward Cunningham

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

`classmethod __upcast_state__``(`*obj_state: Dict[KT, VT]*`)` → Dict[KT, VT]
Upcasts obj_state from the version of the class when the object state was recorded, to be compatible with current version of the class.

`classmethod __upcast__``(`*obj_state: Dict[KT, VT], class_version: int*`)` → Dict[KT, VT]
Must be overridden in domain event classes that set class attribute '__class_version__' to a positive value.

This method is expected to upcast obj_state from method arg 'class_version' to the next version, and return obj_state.

One style of implementation is an if-else block, in which the conditional expressions match on the value of 'class_version'.

The implementation of this method must support upcasting each version of the class from 0 to one less than the current version. For example: a domain class with "__class_version__ = 1" will need to support upcasting "class_version == 0" to version 1; and a domain class with "__class_version__ = 2" will need to support both upcasting "class_version == 0" to version 1, and also upcasting "class_version == 1" to version 2.

To support backward compatibility, the recorded state of old versions of an event class will need to be upcast to work with new versions of the software. Commonly, this involves supplying default values for attributes missing on old versions of events, after an event class has been changed by adding a new attribute.

In a situation where a new version of an event class needs to be used by existing software, it would be necessary also to support forward compatibility. Examples of this situation include the situation of deploying with rolling updates, and the situation where consumers would receive and attempt to handle the new version of the event but cannot be updated before the new event class version is deployed.

To support forward compatibility, it is necessary to restrict changes to be merely additive: either adding new attributes to an event, or adding new behaviours to the type of an existing attribute. Event attributes shouldn't be removed (or renamed), existing attributes should not have aspects of their behaviour removed, and the semantics of an existing attribute should not change. If the type of an attribute value is changed, the new type should substitutable for the old type. The old software will then simply ignore new attributes, and it will simply ignore new aspects of new types of value.

For example, the type of an attribute value can be changed to use a subtype of the previous attribute value type. The possible range of a value can be reduced when changing the type of an attribute value, since all the values of the new type will be supported by the old software.

Increasing the possible range of a value will introduce values that cannot be used by the old software, and changing the type of an attribute value to a supertype will remove behaviour that the old software may depend on.

Of course, if the old software doesn't in fact depend on aspects that are removed, then those aspects can be removed without actually breaking anything.

If backward and forward compatibility is required, and it is felt that an event class needs to be changed that would cause an attribute to be removed or the type of an attribute to be more general, or the range of values of an attribute to increase, then according to Greg Young's book 'Versioning in an Event Sourced System' it is better to to add a new event type. However, if the old software would break because this new event type is not supported, then supporting forward compatibility would be elusive.

In summary, supporting forward compatibility restricts model changes to adding attributes to existing model classes (which implies the versioned event class' upcast method will supply default values when upcasting the state of old versions of the event).

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.`**`Timebucketedlog`**(*name: uuid.UUID*, *bucket_size: Optional[str] = None*, ***kwargs*)

Bases: *`eventsourcing.domain.model.entity.TimestampedVersionedEntity`*

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

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

Supertype for events of time-bucketed log.

class **`Started`**(*originator_version: int = 0*, **args*, ***kwargs*)

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

class **`BucketSizeChanged`**(*originator_version: int*, ***kwargs*)

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

`__init__`(*name: uuid.UUID*, *bucket_size: Optional[str] = None*, ***kwargs*)

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

class `eventsourcing.domain.model.timebucketedlog.`**`TimebucketedlogRepository`**

Bases: `eventsourcing.domain.model.repository.AbstractEntityRepository`

`get_or_create`(*log_name: uuid.UUID*, *bucket_size: str*) → eventsourcing.domain.model.timebucketedlog.Timebucketedlog

Gets or creates a log.

class `eventsourcing.domain.model.timebucketedlog.`**`MessageLogged`**(*message: str*, *originator_id: uuid.UUID*)

Bases: *`eventsourcing.domain.model.events.EventWithTimestamp`*, *`eventsourcing.domain.model.events.EventWithOriginatorID`*, *`eventsourcing.domain.model.events.LoggedEvent`*

`__init__`(*message: str*, *originator_id: uuid.UUID*)

Initialises event attribute values directly from constructor kwargs.

collection

Collections.

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

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

Supertype for events of collection entities.

class **`Created`**(*originator_version: int = 0*, **args*, ***kwargs*)
Bases: `eventsourcing.domain.model.collection.Event`, `eventsourcing.domain.model.entity.Created`

Published when collection is created.

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

Published when collection is discarded.

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

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

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

`mutate`(*obj: eventsourcing.domain.model.collection.Collection*) → None
Updates ("mutates") given 'obj'.

Intended to be overridden by subclasses, as the most concise way of coding a default projection of the event (for example into the state of a domain entity).

The advantage of implementing a default projection using this method rather than __mutate__ is that you don't need to call super or return a value.

Parameters **`obj`** – domain entity to be mutated

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

`mutate`(*obj: eventsourcing.domain.model.collection.Collection*) → None
Updates ("mutates") given 'obj'.

Intended to be overridden by subclasses, as the most concise way of coding a default projection of the event (for example into the state of a domain entity).

The advantage of implementing a default projection using this method rather than __mutate__ is that you don't need to call super or return a value.

Parameters **`obj`** – domain entity to be mutated

class `eventsourcing.domain.model.collection.`**`AbstractCollectionRepository`**
Bases: `eventsourcing.domain.model.repository.AbstractEntityRepository`

array

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

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

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

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

class eventsourcing.domain.model.array.**BigArray**(*array_id*, *repo*)
Bases: eventsourcing.domain.model.array.Array

A virtual array holding items in indexed positions, across a number of Array instances.

Getting and setting items at index position is supported. Slices are supported, and operate across the underlying arrays. Appending is also supported.

BigArray is designed to overcome the concern of needing a single large sequence that may not be suitably stored in any single partiton. In simple terms, if events of an aggregate can fit in a partition, we can use the same size partition to make a tree of arrays that will certainly be capable of sequencing all the events of the application in a single stream.

With normal size base arrays, enterprise applications can expect read and write time to be approximately constant with respect to the number of items in the array.

The array is composed of a tree of arrays, which gives the capacity equal to the size of each array to the power of the size of each array. If the arrays are limited to be about the maximum size of an aggregate event stream (a large number but not too many that would cause there to be too much data in any one partition, let's say 1000s to be safe) then it would be possible to fit such a large number of aggregates in the corresponding BigArray, that we can be confident it would be full.

Write access time in the worst case, and the time to identify the index of the last item in the big array, is proportional to the log of the highest assigned index to 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.

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

get_last_array()
Returns last array in compound.

Return type CompoundSequenceReader

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

__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.

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

`calc_parent``(i, j, h)`
Returns get_big_array and end of span of parent sequence that contains given child.

class `eventsourcing.domain.model.array.`**`AbstractArrayRepository`**`(array_size=10000, *args, **kwargs)`
Bases: `eventsourcing.domain.model.repository.AbstractEntityRepository`

Repository for sequence objects.

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

`__getitem__``(entity_id)` → Any
Returns sequence for given ID.

class `eventsourcing.domain.model.array.`**`AbstractBigArrayRepository`**
Bases: `eventsourcing.domain.model.repository.AbstractEntityRepository`

Repository for compound sequence objects.

`subrepo`
Sub-sequence repository.

`__getitem__``(entity_id)` → Any
Returns sequence for given ID.

1.19.2 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.

sequenceditem

The persistence model for storing events.

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

Bases: `tuple`

sequence_id
Alias for field number 0

position
Alias for field number 1

topic
Alias for field number 2

state
Alias for field number 3

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

static __new__(*_cls*, *sequence_id: uuid.UUID*, *position: int*, *topic: str*, *state: bytes*)
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(***kwds*)
Return a new SequencedItem object replacing specified fields with new values

class eventsourcing.infrastructure.sequenceditem.**StoredEvent**(*originator_id*, *originator_version*, *topic*, *state*)

Bases: `tuple`

originator_id
Alias for field number 0

originator_version
Alias for field number 1

topic
Alias for field number 2

state
Alias for field number 3

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

static __new__(*_cls*, *originator_id: uuid.UUID*, *originator_version: int*, *topic: str*, *state: bytes*)
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(***kwds*)
Return a new StoredEvent object replacing specified fields with new values

sequenceditemmapper

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

class `eventsourcing.infrastructure.sequenceditemmapper.`**AbstractSequencedItemMapper**(***kwargs*)
Bases: `typing.Generic`, `abc.ABC`

__init__(***kwargs*)
Initialises mapper.

item_from_event(*domain_event: TEvent*) → NamedTuple
Constructs and returns a sequenced item for given domain event.

event_from_item(*sequenced_item: NamedTuple*) → TEvent
Constructs and returns a domain event for given sequenced item.

json_dumps(*o: object*) → bytes
Encodes given object as JSON.

json_loads(*s: str*) → object
Decodes given JSON as object.

event_from_topic_and_state(*topic: str*, *state: bytes*) → TEvent
Resolves topic to an event class, decodes state, and constructs an event.

event_from_notification(*notification*)
Reconstructs domain event from an event notification.

Parameters **notification** – The event notification.

Returns A domain event.

class eventsourcing.infrastructure.sequenceditemmapper.**SequencedItemMapper**(*sequenced_item_class: Optional[Type[NamedTup = None, sequence_id_attr_name: Optional[str] = None, position_attr_name: Optional[str] = None, json_encoder_class: Optional[Type[eventsourc = None, sort_keys: bool = False, json_decoder_class: Optional[Type[eventsourc = None, cipher: Optional[eventsourcing.ut = None, compressor: Any = None, other_attr_names: Tuple[str, ...] = ()*)

Bases: `eventsourcing.infrastructure.sequenceditemmapper.AbstractSequencedItemMapper`

Uses JSON to transcode domain events.

`__init__`(*sequenced_item_class: Optional[Type[NamedTuple]] = None, sequence_id_attr_name: Optional[str] = None, position_attr_name: Optional[str] = None, json_encoder_class: Optional[Type[eventsourcing.utils.transcoding.ObjectJSONEncoder]] = None, sort_keys: bool = False, json_decoder_class: Optional[Type[eventsourcing.utils.transcoding.ObjectJSONDecoder]] = None, cipher: Optional[eventsourcing.utils.cipher.aes.AESCipher] = None, compressor: Any = None, other_attr_names: Tuple[str, ...] = ()*)
Initialises mapper.

`item_from_event`(*domain_event: TEvent*) → NamedTuple
Constructs a sequenced item from a domain event.

`construct_item_args`(*domain_event: TEvent*) → Tuple
Constructs attributes of a sequenced item from the given domain event.

`json_dumps`(*o: object*) → bytes
Encodes given object as JSON.

`event_from_item`(*sequenced_item: NamedTuple*) → TEvent
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: str, state: bytes*) → TEvent
Resolves topic to an event class, decodes state, and constructs an event.

`json_loads`(*s: str*) → Dict[KT, VT]
Decodes given JSON as object.

`event_from_notification`(*notification*)
Reconstructs domain event from an event notification.

Parameters **`notification`** – The event notification.

Returns A domain event.

base

Abstract base classes for the infrastructure layer.

class `eventsourcing.infrastructure.base.`**`AbstractRecordManager`**(***kwargs*)
Bases: `abc.ABC`

`__init__`(***kwargs*)
Initialises record manager.

`record_class`
Returns record class to be used by the record manager.

`record_items`(*sequenced_items: Iterable[NamedTuple]*) → None
Writes sequenced items into the datastore.

`record_item`(*sequenced_item: NamedTuple*) → None
Writes sequenced item into the datastore.

`get_item`(*sequence_id: uuid.UUID, position: int*) → NamedTuple
Gets sequenced item from the datastore.

`get_items`(*sequence_id: uuid.UUID, gt: Optional[int] = None, gte: Optional[int] = None, lt: Optional[int] = None, lte: Optional[int] = None, limit: Optional[int] = None, query_ascending: bool = True, results_ascending: bool = True*) → Iterator[NamedTuple]
Iterates over records in sequence.

get_record(*sequence_id: uuid.UUID*, *position: int*) → Any
Gets record at position in sequence.

get_records(*sequence_id: uuid.UUID*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *query_ascending: bool = True*, *results_ascending: bool = True*) → Sequence[Any]
Returns records for a sequence.

all_sequence_ids() → Iterable[uuid.UUID]
Returns all sequence IDs.

delete_record(*record: Any*) → None
Removes permanently given record from the table.

class eventsourcing.infrastructure.base.**BaseRecordManager**(*record_class: type*, *sequenced_item_class: Type[eventsourcing.infrastructure.sequencediten = <class 'eventsourcing.infrastructure.sequenceditem.SequencedItem contiguous_record_ids: bool = False*, *application_name: str = ''*, *pipeline_id: int = 0*, ***kwargs*)
Bases: eventsourcing.infrastructure.base.AbstractRecordManager

__init__(*record_class: type*, *sequenced_item_class: Type[eventsourcing.infrastructure.sequenceditem.SequencedItem] = <class 'eventsourcing.infrastructure.sequenceditem.SequencedItem'>*, *contiguous_record_ids: bool = False*, *application_name: str = ''*, *pipeline_id: int = 0*, ***kwargs*)
Initialises record manager.

record_class
Returns record class to be used by the record manager.

get_item(*sequence_id: uuid.UUID*, *position: int*) → eventsourcing.infrastructure.sequenceditem.SequencedItem
Gets sequenced item from the datastore.

get_items(*sequence_id: uuid.UUID*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *query_ascending: bool = True*, *results_ascending: bool = True*) → Iterator[eventsourcing.infrastructure.sequenceditem.SequencedItem]
Returns sequenced item generator.

list_items(**args*, ***kwargs*) → List[eventsourcing.infrastructure.sequenceditem.SequencedItem]
Returns list of sequenced items.

to_record(*sequenced_item: NamedTuple*) → object
Constructs a record object from given sequenced item object.

from_record(*record: object*) → eventsourcing.infrastructure.sequenceditem.SequencedItem
Constructs and returns a sequenced item object, from given ORM object.

class eventsourcing.infrastructure.base.**RecordManagerWithNotifications**(*record_class: type, sequenced_item_class: Type[eventsourcing.infrastru = <class 'eventsourcing.infrastructure.sequencedi contiguous_record_ids: bool = False, application_name: str = '', pipeline_id: int = 0, **kwargs*)

Bases: *eventsourcing.infrastructure.base.BaseRecordManager*

get_max_notification_id() → int
Return maximum notification ID in pipeline.

get_notification_records(*start: Optional[int] = None, stop: Optional[int] = None, *args, **kwargs*) → Iterable[T_co]
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.

class eventsourcing.infrastructure.base.**RecordManagerWithTracking**(*tracking_record_class: Optional[type] = None, *args, **kwargs*)

Bases: *eventsourcing.infrastructure.base.RecordManagerWithNotifications*

ACID record managers can write tracking records and event records in an atomic transaction, needed for atomic processing in process applications.

