eventsourcing Documentation

Release 9.1.3

John Bywater

Oct 09, 2021

Contents

A library for event sourcing in Python. This project is hosted on GitHub.

CHAPTER 1

Contents

1.1 Introduction

1.1.1 What is event sourcing?

One definition of event sourcing 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.

Whilst the basic event sourcing patterns are quite simple and can be reproduced in code for each project, event sourcing as a persistence mechanism for domain-driven design appears as a "conceptually cohesive mechanism" and so can be partitioned into a "separate lightweight framework".

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."*

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 events, such as the domain events of event-sourced aggregates in a domain-driven design, and snapshots of those aggregates. A variety of schemas and technologies can be used for 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 and examples of event-sourced aggregates and event-sourced applications.

It is possible using this library to define an entire event-driven system of event-sourced applications independently of infrastructure and mode of running. 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 Features

Flexible event store — flexible persistence of domain events. Combines an event mapper and an event recorder in ways that can be easily extended. Mapper uses a transcoder that can be easily extended to support custom model object types. Recorders supporting different databases can be easily substituted and configured with environment variables.

Domain models and applications — base classes for domain model aggregates and applications. Suggests how to structure an event-sourced application.

Application-level encryption and compression — encrypts and decrypts events inside the application. 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. 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.

Snapshotting — reduces access-time for aggregates with many domain events.

Versioning - allows domain model changes to be introduced after an application has been deployed. Both domain events and aggregate 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 aggregate object is reconstructed.

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.

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. Supports materialized views and CQRS.

Event-driven systems — reliable event processing. Event-driven systems can be defined independently of particular persistence infrastructure and mode of running.

Detailed documentation — documentation provides general overview, introduction of concepts, explanation of usage, and detailed descriptions of library classes.

Worked examples — includes examples showing how to develop aggregates, applications and systems.

1.1.4 Design overview

The design of the library follows the notion of a "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 on any other layer. The application layer depends on the domain and infrastructure layers, and the interface layer depends only on the application layer.

1.1.5 Register issues

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

1.2 Installation guide

This version of the library is compatible with Python versions 3.7, 3.8, 3.9, and 3.10. The library's suite of tests is run against these versions and has 100% line and branch coverage.

You can use pip to install the library from the Python Package Index. It is recommended always to install into a virtual environment.

```
$ pip install eventsourcing
```

When including the library in a list of project dependencies, in order to avoid installing future incompatible releases, it is recommended to specify the major and minor version numbers.

As an example, the expression below would install the latest version of the v9.1.x release, allowing future bug fixes released with point version number increments.

```
eventsourcing<=9.1.99999
```

Specifying the major and minor version number in this way will avoid any potentially destabilising additional features introduced with minor version number increments, and also any backwards incompatible changes introduced with major version number increments.

This package depends only on modules from the Python Standard Library, except for the extra options described below.

1.2.1 Install options

Running the install command with different options will 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.

For example, if you want to store cryptographically encrypted events, then install with the `crypto` option. This simply installs PyCryptodome so feel free to make your project depend on that instead.

```
$ pip install "eventsourcing[crypto]"
```

If you want to store events with PostgreSQL, then install with the `postgres` option. This simply installs Psycopg2 so feel free to make your project depend on that instead. Please note, the binary version psycopg2-binary is a convenient alternative for development and testing, but the main package is recommended by the Psycopg2 developers for production usage.

```
$ pip install "eventsourcing[postgres]"
```

Options can be combined, so that if you want to store encrypted events in PostgreSQL, then install with the `crypto` and `postgres` options.

```
$ pip install "eventsourcing[crypto,postgres]"
```

1.2.2 Developers

If you want to install the code for the purpose of developing the library, then fork and clone the GitHub repository and install from the root folder with the 'dev' option. This option will install a number of packages that help with development and documentation, such as the above extra dependencies along with Sphinx, Coverage.py, Black, mypy, Flake8, and isort.

```
$ pip install ".[dev]"
```

Alternatively, the project's Makefile can be used to the same effect with the following command.

```
$ make install
```

Once installed, you can check the unit tests pass and the code is 100% covered by the tests with the following command.

```
$ make test
```

Before the tests will pass, you will need setup PostgreSQL. The following commands will install PostgreSQL on MacOS and setup the database and database user. If you already have PostgreSQL installed, just create the database and user. If you prefer to run PostgreSQL in a Docker container, feel free to do that too.

```
$ brew install postgresql
$ brew services start postgresql
$ psql postgres
postgres=# CREATE DATABASE eventsourcing;
postgres=# CREATE USER eventsourcing WITH PASSWORD 'eventsourcing';
```

You can also check the syntax and static types are correct with the following command (which uses isort, Black, Flake8, and mypy).

```
$ make lint
```

The code can be automatically reformatted using the following command (which uses isort and Black). Flake8 and mypy errors will often need to be fixed by hand.

```
$ make fmt
```

You can build the docs, and make sure they build, with the following command (which uses Sphinx).

```
$ make docs
```

If you wish to submit changes to the library, before submitting a pull request please check all three things (lint, docs, and test) which you can do conveniently with the following command.

```
$ make prepush
```

If you wish to submit a pull request on GitHub, please target the main branch. Improvements of any size are always welcome.

1.3 Support options

I'm very grateful for your interest in this library. It has taken quite a lot of time to create this library. Similarly, it may take some time to understand the library and develop well-designed event-sourced applications.

To supplement the detailed documentation, professional training workshops and development services are available. Friendly community support is also available on the Slack channel.

Please support the continuing development and maintenance of this library by starring the project on GitHub and if possible by making a regular donation. If you have any issues using the library or reading the documentation, please raise an issue on GitHub, feel free to start a discussion in the Slack channel, or create a pull request.

1.3.1 Professional support

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

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

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

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

1.3.2 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.

1.3.3 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.

1.3.4 Support the project

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

1.4 `domain` — Domain models

This module helps with developing event-sourced domain models.

An event-sourced domain model has many event-sourced **aggregates**. The state of an event-sourced aggregate is determined by a sequence of **domain events**. The time needed to reconstruct an aggregate from its domain events can be reduced by using **snapshots**.

1.4.1 Aggregates in DDD

Aggregates are enduring objects which enjoy adventures of change. The book *Domain-Driven Design* by Eric Evans' describes a design pattern called "aggregate" in the following way.

> *"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."*

An aggregate is a cluster of 'entities' and 'value objects'. An entity is an object with a fixed unique identity and other attributes that may vary. A value object does not vary, and does not necessarily have a unique identity. This basic notion of a cluster of software objects is understandable as straightforward object-oriented programming.

An aggregate has a 'root'. The 'root' of an aggregate is an entity. This entity is known as the 'root entity' or the 'aggregate root'. Entities have IDs and the ID of the root entity is used to uniquely identify the cluster of objects in a domain model. Access to the cluster of objects is made through the root entity.

Changes to the cluster of objects are made using 'command methods' defined on the root entity, and the state of the cluster of objects is obtained by using either 'query methods' or properties of the root entity. The idea of distinguishing between command methods (methods that change state but do not return values) and query methods (methods that return values but do not change state) is known as 'command-query separation' or CQS. CQS was devised by Bertrand Meyer and described in his book *Object Oriented Software Construction*.

The 'boundary' of the aggregate is defined by the extent of the cluster of objects. The 'consistency' of the cluster of objects is maintaining by making sure all the changes that result from a single command are recorded atomically. There is only ever one cluster of objects for any given aggregate, so there is no branching, and the atomic changes have a serial order. These two notions of 'consistency' and 'boundary' are combined in the notion in *Domain-Driven Design* of 'consistency boundary'. Whilst we can recognise the cluster of objects as basic object-orientated programming, and we can recognise the use of command and query methods as the more refined pattern called CQS, the 'consistency boundary' notion gives to the aggregates in *Domain-Driven Design* their distinctive character.

1.4.2 Event-sourced aggregates

It is in the Zen of Python that explicit is better than implicit. The changes to an aggregate's cluster of objects will always follow from decisions made by the aggregate, but these decisions had not been directly expressed as objects. It will always be true that a decision itself does not change, but this fact had not been directly expressed.

> *"Explicit is better than implicit."*

To make things explicit, the decisions made in the command methods of an aggregate can be coded and recorded as a sequence of immutable 'domain event' objects, and this sequence can be used to evolve the aggregate's cluster of entities and value objects. Event-sourced aggregates make these things explicit. For each event-sourced aggregate, there is a sequence of domain event objects, and the state of an event-sourced aggregate is determined by its sequence of domain event objects. The state of an aggregate can change, and its sequence of domain events can be augmented. But once created the individual domain event objects do not change. They are what they are. The notion of 'change' is the contrast between successive domain events in an aggregate's sequence (contrasted from the standpoint of the cluster of objects within an aggregate's consistency boundary, which is a standpoint that may change since there must be a function that applies the events to the cluster, and this function can be adjusted. Hence it isn't strictly true to say that the state of an aggregate is determined by a sequence of events. The events merely contribute determination, and the state is in fact determined by a combination of the sequence of events and a function that constructs that state from those events, but I digress. . .)

The state of an aggregate, event-sourced or not, is changed by calling its command methods. In an event-sourced aggregate, the command methods create new domain event objects. The domain events are used to evolve the state of the aggregate. By evolving the state of the aggregate via creating and applying domain events, the domain events can be recorded and used in future to reconstruct the state of the aggregate.

One command may result in many new domain event objects, and a single client request may result in the execution of many commands. To maintain consistency in the domain model, all the domain events triggered by responding to a single client request must be recorded atomically in the order they were created, otherwise the recorded state of the aggregate may become inconsistent with respect to that which was desired or expected.

Aggregate base class

This library's `Aggregate` class is a base class for event-sourced aggregates. It can be imported from the library's `eventsourcing.domain` module.

```
from eventsourcing.domain import Aggregate
```

The *Aggregate* base class can be used to develop event-sourced aggregates. See for example the World aggregate in the *basic example* below. The *Aggregate* base class has three methods which can be used by subclasses:

- the class method *_create()* is used to create aggregate objects;
- the object method *trigger_event()* is used to trigger subsequent events; and
- the object method *collect_events()* is used to collect aggregate events that have been triggered.

These methods are explained below.

Creating new aggregates

Firstly, the *Aggregate* class has a "private" class method *_create()* which can be used to create a new aggregate. It works by creating the first of a new sequence of domain event objects, and uses this domain event object to construct and initialise an instance of the aggregate class. Usually, this "private" method will be called by a "public" class method defined on a subclass of the *Aggregate* base class. For example, see the class method create() of the World aggregate class in the *basic example* below.

The *_create()* method has a required positional argument event_class which is used by the caller to pass a domain event class that will represent the fact that an aggregate was "created". A domain event object of this type will be constructed by this method, and this domain event object will be used to construct and initialise an aggregate object. This method will then return that aggregate object. The *_create()* method also has a required id argument which should be a Python UUID object that will be used to uniquely identify the aggregate in the domain model.

```
from uuid import uuid4

aggregate_id = uuid4()

aggregate = Aggregate._create(Aggregate.Created, id=aggregate_id)
```

The library's *Aggregate* base class is defined with a nested class *Created* which can be used to represent the fact that an aggregate was "created". The *Created* class is defined as a frozen Python data class with four attributes: the ID of an aggregate, a version number, a timestamp, and the *topic* of an aggregate class — see the *Domain events* section below for more information. Except for the ID which is passed as the id argument to the *_create()* method, the values of these other attributes are worked out by the *_create()* method. The *Created* class can be used directly, but is normally subclassed to define a particular "created" event class for a particular aggregate class, with a suitable name and with suitable extra attributes that represent the particular beginning of a particular type of aggregate. A "created" event class should be named using a past participle that describes the beginning of something, such as "Started", "Opened", or indeed "Created".

The *_create()* method also accepts arbitrary keyword-only arguments, which if given will also be used to construct the event object in addition to those mentioned above. The "created" event object will be constructed with these additional arguments, and so the extra method arguments must be matched by the attributes of the "created" event class. (The concrete aggregate class's initializer method __init__() should also be coded to accept these extra arguments.)

Having been created, an aggregate object will have an aggregate ID. The ID is presented by its *id* property. The ID will be identical to the value passed with the id argument to the *_create()* method.

```
assert aggregate.id == aggregate_id
```

A new aggregate instance has a version number. The version number is presented by its *version* property, and is a Python int. The initial version of a newly created aggregate is always 1.

```
assert aggregate.version == 1
```

A new aggregate instance has a *created_on* property which gives the date and time when an aggregate object was created, and is determined by the timestamp attribute of the first event in the aggregate's sequence, which is the "created" event. It is a Python `datetime` object.

```
from datetime import datetime

assert isinstance(aggregate.created_on, datetime)
```

A new aggregate instance also has a *modified_on* property which gives the date and time when an aggregate object was last modified, and is determined by the timestamp attribute of the last event in the aggregate's sequence. It is also a Python `datetime` object.

```
from datetime import datetime

assert isinstance(aggregate.modified_on, datetime)
```

Initially, since there is only one event in the aggregate's sequence, the `created_on` and `modified_on` values are identical, and equal to the timestamp of the "created" event.

```
assert aggregate.created_on == aggregate.modified_on
```

Triggering subsequent events

Secondly, the *Aggregate* class has a method *trigger_event()* which can be called to create subsequent aggregate event objects and apply them to the aggregate. This method is usually called by the command methods of an aggregate to express the decisions that it makes. For example, see the `make_it_so()` method of the `World` class in the *basic example* below.

The *trigger_event()* method has a positional argument `event_class`, which is used to pass the type of aggregate event to be triggered.

```
from eventsourcing.domain import AggregateEvent

aggregate.trigger_event(AggregateEvent)
```

The *Aggregate* class has a nested *Event* class. It is defined as a frozen Python data class with three attributes: the ID of an aggregate, a version number, and a timestamp. It can be used as a base class to define aggregate event classes. The *Created* event class discussed above is a subclass of *Event*. For another example, see the `SomethingHappened` class in the *basic example* below. Aggregate event classes are usually named using past participles to describe what was decided by the command method, such as "Done", "Updated", "Closed", etc. See the *Domain events* section below for more information about aggregate event classes. They can be defined on aggregate classes as nested classes.

The *trigger_event()* method also accepts arbitrary keyword-only arguments, which will be used to construct the aggregate event object. As with the *_create()* method described above, the event object will be constructed with these arguments, and so any extra arguments must be matched by the expected values of the event class. For example `what: str` on the `SomethingHappened` event class in the *basic example* below matches the `what=what` keyword argument passed in the call to the *trigger_event()* method in the `make_it_so()` command.

The `version` will be incremented by 1 for each event that is triggered.

```
assert aggregate.version == 2
```

After triggering a second event, the modified time will be greater than the created time.

```
assert aggregate.modified_on > aggregate.created_on
```

Collecting pending events

Thirdly, the *Aggregate* class has a "public" object method *collect_events()* which can be called to collect the aggregate events that have been created but since either the last call to this method or since the aggregate object was constructed. This method is called without any arguments.

```
from eventsourcing.domain import AggregateCreated

pending_events = aggregate.collect_events()

assert len(pending_events) == 2

assert isinstance(pending_events[0], AggregateCreated)
assert pending_events[0].originator_id == aggregate.id
assert pending_events[0].originator_version == 1
assert pending_events[0].timestamp == aggregate.created_on

assert isinstance(pending_events[1], AggregateEvent)
assert pending_events[1].originator_id == aggregate.id
assert pending_events[1].originator_version == 2
assert pending_events[1].timestamp == aggregate.modified_on
```

1.4.3 Basic example

In the example below, the `World` aggregate is a subclass of the library's base *Aggregate* class. The `__init__()` method extends the super class method and initialises a `history` attribute with an empty Python `list` object.

The `create()` method is a class method that creates and returns a new `World` aggregate object. It calls the base class *_create()* method. It uses its `Created` event class as the value of the `event_class` argument. It uses a version 4 UUID object as the value of the `id` argument. (See the *Namespaced IDs* section below for a discussion about using version 5 UUIDs.)

The `make_it_so()` method is a command method that triggers a `World.SomethingHappened` domain event. It calls the base class *trigger_event()* method. The event is triggered with the method argument `what`.

```
from eventsourcing.domain import Aggregate


class World(Aggregate):
    def __init__(self):
        self.history = []

    @classmethod
    def create(cls):
        return cls._create(cls.Created, id=uuid4())

    class Created(AggregateCreated):
        pass

    def make_it_so(self, what):
        self.trigger_event(self.SomethingHappened, what=what)
```

(continues on next page)

(continued from previous page)

```
    class SomethingHappened(AggregateEvent):
        what: str

        def apply(self, world):
            world.history.append(self.what)
```

The nested `Created` class is defined as a subclass of the base aggregate *`Created`* class. Although in this simple example this `World.Created` event class carries no more attributes than the base class event that it inherits, it's always worth defining all event classes on the concrete aggregate class itself in case these classes need to be modified so that old instances can be upcast to new versions (see *Versioning*). The name of an event class should express your project's ubiquitous language, take the grammatical form of a past participle (either regular or irregular), and describe the type of decision represented by the event class.

The nested `SomethingHappened` class is a frozen data class that extends the base aggregate event class `Aggregate.Event` (also a frozen data class) with a field `what` which is defined as a Python `str`. An `apply()` method is defined which appends the `what` value to the aggregate's `history`. This method is called when the event is triggered (see *Domain events*).

By defining the event class under the command method which triggers it, and then defining an apply() method as part of the event class definition, the story of calling a command method, triggering an event, and evolving the state of the aggregate is expressed neatly in three parts.

Having defined the `World` aggregate class, we can create a new `World` aggregate object by calling the `World.create()` class method.

```
world = World.create()

assert isinstance(world, World)
```

The aggregate's attributes `created_on` and `modified_on` show when the aggregate was created and when it was modified. Since there has only been one domain event, these are initially equal. The values of these attributes are timezone-aware Python `datetime` objects. These values follow from the `timestamp` values of the domain event objects, and represent when the aggregate's first and last domain events were created. The timestamps have no consequences for the operation of the library, and are included to give a general indication to humans of when the domain events occurred.

```
from datetime import datetime

assert world.created_on == world.modified_on
assert isinstance(world.created_on, datetime)
```

We can call the aggregate object methods. The `World` aggregate has a command method `make_it_so()` which triggers the `SomethingHappened` event. The `apply()` method of the `SomethingHappened` class appends the `what` of the event to the `history` of the `world`. So when we call the `make_it_so()` command, the argument `what` will be appended to the `history`.

```
# Commands methods trigger events.
world.make_it_so("dinosaurs")
world.make_it_so("trucks")
world.make_it_so("internet")

# State of aggregate object has changed.
assert world.history[0] == "dinosaurs"
assert world.history[1] == "trucks"
assert world.history[2] == "internet"
```

Now that more than one domain event has been created, the aggregate's `modified_on` value is greater than its `created_on` value.

```
assert world.modified_on > world.created_on
```

The resulting domain events are now held internally in the aggregate in a list of pending events, in the `pending_events` attribute. The pending events can be collected by calling the aggregate's *`collect_events()`* method. These events are pending to be saved, and indeed the library's *application* object has a *`save()`* method which works by calling this method. So far, we have created four domain events and we have not yet collected them, and so there will be four pending events: one `Created` event, and three `SomethingHappened` events.

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

# Collect pending events.
pending_events = world.collect_events()
assert len(pending_events) == 4
assert len(world.pending_events) == 0

assert isinstance(pending_events[0], World.Created)
assert isinstance(pending_events[1], World.SomethingHappened)
assert isinstance(pending_events[2], World.SomethingHappened)
assert isinstance(pending_events[3], World.SomethingHappened)
assert pending_events[1].what == "dinosaurs"
assert pending_events[2].what == "trucks"
assert pending_events[3].what == "internet"

assert pending_events[0].timestamp == world.created_on
assert pending_events[3].timestamp == world.modified_on
```

1.4.4 Domain events

Domain events are created but do not change. They are uniquely identifiable in a domain model by a aggregate ID which identifies the sequence to which they belong and and a version number which determines their position in that sequence.

The library's *`DomainEvent`* class is a base class for domain events. It is defined as a frozen data class with an `originator_id` attribute which is a Python `UUID` that holds an aggregate ID and identifies the sequence to which a domain event object belongs, an `originator_version` attribute which is a Python `int` that holds the version of an aggregate and determines the position of a domain event object in its sequence, and a `timestamp` attribute which is a Python `datetime` that represents when the event was created.

The timestamps have no consequences for the operation of the library. The aggregate events objects are ordered in their sequence by their version numbers, and not by their timestamps. The timestamps exist only to give a general indication to humans of when things occurred.

The library's *`DomainEvent`* class is used (inherited) by the aggregate *`Event`* class. The library's *`Snapshot`* class also inherits from the *`DomainEvent`* class — see *Snapshots* for more information about snapshots. The aggregate *`Event`* class is defined as a subclass of the domain event base class *`DomainEvent`*. Aggregate event objects represent original decisions by a domain model that advance the state of an application.

The aggregate *`Event`* class has a method `mutate()` which adjusts the state of an aggregate. It has an optional argument `aggregate` which is used to pass the aggregate object to which the domain event object pertains into the method when it is called. It returns an optional `aggregate` object, and the return value can be passed in when calling this method on another event object. An initial "created" event can construct an aggregate object, a subsequent event can receive and return an aggregate, and a final "discarded" event can receive an aggregate and return `None`. The

`mutate()` methods of a sequence of aggregate events can be used to reconstruct a copy of the original aggregate object. And indeed the *application repository* object has a *`get()`* method which works by calling these methods.