__init__(*tracking_record_class: Optional[type] = None, *args, **kwargs*) → None
Initialises record manager.

write_records(*records: Iterable[Any], tracking_kwargs: Optional[Dict[str, Union[str, int]]] = None, orm_objs_pending_save: Optional[Sequence[Any]] = None, orm_objs_pending_delete: Optional[Sequence[Any]] = None*) → None
Writes tracking, event and notification records for a process event. :param orm_objs_pending_delete: :param orm_objs_pending_save:

get_max_tracking_record_id(*upstream_application_name: str*) → int
Return maximum tracking record ID for notification from upstream application in pipeline.

has_tracking_record(*upstream_application_name: str*, *pipeline_id: int*, *notification_id: int*) → bool
True if tracking record exists for notification from upstream in pipeline.

get_pipeline_and_notification_id(*sequence_id: uuid.UUID*, *position: int*) → Tuple
Returns pipeline ID and notification ID for event at given position in given sequence.

class eventsourcing.infrastructure.base.**SQLRecordManager**(**args*, ***kwargs*)
Bases: `eventsourcing.infrastructure.base.RecordManagerWithTracking`

Common aspects of SQL record managers, such as SQLAlchemy and Django record managers.

__init__(**args*, ***kwargs*)
Initialises record manager.

record_items(*sequenced_items: Iterable[NamedTuple]*) → None
Writes sequenced items into the datastore.

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

_prepare_insert(*tmpl: str, record_class: type, field_names: List[str], placeholder_for_id: bool = False*) → Any
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.

make_placeholder(*field_name: str*) → str
Returns "placeholder" string for late binding of values to query.

Depends on record manager's adapted database system or adapted ORM.

insert_values
SQL statement that inserts records without ID.

insert_tracking_record
SQL statement that inserts tracking records.

get_record_table_name(*record_class: type*) → str
Returns table name - used in raw queries.

Return type str

class eventsourcing.infrastructure.base.**AbstractEventStore**(*record_manager: TRecordManager*, *event_mapper: eventsourcing.infrastructure.sequenceditemmapper.Abstra*
Bases: `abc.ABC`, `typing.Generic`

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

__init__(*record_manager: TRecordManager*, *event_mapper: eventsourcing.infrastructure.sequenceditemmapper.AbstractSequencedItemMapper*)
Initialises event store object.

Parameters

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

store_events(*events: Iterable[TEvent]*) → None
Put domain event in event store for later retrieval.

iter_events(*originator_id: uuid.UUID*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *is_ascending: bool = True*, *page_size: Optional[int] = None*) → Iterable[TEvent]

Returns iterable of domain events for given entity ID.

list_events(**args*, ***kwargs*) → List[TEvent]

Returns list of domain events for given entity ID.

get_event(*originator_id: uuid.UUID*, *position: int*) → TEvent

Returns a single domain event.

get_most_recent_event(*originator_id: uuid.UUID*, *lt: Optional[int] = None*, *lte: Optional[int] = None*) → Optional[TEvent]

Returns most recent domain event for given entity ID.

all_events() → Iterable[TEvent]

Returns all domain events in the event store.

This works by iterating over all sequences, so doesn't return events in order. Use a Notification Log to project application state.

get_domain_events(*originator_id: uuid.UUID*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *is_ascending: bool = True*, *page_size: Optional[int] = None*) → Iterable[TEvent]

Deprecated. Please use iter_events() instead.

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

items_from_events(*events: Iterable[TEvent]*) → Iterable[NamedTuple]

Maps domain event to sequenced item namedtuple.

Parameters **events** – An iterable of events.

datastore

Base classes for concrete datastore classes.

class eventsourcing.infrastructure.datastore.**DatastoreSettings**

Bases: object

Settings for Datastore.

class `eventsourcing.infrastructure.datastore.`**`AbstractDatastore`**(*settings: TDatastoreSettings*)

Bases: `abc.ABC`, `typing.Generic`

`can_drop_tables = True`
Datastores hold stored event records, used by a record manager.

`__init__`(*settings: TDatastoreSettings*)
Initialize self. See help(type(self)) for accurate signature.

`setup_connection`() → None
Sets up a connection to a datastore.

`close_connection`() → None
Drops connection to a datastore.

`setup_tables`() → None
Sets up tables used to store events.

`setup_table`(*table: Any*) → None
Sets up given table.

`drop_tables`() → None
Drops tables used to store events.

`drop_table`(*table: Any*) → None
Drops given table.

`truncate_tables`() → None
Truncates tables used to store events.

exception `eventsourcing.infrastructure.datastore.`**`DatastoreError`**
Bases: `Exception`

exception `eventsourcing.infrastructure.datastore.`**`DatastoreConnectionError`**
Bases: *`eventsourcing.infrastructure.datastore.DatastoreError`*

exception `eventsourcing.infrastructure.datastore.`**`DatastoreTableError`**
Bases: *`eventsourcing.infrastructure.datastore.DatastoreError`*

cassandra

Classes for event sourcing with Apache Cassandra.

class eventsourcing.infrastructure.cassandra.datastore.**CassandraSettings**(*hosts=None*, *port=None*, *protocol_version=None*, *default_keyspace=None*, *consistency=None*, *replication_factor=None*, *username=None*, *password=None*)

Bases: *eventsourcing.infrastructure.datastore.DatastoreSettings*

__init__(*hosts=None*, *port=None*, *protocol_version=None*, *default_keyspace=None*, *consistency=None*, *replication_factor=None*, *username=None*, *password=None*)
Initialize self. See help(type(self)) for accurate signature.

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

Bases: *eventsourcing.infrastructure.datastore.AbstractDatastore*

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

setup_connection()
Sets up a connection to a datastore.

close_connection()
Drops connection to a datastore.

setup_tables()
Sets up tables used to store events.

setup_table(*table*) → None
Sets up given table.

drop_tables()
Drops tables used to store events.

drop_table(**_*)
Drops given table.

truncate_tables()
Truncates tables used to store events.

class eventsourcing.infrastructure.cassandra.factory.**CassandraInfrastructureFactory**(*record_m*
Op-
tional[Ty
=
None,
se-
*quenced_*
Op-
tional[Ty
=
None,
event_sto
Op-
tional[Ty
=
None,
se-
*quenced_*
Op-
tional[Ty
=
None,
json_enco
Op-
tional[Ty
=
None,
sort_keys
bool
=
False,
json_deco
Op-
tional[Ty
=
None,
in-
te-
ger_seque
Op-
tional[typ
=
None,
times-
tamp_seq
Op-
tional[typ
=
None,
snap-
shot_reco
Op-
tional[typ
=
None,
con-
tigu-
ous_recor
bool
=

Bases: *eventsourcing.infrastructure.factory.InfrastructureFactory*

Infrastructure factory for Cassandra.

record_manager_class
alias of *eventsourcing.infrastructure.cassandra.manager.CassandraRecordManager*

integer_sequenced_record_class
alias of *eventsourcing.infrastructure.cassandra.records.IntegerSequencedRecord*

timestamp_sequenced_record_class
alias of *eventsourcing.infrastructure.cassandra.records.TimestampSequencedRecord*

snapshot_record_class
alias of *eventsourcing.infrastructure.cassandra.records.SnapshotRecord*

class eventsourcing.infrastructure.cassandra.manager.**CassandraRecordManager**(*record_class: type, sequenced_item_class: Type[eventsourcing.in = <class 'eventsourcing.infrastructure.seq contiguous_record_ids: bool = False, application_name: str = '', pipeline_id: int = 0, **kwargs*)

Bases: *eventsourcing.infrastructure.base.BaseRecordManager*

record_items(*sequenced_items*)
Writes sequenced items into the datastore.

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.

all_sequence_ids()
Returns all sequence IDs.

delete_record(*record*)
Removes permanently given record from the table.

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`

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`

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`

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`

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`

django

Infrastructure for event sourcing with the Django ORM. This package functions as a Django application. It can be included in "INSTALLED_APPS" in settings.py in your Django project. There is just one migration, to create tables that do not exist.

class eventsourcing.infrastructure.django.factory.**DjangoInfrastructureFactory**(*tracking_record_c Optional[type] = None*, **args*, ***kwargs*)

Bases: *eventsourcing.infrastructure.factory.InfrastructureFactory*

Infrastructure factory for Django.

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

__init__(*tracking_record_class: Optional[type] = None*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

construct_integer_sequenced_record_manager(***kwargs*) → eventsourcing.infrastructure.base.AbstractRecordManager
Constructs Django record manager.

Returns A Django record manager.

Return type *DjangoRecordManager*

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

Bases: *eventsourcing.infrastructure.base.SQLRecordManager*

write_records(*records: Iterable[Any]*, *tracking_kwargs: Optional[Dict[str, Union[str, int]]] = None*, *orm_objs_pending_save: Optional[Sequence[Any]] = None*, *orm_objs_pending_delete: Optional[Sequence[Any]] = None*) → None
Writes tracking, event and notification records for a process event. :param orm_objs_pending_delete: :param orm_objs_pending_save:

make_placeholder(*_: str*) → str
Returns "placeholder" string for late binding of values to query.

Depends on record manager's adapted database system or adapted ORM.

get_record_table_name(*record_class: type*) → str
Returns table name from SQLAlchemy record class.

get_record(*sequence_id: uuid.UUID*, *position: int*) → Any
Gets record at position in sequence.

get_records(*sequence_id: uuid.UUID*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *query_ascending: bool = True*, *results_ascending: bool = True*) → Sequence[Any]
Returns records for a sequence.

get_notification_records(*start: Optional[int] = None*, *stop: Optional[int] = None*, **args*, ***kwargs*) → Iterable[T_co]
Returns all records in the table.

delete_record(*record: Any*) → None
Permanently removes record from table.

get_max_notification_id() → int
Return maximum notification ID in pipeline.

get_max_tracking_record_id(*upstream_application_name: str*) → int
Return maximum tracking record ID for notification from upstream application in pipeline.

has_tracking_record(*upstream_application_name: str*, *pipeline_id: int*, *notification_id: int*) → bool
True if tracking record exists for notification from upstream in pipeline.

all_sequence_ids() → Iterable[uuid.UUID]
Returns all sequence IDs.

class eventsourcing.infrastructure.django.models.**IntegerSequencedRecord**(*id*, *sequence_id*, *position*, *topic*, *state*)
Bases: `django.db.models.base.Model`

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

class eventsourcing.infrastructure.django.models.**TimestampSequencedRecord**(*id*, *sequence_id*, *position*, *topic*, *state*)
Bases: `django.db.models.base.Model`

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

class eventsourcing.infrastructure.django.models.**SnapshotRecord**(*uid*, *sequence_id*, *position*, *topic*, *state*)
Bases: `django.db.models.base.Model`

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

class eventsourcing.infrastructure.django.models.**EntitySnapshotRecord**(*uid*, *application_name*, *originator_id*, *originator_version*, *topic*, *state*)

Bases: `django.db.models.base.Model`

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

class eventsourcing.infrastructure.django.models.**StoredEventRecord**(*uid*, *application_name*, *originator_id*, *originator_version*, *pipeline_id*, *notification_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`

class eventsourcing.infrastructure.django.models.**NotificationTrackingRecord**(*uid*, *application_name*, *upstream_application_n* *pipeline_id*, *notification_id*)

Bases: `django.db.models.base.Model`

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

sqlalchemy

Classes for event sourcing with SQLAlchemy.

class `eventsourcing.infrastructure.sqlalchemy.datastore.`**`SQLAlchemySettings`**(*uri: Optional[str] = None, pool_size: Optional[int] = None*)

Bases: *`eventsourcing.infrastructure.datastore.DatastoreSettings`*

`__init__`(*uri: Optional[str] = None, pool_size: Optional[int] = None*)
Initialize self. See help(type(self)) for accurate signature.

class `eventsourcing.infrastructure.sqlalchemy.datastore.`**`SQLAlchemyDatastore`**(*settings: eventsourcing.infrastructure.sql base: sqlalchemy.ext.declar = <class 'sqlalchemy.ext.decla tables: Optional[Sequence[T_co = None, connection_strategy: str = 'plain', session: Union[sqlalchemy.orm sqlalchemy.orm.scopi None] = None*)

Bases: *`eventsourcing.infrastructure.datastore.AbstractDatastore`*

`__init__`(*settings: eventsourcing.infrastructure.sqlalchemy.datastore.SQLAlchemySettings, base: sqlalchemy.ext.declarative.api.DeclarativeMeta = <class 'sqlalchemy.ext.declarative.api.Base'>, tables: Optional[Sequence[T_co]] = None, connection_strategy: str = 'plain', session: Union[sqlalchemy.orm.session.Session, sqlalchemy.orm.scoping.scoped_session, None] = None*)

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

setup_connection() → None
Sets up a connection to a datastore.

setup_tables() → None
Sets up tables used to store events.

setup_table(*table: Any*) → None
Sets up given table.

drop_tables() → None
Drops tables used to store events.

drop_table(*table: Any*) → None
Drops given table.

truncate_tables() → None
Truncates tables used to store events.

close_connection() → None
Drops connection to a datastore.

class eventsourcing.infrastructure.sqlalchemy.factory.**SQLAlchemyInfrastructureFactory**(*sessio Any, uri: Op- tional[= None, pool_s Op- tional[= None, track- ing_re Op- tional[= None, *args, **kwa*

Bases: *eventsourcing.infrastructure.factory.InfrastructureFactory*

Infrastructure factory for SQLAlchemy infrastructure.

record_manager_class
alias of *eventsourcing.infrastructure.sqlalchemy.manager.SQLAlchemyRecordManager*

integer_sequenced_record_class
alias of *eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord*

timestamp_sequenced_record_class
alias of *eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedNoIDRecord*

snapshot_record_class
alias of *eventsourcing.infrastructure.sqlalchemy.records.SnapshotRecord*

tracking_record_class
alias of *eventsourcing.infrastructure.sqlalchemy.records.NotificationTrackingRecord*

__init__(*session: Any*, *uri: Optional[str] = None*, *pool_size: Optional[int] = None*, *tracking_record_class: Optional[type] = None*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

construct_integer_sequenced_record_manager(***kwargs*) → eventsourcing.infrastructure.base.AbstractRecordManager
Constructs SQLAlchemy record manager.