```
copy = None
for domain_event in pending_events:
    copy = domain_event.mutate(copy)

assert isinstance(copy, World)
assert copy.id == world.id
assert copy.version == world.version
assert copy.created_on == world.created_on
assert copy.modified_on == world.modified_on
assert copy.history == world.history
```

The aggregate *`Event`* class has a method `apply()`. Like the `mutate()` method, it also has an argument `aggregate` which is used to pass the aggregate object to which the domain event object pertains into the method when it is called. The `mutate()` method calls the event's `apply()` method before it returns. The base class `apply()` method body is empty, and so this method can be simply overridden (implemented without a call to the superclass method). It is also not expected to return a value (any value that it does return will be ignored). Hence this method can be simply and conveniently implemented in aggregate event classes to apply the event attribute values to the aggregate.

The `mutate()` and `apply()` methods of aggregate events effectively implement the "aggregate projection", which means the function by which the events are processed to reconstruct the state of the aggregate. An alternative to use `apply()` methods on the event classes is to define apply methods on the aggregate class. A base `Event` class can be defined on the aggregate class which simply calls an `apply()` method on the aggregate class. This aggregate `apply()` method can be decorated with the `@singledispatchmethod` decorator, and then event-specific methods can be defined and registered that will apply the events to the aggregate. See the `Cargo` aggregate of the *Cargo Shipping example* for details. A further alternative is to use the *declarative syntax*.

The aggregate *`Created`* class represents the creation of an aggregate object instance. It is defined as a frozen data class that extends the base class *`Event`* with its attribute `originator_topic` which is Python `str`. The value of this attribute will be a *topic* that describes the path to the aggregate instance's class. It has a `mutate()` method which constructs an aggregate object after resolving the `originator_topic` value to an aggregate class. It does not call `apply()` since the aggregate class `__init__()` method receives the "created" event attribute values and can fully initialise the aggregate object.

Domain event objects are usually created by aggregate methods, as part of a sequence that determines the state of an aggregate. The attribute values of new event objects are decided by these methods before the event is created. For example, the aggregate's *`_create()`* method uses the given value of its `id` argument as the new event's `originator_id`. It sets the `originator_version` to the value of `1`. It derives the `originator_topic` value from the aggregate class. And it calls Python's `datetime.now()` to create the `timestamp` value.

Similarly, the aggregate *`trigger_event()`* method uses the `id` attribute of the aggregate as the `originator_id` of the new domain event. It uses the current aggregate `version` to create the next version number (by adding `1`) and uses this value as the `originator_version` of the new domain event. It calls `datetime.now()` to create the `timestamp` value of the new domain event.

The timestamp values are "timezone aware" datetime objects. The default timezone is UTC, as defined by Python's `datetime.timezone.utc`. It is recommended to store date-times as UTC values, and convert to a local timezone in the interface layer according to the particular timezone of a particular user. However, if necessary, this default can be changed either by assigning a `datetime.tzinfo` object to the `TZINFO` attribute of the *`eventsourcing.domain`* module. The `eventsourcing.domain.TZINFO` value can also be configured using environment variables, by setting the environment variable `TZINFO_TOPIC` to a string that describes the *topic* of a Python `datetime.tzinfo` object (for example `'datetime:timezone.utc'`).

1.4.5 Snapshots

Snapshots speed up aggregate access time, by avoiding the need to retrieve and apply all the domain events when reconstructing an aggregate object instance. The library's *Snapshot* class can be used to create and restore snapshots of aggregate object instances. See *Snapshotting* in the application module documentation for more information about taking snapshots in an event-sourced application.

The *Snapshot* class is defined as a subclass of the domain event base class *DomainEvent*. It is defined as a frozen data class and extends the base class with attributes `topic` and `state`, which hold the topic of an aggregate object class and the current state of an aggregate object.

```
from eventsourcing.domain import Snapshot
```

The class method *take()* can be used to create a snapshot of an aggregate object.

```
snapshot = Snapshot.take(world)

assert isinstance(snapshot, Snapshot)
assert snapshot.originator_id == world.id
assert snapshot.originator_version == world.version
assert snapshot.topic == "__main__:World", snapshot.topic
assert snapshot.state["history"] == world.history
assert snapshot.state["_created_on"] == world.created_on
assert snapshot.state["_modified_on"] == world.modified_on
assert len(snapshot.state) == 3
```

A snapshot's *mutate()* method can be used to reconstruct its aggregate object instance.

```
copy = snapshot.mutate(None)

assert isinstance(copy, World)
assert copy.id == world.id
assert copy.version == world.version
assert copy.created_on == world.created_on
assert copy.modified_on == world.modified_on
assert copy.history == world.history
```

The signature of the *mutate()* method is the same as the domain event object method of the same name, so that when reconstructing an aggregate, a list that starts with a snapshot and continues with the subsequent domain event objects can be treated in the same way as a list of all the domain event objects of an aggregate. This similarity is needed by the application *repository*, since some specialist event stores (e.g. AxonDB) return a snapshot as the first domain event.

1.4.6 Initial Version Number

By default, the aggregates have an initial version number of `1`. Sometimes it may be desired, or indeed necessary, to use a different initial version number.

In the example below, the initial version number of the class `MyAggregate` is defined to be `0`.

```
class MyAggregate(Aggregate):
    INITIAL_VERSION = 0

aggregate = MyAggregate()
assert aggregate.version == 0
```

If all aggregates in a domain model need to use the same non-default version number, then a base class can be defined and used by the aggregates of the domain model on which `INITIAL_VERSION` is set to the preferred value. Some people may wish to set the preferred value on the library *`Aggregate`* class.

1.4.7 Versioning

Versioning allows aggregate and domain event classes to be modified after an application has been deployed.

On both aggregate and domain event classes, the class attribute `class_version` can be used to indicate the version of the class. This attribute is inferred to have a default value of `1`. If the data model is changed, by adding or removing or renaming or changing the meaning of values of attributes, subsequent versions should be given a successively higher number than the previously deployed version. Static methods of the form `upcast_vX_vY()` will be called to update the state of a stored event or snapshot from a lower version `X` to the next higher version `Y`. Such upcast methods will be called to upcast the state from the version of the class with which it was created to the version of the class which will be reconstructed. For example, upcasting the stored state of an object created at version `2` of a class that will be used to reconstruct an object at version `4` of the class will involve calling upcast methods `upcast_v2_v3()`, and `upcast_v3_v4()`. If you aren't using snapshots, you don't need to define upcast methods or version numbers on the aggregate class.

In the example below, version `1` of the class `MyAggregate` is defined with an attribute `a`.

```
class MyAggregate(Aggregate):
    def __init__(self, a:str):
        self.a = a

    @classmethod
    def create(cls, a:str):
        return cls._create(cls.Created, id=uuid4(), a=a)

    class Created(Aggregate.Created):
        a: str
```

After an application that uses the above aggregate class has been deployed, its `Created` events will have been created and stored with the `a` attribute defined. If subsequently the attribute `b` is added to the definition of the `Created` event, in order for the existing stored events to be constructed in a way that satisfies the new version of the class, the stored events will need to be upcast to have a value for `b`. In the example below, the static method `upcast_v1_v2()` defined on the `Created` event sets a default value for `b` in the given `state`. The class attribute `class_version` is set to `2`. The same treatment is given to the aggregate class as the domain event class, so that snapshots can be upcast.

```
class MyAggregate(Aggregate):
    def __init__(self, a:str, b:int):
        self.a = a
        self.b = b

    @classmethod
    def create(cls, a:str, b: int = 0):
        return cls._create(cls.Created, id=uuid4(), a=a, b=b)

    class Created(Aggregate.Created):
        a: str
        b: int

        class_version = 2

        @staticmethod
```

(continues on next page)

(continued from previous page)

```
        def upcast_v1_v2(state):
            state["b"] = 0

    class_version = 2

    @staticmethod
    def upcast_v1_v2(state):
        state["b"] = 0
```

After an application that uses the above version 2 aggregate class has been deployed, its `Created` events will have be created and stored with both the `a` and `b` attributes. If subsequently the attribute `c` is added to the definition of the `Created` event, in order for the existing stored events from version 1 to be constructed in a way that satisfies the new version of the class, they will need to be upcast to include a value for `b` and `c`. The existing stored events from version 2 will need to be upcast to include a value for `c`. The additional static method `upcast_v2_v3()` defined on the `Created` event sets a default value for `c` in the given `state`. The class attribute `class_version` is set to `3`. The same treatment is given to the aggregate class as the domain event class, so that any snapshots will be upcast.

```
class MyAggregate(Aggregate):
    def __init__(self, a:str, b:int, c:float):
        self.a = a
        self.b = b
        self.c = c

    @classmethod
    def create(cls, a:str, b: int = 0, c: float = 0.0):
        return cls._create(cls.Created, id=uuid4(), a=a, b=b, c=c)

    class Created(Aggregate.Created):
        a: str
        b: int
        c: float

        class_version = 3

        @staticmethod
        def upcast_v1_v2(state):
            state["b"] = 0

        @staticmethod
        def upcast_v2_v3(state):
            state["c"] = 0.0

    class_version = 3

    @staticmethod
    def upcast_v1_v2(state):
        state["b"] = 0

    @staticmethod
    def upcast_v2_v3(state):
        state["c"] = 0.0
```

If subsequently a new event is added that manipulates a new attribute that is expected to be initialised when the aggregate is created, in order that snapshots from earlier version will be upcast, the aggregate class attribute `class_version` will need to be set to `4` and a static method `upcast_v3_v4()` defined on the aggregate class which upcasts the state of a previously created snapshot. In the example below, the new attribute `d` is initialised in

the `__init__()` method, and a domain event which updates `d` is defined. Since the `Created` event class has not changed, it remains at version `3`.