Returns An SQLAlchemy record manager.

Return type *SQLAlchemyRecordManager*

construct_record_manager(*record_class: Optional[type]*, *sequenced_item_class: Optional[Type[NamedTuple]] = None*, ***kwargs*) → eventsourcing.infrastructure.base.AbstractRecordManager
Constructs SQLAlchemy record manager.

Returns An SQLAlchemy record manager.

Return type *SQLAlchemyRecordManager*

construct_datastore() → Optional[eventsourcing.infrastructure.datastore.AbstractDatastore]
Constructs SQLAlchemy datastore.

Return type *SQLAlchemyDatastore*

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

__init__(*session: Any*, **args*, ***kwargs*)
Initialises record manager.

_prepare_insert(*tmpl: Any, record_class: type, field_names: List[str], placeholder_for_id: bool = False*) → Any
With transaction isolation level of "read committed" this should generate records with a contiguous sequence of integer IDs, assumes an indexed ID column, the database-side SQL max function, the insert-select-from form, and optimistic concurrency control.

make_placeholder(*field_name: str*) → str
Returns "placeholder" string for late binding of values to query.

Depends on record manager's adapted database system or adapted ORM.

write_records(*records: Iterable[Any]*, *tracking_kwargs: Optional[Dict[str, Union[str, int]]] = None*, *orm_objs_pending_save: Optional[Sequence[Any]] = None*, *orm_objs_pending_delete: Optional[Sequence[Any]] = None*) → None
Writes tracking, event and notification records for a process event. :param orm_objs_pending_delete: :param orm_objs_pending_save:

get_records(*sequence_id: uuid.UUID*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *query_ascending: bool = True*, *results_ascending: bool = True*) → Sequence[Any]
Returns records for a sequence.

get_notification_records(*start: Optional[int] = None*, *stop: Optional[int] = None*, **args*, ***kwargs*) → Iterable[T_co]
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: uuid.UUID*, *position: int*) → Any
Gets record at position in sequence.

get_max_notification_id() → int
Return maximum notification ID in pipeline.

get_max_tracking_record_id(*upstream_application_name: str*) → int
Return maximum tracking record ID for notification from upstream application in pipeline.

has_tracking_record(*upstream_application_name: str*, *pipeline_id: int*, *notification_id: int*) → bool
True if tracking record exists for notification from upstream in pipeline.

all_sequence_ids() → Iterable[uuid.UUID]
Returns all sequence IDs.

delete_record(*record: Any*) → None
Permanently removes record from table.

get_record_table_name(*record_class: type*) → str
Returns table name - used in raw queries.

Return type str

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.

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.

eventsourcing.infrastructure.sqlalchemy.records.**IntegerSequencedRecord**
alias of *eventsourcing.infrastructure.sqlalchemy.records.IntegerSequencedWithIDRecord*

class eventsourcing.infrastructure.sqlalchemy.records.**TimestampSequencedWithIDRecord**(***kwarg*
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.

class eventsourcing.infrastructure.sqlalchemy.records.**TimestampSequencedNoIDRecord**(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

`__init__`(***kwargs*)
A simple constructor that allows initialization from kwargs.

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

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

`eventsourcing.infrastructure.sqlalchemy.records.`**`TimestampSequencedRecord`**
alias of *`eventsourcing.infrastructure.sqlalchemy.records.TimestampSequencedNoIDRecord`*

`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.

`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.

`class` `eventsourcing.infrastructure.sqlalchemy.records.`**`StoredEventRecord`**(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

`__init__`(***kwargs*)
A simple constructor that allows initialization from kwargs.

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

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

`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.

popo

Infrastructure for event sourcing with "plain old Python objects".

class eventsourcing.infrastructure.popo.factory.**PopoInfrastructureFactory**(*record_manager_class: Optional[Type[eventsourcin = None, sequenced_item_class: Optional[Type[NamedTuple = None, event_store_class: Optional[Type[eventsourcin = None, sequenced_item_mapper_c Optional[Type[eventsourcin = None, json_encoder_class: Optional[Type[json.encode = None, sort_keys: bool = False, json_decoder_class: Optional[Type[json.decode = None, integer_sequenced_record_c Optional[type] = None, timestamp_sequenced_record Optional[type] = None, snapshot_record_class: Optional[type] = None, contiguous_record_ids: bool =*

Bases: *eventsourcing.infrastructure.factory.InfrastructureFactory*

record_manager_class
alias of *eventsourcing.infrastructure.popo.manager.PopoRecordManager*

integer_sequenced_record_class
alias of *eventsourcing.infrastructure.popo.records.IntegerSequencedRecord*

snapshot_record_class
alias of *eventsourcing.infrastructure.popo.records.SnapshotRecord*

class eventsourcing.infrastructure.popo.manager.**PopoRecordManager**(**args*, ***kwargs*)
Bases: *eventsourcing.infrastructure.base.RecordManagerWithTracking*

__init__(**args*, ***kwargs*)
Initialises record manager.

all_sequence_ids() → List[uuid.UUID]
Returns all sequence IDs.

delete_record(*record: Any*) → None
Removes permanently given record from the table.

get_max_notification_id() → int
Return maximum notification ID in pipeline.

get_notification_records(*start: Optional[int] = None*, *stop: Optional[int] = None*, **args*, ***kwargs*) → Iterable[T_co]
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_max_tracking_record_id(*upstream_application_name: str*) → int
Return maximum tracking record ID for notification from upstream application in pipeline.

get_record(*sequence_id: uuid.UUID*, *position: int*) → Any
Gets record at position in sequence.

get_records(*sequence_id: uuid.UUID*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *query_ascending: bool = True*, *results_ascending: bool = True*) → Sequence[Any]
Returns records for a sequence.

has_tracking_record(*upstream_application_name: str*, *pipeline_id: int*, *notification_id: int*) → bool
True if tracking record exists for notification from upstream in pipeline.

record_items(*sequenced_items: Iterable[NamedTuple]*) → None
Writes sequenced items into the datastore.

write_records(*records: Iterable[Any], tracking_kwargs: Optional[Dict[str, Union[str, int]]] = None, orm_objs_pending_save: Optional[Sequence[Any]] = None, orm_objs_pending_delete: Optional[Sequence[Any]] = None*) → None
Writes tracking, event and notification records for a process event. :param orm_objs_pending_delete: :param orm_objs_pending_save:

to_record(*sequenced_item: NamedTuple*) → object
Constructs a record object from given sequenced item object.

class eventsourcing.infrastructure.popo.mapper.**SequencedItemMapperForPopo**(*sequenced_item_class: Optional[Type[NamedTuple* *= None*, *sequence_id_attr_name: Optional[str] = None*, *position_attr_name: Optional[str] = None*, *json_encoder_class: Optional[Type[eventsourcin* *= None*, *sort_keys: bool = False*, *json_decoder_class: Optional[Type[eventsourcin* *= None*, *cipher: Optional[eventsourcing.util* *= None*, *compressor: Any = None*, *other_attr_names: Tuple[str, ...] = ()*)

Bases: *eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper*

class eventsourcing.infrastructure.popo.records.**IntegerSequencedRecord**(*sequenced_item: NamedTuple*)

Bases: object

Encapsulates sequenced item tuple (containing real event object).

__init__(*sequenced_item: NamedTuple*)
Initialize self. See help(type(self)) for accurate signature.

class eventsourcing.infrastructure.popo.records.**SnapshotRecord**(*sequenced_item: NamedTuple*)
Bases: *eventsourcing.infrastructure.popo.records.IntegerSequencedRecord*

class eventsourcing.infrastructure.popo.records.**StoredEventRecord**(*sequenced_item: NamedTuple*)
Bases: *eventsourcing.infrastructure.popo.records.IntegerSequencedRecord*

Encapsulates sequenced item tuple (containing real event object).

Allows other attributes to be set, such as notification ID.

eventstore

The event store provides the interface to the event sourcing persistence mechanism that is used by applications.

class eventsourcing.infrastructure.eventstore.**EventStore**(*record_manager: TRecordManager, event_mapper: eventsourcing.infrastructure.sequenceditemmapper.AbstractS*
Bases: *eventsourcing.infrastructure.base.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.

iterator_class
alias of *eventsourcing.infrastructure.iterators.SequencedItemIterator*

store_events(*events: Iterable[TEvent]*) → None
Appends given domain event, or list of domain events, to their sequence.

Parameters events – domain event, or list of domain events

items_from_events(*events: Iterable[TEvent]*) → Iterable[NamedTuple]
Maps domain event to sequenced item namedtuple.

An iterable of events.

iter_events(*originator_id: uuid.UUID, gt: Optional[int] = None, gte: Optional[int] = None, lt: Optional[int] = None, lte: Optional[int] = None, limit: Optional[int] = None, is_ascending: bool = True, page_size: Optional[int] = None*) → Iterable[TEvent]
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_event`(*originator_id: uuid.UUID*, *position: int*) → TEvent

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_most_recent_event`(*originator_id: uuid.UUID*, *lt: Optional[int] = None*, *lte: Optional[int] = None*) → Optional[TEvent]

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

`all_events`() → Iterable[TEvent]

Yields all domain events in the event store.

This method iterates over the sequence IDs, and returns all the events for each sequence effectively concatenated together.

Use a notification log to propagate events from an application as a stable append-only sequence.

eventsourcedrepository

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

class eventsourcing.infrastructure.eventsourcedrepository.**EventSourcedRepository**(*event_store: eventsourcing.infrastruct use_cache: bool = False, snapshot_strategy: Optional[eventso = None, mutator_func: Optional[Callabl TVersionedEvent], Optional[TVersio = None, **kwargs*)

Bases: `eventsourcing.domain.model.repository.AbstractEntityRepository`

`__init__`(*event_store: eventsourcing.infrastructure.base.AbstractEventStore, use_cache: bool = False, snapshot_strategy: Optional[eventsourcing.infrastructure.snapshotting.AbstractSnapshotStrategy] = None, mutator_func: Optional[Callable[[Optional[TVersionedEntity], TVersionedEvent], Optional[TVersionedEntity]]] = None, **kwargs*)

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

`event_store`

Returns event store object used by this repository.

`__contains__`(*entity_id: uuid.UUID*) → bool

Returns a boolean value according to whether entity with given ID exists.

`__getitem__`(*entity_id: uuid.UUID*) → TVersionedEntity

Returns entity with given ID.

Parameters **`entity_id`** – ID of entity in the repository.

Raises ***`RepositoryKeyError`*** – If the entity is not found.

`get_entity`(*entity_id: uuid.UUID, at: Optional[int] = None*) → Optional[TVersionedEntity]

Returns entity with given ID, optionally at a version.

Returns None if entity not found.

`get_and_project_events`(*entity_id: uuid.UUID, gt: Optional[int] = None, gte: Optional[int] = None, lt: Optional[int] = None, lte: Optional[int] = None, limit: Optional[int] = None, initial_state: Optional[TVersionedEntity] = None, query_descending: bool = False*) → Optional[TVersionedEntity]

Reconstitutes requested domain entity from domain events found in event store.

project_events (*initial_state: Optional[TVersionedEntity], domain_events: Iterable[TVersionedEvent]*) → Optional[TVersionedEntity]

Evolves initial_state using the domain_events and a mutator function.

Applies a mutator function cumulatively to a sequence of domain events, so as to mutate the initial value to a mutated value.

This class's mutate() method is used as the default mutator function, but custom behaviour can be introduced by passing in a 'mutator_func' argument when constructing this class, or by overridding the mutate() method.

static mutate (*initial: Optional[TVersionedEntity], event: TVersionedEvent*) → Optional[TVersionedEntity]

Default mutator function, which uses __mutate__() method on event object to mutate initial state.

Parameters

- **initial** – Initial state to be mutated by this function.
- **event** – Event that causes the initial state to be mutated.

Returns Returns the mutated state.

take_snapshot (*entity_id: uuid.UUID, lt: Optional[int] = None, lte: Optional[int] = None*) → Optional[eventsourcing.domain.model.events.AbstractSnapshot]

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.

iterators

Different ways of getting sequenced items from a datastore.

class eventsourcing.infrastructure.iterators.**AbstractSequencedItemIterator**(*record_manager: eventsourcing.infrastructure.base.*, *sequence_id: uuid.UUID*, *page_size: Optional[int] = None*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *is_ascending: bool = True*)

Bases: collections.abc.Iterable, typing.Generic

__init__(*record_manager: eventsourcing.infrastructure.base.AbstractRecordManager*, *sequence_id: uuid.UUID*, *page_size: Optional[int] = None*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *is_ascending: bool = True*)

Initialises sequenced item iterator.

Parameters

- **record_manager** – The record manager used to get sequenced items.
- **sequence_id** – The id of the sequence being iterated over.
- **page_size** – The number of items requested from the record manager.
- **gt** – Exclusive lower bound on position of items returned.

- **gte** – Inclusive lower bound on position of items returned.
- **lt** – Exclusive upper bound on position of items returned.
- **lte** – Inclusive upper bound on position of items returned.
- **limit** – Limit to the number of items returned.
- **is_ascending** – Whether or not to iterate in ascending order.

_inc_page_counter() → None
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() → None
Increments the query counter.