```
class MyAggregate(Aggregate):
    def __init__(self, a:str, b:int, c:float):
        self.a = a
        self.b = b
        self.c = c
        self.d = False

    @classmethod
    def create(cls, a:str, b: int = 0, c: float = 0.0):
        return cls._create(cls.Created, id=uuid4(), a=a, b=b, c=c)

    class Created(Aggregate.Created):
        a: str
        b: int
        c: float

        class_version = 3

        @staticmethod
        def upcast_v1_v2(state):
            state["b"] = 0

        @staticmethod
        def upcast_v2_v3(state):
            state["c"] = 0.0

    def set_d(self, d: bool):
        self.trigger_event(self.DUpdated, d=d)

    class DUpdated(AggregateEvent):
        d: bool

        def apply(self, aggregate: "Aggregate") -> None:
            aggregate.d = self.d

    class_version = 4

    @staticmethod
    def upcast_v1_v2(state):
        state["b"] = 0

    @staticmethod
    def upcast_v2_v3(state):
        state["c"] = 0.0

    @staticmethod
    def upcast_v3_v4(state):
        state["d"] = False
```

If the value objects used by your events also change, you may also need to define new transcodings with new names. Simply register the new transcodings after the old, and use a modified `name` value for the transcoding. In this way, the existing encoded values will be decoded by the old transcoding, and the new instances of the value object class will be encoded with the new version of the transcoding.

In order to support forward compatibility as well as backward compatibility, so that consumers designed for old versions will not be broken by modifications, it is advisable to restrict changes to existing types to be additions only,

so that existing attributes are unchanged. If existing aspects need to be changed, for example by renaming or removing an attribute of an event, then it is advisable to define a new type. This approach depends on consumers overlooking or ignoring new attribute and new types, but they may effectively be broken anyway by such changes if they no longer see any data.

Including model changes in the domain events may help to inform consumers of changes to the model schema, and may allow the domain model itself to be validated, so that classes are marked with new versions if the attributes have changed. This may be addressed by a future version of this library. Considering model code changes as a sequence of immutable events brings the state of the domain model code itself into the same form of event-oriented consideration as the consideration of the state an application as a sequence of events.

1.4.8 Namespaced IDs

Aggregates can be created with version 5 UUIDs so that their IDs can be generated from a given name in a namespace. They can be used for example to create IDs for aggregates with fixed names that you want to identify by name. For example, you can use this technique to identify a system configuration object. This technique can also be used to identify index aggregates that hold the IDs of aggregates with mutable names, or used to index other mutable attributes of an event sourced aggregate. It isn't possible to change the ID of an existing aggregate, because the domain events will need to be stored together in a single sequence. And so, using an index aggregate that has an ID that can be recreated from a particular value of a mutable attribute of another aggregate to hold the ID of that aggregate with makes it possible to identify that aggregate from that particular value. Such index aggregates can be updated when the mutable attribute changes, or not.

For example, if you have a collection of page aggregates with names that might change, and you want to be able to identify the pages by name, then you can create index aggregates with version 5 UUIDs that are generated from the names, and put the IDs of the page aggregates in the index aggregates. The aggregate classes `Page` and `Index` in the example code below show how this can be done.

If we imagine we can save these page and index aggregates and retrieve them by ID, we can imagine retrieving a page aggregate using its name by firstly recreating an index ID from the page name, retrieving the index aggregate using that ID, getting the page ID from the index aggregate, and then using that ID to retrieve the page aggregate. When the name is changed, a new index aggregate can be saved along with the page, so that later the page aggregate can be retrieved using the new name. See the discussion about *saving multiple aggregates* to see an example of how this can work.

```
from uuid import NAMESPACE_URL, uuid5, UUID
from typing import Optional

from eventsourcing.domain import Aggregate


class Page(Aggregate):
    def __init__(self, name: str, body: str):
        self.name = name
        self.body = body

    @classmethod
    def create(cls, name: str, body: str = ""):
        return cls._create(
            id=uuid4(),
            event_class=cls.Created,
            name=name,
            body=body
        )

    class Created(AggregateCreated):
```

(continues on next page)

(continued from previous page)

```
        name: str
        body: str

    def update_name(self, name: str):
        self.trigger_event(self.NameUpdated, name=name)

    class NameUpdated(AggregateEvent):
        name: str

        def apply(self, page: "Page"):
            page.name = self.name


class Index(Aggregate):
    def __init__(self, name: str, ref: UUID):
        self.name = name
        self.ref = ref

    @classmethod
    def create(cls, name: str, ref: UUID):
        return cls._create(
            event_class=cls.Created,
            id=cls.create_id(page.name),
            name=page.name,
            ref=page.id
        )

    @staticmethod
    def create_id(name: str):
        return uuid5(NAMESPACE_URL, f"/pages/{name}")

    class Created(AggregateCreated):
        name: str
        ref: UUID

    def update_ref(self, ref):
        self.trigger_event(self.RefUpdated, ref=ref)

    class RefUpdated(AggregateEvent):
        ref: Optional[UUID]

        def apply(self, index: "Index"):
            index.ref = self.ref
```

We can use the classes above to create a "page" aggregate with a name that we will then change. We can at the same time create an index object for the page.

```
page = Page.create(name="Erth")
index1 = Index.create(page.name, page.id)
```

Let's imagine these two aggregate are saved together, and having been saved can be retrieved by ID. See the discussion about *saving multiple aggregates* to see how this works in an application object.

We can use the page name to recreate the index ID, and use the index ID to retrieve the index aggregate. We can then obtain the page ID from the index aggregate, and then use the page ID to get the page aggregate.

```
index_id = Index.create_id("Erth")
assert index_id == index1.id
assert index1.ref == page.id
```

Now let's imagine we want to correct the name of the page. We can update the name of the page, and create another index aggregate for the new name, so that later we can retrieve the page using its new name.

```
page.update_name("Earth")
index2 = Index.create(page.name, page.id)
```

We can drop the reference from the old index, so that it can be used to refer to a different page.

We can now use the new name to get the ID of the second index aggregate, and imagine using the second index aggregate to get the ID of the page.

```
index_id = Index.create_id("Earth")
assert index_id == index2.id
assert index2.ref == page.id
```

Saving and retrieving aggregates by ID is demonstrated in the discussion about *saving multiple aggregates* in the *applications* documentation.

1.4.9 Declarative syntax

You may have noticed a certain amount of repetition in the definitions of the aggregates above. In several places, the same argument was defined in a command method, on an event class, and in an apply method. The library offers a more concise way to express aggregates by using a declarative syntax.

Create new aggregate by calling the aggregate class directly

A new event sourced aggregate can be created by calling the aggregate class directly. You don't actually need to define a class method to do this, although you may wish to express your project's ubiquitous language by doing so.

Calling the aggregate class directly will firstly create a created event (an instance of the aggregate's created event class) and use that event object to construct an instance of the aggregate class.

```
class MyAggregate(Aggregate):
    class Created(Aggregate.Created):
        pass


# Call the class directly.
agg = MyAggregate()

# There is one pending event.
pending_events = agg.collect_events()
assert len(pending_events) == 1
assert isinstance(pending_events[0], MyAggregate.Created)

# The pending event can be used to reconstruct the aggregate.
copy = pending_events[0].mutate(None)
assert copy.id == agg.id
assert copy.created_on == agg.created_on
```

Using the init method to define the created event class

If a created event class is not defined on an aggregate class, one will be automatically defined. The attributes of this event class will be derived by inspecting the signature of the `__init__()` method. The example below has an init method that has a `name` argument. Because this example doesn't have a created event class defined explicitly on the aggregate class, a created event class will be defined automatically to match the signature of the init method. That is, a created event class will be defined that has an attribute `name`.

```
class MyAggregate(Aggregate):
    def __init__(self, name):
        self.name = name


# Call the class with a 'name' argument.
agg = MyAggregate(name="foo")
assert agg.name == "foo"

# There is one pending event.
pending_events = agg.collect_events()
assert len(pending_events) == 1

# The pending event is a created event.
assert isinstance(pending_events[0], MyAggregate.Created)

# The created event has a 'name' attribute.
pending_events[0].name == "foo"

# The created event can be used to reconstruct the aggregate.
copy = pending_events[0].mutate(None)
assert copy.name == agg.name
```

Dataclass-style init methods

Python's dataclass annotations can be used to define an aggregate's `__init__()` method. A created event class can be automatically defined from this method.

```
from dataclasses import dataclass

@dataclass
class MyAggregate(Aggregate):
    name: str


# Create a new aggregate.
agg = MyAggregate(name="foo")

# The aggregate has a 'name' attribute
assert agg.name == "foo"

# The created event has a 'name' attribute.
pending_events = agg.collect_events()
pending_events[0].name == "foo"
```

Optional arguments can be defined by providing default values on the dataclass attribute definitions.

```
from dataclasses import dataclass

@dataclass
class MyAggregate(Aggregate):
    name: str = "bar"


# Call the class without a name.
agg = MyAggregate()
assert agg.name == "bar"

# Call the class with a name.
agg = MyAggregate("foo")
assert agg.name == "foo"
```

Anything that works on a dataclass should work here too. For example, you can define non-init argument attributes by using the `field` feature of the dataclasses module.

```
from dataclasses import field
from typing import List

@dataclass
class MyAggregate(Aggregate):
    history: List[str] = field(default_factory=list, init=False)


# Create a new aggregate.
agg = MyAggregate()

# The aggregate has a list.
assert agg.history == []
```

Please note, when using the dataclass-style for defining `__init__()` methods, using the `@dataclass` decorator will inform your IDE of the method signature. The annotations will in any case be used to create an `__init__()` method when the class does not already have an `__init__()`. Using the dataclass decorator merely enables code completion and syntax checking, but the code will run just the same with or without the `@dataclass` decorator being applied to aggregate classes that are defined using this style.

Declaring the created event class name

To give the created event class a particular name, use the class argument 'created_event_name'.

```
class MyAggregate(Aggregate, created_event_name="Started"):
    name: str

# Create a new aggregate.
agg = MyAggregate("foo")

# The created event class is called "Started".
pending_events = agg.collect_events()
assert isinstance(pending_events[0], MyAggregate.Started)
```

This is equivalent to declaring the created event class explicitly on the aggregate class using a particular name.

```
class MyAggregate(Aggregate):
    class Started(Aggregate.Created):
```

(continues on next page)

(continued from previous page)

```
        pass

# Create a new aggregate.
agg = MyAggregate()

# The created event class is called "Started".
pending_events = agg.collect_events()
assert isinstance(pending_events[0], MyAggregate.Started)
```

If more than one created event class is defined on the aggregate class, perhaps because the name of the created event class was changed and there are stored events that were created using the old created event class that still need to be supported, the `created_event_name` class argument can be used to identify which created event class is the one to use when creating new aggregate instances. This can be combined with upcasting old events, discussed above.

```
class MyAggregate(Aggregate, created_event_name="Started"):
    class Created(Aggregate.Created):
        pass

    class Started(Aggregate.Created):
        pass


# Create a new aggregate.
agg = MyAggregate()

# The created event class is called "Started".
pending_events = agg.collect_events()
assert isinstance(pending_events[0], MyAggregate.Started)
```

If the `created_event_name` argument is used but the value does not match the name of one the created event classes that are explicitly defined on the aggregate class, then an event class will be automatically defined, and it will be used when creating new aggregate instances.

```
class MyAggregate(Aggregate, created_event_name="Opened"):
    class Created(Aggregate.Created):
        pass

    class Started(Aggregate.Created):
        pass


# Create a new aggregate.
agg = MyAggregate()

# The created event class is called "Opened".
pending_events = agg.collect_events()
assert isinstance(pending_events[0], MyAggregate.Opened)
```

Defining the aggregate ID

By default, the aggregate ID will be a version 4 UUID, automatically generated when a new aggregate is created. However, the aggregate ID can also be defined as a function of the arguments used to create the aggregate. You can do this by defining a `create_id()` method.