__iter__() → Iterator[NamedTuple]
Yields a continuous sequence of items.

class eventsourcing.infrastructure.iterators.**SequencedItemIterator**(*record_manager: eventsourcing.infrastructure.base.AbstractRec*, *sequence_id: uuid.UUID*, *page_size: Optional[int] = None*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *is_ascending: bool = True*)

Bases: *eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator*

__iter__() → Iterator[NamedTuple]
Yields a continuous sequence of items from "pages" of sequenced items retrieved using the record manager.

class eventsourcing.infrastructure.iterators.**ThreadedSequencedItemIterator**(*record_manager: eventsourcing.infrastructure.base. sequence_id: uuid.UUID*, *page_size: Optional[int] = None*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *limit: Optional[int] = None*, *is_ascending: bool = True*)

Bases: *eventsourcing.infrastructure.iterators.AbstractSequencedItemIterator*

__iter__() → Iterator[NamedTuple]

Yields a continuous sequence of items.

class eventsourcing.infrastructure.iterators.**GetEntityEventsThread**(*record_manager: eventsourcing.infrastructure.base.AbstractRec*
sequence_id: uuid.UUID, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *page_size: Optional[int] = None*, *is_ascending: bool = True*, **args*, ***kwargs*)

Bases: threading.Thread

__init__(*record_manager: eventsourcing.infrastructure.base.AbstractRecordManager*, *sequence_id: uuid.UUID*, *gt: Optional[int] = None*, *gte: Optional[int] = None*, *lt: Optional[int] = None*, *lte: Optional[int] = None*, *page_size: Optional[int] = None*, *is_ascending: bool = True*, **args*, ***kwargs*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run() → None

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.

factory

Infrastructure factory.

class eventsourcing.infrastructure.factory.**InfrastructureFactory**(*record_manager_class: Optional[Type[eventsourcing.infrastructu = None, sequenced_item_class: Optional[Type[NamedTuple]] = None, event_store_class: Optional[Type[eventsourcing.infrastructu = None, sequenced_item_mapper_class: Optional[Type[eventsourcing.infrastructu = None, json_encoder_class: Optional[Type[json.encoder.JSONEncod = None, sort_keys: bool = False, json_decoder_class: Optional[Type[json.decoder.JSONDecod = None, integer_sequenced_record_class: Optional[type] = None, timestamp_sequenced_record_class: Optional[type] = None, snapshot_record_class: Optional[type] = None, contiguous_record_ids: bool = False, application_name: Optional[str] = None, pipeline_id: int = 0*)

Bases: `typing.Generic`

Base class for infrastructure factories.

`__init__`(*record_manager_class: Optional[Type[eventsourcing.infrastructure.base.AbstractRecordManager]] = None*, *sequenced_item_class: Optional[Type[NamedTuple]] = None*, *event_store_class: Optional[Type[eventsourcing.infrastructure.base.AbstractEventStore]] = None*, *sequenced_item_mapper_class: Optional[Type[eventsourcing.infrastructure.sequenceditemmapper.AbstractSequence*
= None, *json_encoder_class: Optional[Type[json.encoder.JSONEncoder]] = None*, *sort_keys: bool = False*, *json_decoder_class: Optional[Type[json.decoder.JSONDecoder]] = None*, *integer_sequenced_record_class: Optional[type] = None*, *timestamp_sequenced_record_class: Optional[type] = None*, *snapshot_record_class: Optional[type] = None*, *contiguous_record_ids: bool = False*, *application_name: Optional[str] = None*, *pipeline_id: int = 0*)

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

`sequenced_item_class`

alias of `eventsourcing.infrastructure.sequenceditem.SequencedItem`

`sequenced_item_mapper_class`

alias of `eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper`

`construct_integer_sequenced_record_manager`(*integer_sequenced_record_class=None*, ***kwargs*) → eventsourcing.infrastructure.base.AbstractRecordManager

Constructs an integer sequenced record manager.

`construct_timestamp_sequenced_record_manager`(***kwargs*) → eventsourcing.infrastructure.base.AbstractRecordManager

Constructs a timestamp sequenced record manager.

`construct_snapshot_record_manager`(***kwargs*) → eventsourcing.infrastructure.base.AbstractRecordManager

Constructs a snapshot record manager.

`construct_record_manager`(*record_class: Optional[type]*, *sequenced_item_class: Optional[Type[NamedTuple]] = None*, ***kwargs*) → eventsourcing.infrastructure.base.AbstractRecordManager

Constructs an record manager.

`construct_sequenced_item_mapper`(*cipher: Optional[eventsourcing.utils.cipher.aes.AESCipher]*, *compressor: Any*) → eventsourcing.infrastructure.sequenceditemmapper.AbstractSequencedItemMapper

Constructs sequenced item mapper object.

Returns Sequenced item mapper object.

Return type eventsourcing.infrastructure.sequenceditemmapper

.AbstractSequencedItemMapper

`construct_integer_sequenced_event_store`(*cipher: Optional[eventsourcing.utils.cipher.aes.AESCipher]*, *compressor: Any*) → eventsourcing.infrastructure.base.AbstractEventStore

Constructs an integer sequenced event store.

`construct_datastore`() → Optional[eventsourcing.infrastructure.datastore.AbstractDatastore]

Constructs datastore object.

Returns Concrete datastore object object.

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: uuid.UUID*, *lt: Optional[int] = None*, *lte: Optional[int] = None*) → Optional[eventsourcing.domain.model.events.AbstractSnapshot]

Gets the last snapshot for entity, optionally until a particular version number.

Return type *Snapshot*

take_snapshot(*entity_id: uuid.UUID*, *entity: object*, *last_event_version: int*) → eventsourcing.domain.model.snapshot.Snapshot

Takes a snapshot of entity, using given ID, state and version number.

class eventsourcing.infrastructure.snapshotting.**EventSourcedSnapshotStrategy**(*snapshot_store: eventsourcing.infrastructure.ba eventsourcing.infrastructure.ba eventsourcing.infrastructure.ba*

Bases: *eventsourcing.infrastructure.snapshotting.AbstractSnapshotStrategy*

Snapshot strategy that uses an event sourced snapshot.

__init__(*snapshot_store: eventsourcing.infrastructure.base.AbstractEventStore[eventsourcing.domain.model.events.Abstract eventsourcing.infrastructure.base.AbstractRecordManager][eventsourcing.domain.model.events.AbstractSnapshot, eventsourcing.infrastructure.base.AbstractRecordManager]*)

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

get_snapshot(*entity_id: uuid.UUID*, *lt: Optional[int] = None*, *lte: Optional[int] = None*) → Optional[eventsourcing.domain.model.events.AbstractSnapshot]

Gets the last snapshot for entity, optionally until a particular version number.

Return type *Snapshot*

take_snapshot(*entity_id: uuid.UUID*, *entity: object*, *last_event_version: int*) → eventsourcing.domain.model.snapshot.Snapshot

Creates a Snapshot from the given state, and appends it to the snapshot store.

Return type *Snapshot*

timebucketedlog_reader

Reader for timebucketed logs.

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.eventsourcedrepository.EventSourcedRepository*

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

Bases: *eventsourcing.domain.model.array.AbstractBigArrayRepository*, *eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository*

subrepo_class
alias of *ArrayRepository*

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

subrepo
Sub-sequence repository.

class eventsourcing.infrastructure.repositories.collection_repo.**CollectionRepository**(*event_st eventsou ing.infra use_cac bool = False, snap- shot_str Op- tional[e = None, mu- ta- tor_func Op- tional[C TVer- sionedE vent], Op- tional[T = None, **kwarg*

Bases: *eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository*, *eventsourcing.domain.model.collection.AbstractCollectionRepository*

Event sourced repository for the Collection domain model entity.

class eventsourcing.infrastructure.repositories.timebucketedlog_repo.**TimebucketedlogRepo**(*ev*
ev
in
us
bo
=
Fa
sn
sh
O
tic
=
No
m
ta
to
O
tic
TV
sic
ve
O
tic
=
No
****

Bases: *eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository*, *eventsourcing.domain.model.timebucketedlog.TimebucketedlogRepository*

Event sourced repository for the Example domain model entity.

integersequencegenerators

Different ways of generating sequences of integers.

class eventsourcing.infrastructure.integersequencegenerators.base.**AbstractIntegerSequenceGe**

Bases: object

Abstract base class for generating a sequence of integers.

__next__() → int

Returns the next item in the container.

class eventsourcing.infrastructure.integersequencegenerators.base.**SimpleIntegerSequenceGene**

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

Generates a sequence of integers, by simply incrementing a Python int.

__init__(*i: int = 0*)

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

__next__() → int

Returns the next item in the container.

class eventsourcing.infrastructure.integersequencegenerators.redisincr.**RedisIncr**(*redis: Optional[redis.cl = None*, *key: Optional[str] = None*)

Bases: *eventsourcing.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.

__init__(*redis: Optional[redis.client.Redis] = None*, *key: Optional[str] = None*)
Initialize self. See help(type(self)) for accurate signature.

__next__() → int
Returns the next item in the container.

1.19.3 application

The application layer brings together the domain and infrastructure layers.

simple

class `eventsourcing.application.simple.`**`ProcessEvent`**(*domain_events: Iterable[TDomainEvent], tracking_kwargs: Optional[Dict[str, Union[str, int]]] = None, causal_dependencies: Optional[List[Dict[str, int]]] = None, orm_objs_pending_save: Sequence[Any] = (), orm_objs_pending_delete: Sequence[Any] = ()*)

Bases: *`eventsourcing.whitehead.ActualOccasion`*, `typing.Generic`

`__init__`(*domain_events: Iterable[TDomainEvent], tracking_kwargs: Optional[Dict[str, Union[str, int]]] = None, causal_dependencies: Optional[List[Dict[str, int]]] = None, orm_objs_pending_save: Sequence[Any] = (), orm_objs_pending_delete: Sequence[Any] = ()*)

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

class eventsourcing.application.simple.**SimpleApplication**(*name: str = '', persistence_policy: Optional[eventsourcing.application.policies.Persister = None, persist_event_type: Union[Type[eventsourcing.domain.model.events.L Tuple[Type[eventsourcing.domain.model.events.Dom None] = None, cipher_key: Optional[str] = None, compressor: Any = None, sequenced_item_class: Optional[Type[NamedTuple]] = None, sequenced_item_mapper_class: Optional[Type[eventsourcing.infrastructure.sequence = None, record_manager_class: Optional[Type[eventsourcing.infrastructure.base.Abs = None, stored_event_record_class: Optional[type] = None, event_store_class: Optional[Type[eventsourcing.infrastructure.eventstor = None, snapshot_record_class: Optional[type] = None, setup_table: bool = True, contiguous_record_ids: bool = True, pipeline_id: int = 0, json_encoder_class: Optional[Type[json.encoder.JSONEncoder]] = None, sort_keys: bool = False, json_decoder_class: Optional[Type[json.decoder.JSONDecoder]] = None, notification_log_section_size: Optional[int] = None, use_cache: bool = False*)

Bases: *eventsourcing.application.pipeline.Pipeable*, typing.Generic

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.

infrastructure_factory_class

alias of *eventsourcing.infrastructure.factory.InfrastructureFactory*

repository_class
alias of *eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository*

__init__(*name: str = '', persistence_policy: Optional[eventsourcing.application.policies.PersistencePolicy] = None, persist_event_type: Union[Type[eventsourcing.domain.model.events.DomainEvent], Tuple[Type[eventsourcing.domain.model.events.DomainEvent]], None] = None, cipher_key: Optional[str] = None, compressor: Any = None, sequenced_item_class: Optional[Type[NamedTuple]] = None, sequenced_item_mapper_class: Optional[Type[eventsourcing.infrastructure.sequenceditemmapper.SequencedItemMapper]] = None, record_manager_class: Optional[Type[eventsourcing.infrastructure.base.AbstractRecordManager]] = None, stored_event_record_class: Optional[type] = None, event_store_class: Optional[Type[eventsourcing.infrastructure.eventstore.EventStore]] = None, snapshot_record_class: Optional[type] = None, setup_table: bool = True, contiguous_record_ids: bool = True, pipeline_id: int = 0, json_encoder_class: Optional[Type[json.encoder.JSONEncoder]] = None, sort_keys: bool = False, json_decoder_class: Optional[Type[json.decoder.JSONDecoder]] = None, notification_log_section_size: Optional[int] = None, use_cache: bool = False*)
Initialises application object.

Parameters

- **name** – Name of application.
- **persistence_policy** – Persistence policy object.
- **persist_event_type** – Tuple of domain event classes to be persisted.
- **cipher_key** – Base64 unicode string cipher key.
- **compressor** – Compressor used to compress serialized event state.
- **sequenced_item_class** – Named tuple for mapping and recording events.
- **sequenced_item_mapper_class** – Object class for mapping stored events.
- **record_manager_class** – Object class for recording stored events.
- **stored_event_record_class** – Object class for event records.
- **event_store_class** – Object class uses to store and retrieve domain events.
- **snapshot_record_class** – Object class used to represent snapshots.
- **setup_table** – Option to create database tables when application starts.
- **contiguous_record_ids** – Whether or not to delegate notification ID generation to the record manager (to guarantee there will be no gaps).
- **pipeline_id** – ID of instance of system pipeline expressions.
- **json_encoder_class** – Object class used to encode object as JSON strings.
- **json_decoder_class** – Object class used to decode JSON strings as objects.
- **notification_log_section_size** – Number of notification items in a section.
- **use_cache** – Whether or not to keep aggregates in memory (saves replaying when accessing again, but uses memory).

event_store_class
alias of *eventsourcing.infrastructure.eventstore.EventStore*

construct_infrastructure(**args, **kwargs*) → None
Constructs infrastructure for application.

construct_infrastructure_factory(**args*, ***kwargs*) → eventsourcing.infrastructure.factory.InfrastructureFactory
Constructs infrastructure factory object.

construct_datastore() → None
Constructs datastore object (used to create and drop database tables).

construct_event_store() → None
Constructs event store object.

construct_repository(***kwargs*) → None
Constructs repository object.

setup_table() → None
Sets up the database table using event store's record class.

drop_table() → None
Drops the database table using event store's record class.

construct_notification_log() → None
Constructs notification log object.

construct_persistence_policy() → None
Constructs persistence policy object.

change_pipeline(*pipeline_id: int*) → None
Switches pipeline being used by this application object.

close() → None
Closes the application for further use.

The persistence policy is closed, and the application's connection to the database is closed.

__enter__() → T
Supports use of application as context manager.

__exit__(*exc_type: Any*, *exc_val: Any*, *exc_tb: Any*) → None
Closes application when exiting context manager.

classmethod mixin(*infrastructure_class: type*) → T
Returns subclass that inherits also from given infrastructure class.

save(*aggregates=()*, *orm_objects_pending_save=()*, *orm_objects_pending_delete=()*) → None
Saves state of aggregates, and ORM objects.