```
class MyAggregate(Aggregate):
    name: str

    @staticmethod
    def create_id(name: str):
        return uuid5(NAMESPACE_URL, f"/my_aggregates/{name}")


# Create a new aggregate.
agg = MyAggregate(name="foo")
assert agg.name == "foo"

# The aggregate ID is a version 5 UUID.
assert agg.id == MyAggregate.create_id("foo")
```

If a `create_id()` method is defined on the aggregate class, the base class method *`create_id()`* will be overridden. The arguments used in this method must be a subset of the arguments used to create the aggregate. The base class method simply returns a version 4 UUID, which is the default behaviour for generating aggregate IDs.

Alternatively, an 'id' attribute can be declared on the aggregate class, and an ID supplied directly when creating new aggregates.

```
def create_id(name: str):
    return uuid5(NAMESPACE_URL, f"/my_aggregates/{name}")

class MyAggregate(Aggregate):
    id: UUID


# Create an ID.
agg_id = create_id(name="foo")

# Create an aggregate with the ID.
agg = MyAggregate(id=agg_id)
assert agg.id == agg_id
```

When defining an explicit `__init__()` method, the `id` argument can be set on the object as `self._id`. Assigning to `self.id` won't work because `id` is defined as a read-only property on the base aggregate class.

```
class MyAggregate(Aggregate):
    def __init__(self, id: UUID):
        self._id = id


# Create an aggregate with the ID.
agg = MyAggregate(id=agg_id)
assert agg.id == agg_id
```

The `@event` decorator

A more concise way of expressing the concerns around defining, triggering and applying subsequent aggregate events can be achieved by using the library function *`event()`* to decorate aggregate command methods.

When decorating a method with the `@event` decorator, the method signature will be used to automatically define an aggregate event class. And when the method is called, the event will firstly be triggered with the values given when calling the method, so that an event is created and used to mutate the state of the aggregate. The body of the

decorated method will be used as the `apply()` method of the event both after the event has been triggered and when the aggregate is reconstructed from stored events. The name of the event class can be passed to the decorator.

```
from eventsourcing.domain import event

class MyAggregate(Aggregate):
    name: str

    @event("NameUpdated")
    def update_name(self, name):
        self.name = name


# Create an aggregate.
agg = MyAggregate(name="foo")
assert agg.name == "foo"

# Update the name.
agg.update_name("bar")
assert agg.name == "bar"

# There are two pending events.
pending_events = agg.collect_events()
assert len(pending_events) == 2
assert pending_events[0].name == "foo"

# The second pending event is a 'NameUpdated' event.
assert isinstance(pending_events[1], MyAggregate.NameUpdated)

# The second pending event has a 'name' attribute.
assert pending_events[1].name == "bar"
```

Inferring the event class name from the method name

The `@event` decorator can be used without providing the name of an event. If the decorator is used without any arguments, the name of the event will be derived from the method name. The method name is assumed to be lower case and underscore-separated. The name of the event class is constructed by firstly splitting the name of the method by its underscore characters, then by capitalising the resulting parts, and then by concatenating the capitalised parts to give an "upper camel case" class name. For example, a method name `name_updated` would give an event class name `NameUpdated`.

```
from eventsourcing.domain import event

class MyAggregate(Aggregate):
    name: str

    @event
    def name_updated(self, name):
        self.name = name


# Create an aggregate.
agg = MyAggregate(name="foo")
assert agg.name == "foo"

# Update the name.
```

(continues on next page)

(continued from previous page)

```
agg.name_updated("bar")
assert agg.name == "bar"

# There are two pending events.
pending_events = agg.collect_events()
assert len(pending_events) == 2
assert pending_events[0].name == "foo"

# The second pending event is a 'NameUpdated' event.
assert isinstance(pending_events[1], MyAggregate.NameUpdated)

# The second pending event has a 'name' attribute.
assert pending_events[1].name == "bar"
```

However, this creates a slight tension in the naming conventions because methods should normally be named using the imperative form and event names should normally be past participles. However, this can be useful when naming methods that will be only called by aggregate command methods under certain conditions.

For example, if an attempt is made to update the value of an attribute, but the given value happens to be identical to the existing value, then it might be desirable to skip on having an event triggered.

```
class MyAggregate(Aggregate):
    name: str

    def update_name(self, name):
        if name != self.name:
            self.name_updated(name)

    @event
    def name_updated(self, name):
        self.name = name

# Create an aggregate.
agg = MyAggregate(name="foo")
assert agg.name == "foo"

# Update the name lots of times.
agg.update_name("foo")
agg.update_name("foo")
agg.update_name("foo")
agg.update_name("bar")
agg.update_name("bar")
agg.update_name("bar")
agg.update_name("bar")

# There are two pending events (not eight).
pending_events = agg.collect_events()
assert len(pending_events) == 2, len(pending_events)
```

The World aggregate class revisited

Using the declarative syntax described above, the `World` aggregate in the *basic example* above can be expressed more concisely in the following way.

In the example below, the `World` aggregate's created event is automatically defined by inspecting the aggregate's `__init__()` method. The created event is named `Created`. The `World.SomethingHappened` event is auto-

matically defined by inspecting the decorated `make_it_so()` method. The event class name "SomethingHappened" is given to the event decorator. The body of the decorated `make_it_so()` method will be used as the `apply()` method of the `World.SomethingHappened` event, both when the event is triggered and when the aggregate is reconstructed from stored events.

```
from eventsourcing.domain import event


class World(Aggregate):
    def __init__(self):
        self.history = []

    @event("SomethingHappened")
    def make_it_so(self, what):
        self.history.append(what)
```

The `World` aggregate class can be called directly. Calling the class directly will call the *`Aggregate._create()`* method with the automatically defined `World.Created` event. Calling the `make_it_so()` method will trigger a `World.SomethingHappened` event, and this event will be used to mutate the state of the aggregate, such that the `make_it_so()` method argument `what` will eventually be appended to the aggregate's `history` attribute.

```
world = World()
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"
assert len(world.collect_events()) == 4
```

The Page and Index aggregates revisited

The `Page` and `Index` aggregates defined in the above *discussion about namespaced IDs* can be expressed more concisely in the following way.

```
from dataclasses import dataclass


@dataclass
class Page(Aggregate):
    name: str
    body: str = ""

    @event("NameUpdated")
    def update_name(self, name: str):
        self.name = name


@dataclass
class Index(Aggregate):
    name: str
    ref: Optional[UUID]

    @staticmethod
    def create_id(name: str):
```

(continues on next page)

(continued from previous page)

```
        return uuid5(NAMESPACE_URL, f"/pages/{name}")

    @event("RefUpdated")
    def update_ref(self, ref: Optional[UUID]):
        self.ref = ref


# Create new page and index aggregates.
page = Page(name="Erth")
index1 = Index(name=page.name, ref=page.id)

# The page name can be used to recreate
# the index ID. The index ID can be used
# to retrieve the index aggregate, which
# gives the page ID, and then the page ID
# can be used to retrive the page aggregate.
index_id = Index.create_id(name="Erth")
assert index_id == index1.id
assert index1.ref == page.id
assert index1.name == page.name

# Later, the page name can be updated,
# and a new index created for the page.
page.update_name(name="Earth")
index1.update_ref(ref=None)
index2 = Index(name=page.name, ref=page.id)

# The new page name can be used to recreate
# the new index ID. The new index ID can be
# used to retrieve the new index aggregate,
# which gives the page ID, and then the page
# ID can be used to retrieve the renamed page.
index_id = Index.create_id(name="Earth")
assert index_id == index2.id
assert index2.ref == page.id
assert index2.name == page.name
```

Non-trivial command methods

Tn the examples above, the work of the command methods is "trivial", in that the command method arguments are always used directly as the aggregate event attribute values. But often a command method needs to do some work before triggering an event. The event attributes may not be the same as the command method arguments. The logic of the command may be such that under some conditions an event should not be triggered.

As a final example, consider the following `Order` class. It is an ordinary Python object class. Its `__init__()` method takes a `name` argument. The method `confirm()` sets the attribute `confirmed_at`. The method `pickup()` checks that the order has been confirmed before calling the `_pickup()` method which sets the attribute `pickedup_at`. If the order has not been confirmed, an exception will be raised. That is, whilst the `confirm()` command method is trivial in that its arguments are always used as the event attributes, the `pickup()` method is non-trivial in that it will only trigger an event if the order has been confirmed. That means we can't decorate the `pickup()` method with the `@event` decorator without triggering an unwanted event.

```
class Order:
    def __init__(self, name):
        self.name = name
```

(continues on next page)

(continued from previous page)

```
        self.confirmed_at = None
        self.pickedup_at = None

    def confirm(self, at):
        self.confirmed_at = at

    def pickup(self, at):
        if self.confirmed_at:
            self._pickup(at)
        else:
            raise RuntimeError("Order is not confirmed")

    def _pickup(self, at):
        self.pickedup_at = at
```

This ordinary Python class can used in the usual way. We can construct a new instance of the class, and call its command methods.

```
# Start a new order, confirm, and pick up.
order = Order("my order")

try:
    order.pickup(datetime.now())
except RuntimeError:
    pass
else:
    raise AssertionError("shouldn't get here")

order.confirm(datetime.now())
order.pickup(datetime.now())
```

This ordinary Python class can be easily converted into an event sourced aggregate by applying the library's `@event` decorator to the `confirm()` and `_pickup()` methods.

Because the command methods are decorated in this way, when the `confirm()` method is called, an `Order.Confirmed` event will be triggered. When the `_pickup()` method is called, an `Order.PickedUp` event will be triggered. Those event classes are defined automatically from the method signatures. The decorating of the `_pickup()` method and not of the `pickup()` method is a good example of a command method that needs to do some work before an event is triggered. The body of the `pickup()` method is only executed when the command method is called, whereas the body of the `_pickup()` method is executed each time the event is applied to evolve the state of the aggregate.

```
class Order(Aggregate):
    def __init__(self, name):
        self.name = name
        self.confirmed_at = None
        self.pickedup_at = None

    @event("Confirmed")
    def confirm(self, at):
        self.confirmed_at = at

    def pickup(self, at):
        if self.confirmed_at:
            self._pickup(at)
        else:
```

(continues on next page)

(continued from previous page)

```
            raise RuntimeError("Order is not confirmed")

    @event("PickedUp")
    def _pickup(self, at):
        self.pickedup_at = at
```

We can use the event sourced `Order` aggregate in the same way as the undecorated ordinary Python `Order` class. The event sourced version has the advantage that using it will trigger a sequence of aggregate events that can be persisted in a database and used in future to determine the state of the order.