All of the pending events of the aggregates, along with the ORM objects, are recorded atomically as a process event.

Then a "prompt to pull" is published, and, if the repository cache is in use, then puts the aggregates in the cache.

Parameters

- **aggregates** – One or many aggregates.
- **orm_objects_pending_save** – Sequence of ORM objects to be saved.
- **orm_objects_pending_delete** – Sequance of ORM objects to be deleted.

record_process_event(*process_event: eventsourcing.application.simple.ProcessEvent*) → List[T]
Records a process event.

Converts the domain events of the process event to event record objects, and writes the event records and the ORM objects to the database using the application's event store's record manager.

Parameters **`process_event`** – An instance of *`ProcessEvent`*

Returns A list of event records.

`construct_event_records`(*pending_events: Iterable[TAggregateEvent], causal_dependencies: Optional[List[Dict[str, int]]]*) → List[T]
Constructs event records from domain events.

Parameters

- **`pending_events`** – An iterable of domain events.
- **`causal_dependencies`** – A list of causal dependencies.

Returns A list of event records.

`publish_prompt`(*head_notification_id=None*)
Publishes a "prompt to pull" (instance of *`PromptToPull`*).

Parameters **`head_notification_id`** – Maximum notification ID of event records to be pulled.

class eventsourcing.application.simple.**ApplicationWithConcreteInfrastructure**(*name: str = ”, persistence_policy: Optional[eventsourcing = None, persist_event_type: Union[Type[eventso Tuple[Type[eventsourc None] = None, cipher_key: Optional[str] = None, compressor: Any = None, sequenced_item_class Optional[Type[Named = None, sequenced_item_mapp Optional[Type[eventso = None, record_manager_cl Optional[Type[eventso = None, stored_event_record Optional[type] = None, event_store_class: Optional[Type[eventso = None, snapshot_record_class:*

Bases: `eventsourcing.application.simple.SimpleApplication`

Base class for application classes that have actual infrastructure.

class `eventsourcing.application.simple.`**`Prompt`**

Bases: `eventsourcing.whitehead.ActualOccasion`

class `eventsourcing.application.simple.`**`PromptToPull`**(*process_name: str*, *pipeline_id: int*, *head_notification_id=None*)

Bases: `eventsourcing.application.simple.Prompt`

`__init__`(*process_name: str*, *pipeline_id: int*, *head_notification_id=None*)

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

`__eq__`(*other: object*) → bool

Return self==value.

`__repr__`() → str

Return repr(self).

policies

class `eventsourcing.application.policies.`**`PersistencePolicy`**(*event_store: eventsourcing.infrastructure.base.AbstractEventStore*, *persist_event_type: Union[type, Tuple, None] = None*)

Bases: `object`

Stores events of given type to given event store, whenever they are published.

`__init__`(*event_store: eventsourcing.infrastructure.base.AbstractEventStore*, *persist_event_type: Union[type*, *Tuple*, *None] = None*)

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

class `eventsourcing.application.policies.`**`SnapshottingPolicy`**(*repository: eventsourcing.domain.model.repository.AbstractEntityR*
snapshot_store: eventsourcing.infrastructure.base.AbstractEventStore[ev
eventsourcing.infrastructure.base.AbstractRecordManag
eventsourcing.infrastructure.base.AbstractRecordManag
persist_event_type: Union[type, Tuple, None] = (<class 'eventsourcing.domain.model.events.EventWithOriginato
period: int = 0)

Bases: `typing.Generic`

`__init__`(*repository: eventsourcing.domain.model.repository.AbstractEntityRepository, snapshot_store: eventsourcing.infrastructure.base.AbstractEventStore[eventsourcing.domain.model.events.AbstractSnap eventsourcing.infrastructure.base.AbstractRecordManager][eventsourcing.domain.model.events.AbstractSnapshot, eventsourcing.infrastructure.base.AbstractRecordManager], persist_event_type: Union[type, Tuple, None] = (<class 'eventsourcing.domain.model.events.EventWithOriginatorVersion'>,), period: int = 0*)

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

notificationlog

class `eventsourcing.application.notificationlog.`**`Section`**(*section_id: str, items: List[T], previous_id: Optional[str] = None, next_id: Optional[str] = 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.

`__init__`(*section_id: str, items: List[T], previous_id: Optional[str] = None, next_id: Optional[str] = None*)

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

class `eventsourcing.application.notificationlog.`**`AbstractNotificationLog`**

Bases: `abc.ABC`

Presents a sequence of sections from a sequence of notifications.

`__getitem__`(*section_id: str*) → eventsourcing.application.notificationlog.Section

Get section of notification log.

Parameters **`section_id`** – ID of a section of the notification log.

`section_size`

Size of section of notification log.

class `eventsourcing.application.notificationlog.`**`LocalNotificationLog`**(*section_size: Optional[int] = None*)

Bases: *`eventsourcing.application.notificationlog.AbstractNotificationLog`*

Presents a sequence of sections from a sequence of notifications.

`__init__`(*section_size: Optional[int] = None*)

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

`section_size`

Size of section of notification log.

`__getitem__`(*section_id: str*) → eventsourcing.application.notificationlog.Section

Get section of notification log.

Parameters **`section_id`** – ID of a section of the notification log.

`get_next_position`() → int

Returns next unoccupied position in zero-based sequence.

Since the notification IDs are one-based, the next position in the zero-based sequence equals the current max notification ID which is 1-based. If there are no records, the max notification ID will be None, and the next position is zero.

Returns Non-negative integer.

Return type int

get_items(*start: int*, *stop: int*) → Sequence[Any]
Returns items for section.

class eventsourcing.application.notificationlog.**RecordManagerNotificationLog**(*record_manager: eventsourcing.infrastructure.ba sec- tion_size: Op- tional[int] = None*)

Bases: *eventsourcing.application.notificationlog.LocalNotificationLog*

Local notification log that gets notifications from a record manager.

__init__(*record_manager: eventsourcing.infrastructure.base.AbstractRecordManager*, *section_size: Optional[int] = None*)
Initialize self. See help(type(self)) for accurate signature.

get_items(*start: int, stop: Optional[int]*) → Sequence[Any]
Returns notification in log.

Parameters

- **start** – Inclusive start position in log.
- **stop** – Inclusive stop position in log.

Returns

get_next_position() → int
Returns next unoccupied position in zero-based sequence.

Since the notification IDs are one-based, the next position in the zero-based sequence equals the current max notification ID which is 1-based. If there are no records, the max notification ID will be None, and the next position is zero.

Returns Non-negative integer.

Return type int

class eventsourcing.application.notificationlog.**BigArrayNotificationLog**(*big_array: eventsourcing.domain.model.array.Big sec- tion_size: int*)

Bases: *eventsourcing.application.notificationlog.LocalNotificationLog*

Notification log that uses the BigArray class.

__init__(*big_array: eventsourcing.domain.model.array.BigArray*, *section_size: int*)
Initialize self. See help(type(self)) for accurate signature.

get_items(*start: int*, *stop: int*) → Sequence[Any]
Returns items for section.

get_next_position() → int
Returns next unoccupied position in zero-based sequence.

Returns Non-negative integer.

Return type int

class eventsourcing.application.notificationlog.**NotificationLogReader**(*notification_log: eventsourcing.application.notificationlog use_direct_query_if_available bool = False*)

Bases: abc.ABC

__init__(*notification_log: eventsourcing.application.notificationlog.AbstractNotificationLog*, *use_direct_query_if_available: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

seek(*position: int*) → None
Sets position of reader in notification log sequence.

This represents the position of the last notification read by the reader. The next notification returned by the reader will be the next position.

Parameters **position** (*int*) – Position is notification log sequence.

Raises **ValueError** – if the position is less than zero

initial_section_id
Returns initial section ID used to start getting linked sections from the notification log.

Slight departure from Vaughn Vernon's design by not using 'current' as initial section ID, but a section ID that just includes the "next" position, which the notification log can use to return the section containing this position. This avoids lengthy back- tracking when reader has a lot of notifications to catch-up on.

This property has been extracted in order to allow a subclass to adjust this default behaviour.

It would be possible to calculate the actual section ID from the current reader position. Using section ID of an actual section may hit a cache and avoid troubling the server, but the reader would need to know the section size of the notification log it is reading. If we don't know section size, perhaps it is a remote notification log, we can use 'current' to hit a cache. In future, it might be possible to ask the notification log to disclose it's section size, or compute an actual section ID for a given position.

Returns A notification log section ID.

Return type str

read_list(*advance_by: Optional[int] = None*) → List[Dict[str, Any]]
Deprecated in 8.0.0.

Please use list_notifications() instead.

read_items(*stop_index: Optional[int] = None*, *advance_by: Optional[int] = None*) → Iterator[Dict[str, Any]]
Deprecated in 8.0.0.

Please use iter_notifications() instead.

django

class eventsourcing.application.django.**DjangoApplication**(*tracking_record_class: Any = None*, **args*, ***kwargs*)

Bases: *eventsourcing.application.simple.ApplicationWithConcreteInfrastructure*

infrastructure_factory_class
alias of *eventsourcing.infrastructure.django.factory.DjangoInfrastructureFactory*

__init__(*tracking_record_class: Any = None*, **args*, ***kwargs*)
Initialises application object.

Parameters

- **name** – Name of application.
- **persistence_policy** – Persistence policy object.
- **persist_event_type** – Tuple of domain event classes to be persisted.
- **cipher_key** – Base64 unicode string cipher key.
- **compressor** – Compressor used to compress serialized event state.
- **sequenced_item_class** – Named tuple for mapping and recording events.
- **sequenced_item_mapper_class** – Object class for mapping stored events.
- **record_manager_class** – Object class for recording stored events.
- **stored_event_record_class** – Object class for event records.
- **event_store_class** – Object class uses to store and retrieve domain events.
- **snapshot_record_class** – Object class used to represent snapshots.
- **setup_table** – Option to create database tables when application starts.
- **contiguous_record_ids** – Whether or not to delegate notification ID generation to the record manager (to guarantee there will be no gaps).
- **pipeline_id** – ID of instance of system pipeline expressions.
- **json_encoder_class** – Object class used to encode object as JSON strings.
- **json_decoder_class** – Object class used to decode JSON strings as objects.
- **notification_log_section_size** – Number of notification items in a section.
- **use_cache** – Whether or not to keep aggregates in memory (saves replaying when accessing again, but uses memory).

construct_infrastructure(**args*, ***kwargs*) → None
Constructs infrastructure for application.

classmethod reset_connection_after_forking() → None
Resets database connection after forking.

popo

class eventsourcing.application.popo.**PopoApplication**(*name: str = ”, persistence_policy: Optional[eventsourcing.application.policies.PersistencePol = None, persist_event_type: Union[Type[eventsourcing.domain.model.events.Domai Tuple[Type[eventsourcing.domain.model.events.DomainEv None] = None, cipher_key: Optional[str] = None, compressor: Any = None, sequenced_item_class: Optional[Type[NamedTuple]] = None, sequenced_item_mapper_class: Optional[Type[eventsourcing.infrastructure.sequenceditem = None, record_manager_class: Optional[Type[eventsourcing.infrastructure.base.AbstractR = None, stored_event_record_class: Optional[type] = None, event_store_class: Optional[Type[eventsourcing.infrastructure.eventstore.Even = None, snapshot_record_class: Optional[type] = None, setup_table: bool = True, contiguous_record_ids: bool = True, pipeline_id: int = 0, json_encoder_class: Optional[Type[json.encoder.JSONEncoder]] = None, sort_keys: bool = False, json_decoder_class: Optional[Type[json.decoder.JSONDecoder]] = None, notification_log_section_size: Optional[int] = None, use_cache: bool = False*)

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`

sqlalchemy

class eventsourcing.application.sqlalchemy.**SQLAlchemyApplication**(*uri: Optional[str] = None*, *session: Optional[Any] = None*, *tracking_record_class: Any = None*, ***kwargs*)

Bases: *eventsourcing.application.simple.ApplicationWithConcreteInfrastructure*

infrastructure_factory_class
alias of *eventsourcing.infrastructure.sqlalchemy.factory.SQLAlchemyInfrastructureFactory*

stored_event_record_class
alias of *eventsourcing.infrastructure.sqlalchemy.records.StoredEventRecord*

snapshot_record_class
alias of *eventsourcing.infrastructure.sqlalchemy.records.EntitySnapshotRecord*

__init__(*uri: Optional[str] = None*, *session: Optional[Any] = None*, *tracking_record_class: Any = None*, ***kwargs*)
Initialises application object.

Parameters

- **name** – Name of application.
- **persistence_policy** – Persistence policy object.
- **persist_event_type** – Tuple of domain event classes to be persisted.
- **cipher_key** – Base64 unicode string cipher key.
- **compressor** – Compressor used to compress serialized event state.
- **sequenced_item_class** – Named tuple for mapping and recording events.
- **sequenced_item_mapper_class** – Object class for mapping stored events.
- **record_manager_class** – Object class for recording stored events.
- **stored_event_record_class** – Object class for event records.
- **event_store_class** – Object class uses to store and retrieve domain events.
- **snapshot_record_class** – Object class used to represent snapshots.
- **setup_table** – Option to create database tables when application starts.
- **contiguous_record_ids** – Whether or not to delegate notification ID generation to the record manager (to guarantee there will be no gaps).
- **pipeline_id** – ID of instance of system pipeline expressions.
- **json_encoder_class** – Object class used to encode object as JSON strings.
- **json_decoder_class** – Object class used to decode JSON strings as objects.

- **notification_log_section_size** – Number of notification items in a section.
- **use_cache** – Whether or not to keep aggregates in memory (saves replaying when accessing again, but uses memory).

construct_infrastructure(**args*, ***kwargs*) → None
Constructs infrastructure for application.

construct_infrastructure_factory(**args*, ***kwargs*) → eventsourcing.infrastructure.factory.InfrastructureFactory
Constructs infrastructure factory object.

construct_datastore() → None
Constructs datastore object (used to create and drop database tables).

process

class eventsourcing.application.process.**PromptToQuit**
Bases: *eventsourcing.application.simple.Prompt*

class eventsourcing.application.process.**WrappedRepository**(*repository: eventsourcing.infrastructure.eventsourcedrepository.EventS ~TAggregateEvent][TAggregate, TAggregateEvent]*)
Bases: typing.Generic

Used to wrap an event sourced repository for use in process application policy so that use of, and changes to, domain model aggregates can be automatically detected and recorded.