```
order = Order("my order")
order.confirm(datetime.now())
order.pickup(datetime.now())

# Check the state of the order.
assert order.name == "my order"
assert isinstance(order.confirmed_at, datetime)
assert isinstance(order.pickedup_at, datetime)
assert order.pickedup_at > order.confirmed_at

# Check the triggered events determine the state of the order.
pending_events = order.collect_events()
copy = None
for e in pending_events:
    copy = e.mutate(copy)
assert copy.name == order.name
assert copy.created_on == order.created_on
assert copy.modified_on == order.modified_on
assert copy.confirmed_at == order.confirmed_at
assert copy.pickedup_at == order.pickedup_at
```

Raising exceptions in the body of decorated methods

It is actually possible to decorate the `pickup()` command method with the `@event` decorator, but if a decorated command method has conditional logic that would mean the state of the aggregate should not be evolved, you must take care to raise an exception rather than returning early, and raise an exception before changing the state of the aggregate at all. By raising an exception in the body of a decorated method, the triggered event will not in fact be appended to the aggregate's list of pending events, and it will be as if it never happened. However, the conditional expression will be perhaps needlessly evaluated each time the aggregate is reconstructed from stored events. Of course this conditional logic may be useful and considered as validation of the projection of earlier events, for example checking the the `Confirmed` event is working properly.

If you wish to use this style, just make sure to raise an exception rather than returning early, and make sure not to change the state of the aggregate if an exception may be raised later. Returning early will mean the event will be appended to the list of pending events. Changing the state before raising an exception will the state will be different when the aggregate is reconstructed from stored events. So if your method does change state and then raise an exception, make sure to obtain a fresh version of the aggregate before continuing to trigger events.

```
class Order(Aggregate):
    def __init__(self, name):
        self.name = name
        self.confirmed_at = None
        self.pickedup_at = None
```

(continues on next page)

(continued from previous page)

```
    @event("Confirmed")
    def confirm(self, at):
        self.confirmed_at = at

    @event("PickedUp")
    def pickup(self, at):
        if self.confirmed_at:
            self.pickedup_at = at
        else:
            raise RuntimeError("Order is not confirmed")


    # Creating the aggregate causes one pending event.
    order = Order("name")
    assert len(order.pending_events) == 1

    # Call pickup() too early raises an exception.
    try:
        order.pickup(datetime.now())
    except RuntimeError:
        pass
    else:
        raise Exception("Shouldn't get here")

    # There is still only one pending event.
    assert len(order.pending_events) == 1
```

Recording command arguments and reprocessing them each time the aggregate is reconstructed is perhaps best described as "command sourcing".

In many cases, a command will do some work and trigger an aggregate event that has attributes that are different from the command, and in those cases it is necessary to have two different methods with different signatures: a command method that is not decorated and a decorated method that triggers and applies an aggregate event. This second method may arguably be well named by using a past participle rather than the imperative form.

The `@aggregate` decorator

Just for fun, the library's *`aggregate()`* function can be used to declare event sourced aggregate classes. This is equivalent to inheriting from the library's *`Aggregate`* class. The created event name can be defined using the `created_event_name` argument of the decorator. However, it is recommended to inherit from the *`Aggregate`* class rather than using the `@aggregate` decorator so that full the *`Aggregate`* class definition will be visible to your IDE.

```
from eventsourcing.domain import aggregate


@aggregate(created_event_name="Started")
class Order:
    def __init__(self, name):
        self.name = name


order = Order("my order")
pending_events = order.collect_events()
assert isinstance(pending_events[0], Order.Started)
```

1.4.10 Topics

A "topic" in this library is a string formed from joining with a colon character (`':'`) the path to a Python module (e.g. `'eventsourcing.domain'`) with the qualified name of an object in that module (e.g. `'Aggregate.Created'`). For example `'eventsourcing.domain:Aggregate.Created'` describes the path to the library's *Created* class. The library's *utils* module contains the functions *resolve_topic()* and *get_topic()* which are used in the library to resolve a given topic to a Python object, and to construct a topic for a given Python object.

1.4.11 Classes

class eventsourcing.domain.**MetaDomainEvent**(**args*, ***kwargs*)

Bases: abc.ABCMeta

static **__new__**(*mcs*, *name: str*, *bases: tuple*, *cls_dict: dict*) → eventsourcing.domain.MetaDomainEvent

Create and return a new object. See help(type) for accurate signature.

__init__(**args*, ***kwargs*) → None

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

class eventsourcing.domain.**DomainEvent**(*originator_id: uuid.UUID*, *originator_version: int*, *timestamp: datetime.datetime*)

Bases: abc.ABC

Base class for domain events, such as aggregate *AggregateEvent* and aggregate *Snapshot*.

Constructor arguments:

Parameters

- **originator_id** (*UUID*) – ID of originating aggregate.
- **originator_version** (*int*) – version of originating aggregate.
- **timestamp** (*datetime*) – date-time of the event

class eventsourcing.domain.**AggregateEvent**(*originator_id: uuid.UUID*, *originator_version: int*, *timestamp: datetime.datetime*)

Bases: *eventsourcing.domain.DomainEvent*, typing.Generic

Base class for aggregate events. Subclasses will model decisions made by the domain model aggregates.

Constructor arguments:

Parameters

- **originator_id** (*UUID*) – ID of originating aggregate.
- **originator_version** (*int*) – version of originating aggregate.
- **timestamp** (*datetime*) – date-time of the event

mutate(*obj: Optional[TAggregate]*) → Optional[TAggregate]

Changes the state of the aggregate according to domain event attributes.

apply(*aggregate: TAggregate*) → None

Applies the domain event to the aggregate.

class eventsourcing.domain.**AggregateCreated**(*originator_id: uuid.UUID*, *originator_version: int*, *timestamp: datetime.datetime*, *originator_topic: str*)

Bases: *eventsourcing.domain.AggregateEvent*

Domain event for when aggregate is created.

Constructor arguments:

Parameters

- **originator_id** (*UUID*) – ID of originating aggregate.
- **originator_version** (*int*) – version of originating aggregate.
- **timestamp** (*datetime*) – date-time of the event
- **originator_topic** (*str*) – topic for the aggregate class

mutate(*obj: Optional[TAggregate]*) → TAggregate
Constructs aggregate instance defined by domain event object attributes.

eventsourcing.domain.**event**(*arg: Union[function, str, Type[eventsourcing.domain.AggregateEvent], None] = None*) → eventsourcing.domain.CommandMethodDecorator
Can be used to decorate an aggregate method so that when the method is called an event is triggered. The body of the method will be used to apply the event to the aggregate, both when the event is triggered and when the aggregate is reconstructed from stored events.

```
class MyAggregate(Aggregate):
    @event("NameChanged")
    def set_name(self, name: str):
        self.name = name
```

. . . is equivalent to. . .

```
class MyAggregate(Aggregate):
    def set_name(self, name: str):
        self.trigger_event(self.NameChanged, name=name)

    class NameChanged(Aggregate.Event):
        name: str

        def apply(self, aggregate):
            aggregate.name = self.name
```

In the example above, the event "NameChanged" is defined automatically by inspecting the signature of the *set_name()* method. If it is preferred to declare the event class explicitly, for example to define upcasting of old events, the event class itself can be mentioned in the event decorator rather than just providing the name of the event as a string.

```
class MyAggregate(Aggregate):

    class NameChanged(Aggregate.Event):
        name: str

    @event(NameChanged)
    def set_name(self, name: str):
        aggregate.name = self.name
```

eventsourcing.domain.**triggers**(*arg: Union[function, str, Type[eventsourcing.domain.AggregateEvent], None] = None*) → eventsourcing.domain.CommandMethodDecorator
Can be used to decorate an aggregate method so that when the method is called an event is triggered. The body of the method will be used to apply the event to the aggregate, both when the event is triggered and when the aggregate is reconstructed from stored events.

```
class MyAggregate(Aggregate):
    @event("NameChanged")
    def set_name(self, name: str):
        self.name = name
```

. . . is equivalent to. . .

```
class MyAggregate(Aggregate):
    def set_name(self, name: str):
        self.trigger_event(self.NameChanged, name=name)

    class NameChanged(Aggregate.Event):
        name: str

        def apply(self, aggregate):
            aggregate.name = self.name
```

In the example above, the event "NameChanged" is defined automatically by inspecting the signature of the *set_name()* method. If it is preferred to declare the event class explicitly, for example to define upcasting of old events, the event class itself can be mentioned in the event decorator rather than just providing the name of the event as a string.

```
class MyAggregate(Aggregate):

    class NameChanged(Aggregate.Event):
        name: str

    @event(NameChanged)
    def set_name(self, name: str):
        aggregate.name = self.name
```

class eventsourcing.domain.**UnboundCommandMethodDecorator**(*event_decorator: eventsourcing.domain.CommandMethodDecorator*)

Bases: object

Wraps an EventDecorator instance when attribute is accessed on an aggregate class.

__init__(*event_decorator: eventsourcing.domain.CommandMethodDecorator*)

Parameters **event_decorator** (*CommandMethodDecorator*) –

class eventsourcing.domain.**BoundCommandMethodDecorator**(*event_decorator: eventsourcing.domain.CommandMethodDecorator*, *aggregate: TAggregate*)

Bases: object

Wraps an EventDecorator instance when attribute is accessed on an aggregate so that the aggregate methods can be accessed.

__init__(*event_decorator: eventsourcing.domain.CommandMethodDecorator*, *aggregate: TAggregate*)

Parameters

- **event_decorator** (*CommandMethodDecorator*) –
- **aggregate** (Aggregate) –

__call__(**args*, ***kwargs*) → None
Call self as a function.

class eventsourcing.domain.**DecoratedEvent**(**args*, ***kwds*)
Bases: *eventsourcing.domain.AggregateEvent*

apply(*aggregate: TAggregate*) → None
Applies event to aggregate by calling method decorated by @event.

class eventsourcing.domain.**MetaAggregate**(**args*, *created_event_name: Optional[str] = None*)
Bases: abc.ABCMeta

static __new__(*mcs*, **args*, ***kwargs*) → eventsourcing.domain.MetaAggregate
Create and return a new object. See help(type) for accurate signature.

__init__(**args*, *created_event_name: Optional[str] = None*) → None
Initialize self. See help(type(self)) for accurate signature.

__call__(**args*, ***kwargs*) → TAggregate
Call self as a function.

static create_id(***kwargs*) → uuid.UUID
Returns a new aggregate ID.

_create(*event_class: Type[TAggregateCreated], *, id: Optional[uuid.UUID] = None, **kwargs*) → TAggregate
Factory method to construct a new aggregate object instance.

class eventsourcing.domain.**Aggregate**
Bases: abc.ABC

Base class for aggregate roots.

class Event(**args*, ***kwds*)
Bases: *eventsourcing.domain.AggregateEvent*

class Created(**args*, ***kwds*)
Bases: *eventsourcing.domain.AggregateCreated*

static __new__(*cls*, **args*, ***kwargs*) → Any
Create and return a new object. See help(type) for accurate signature.

__eq__(*other: Any*) → bool
Return self==value.

__repr__() → str
Return repr(self).

__base_init__(*id: uuid.UUID*, *version: int*, *timestamp: datetime.datetime*) → None
Initialises an aggregate object with an *id*, a *version* number, and a timestamp. The internal *pending_events* list is also initialised.

id
The ID of the aggregate.

version
The version number of the aggregate.

created_on
The date and time when the aggregate was created.

modified_on
The date and time when the aggregate was last modified.

pending_events
A list of pending events.

trigger_event(*event_class: Type[TAggregateEvent], **kwargs*) → None
Triggers domain event of given type, by creating an event object and using it to mutate the aggregate.

collect_events() → List[eventsourcing.domain.AggregateEvent]
Collects and returns a list of pending aggregate *AggregateEvent* objects.

eventsourcing.domain.**aggregate**(*cls: Optional[eventsourcing.domain.MetaAggregate] = None*, ***, *created_event_name: Optional[str] = None*) → Union[eventsourcing.domain.MetaAggregate, Callable]
Converts the class that was passed in to inherit from Aggregate.

```
@aggregate
class MyAggregate:
    pass
```

. . . is equivalent to. . .

```
class MyAggregate(Aggregate):
    pass
```

exception eventsourcing.domain.**VersionError**
Bases: Exception

Raised when a domain event can't be applied to an aggregate due to version mismatch indicating the domain event is not the next in the aggregate's sequence of events.

class eventsourcing.domain.**Snapshot**(*originator_id: uuid.UUID*, *originator_version: int*, *timestamp: datetime.datetime*, *topic: str*, *state: dict*)
Bases: *eventsourcing.domain.DomainEvent*

Snapshots represent the state of an aggregate at a particular version.

Constructor arguments:

Parameters

- **originator_id** (*UUID*) – ID of originating aggregate.
- **originator_version** (*int*) – version of originating aggregate.
- **timestamp** (*datetime*) – date-time of the event
- **topic** (*str*) – string that includes a class and its module
- **state** (*dict*) – version of originating aggregate.

classmethod take(*aggregate: TAggregate*) → eventsourcing.domain.Snapshot
Creates a snapshot of the given *Aggregate* object.

mutate(*_: None = None*) → TAggregate
Reconstructs the snapshotted *Aggregate* object.

eventsourcing.utils.**get_topic**(*cls: type*) → str
Returns a string that locates the given class.

eventsourcing.utils.**resolve_topic**(*topic: str*) → Any
Returns an object located by the given string.

eventsourcing.utils.**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.utils.**strtobool**(*val: str*) → bool

Convert a string representation of truth to True or False.

True values are 'y', 'yes', 't', 'true', 'on', and '1'; false values are 'n', 'no', 'f', 'false', 'off', and '0'. Raises ValueError if 'val' is anything else.

eventsourcing.utils.**random**() → x in the interval [0, 1).

1.5 application — Applications

This module helps with developing event-sourced applications.

An event-sourced **application** object has command and query methods used by clients to interact with its domain model. An application object has an event-sourced **repository** used to obtain already existing event-sourced aggregates. It also has a **notification log** that is used to propagate the state of the application as a sequence of domain event notifications.

1.5.1 Domain-driven design

The book *Domain-Driven Design* describes a "layered architecture" with four layers: interface, application, domain, and infrastructure. The application layer depends on the domain and infrastructure layers. The interface layer depends on the application layer.

Generally speaking, the application layer implements commands which change the state of the application, and queries which present the state of the application. The commands and queries ("application services") are called from the interface layer. By keeping the application and domain logic in the application and domain layers, different interfaces can be developed for different technologies without duplicating application and domain logic.

The discussion below continues these ideas, by combining event-sourced aggregates and persistence objects in an application object that implements "application services" as object methods.

1.5.2 Application objects

An event-sourced application object combines a domain model with a cohesive mechanism for storing and retrieving domain events.

The library's *Application* object class brings together objects from the *domain* and *persistence* modules. It can be subclassed to develop event-sourced applications. The general idea is to name your application object class after the domain supported by its domain model, and then define command and query methods that allow interfaces to create,

read, update and delete your domain model aggregates. Domain model aggregates are discussed in the *domain module documentation*. The "ubiquitous language" of your project should guide the names of the application's command and query methods, along with those of its *domain model aggregates*.

The *Application* class defines an object method *save()* which can be used to update the recorded state of one or many *domain model aggregates*. The *save()* method functions by using the aggregate's *collect_events()* method to collect pending domain events; the pending domain events are stored by calling the *put()* method of application's *event store*.

The *Application* class defines an object attribute repository which holds an *event-sourced repository*. The repository's *get()* method can be used by your application's command and query methods to obtain already existing aggregates.

The *Application* class defines an object attribute log which holds a *local notification log*. The notification log can be used to propagate the state of an application as a sequence of domain event notifications.

The *Application* class defines an object method *take_snapshot()* which can be used for *snapshotting* existing aggregates. Snapshotting isn't necessary, but can help to reduce the time it takes to access aggregates with lots of domain events.

1.5.3 Basic example

In the example below, the Worlds application extends the library's application object base class. The World aggregate is defined and discussed as the *basic example* in the domain module documentation.

The Worlds application's create_world() method is a command method that creates and saves new World aggregates, returning a new world_id that can be used to identify the aggregate on subsequence method calls. It saves the new aggregate by calling the base class save() method.

The Worlds application's make_it_so() method is a command method that obtains an existing World aggregate from the repository, then calls the aggregate's command method make_it_so(), and then saves the aggregate by calling the application's save() method.

The Worlds application's get_world_history() method is a query method that presents the current history of an existing aggregate.

```
from typing import List
from uuid import UUID

from eventsourcing.application import Application


class Worlds(Application):
    def create_world(self) -> UUID:
        world = World.create()
        self.save(world)
        return world.id

    def make_it_so(self, world_id: UUID, what: str):
        world = self.repository.get(world_id)
        world.make_it_so(what)
        self.save(world)

    def get_world_history(self, world_id: UUID) -> List[str]:
        world = self.repository.get(world_id)
        return list(world.history)
```

In the example below, an instance of the Worlds application is constructed. A new World aggregate is created by calling the create_world() method. Three items are added to its history: "dinosaurs", "trucks", and "internet" by

calling the `make_it_so()` application command with the `world_id` aggregate ID. The history of the aggregate is obtained when the `get_world_history()` method is called.

```
application = Worlds()

world_id = application.create_world()

application.make_it_so(world_id, "dinosaurs")
application.make_it_so(world_id, "trucks")
application.make_it_so(world_id, "internet")

history = application.get_world_history(world_id)
assert history[0] == "dinosaurs"
assert history[1] == "trucks"
assert history[2] == "internet"
```

By default, the application object uses the "Plain Old Python Object" infrastructure which has stored domain events in memory only. To store the domain events in a real database, you will need to *configure persistence*.

1.5.4 Repository

A repository is used to get the already existing aggregates of the application's domain model.

The application object's `repository` attribute has an instance of the library's *`Repository`* class.

The repository's *`get()`* method is used to obtain already existing aggregates. It uses the event store's *`get()`* method to retrieve the already existing *domain event objects* of the requested aggregate, and the *`mutate()`* methods of the *domain event objects* to reconstruct the state of the requested aggregate. The repository's *`get()`* method accepts two arguments: `aggregate_id` and `version`:

The `aggregate_id` argument is required, and should be the ID of an already existing aggregate. If the aggregate is not found, the exception *`AggregateNotFound`* will be raised.

The `version` argument is optional, and represents the required version of the aggregate. If the requested version is greater than the highest available version of the aggregate, the highest available version of the aggregate will be returned.

```
world_latest = application.repository.get(world_id)

assert world_latest.version == 4
assert len(world_latest.history) == 3


world_v1 = application.repository.get(world_id, version=1)

assert world_v1.version == 1
assert len(world_v1.history) == 0

world_v2 = application.repository.get(world_id, version=2)

assert world_v2.version == 2
assert len(world_v2.history) == 1
assert world_v2.history[-1] == "dinosaurs"

world_v3 = application.repository.get(world_id, version=3)

assert world_v3.version == 3
```

(continues on next page)

(continued from previous page)

```
assert len(world_v3.history) == 2
assert world_v3.history[-1] == "trucks"

world_v4 = application.repository.get(world_id, version=4)

assert world_v4.version == 4
assert len(world_v4.history) == 3
assert world_v4.history[-1] == "internet"

world_v5 = application.repository.get(world_id, version=5)

assert world_v5.version == 4  # There is no version 5.
assert len(world_v5.history) == 3
assert world_v5.history[-1] == "internet"
```

1.5.5 Notification log

A notification log can be used to propagate the state of an application as a sequence of domain event notifications.

The application object's `log` attribute has an instance of the library's *`LocalNotificationLog`* class. The notification log presents linked sections of *notification objects*. The sections are instances of the library's *`Section`* class.

Each event notification has an `id` that has the unique integer ID of the event notification. The event notifications are ordered by their IDs, with later event notifications having higher values than earlier ones.

A notification log section is identified by a section ID string that comprises two integers separated by a comma, for example `"1,10"`. The first integer specifies the notification ID of the first event notification included in the section. The second integer specifies the notification ID of the second event notification included in the section. Sections are requested from the notification using the Python square bracket syntax, for example `application.log["1,10"]`.

The notification log will return a section that has no more than the requested number of event notifications. Sometimes there will be less event notifications in the recorded sequence of event notifications than are needed to fill the section, in which case less than the number of event notifications will be included in the returned section. On the other hand, there may be gaps in the recorded sequence of event notifications, in which case the last event notification included in the section may have a notification ID that is greater than that which was specified in the requested section ID.

A notification log section has an attribute `section_id` that has the section ID. The section ID value will represent the event notification ID of the first and the last event notification included in the section. If there are no event notifications, the section ID will be `None`.

A notification log section has an attribute `items` that has the list of *notification objects* included in the section.

A notification log section has an attribute `next_id` that has the section ID of the next section in the notification log. If the notification log section has less event notifications that were requested, the `next_id` value will be `None`.

In the example above, there are four domain events in the domain model, and so there are four notifications in the notification log.

```
from eventsourcing.persistence import Notification

section = application.log["1,10"]

assert len(section.items) == 4
assert section.id == "1,4"
```

(continues on next page)

(continued from previous page)

```
assert section.next_id is None

assert isinstance(section.items[0], Notification)
assert section.items[0].id == 1
assert section.items[1].id == 2
assert section.items[2].id == 3
assert section.items[3].id == 4

assert section.items[0].originator_id == world_id
assert section.items[1].originator_id == world_id
assert section.items[2].originator_id == world_id
assert section.items[3].originator_id == world_id

assert section.items[0].originator_version == 1
assert section.items[1].originator_version == 2
assert section.items[2].originator_version == 3
assert section.items[3].originator_version == 4

assert "World.Created" in section.items[0].topic
assert "World.SomethingHappened" in section.items[1].topic
assert "World.SomethingHappened" in section.items[2].topic
assert "World.SomethingHappened" in section.items[3].topic

assert b"dinosaurs" in section.items[1].state
assert b"trucks" in section.items[2].state
assert b"internet" in section.items[3].state
```

A domain event can be reconstructed from an event notification by calling the application's mapper method *to_domain_event()*. If the application is configured to encrypt stored events, the event notification will also be encrypted, but the mapper will decrypt the event notification.