Implements a "dictionary like" interface, so that aggregates can be accessed by ID.

__init__(*repository: eventsourcing.infrastructure.eventsourcedrepository.EventSourcedRepository[~TAggregate, ~TAggregateEvent][TAggregate, TAggregateEvent]*) → None
Initialize self. See help(type(self)) for accurate signature.

save_orm_obj(*orm_obj: Any*) → None
Includes orm_obj in "process event", so that projections into custom ORM objects is as reliable with respect to sudden restarts as "normal" domain event processing in a process application.

delete_orm_obj(*orm_obj: Any*) → None
Includes orm_obj in "process event", so that projections into custom ORM objects is as reliable with respect to sudden restarts as "normal" domain event processing in a process application.

class eventsourcing.application.process.**ProcessApplication**(*name: str = ''*, *policy: Optional[function] = None*, *setup_table: bool = False*, *use_direct_query_if_available: bool = False*, *notification_log_reader_class: Optional[Type[eventsourcing.application.notificat = None*, *apply_policy_to_generated_events: bool = False*, ***kwargs*)
Bases: *eventsourcing.application.simple.SimpleApplication*

`__init__`(*name: str = ”*, *policy: Optional[function] = None*, *setup_table: bool = False*, *use_direct_query_if_available: bool = False*, *notification_log_reader_class: Optional[Type[eventsourcing.application.notificationlog.NotificationLogReader]] = None*, *apply_policy_to_generated_events: bool = False*, ***kwargs*)

Initialises application object.

Parameters

- **`name`** – Name of application.
- **`persistence_policy`** – Persistence policy object.
- **`persist_event_type`** – Tuple of domain event classes to be persisted.
- **`cipher_key`** – Base64 unicode string cipher key.
- **`compressor`** – Compressor used to compress serialized event state.
- **`sequenced_item_class`** – Named tuple for mapping and recording events.
- **`sequenced_item_mapper_class`** – Object class for mapping stored events.
- **`record_manager_class`** – Object class for recording stored events.
- **`stored_event_record_class`** – Object class for event records.
- **`event_store_class`** – Object class uses to store and retrieve domain events.
- **`snapshot_record_class`** – Object class used to represent snapshots.
- **`setup_table`** – Option to create database tables when application starts.
- **`contiguous_record_ids`** – Whether or not to delegate notification ID generation to the record manager (to guarantee there will be no gaps).
- **`pipeline_id`** – ID of instance of system pipeline expressions.
- **`json_encoder_class`** – Object class used to encode object as JSON strings.
- **`json_decoder_class`** – Object class used to decode JSON strings as objects.
- **`notification_log_section_size`** – Number of notification items in a section.
- **`use_cache`** – Whether or not to keep aggregates in memory (saves replaying when accessing again, but uses memory).

`notification_log_reader_class`

alias of `eventsourcing.application.notificationlog.NotificationLogReader`

`close`() → None

Closes the application for further use.

The persistence policy is closed, and the application’s connection to the database is closed.

`publish_prompt_for_events`(*_: Optional[Iterable[eventsourcing.whitehead.ActualOccasion]] = None*) → 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.

`follow`(*upstream_application_name: str*, *notification_log: eventsourcing.application.notificationlog.AbstractNotificationLog*) → None

Sets up process application to follow the given notification log of an upstream application.

Parameters

- **`upstream_application_name`** – Name of the upstream application.
- **`notification_log`** – Notification log that will be processed.

`run`(*prompt: Optional[eventsourcing.application.simple.Prompt] = None*, *advance_by: Optional[int] = None*) → int

Pulls event notifications from notification logs being followed by this process application, and processes the contained domain events.

Parameters

- **`prompt`** – Optional prompt, specifying a particular notification log.
- **`advance_by`** – Maximum event notifications to process.

Returns Returns number of events that have been processed.

`check_causal_dependencies`(*upstream_name*, *causal_dependencies_json*)

Checks the causal dependencies are satisfied (have already been processed).

Parameters

- **`upstream_name`** – Name of the upstream application being processed.
- **`causal_dependencies_json`** – Pipelines and positions in notification logs.

Raises ***`CausalDependencyFailed`*** – If causal dependencies are not satisfied.

`process_upstream_event`(*domain_event: TAggregateEvent*, *notification_id: int*, *upstream_name: str*) → Tuple[List[TAggregateEvent], List[T]]

Processes given domain event from an upstream notification log.

Calls the process application policy, and then records a process event, hence recording atomically all new domain events created by the call to the policy along with any ORM objects that may result.

Parameters

- **`domain_event`** – Domain event to be processed.
- **`notification_id`** – Position in notification log.
- **`upstream_name`** – Name of upstream application.

Returns Returns a list of new domain events.

`event_from_notification`(*notification: Dict[str, Any]*) → TAggregateEvent

Reconstructs domain event from an event notification.

Parameters **`notification`** – The event notification.

Returns A domain event.

`call_policy`(*domain_event: TAggregateEvent*) → Tuple[List[TAggregateEvent], List[Dict[str, int]], List[Any], List[Any]]

Calls the process application policy with the given domain event.

Parameters **`domain_event`** – Domain event that will be given to the policy.

Returns Returns a list of domain events, and a list of causal dependencies.

`policy`(*repository: eventsourcing.application.process.WrappedRepository[~TAggregate, ~TAggregateEvent][TAggregate, TAggregateEvent]*, *event: TAggregateEvent*) → Union[TAggregate, Sequence[TAggregate], None]

Empty method, can be overridden in subclasses to implement concrete policy.

collect_pending_events(*aggregates: Sequence[TAggregate]*) → List[TAggregateEvent]
Collects all the pending events from the given sequence of aggregates.

Parameters **aggregates** – Sequence of aggregates.

Returns Returns a list of domain events.

setup_table() → None
Sets up the database table using event store's record class.

drop_table() → None
Drops the database table using event store's record class.

class eventsourcing.application.process.**ProcessApplicationWithSnapshotting**(*snapshot_period: int = 0*, ***kwargs*)
Bases: *eventsourcing.application.snapshotting.SnapshottingApplication*, *eventsourcing.application.process.ProcessApplication*

Supplements process applications that will use snapshotting.

take_snapshots(*new_events: Sequence[TAggregateEvent]*) → None
Takes snapshot of aggregates, according to the policy.

Parameters **new_events** – Domain events used to detect if a snapshot is to be taken.

decorators

eventsourcing.application.decorators.**applicationpolicy**(*arg: Callable*) → Callable
Decorator for application policy method.

Allows policy to be built up from methods registered for different event classes.

eventsourcing.application.decorators.**applicationpolicy2**(*arg: Callable*) → Callable
This one doesn't use weakrefs.

command

class eventsourcing.application.command.**CommandProcess**(*name: str = ”*, *policy: Optional[function] = None*, *setup_table: bool = False*, *use_direct_query_if_available: bool = False*, *notification_log_reader_class: Optional[Type[eventsourcing.application.notificationlog* *= None*, *apply_policy_to_generated_events: bool = False*, ***kwargs*)
Bases: *eventsourcing.application.process.ProcessApplication*

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

pipeline

class eventsourcing.application.pipeline.**PipelineExpression**(*left: Union[PipelineExpression, PipeableMetaclass], right: Union[PipelineExpression, PipeableMetaclass]*)

Bases: `object`

Implements a left-to-right association between two objects.

__init__(*left: Union[PipelineExpression, PipeableMetaclass], right: Union[PipelineExpression, PipeableMetaclass]*)
Initialize self. See help(type(self)) for accurate signature.

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

Meta class for pipeable classes.

__or__(*other: Union[PipelineExpression, PipeableMetaclass]*) → eventsourcing.application.pipeline.PipelineExpression
Implements bitwise or operator '|' as a pipe between pipeable classes.

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

Base class for pipeable classes.

snapshotting

class eventsourcing.application.snapshotting.**SnapshottingApplication**(*snapshot_period: int = 0, **kwargs*)

Bases: *eventsourcing.application.simple.SimpleApplication*

__init__(*snapshot_period: int = 0, **kwargs*)
Initialises application object.

Parameters

- **name** – Name of application.
- **persistence_policy** – Persistence policy object.
- **persist_event_type** – Tuple of domain event classes to be persisted.
- **cipher_key** – Base64 unicode string cipher key.
- **compressor** – Compressor used to compress serialized event state.
- **sequenced_item_class** – Named tuple for mapping and recording events.
- **sequenced_item_mapper_class** – Object class for mapping stored events.
- **record_manager_class** – Object class for recording stored events.
- **stored_event_record_class** – Object class for event records.
- **event_store_class** – Object class uses to store and retrieve domain events.
- **snapshot_record_class** – Object class used to represent snapshots.

- **setup_table** – Option to create database tables when application starts.
- **contiguous_record_ids** – Whether or not to delegate notification ID generation to the record manager (to guarantee there will be no gaps).
- **pipeline_id** – ID of instance of system pipeline expressions.
- **json_encoder_class** – Object class used to encode object as JSON strings.
- **json_decoder_class** – Object class used to decode JSON strings as objects.
- **notification_log_section_size** – Number of notification items in a section.
- **use_cache** – Whether or not to keep aggregates in memory (saves replaying when accessing again, but uses memory).

construct_event_store() → None
Constructs event store object.

construct_repository(***kwargs*) → None
Constructs repository object.

construct_persistence_policy() → None
Constructs persistence policy object.

setup_table() → None
Sets up the database table using event store's record class.

drop_table() → None
Drops the database table using event store's record class.

close() → None
Closes the application for further use.

The persistence policy is closed, and the application's connection to the database is closed.

1.19.4 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.**RemoteNotificationLog**(*base_url: str, json_decoder_class: Optional[Type[json.decoder.JSOND = None*)

Bases: *eventsourcing.application.notificationlog.AbstractNotificationLog*

Presents notification log sections retrieved an HTTP API that presents notification log sections in JSON format, for example by using a NotificationLogView.

__init__(*base_url: str, json_decoder_class: Optional[Type[json.decoder.JSONDecoder]] = None*)
Initialises remote notification log object.

Parameters

- **base_url** (*str*) – A URL for the HTTP API.

- **json_decoder_class** (*JSONDecoder*) – used to deserialize remote sections.

section_size
Size of section of notification log.

__getitem__(*section_id: str*) → eventsourcing.application.notificationlog.Section
Returns a section of notification log.

Parameters **section_id** (*str*) – ID of the section of the notification log.

Returns Identified section of notification log.

Return type *Section*

class eventsourcing.interface.notificationlog.**NotificationLogView**(*notification_log: eventsourcing.application.notificationlog.Local json_encoder: eventsourcing.utils.transcoding.ObjectJSONEn*

Bases: object

Presents sections of a notification log in JSON format.

Can be used to make an HTTP API that can be used remotely, for example by a RemoteNotificationLog.

__init__(*notification_log: eventsourcing.application.notificationlog.LocalNotificationLog, json_encoder: eventsourcing.utils.transcoding.ObjectJSONEncoder*)
Initialises notification log view object.

Parameters

- **notification_log** – A notification log object
- **json_encoder_class** – JSON encoder class

present_resource(*name: str*) → bytes
Returns a resource of the notification log in JSON format.

Supports returning section of the notification log.

Also supports returning section_size of notification log, if section_id has special value 'section_size'.

Parameters **name** – Name of the resource, e.g. a section ID.

Returns Identified resource of notification log view in JSON format.

1.19.5 system

The system layer brings together different process applications within a system definition, and provides various system runners for running a system.

definition

class eventsourcing.system.definition.**System**(**pipeline_exprs*, ***kwargs*)

Bases: object

A system object has a set of pipeline expressions, which involve process application classes. A system object can be run using a system runner.

__init__(**pipeline_exprs*, ***kwargs*)

Initialises a "process network" system object.

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

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

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

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

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

Parameters pipeline_exprs – Pipeline expressions involving process application

classes.

construct_app(*process_class: Type[TProcessApplication], infrastructure_class: Optional[Type[eventsourcing.application.simple.ApplicationWithConcreteInfrastructure]] = None, **kwargs*) → TProcessApplication

Constructs process application from given process_class.

__enter__() → eventsourcing.system.definition.AbstractSystemRunner

Supports running a system object directly as a context manager.

The system is run with the SingleThreadedRunner.

__exit__(*exc_type: Any*, *exc_val: Any*, *exc_tb: Any*) → None

Supports usage of a system object as a context manager.

bind(*infrastructure_class: Type[eventsourcing.application.simple.ApplicationWithConcreteInfrastructure]*) → TSystem

Constructs a system object that has an infrastructure class from a system object constructed without infrastructure class.

Raises ProgrammingError if already have an infrastructure class.

Parameters infrastructure_class –

Returns System object that has an infrastructure class.

Return type *System*

class eventsourcing.system.definition.**AbstractSystemRunner**(*system: eventsourcing.system.definition.System, infrastructure_class: Optional[Type[eventsourcing.application.simple.A = None, setup_tables: bool = False, use_direct_query_if_available: bool = False*)

Bases: abc.ABC

__init__(*system: eventsourcing.system.definition.System*, *infrastructure_class: Optional[Type[eventsourcing.application.simple.ApplicationWithConcreteInfrastructure]] = None*, *setup_tables: bool = False*, *use_direct_query_if_available: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

__enter__() → TSystemRunner
Supports usage of a system runner as a context manager.

__exit__(*exc_type: Any*, *exc_val: Any*, *exc_tb: Any*) → None
Supports usage of a system runner as a context manager.

start() → None
Starts running the system.

Abstract method which must be implemented on concrete descendants.

close() → None
Closes a running system.

runner

class eventsourcing.system.runner.**InProcessRunner**(*system: eventsourcing.system.definition.System*, *infrastructure_class: Optional[Type[eventsourcing.application.simple.ApplicationWi*
= None, *setup_tables: bool = False*, *use_direct_query_if_available: bool = False*)
Bases: `eventsourcing.system.definition.AbstractSystemRunner`

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

start() → None
Starts running the system.

Abstract method which must be implemented on concrete descendants.

handle_prompt(*prompt: eventsourcing.application.simple.Prompt*) → None
Handles publication of a prompt.

Abstract method which must be implemented on concrete descendants.

close() → None
Closes a running system.

class eventsourcing.system.runner.**SingleThreadedRunner**(*system: eventsourcing.system.definition.System*, *infrastructure_class: Optional[Type[eventsourcing.application.simple.Applic*
= None, ***kwargs*)
Bases: `eventsourcing.system.runner.InProcessRunner`

Runs a system in the current thread.

__init__(*system: eventsourcing.system.definition.System*, *infrastructure_class: Optional[Type[eventsourcing.application.simple.ApplicationWithConcreteInfrastructure]] = None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

handle_prompt(*prompt: eventsourcing.application.simple.Prompt*) → None
Handles publication of a prompt.

Abstract method which must be implemented on concrete descendants.

run_followers(*prompt: eventsourcing.application.simple.Prompt*) → None
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.system.runner.**MultiThreadedRunner**(*system: eventsourcing.system.definition.System*, *poll_interval: Optional[int] = None*, *clock_speed: Union[int, float, None] = None*, ***kwargs*)
Bases: *eventsourcing.system.runner.InProcessRunner*

Runs a system with a thread for each process.

__init__(*system: eventsourcing.system.definition.System*, *poll_interval: Optional[int] = None*, *clock_speed: Union[int, float, None] = None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

start() → None
Starts running the system.

Abstract method which must be implemented on concrete descendants.

handle_prompt(*prompt: eventsourcing.application.simple.Prompt*) → None
Handles publication of a prompt.

Abstract method which must be implemented on concrete descendants.

close() → None
Closes a running system.

class eventsourcing.system.runner.**PromptOutbox**
Bases: typing.Generic

Has a collection of downstream prompt inboxes.

__init__() → None
Initialize self. See help(type(self)) for accurate signature.

put(*prompt: Union[eventsourcing.application.simple.PromptToPull, str]*) → None
Puts prompt in each downstream inbox (an actual queue).

class eventsourcing.system.runner.**PromptQueuedApplicationThread**(***, *process: eventsourcing.application.process.ProcessApplica*
poll_interval: int = 5, *inbox: queue.Queue*, *outbox: Optional[eventsourcing.system.runner.Pro*
clock_event: Optional[threading.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.

`__init__`(**, process: eventsourcing.application.process.ProcessApplication, poll_interval: int = 5, inbox: queue.Queue, outbox: Optional[eventsourcing.system.runner.PromptOutbox], clock_event: Optional[threading.Event] = None*)

Initialises the thread with a process application object, and inbox and outbox.

Parameters

- **`process`** (`ProcessApplication`) – Application object.
- **`poll_interval`** (*`int`*) – Interval to check for upstream events.
- **`inbox`** (*`Queue`*) – For incoming prompts.
- **`outbox`** (`PromptOutbox`) – For outgoing prompts.
- **`clock_event`** – Event that "clocks" this thread (optional).

`run`() → None

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.system.runner.`**`SteppingRunner`**(*normal_speed: int = 1, scale_factor: int = 1, is_verbose: bool = False, *args, **kwargs*)

Bases: *`eventsourcing.system.runner.InProcessRunner`*

`__init__`(*normal_speed: int = 1, scale_factor: int = 1, is_verbose: bool = False, *args, **kwargs*)

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

class `eventsourcing.system.runner.`**`SteppingSingleThreadedRunner`**(**args, **kwargs*)

Bases: *`eventsourcing.system.runner.SteppingRunner`*

`__init__`(**args, **kwargs*)

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

`start`() → None

Starts running the system.

Abstract method which must be implemented on concrete descendants.

`handle_prompt`(*prompt: eventsourcing.application.simple.Prompt*) → None

Ignores prompts.

`close`() → None

Closes a running system.

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

Bases: `threading.Thread`

`__init__`(**args, **kwargs*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

class eventsourcing.system.runner.**ProcessRunningClockThread**(**, normal_speed: int, scale_factor: int, stop_event: threading.Event, is_verbose: bool = False, seen_prompt_events: Dict[str, threading.Event], processes: Dict[str, eventsourcing.application.process.ProcessApplication], use_direct_query_if_available: bool = False*)

Bases: *eventsourcing.system.runner.ClockThread*

__init__(**, normal_speed: int, scale_factor: int, stop_event: threading.Event, is_verbose: bool = False, seen_prompt_events: Dict[str, threading.Event], processes: Dict[str, eventsourcing.application.process.ProcessApplication], use_direct_query_if_available: bool = False*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run() → None

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.system.runner.**SteppingMultiThreadedRunner**(**args, **kwargs*)

Bases: *eventsourcing.system.runner.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.

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

`handle_prompt`(*prompt: eventsourcing.application.simple.Prompt*) → None
Handles publication of a prompt.

Abstract method which must be implemented on concrete descendants.

`start`() → None
Starts running the system.

Abstract method which must be implemented on concrete descendants.

`close`() → None
Closes a running system.

class `eventsourcing.system.runner.`**`BarrierControlledApplicationThread`**(*process: eventsourcing.application.process.Process*
fetch_barrier: threading.Barrier, *execute_barrier: threading.Barrier*, *stop_event: threading.Event*)

Bases: `threading.Thread`

`__init__`(*process: eventsourcing.application.process.ProcessApplication*, *fetch_barrier: threading.Barrier*, *execute_barrier: threading.Barrier*, *stop_event: threading.Event*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

`run`() → None
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.system.runner.**BarrierControllingClockThread**(*normal_speed: int, scale_factor: int, tick_interval: Union[int, float, None], fetch_barrier: threading.Barrier, execute_barrier: threading.Barrier, stop_event: threading.Event, is_verbose: bool = False*)

Bases: *eventsourcing.system.runner.ClockThread*

__init__(*normal_speed: int, scale_factor: int, tick_interval: Union[int, float, None], fetch_barrier: threading.Barrier, execute_barrier: threading.Barrier, stop_event: threading.Event, is_verbose: bool = False*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run() → None

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.

multiprocess

class eventsourcing.system.multiprocess.**MultiprocessRunner**(*system: eventsourcing.system.definition.System*, *pipeline_ids: Sequence[int] = (0,)*, *poll_interval: Optional[int] = None*, *setup_tables: bool = False*, *sleep_for_setup_tables: int = 0*, ***kwargs*)

Bases: *eventsourcing.system.definition.AbstractSystemRunner*

__init__(*system: eventsourcing.system.definition.System*, *pipeline_ids: Sequence[int] = (0,)*, *poll_interval: Optional[int] = None*, *setup_tables: bool = False*, *sleep_for_setup_tables: int = 0*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

start() → None
Starts running the system.

Abstract method which must be implemented on concrete descendants.

close() → None
Closes a running system.

class eventsourcing.system.multiprocess.**OperatingSystemProcess**(*application_process_class: Type[eventsourcing.application.process.P infrastructure_class: Type[eventsourcing.application.simple.A upstream_names: List[str], inbox: queue.Queue, outbox: Optional[eventsourcing.system.runner.Prom str]][Tuple[int, str]] = None, pipeline_id: int = 0, poll_interval: int = 5, setup_tables: bool = False, *args, **kwargs*)

Bases: multiprocessing.context.Process

__init__(*application_process_class: Type[eventsourcing.application.process.ProcessApplication], infrastructure_class: Type[eventsourcing.application.simple.ApplicationWithConcreteInfrastructure], upstream_names: List[str], inbox: queue.Queue, outbox: Optional[eventsourcing.system.runner.PromptOutbox[typing.Tuple[int, str]][Tuple[int, str]]] = None, pipeline_id: int = 0, poll_interval: int = 5, setup_tables: bool = False, *args, **kwargs*)
Initialize self. See help(type(self)) for accurate signature.

run() → None
Method to be run in sub-process; can be overridden in sub-class

grpc

class eventsourcing.system.grpc.runner.**GrpcRunner**(**args*, *pipeline_ids=(0,)*, *push_prompt_interval=0.25*, ***kwargs*)
Bases: `eventsourcing.system.definition.AbstractSystemRunner`

System runner that uses gRPC to communicate between process applications.

__init__(**args*, *pipeline_ids=(0,)*, *push_prompt_interval=0.25*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

generate_ports(*start: int*)
Generator that yields a sequence of ports from given start number.

create_address()
Creates a new address for a gRPC server.

start()
Starts running a system of process applications.

start_processor(*application_topic*, *pipeline_id*, *infrastructure_topic*, *setup_table*, *address*, *upstreams*, *downstreams*)
Starts a gRPC process.

close() → None
Stops all gRPC processes started by the runner.

stop_process(*process*)
Stops given gRPC process.

kill_process(*process*)
Kills given gRPC process, if it still running.

listen(*name*, *processor_clients*)
Constructs a listener using the given clients.

class eventsourcing.system.grpc.runner.**ClientWrapper**(*client: eventsourcing.system.grpc.processor.ProcessorClient*)
Bases: `object`

Wraps a gRPC client, and returns a MethodWrapper when attributes are accessed.

__init__(*client: eventsourcing.system.grpc.processor.ProcessorClient*)
Initialize self. See help(type(self)) for accurate signature.

class eventsourcing.system.grpc.runner.**MethodWrapper**(*client: eventsourcing.system.grpc.processor.ProcessorClient*, *method_name: str*)
Bases: `object`

Wraps a gRPC client, and invokes application method name when called.

__init__(*client: eventsourcing.system.grpc.processor.ProcessorClient*, *method_name: str*)
Initialize self. See help(type(self)) for accurate signature.

__call__(**args*, ***kwargs*)
Call self as a function.

class eventsourcing.system.grpc.runner.**ProcessorListener**(*name, address, clients: List[eventsourcing.system.grpc.processor.Process*

Bases: eventsourcing.system.grpc.processor_pb2_grpc.ProcessorServicer

Starts server and uses clients to request prompts from connected servers.

__init__(*name, address, clients: List[eventsourcing.system.grpc.processor.ProcessorClient]*)
Initialize self. See help(type(self)) for accurate signature.

serve()
Starts server.

Ping(*request, context*)
Missing associated documentation comment in .proto file

Prompt(*request, context*)
Missing associated documentation comment in .proto file

prompt(*upstream_name*)
Sets prompt events for given upstream process.

class eventsourcing.system.grpc.processor.**NotificationLogView**(*notification_log: eventsourcing.application.notificationlog.LocalNotifi json_encoder: eventsourcing.utils.transcoding.ObjectJSONEncoder*

Bases: object

Presents sections of notification log for gRPC server.

__init__(*notification_log: eventsourcing.application.notificationlog.LocalNotificationLog, json_encoder: eventsourcing.utils.transcoding.ObjectJSONEncoder*)
Initialize self. See help(type(self)) for accurate signature.

class eventsourcing.system.grpc.processor.**StartClient**(*clients, name, address*)
Bases: threading.Thread

Thread that creates a gRPC client and connects to a gRPC server.

__init__(*clients, name, address*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()
Creates client and connects to address.

class eventsourcing.system.grpc.processor.**PullNotifications**(*prompt_event: threading.Event*, *reader: eventsourcing.application.notificationlog.NotificationLo*, *process_application: eventsourcing.application.process.ProcessApplication*, *event_queue: queue.Queue*, *upstream_name: str*, *has_been_stopped: threading.Event*)

Bases: `threading.Thread`

Thread the pulls notifications from upstream process application.

__init__(*prompt_event: threading.Event*, *reader: eventsourcing.application.notificationlog.NotificationLogReader*, *process_application: eventsourcing.application.process.ProcessApplication*, *event_queue: queue.Queue*, *upstream_name: str*, *has_been_stopped: threading.Event*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run() → None

Loops over waiting for prompt event to be set, reads event notifications from reader, gets domain events from notifications, and puts domain events on the queue of unprocessed events.

set_reader_position()

Sets reader position from recorded position.

class eventsourcing.system.grpc.processor.**RemoteNotificationLog**(*client: eventsourcing.system.grpc.processor.ProcessorCli*, *json_decoder: eventsourcing.utils.transcoding.ObjectJSONDecoc*, *section_size: int*)

Bases: `eventsourcing.application.notificationlog.AbstractNotificationLog`

Notification log that get notification log sections using gRPC client.

__init__(*client: eventsourcing.system.grpc.processor.ProcessorClient*, *json_decoder: eventsourcing.utils.transcoding.ObjectJSONDecoder*, *section_size: int*)

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

section_size

Size of section of notification log.

`__getitem__`(*section_id: str*) → eventsourcing.application.notificationlog.Section
Get section of notification log.

Parameters **`section_id`** – ID of a section of the notification log.

class `eventsourcing.system.grpc.processor.`**`ProcessorServer`**(*application_topic*, *pipeline_id*, *infrastructure_topic*, *setup_table*, *address*, *upstreams*, *downstreams*, *push_prompt_interval*)

Bases: `eventsourcing.system.grpc.processor_pb2_grpc.ProcessorServicer`

`__init__`(*application_topic*, *pipeline_id*, *infrastructure_topic*, *setup_table*, *address*, *upstreams*, *downstreams*, *push_prompt_interval*)
Initialize self. See help(type(self)) for accurate signature.

`serve`()
Starts gRPC server.

`wait_for_termination`()
Runs until termination of process.

`Ping`(*request*, *context*)
Missing associated documentation comment in .proto file

`Lead`(*request*, *context*)
Missing associated documentation comment in .proto file

`lead`(*downstream_name*, *downstream_address*)
Starts client and registers downstream to receive prompts.

`start_client`(*name*, *address*)
Starts client connected to given address.

`Prompt`(*request*, *context*)
Missing associated documentation comment in .proto file

`prompt`(*upstream_name*)
Set prompt event for upstream name.

`GetNotifications`(*request*, *context*)
Missing associated documentation comment in .proto file

`get_notification_log_section`(*section_id*)
Returns section for given section ID.

`CallApplicationMethod`(*request*, *context*)
Missing associated documentation comment in .proto file

`stop`(**args*)
Stops the gRPC server.

actors

class eventsourcing.system.thespian.**ThespianRunner**(*system: eventsourcing.system.definition.System*, *pipeline_ids=(0,)*, *system_actor_name='system'*, *shutdown_on_close=False*, ***kwargs*)

Bases: `eventsourcing.system.definition.AbstractSystemRunner`

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

__init__(*system: eventsourcing.system.definition.System*, *pipeline_ids=(0,)*, *system_actor_name='system'*, *shutdown_on_close=False*, ***kwargs*)

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

start()

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

close()

Stops all the actors running a system of process applications.

class eventsourcing.system.thespian.**SystemActor**

Bases: `thespian.actors.Actor`

__init__()

Called to initialize the Actor.

Override this initialization method as needed in defined Actors.

N.B. Currently the Actor is not yet fully realized in the ActorSystem when __init__ is invoked. This means that the Actor __init__ cannot invoke any ActorSystem-related operations (no .send(), .handleDeadLetters(), .notifyOnSystemRegistrationChanges(), etc.)

Also note that there is post-__init__ processing of a created Actor object by the ActorSystem that is necessary for it to become a full Actor. The Actor's __init__() must not perform Actor-related operations, and the __init__() is not sufficient to *fully* initialize an Actor object. This ensures that the ActorSystem is involved in the creation of a useable Actor (i.e. the ActorSystem is the Factory for an Actor).

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.system.thespian.**PipelineActor**

Bases: `thespian.actors.Actor`

__init__()

Called to initialize the Actor.

Override this initialization method as needed in defined Actors.

N.B. Currently the Actor is not yet fully realized in the ActorSystem when __init__ is invoked. This means that the Actor __init__ cannot invoke any ActorSystem-related operations (no .send(), .handleDeadLetters(), .notifyOnSystemRegistrationChanges(), etc.)

Also note that there is post-__init__ processing of a created Actor object by the ActorSystem that is necessary for it to become a full Actor. The Actor's __init__() must not perform Actor-related operations, and the __init__() is not sufficient to *fully* initialize an Actor object. This ensures that the ActorSystem is involved in the creation of a useable Actor (i.e. the ActorSystem is the Factory for an Actor).

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.system.thespian.`**ProcessMaster**
Bases: `thespian.actors.Actor`

__init__ ()
Called to initialize the Actor.

Override this initialization method as needed in defined Actors.

N.B. Currently the Actor is not yet fully realized in the ActorSystem when __init__ is invoked. This means that the Actor __init__ cannot invoke any ActorSystem-related operations (no .send(), .handleDeadLetters(), .notifyOnSystemRegistrationChanges(), etc.)

Also note that there is post-__init__ processing of a created Actor object by the ActorSystem that is necessary for it to become a full Actor. The Actor's __init__() must not perform Actor-related operations, and the __init__() is not sufficient to *fully* initialize an Actor object. This ensures that the ActorSystem is involved in the creation of a useable Actor (i.e. the ActorSystem is the Factory for an Actor).

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.system.thespian.`**ProcessSlave**
Bases: `thespian.actors.Actor`

__init__ ()
Called to initialize the Actor.

Override this initialization method as needed in defined Actors.

N.B. Currently the Actor is not yet fully realized in the ActorSystem when __init__ is invoked. This means that the Actor __init__ cannot invoke any ActorSystem-related operations (no .send(), .handleDeadLetters(), .notifyOnSystemRegistrationChanges(), etc.)

Also note that there is post-__init__ processing of a created Actor object by the ActorSystem that is necessary for it to become a full Actor. The Actor's __init__() must not perform Actor-related operations, and the __init__() is not sufficient to *fully* initialize an Actor object. This ensures that the ActorSystem is involved in the creation of a useable Actor (i.e. the ActorSystem is the Factory for an Actor).

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.system.ray.`**RayRunner** (*system: eventsourcing.system.definition.System*, *pipeline_ids=(0,)*, *poll_interval: Optional[int] = None*, *setup_tables: bool = False*, *sleep_for_setup_tables: int = 0*, *db_uri: Optional[str] = None*, ***kwargs*)
Bases: `eventsourcing.system.definition.AbstractSystemRunner`

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

__init__ (*system: eventsourcing.system.definition.System*, *pipeline_ids=(0,)*, *poll_interval: Optional[int] = None*, *setup_tables: bool = False*, *sleep_for_setup_tables: int = 0*, *db_uri: Optional[str] = None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

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

close()
Closes a running system.

1.19.6 utils

The utils package contains common functions that are used in more than one layer.

topic

eventsourcing.utils.topic.**get_topic**(*domain_class: type*) → str
Returns a string describing a class.

Parameters **domain_class** – A class.

Returns A string describing the class.

eventsourcing.utils.topic.**resolve_topic**(*topic: str*) → Any
Resolves topic to the object it references.

Parameters **topic** – A string describing a code object (e.g. an object class).

Raises ***TopicResolutionError*** – If there is no such class.

Returns Code object that the topic references.

eventsourcing.utils.topic.**resolve_attr**(*obj: Any*, *path: str*) → Any
A recursive version of getattr for navigating dotted paths.

Parameters

- **obj** – An object for which we want to retrieve a nested attribute.
- **path** – A dot separated string containing zero or more attribute names.

Raises **AttributeError** – If there is no such attribute.

Returns The attribute referred to by the path.

eventsourcing.utils.topic.**reconstruct_object**(*obj_class: Type[T]*, *obj_state: Dict[str, Any]*) → T
Reconstructs object from given class and state.

Parameters

- **obj_class** – Class of object to be reconstructed.
- **obj_state** – State of object to be reconstructed.

Returns Reconstructed object.

transcoding

eventsourcing.utils.transcoding.**encoderpolicy**(*arg=None*)
Decorator for encoder policy.

Allows default behaviour to be built up from methods registered for different types of things, rather than chain of isinstance() calls in a long if-else block.

`eventsourcing.utils.transcoding.`**`decoderpolicy`**(*arg=None*)

Decorator for decoder policy.

Allows default behaviour to be built up from methods registered for different named keys, rather than chain of "in dict" queries in a long if-else block.

class `eventsourcing.utils.transcoding.`**`ObjectJSONEncoder`**(*sort_keys=False*)

Bases: `json.encoder.JSONEncoder`

`__init__`(*sort_keys=False*)

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

`encode`(*o*) → bytes

Return a JSON string representation of a Python data structure.

```
>>> from json.encoder import JSONEncoder
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

class `eventsourcing.utils.transcoding.`**`ObjectJSONDecoder`**(*object_hook=None*, ***kwargs*)

Bases: `json.decoder.JSONDecoder`

`__init__`(*object_hook=None*, ***kwargs*)

`object_hook`, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given `dict`. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).

`object_pairs_hook`, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of `object_pairs_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders. If `object_hook` is also defined, the `object_pairs_hook` takes priority.

`parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default this is equivalent to float(num_str). This can be used to use another datatype or parser for JSON floats (e.g. decimal.Decimal).

`parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default this is equivalent to int(num_str). This can be used to use another datatype or parser for JSON integers (e.g. float).

`parse_constant`, if specified, will be called with one of the following strings: -Infinity, Infinity, NaN. This can be used to raise an exception if invalid JSON numbers are encountered.

If `strict` is false (true is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0-31 range, including `'\t'` (tab), `'\n'`, `'\r'` and `'\0'`.

cipher

class `eventsourcing.utils.cipher.aes.`**`AESCipher`**(*cipher_key: bytes*)

Bases: `object`

Cipher strategy that uses Crypto library AES cipher in GCM mode.

`__init__`(*cipher_key: bytes*)

Initialises AES cipher strategy with `cipher_key`.

Parameters **`cipher_key`** – 16, 24, or 32 random bytes

`encrypt`(*plaintext: bytes*) → bytes

Return ciphertext for given plaintext.

`decrypt`(*ciphertext: bytes*) → bytes

Return plaintext for given ciphertext.

random

`eventsourcing.utils.random.`**`encoded_random_bytes`**(*num_bytes: int*) → str

Generates random bytes, encoded as Base64 unicode string.

Parameters **`num_bytes`** – Number of random bytes to generate.

Returns Random bytes of specified length, encoded as Base64 unicode string.

`eventsourcing.utils.random.`**`encode_random_bytes`**(*num_bytes: int*) → str

Generates random bytes, encoded as Base64 unicode string.

Parameters **`num_bytes`** – Number of random bytes to generate.

Returns Random bytes of specified length, encoded as Base64 unicode string.

`eventsourcing.utils.random.`**`random_bytes`**(*num_bytes: int*) → bytes

Generates random bytes.

Parameters **`num_bytes`** – Number of random bytes to generate.

Returns Random bytes of specified length.

Type bytes

`eventsourcing.utils.random.`**`encode_bytes`**(*value: bytes*) → str

Encodes value as Base64 unicode string.

Parameters **`value`** – Bytes to be encoded.

`eventsourcing.utils.random.`**`decode_bytes`**(*value: str*) → bytes

Decodes bytes from Base64 encoded unicode string.

Parameters **`value`** (`str`) – Base64 string.

Returns Bytes that were encoded with Base64.

Return type bytes

times

`eventsourcing.utils.times.`**`decimaltimestamp_from_uuid`**(*value: uuid.UUID*) → decimal.Decimal

Return a floating point unix timestamp from UUID value.

Parameters **`value`** –

Returns Unix timestamp in seconds, with microsecond precision.

Return type Decimal

`eventsourcing.utils.times.`**`timestamp_long_from_uuid`**(*value: uuid.UUID*) → int

Returns an integer value representing a unix timestamp in tenths of microseconds.

Parameters **`value`** –

Returns Unix timestamp integer in tenths of microseconds.

Return type int

`eventsourcing.utils.times.`**`decimaltimestamp`**(*t: Optional[float] = None*) → decimal.Decimal

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().

Return type Decimal

`eventsourcing.utils.times.`**`datetime_from_timestamp`**(*t: Union[decimal.Decimal, float]*) → datetime.datetime

Returns naive UTC datetime from decimal UNIX timestamps such as time.time().

Parameters **`t`** – timestamp, either Decimal or float

Returns datetime.datetime object

1.19.7 whitehead

This module contains three base classes, which distinguish between "actual occasion" (which "domain model event" is an example of) and "enduring object" (which "domain model aggregate" is an example of). These terms "actual occasion" and "enduring object" are taken from Alfred North Whitehead's Process and Reality (published 1929). The base classes both inherit from a base class "event" because, in Whitehead's system, an enduring object is an event, and so is an actual occasion.

`class` `eventsourcing.whitehead.`**`Event`**

Bases: `object`

"I shall use the term 'event' in the more general sense of a nexus of actual occasions, inter-related in some determinate fashion in one extensive quantum. An actual occasion is the limiting type of an event with only one member."

Alfred North Whitehead, 1929

class `eventsourcing.whitehead.`**`ActualOccasion`**
Bases: *`eventsourcing.whitehead.Event`*

"'Actual entities' – also termed 'actual occasions' – are the final real things of which the world is made up. There is no going behind actual entities to find anything more real."

"Just as 'potentiality for process' is the meaning of the more general term 'entity' or 'thing'; so 'decision' is the additional meaning imported by the word 'actual' into the phrase 'actual entity'. 'Actuality' is the decision amid 'potentiality'. It represents stubborn fact which cannot be evaded."

"Actual entities perish, but do not change; they are what they are."

Alfred North Whitehead, 1929

class `eventsourcing.whitehead.`**`EnduringObject`**
Bases: *`eventsourcing.whitehead.Event`*

"The notions of 'social order' and of 'personal order' cannot be omitted from this preliminary sketch. A 'society' in the sense in which that term is here used, is a nexus with social order; and an 'enduring object' or 'enduring creature' is a society whose social order has taken the special form of 'personal order.'"

"A nexus enjoys 'personal order' when (a) it is a 'society' and when the genetic relatedness of its members orders these members 'serially'."

Alfred North Whitehead, 1929

1.19.8 exceptions

A few exception classes are defined by the library to indicate particular kinds of error.

exception `eventsourcing.exceptions.`**`EventSourcingError`**
Bases: `Exception`

Base eventsourcing exception.

exception `eventsourcing.exceptions.`**`TopicResolutionError`**
Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raised when unable to resolve a topic to a Python class.

exception `eventsourcing.exceptions.`**`EntityVersionNotFound`**
Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raise when accessing an entity version that does not exist.

exception `eventsourcing.exceptions.`**`RecordConflictError`**
Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raised when database raises an integrity error.

exception `eventsourcing.exceptions.`**`PromptFailed`**
Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raised when prompt fails.

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.`**`MismatchedOriginatorError`**
Bases: *`eventsourcing.exceptions.ConsistencyError`*

Raised when applying an event to an inappropriate object.

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.`**`MutatorRequiresTypeNotInstance`**
Bases: *`eventsourcing.exceptions.ConsistencyError`*

Raised when mutator function received a class rather than an 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.`**`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.`**`HeadHashError`**
Bases: *`eventsourcing.exceptions.DataIntegrityError`*, *`eventsourcing.exceptions.MismatchedOriginatorError`*

Raised when applying an event with hash different from aggregate head.

exception `eventsourcing.exceptions.`**`EntityIsDiscarded`**
Bases: `AssertionError`

Raised when access to a recently discarded entity object is attempted.

exception `eventsourcing.exceptions.`**`ProgrammingError`**
Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raised when programming errors are encountered.

exception `eventsourcing.exceptions.`**`RepositoryKeyError`**
Bases: `KeyError`, *`eventsourcing.exceptions.EventSourcingError`*

Raised when using entity repository's dictionary like interface to get an entity that does not exist.

exception `eventsourcing.exceptions.`**`ArrayIndexError`**
Bases: `IndexError`, *`eventsourcing.exceptions.EventSourcingError`*

Raised when appending item to an array that is full.

exception `eventsourcing.exceptions.`**`DatasourceSettingsError`**
Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raised when an error is detected in settings for a datasource.

exception `eventsourcing.exceptions.`**`OperationalError`**

Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raised when an operational error is encountered.

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.`**`TrackingRecordNotFound`**

Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raised when a tracking record is not found.

exception `eventsourcing.exceptions.`**`CausalDependencyFailed`**

Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raised when a causal dependency fails (after its tracking record not found).

exception `eventsourcing.exceptions.`**`EventRecordNotFound`**

Bases: *`eventsourcing.exceptions.EventSourcingError`*

Raised when an event record is not found.

exception `eventsourcing.exceptions.`**`EncoderTypeError`**

Bases: `TypeError`

Python Module Index

Index

Symbols