```
domain_event = application.mapper.to_domain_event(section.items[0])
assert isinstance(domain_event, World.Created)
assert domain_event.originator_id == world_id

domain_event = application.mapper.to_domain_event(section.items[3])
assert isinstance(domain_event, World.SomethingHappened)
assert domain_event.originator_id == world_id
assert domain_event.what == "internet"
```

1.5.6 Snapshotting

If the reconstruction of an aggregate depends on obtaining and replaying a relatively large number of domain event objects, it can take a relatively long time to reconstruct the aggregate. Snapshotting aggregates can help to reduce access time of aggregates with lots of domain events.

Snapshots are stored separately from the aggregate events. When snapshotting is enabled, the application object will have a snapshot store assigned to the attribute 'snapshots'. By default, snapshotting is not enabled, and the 'snapshots' attribute has the value `None`.

```
assert application.snapshots is None
```

Enabling snapshotting

To enable snapshotting in application objects, the environment variable `IS_SNAPSHOTTING_ENABLED` may be set to a valid "true" value. The function *strtobool()* module is used to interpret the value of this environment variable, so that strings `"y"`, `"yes"`, `"t"`, `"true"`, `"on"` and `"1"` are considered to be "true" values, and `"n"`, `"no"`, `"f"`, `"false"`, `"off"` and `"0"` are considered to be "false" values, and other values are considered to be invalid. The default is for an application's snapshotting functionality to be not enabled.

Application environment variables can be passed into the application using the `env` argument when constructing an application object. Snapshotting can be enabled (or disabled) for an individual application object in this way.

```
application = Worlds(env={"IS_SNAPSHOTTING_ENABLED": "y"})
assert application.snapshots is not None
```

Application environment variables can be also be set in the operating system environment. Setting operating system environment variables will affect all applications created in that environment.

```
import os

os.environ["IS_SNAPSHOTTING_ENABLED"] = "y"

application = Worlds()

assert application.snapshots is not None

del os.environ["IS_SNAPSHOTTING_ENABLED"]
```

Values passed into the application object will override operating system environment variables.

```
os.environ["IS_SNAPSHOTTING_ENABLED"] = "y"
application = Worlds(env={"IS_SNAPSHOTTING_ENABLED": "n"})

assert application.snapshots is None

del os.environ["IS_SNAPSHOTTING_ENABLED"]
```

Snapshotting can also be enabled for all instances of an application class by setting the boolean attribute 'is_snapshotting_enabled' on the application class.

```
class WorldsWithSnapshottingEnabled(Worlds):
    is_snapshotting_enabled = True

application = WorldsWithSnapshottingEnabled()
assert application.snapshots is not None
```

However, this setting will also be overridden by both the construct arg `env` and by the operating system environment. The example below demonstrates this by extending the *Worlds* application class defined above.

```
application = WorldsWithSnapshottingEnabled(env={"IS_SNAPSHOTTING_ENABLED": "n"})
assert application.snapshots is None

os.environ["IS_SNAPSHOTTING_ENABLED"] = "n"
application = WorldsWithSnapshottingEnabled()
assert application.snapshots is None
del os.environ["IS_SNAPSHOTTING_ENABLED"]
```

Taking snapshots

The application method *take_snapshot()* can be used to create a snapshot of the state of an aggregate. The ID of an aggregate to be snapshotted must be passed when calling this method. By passing in the ID (and optional version number), rather than an actual aggregate object, the risk of snapshotting a somehow "corrupted" aggregate object that does not represent the actually recorded state of the aggregate is avoided.

```
application = Worlds(env={"IS_SNAPSHOTTING_ENABLED": "y"})
world_id = application.create_world()

application.make_it_so(world_id, "dinosaurs")
application.make_it_so(world_id, "trucks")
application.make_it_so(world_id, "internet")

application.take_snapshot(world_id)
```

Snapshots are stored separately from the aggregate events, but snapshot objects are implemented as a kind of domain event, and snapshotting uses the same mechanism for storing snapshots as for storing aggregate events. When snapshotting is enabled, the application object attribute `snapshots` is an event store dedicated to storing snapshots. The snapshots can be retrieved from the snapshot store using the *get()* method. We can get the latest snapshot by selecting in descending order with a limit of 1.

```
snapshots = application.snapshots.get(world_id, desc=True, limit=1)

snapshots = list(snapshots)
assert len(snapshots) == 1, len(snapshots)
snapshot = snapshots[0]

assert snapshot.originator_id == world_id
assert snapshot.originator_version == 4
```

When snapshotting is enabled, the application repository looks for snapshots in this way. If a snapshot is found by the aggregate repository when retrieving an aggregate, then only the snapshot and subsequent aggregate events will be retrieved and used to reconstruct the state of the aggregate.

Automatic snapshotting

Automatic snapshotting of aggregates at regular intervals can be enabled by setting the application class attribute 'snapshotting_intervals'. The 'snapshotting_intervals' should be a mapping of aggregate classes to integers which represent the snapshotting interval. When aggregates are saved, snapshots will be taken if the version of aggregate coincides with the specified interval. The example below demonstrates this by extending the *Worlds* application class with *World* aggregates snapshotted every 2 events.

```
class WorldsWithAutomaticSnapshotting(Worlds):
    snapshotting_intervals = {World: 2}


application = WorldsWithAutomaticSnapshotting()

world_id = application.create_world()

application.make_it_so(world_id, "dinosaurs")
application.make_it_so(world_id, "trucks")
application.make_it_so(world_id, "internet")

```

(continues on next page)

(continued from previous page)

```
snapshots = application.snapshots.get(world_id)
snapshots = list(snapshots)

assert len(snapshots) == 2

assert snapshots[0].originator_id == world_id
assert snapshots[0].originator_version == 2

assert snapshots[1].originator_id == world_id
assert snapshots[1].originator_version == 4
```

In practice, a suitable interval would most likely be larger than 2. And 'snapshotting_intervals' would be defined on your application class and not a subclass.

1.5.7 Configuring persistence

The example above uses the application's default persistence infrastructure. By default, the application object uses the library's "plain old Python objects" *infrastructure factory*, which provides the application with infrastructure classes that simply keep stored events in a data structure in memory.

To use alternative persistence infrastructure, you will need to set the environment variable `INFRASTRUCTURE_FACTORY` to the *topic* of another infrastructure factory object class that will construct alternative application persistence objects. Using alternative persistence infrastructure will normally involve setting particular environment variables that configure access to a real database, such as a database name, a user name, and a password.

The example below shows how to configure the application to use the library's *SQLite infrastructure*. In the case of the library's SQLite factory, the environment variable `SQLITE_DBNAME` must be set to a file path. And if the tables already exist, the `CREATE_TABLE` may be set to a "false" value (`"n"`, `"no"`, `"f"`, `"false"`, `"off"`, or `"0"`). The function *`strtobool()`* is used to interpret the value of this environment variable.

```
from tempfile import NamedTemporaryFile

tmpfile = NamedTemporaryFile(suffix="_eventsourcing_test.db")
tmpfile.name

os.environ["INFRASTRUCTURE_FACTORY"] = "eventsourcing.sqlite:Factory"
os.environ["SQLITE_DBNAME"] = tmpfile.name
application = Worlds()

world_id = application.create_world()

application.make_it_so(world_id, "dinosaurs")
application.make_it_so(world_id, "trucks")
application.make_it_so(world_id, "internet")
```

By using a file on disk, the named temporary file `tmpfile` above, the state of the application will endure after the application has been reconstructed. The database table only needs to be created once, and so when creating an application for an already existing database the environment variable `CREATE_TABLE` may be set to a "false" value (`"n"`, `"no"`, `"f"`, `"false"`, `"off"`, `"0"`).

```
os.environ["INFRASTRUCTURE_FACTORY"] = "eventsourcing.sqlite:Factory"

application = Worlds()

```

(continues on next page)

(continued from previous page)

```
history = application.get_world_history(world_id)
assert history[0] == "dinosaurs"
assert history[1] == "trucks"
assert history[2] == "internet"
```

1.5.8 Registering custom transcodings

The application's persistence mechanism serialises the domain events, using the library's transcoder. If your aggregates' domain event objects have objects of types that are not already supported by the transcoder, for example custom value objects, custom *transcodings* for these objects will need to be implemented and registered with the application's transcoder.

The application method *register_transcodings()* can be extended to register custom transcodings for custom value objects used in your application's domain events. The library's application base class registers transcodings for `UUID`, `Decimal`, and `datetime` objects.

For example, to define and register a *Transcoding* for the Python `date` class, define a class such as the `DateAsISO` class below, and extend the application *register_transcodings()* method by calling the `super()` method with the given `transcoder` argument, and then the transcoder's *register()* method once for each of your custom transcodings.

```
from datetime import date
from typing import Union

from eventsourcing.persistence import Transcoder, Transcoding


class MyApplication(Application):
    def register_transcodings(self, transcoder: Transcoder):
        super().register_transcodings(transcoder)
        transcoder.register(DateAsISO)


class DateAsISO(Transcoding):
    type = date
    name = "date_iso"

    def encode(self, o: date) -> str:
        return o.isoformat()

    def decode(self, d: Union[str, dict]) -> date:
        assert isinstance(d, str)
        return date.fromisoformat(d)
```

1.5.9 Encryption and compression

Application-level encryption is useful for encrypting the state of the application "on the wire" and "at rest". Compression is useful for reducing transport time of domain events and domain event notifications across a network and for reducing the total size of recorded application state.

The library's *AESCipher* class can be used to encrypt stored domain events. The Python `zlib` module can be used to compress stored domain events. It is encapsulated by the library's *ZlibCompressor* class.

To enable encryption and compression, set the environment variables `CIPHER_TOPIC` (a *topic* to a cipher class), `CIPHER_KEY` (a valid encryption key), and `COMPRESSOR_TOPIC` (*topic* for a compressor class).

When using the library's *AESCipher* class, you can use its static method *create_key()* to generate a valid encryption key.

```
import os

from eventsourcing.cipher import AESCipher

# Generate a cipher key (keep this safe).
cipher_key = AESCipher.create_key(num_bytes=32)

# Configure cipher key.
os.environ["CIPHER_KEY"] = cipher_key

# Configure cipher topic.
os.environ["CIPHER_TOPIC"] = "eventsourcing.cipher:AESCipher"

# Configure compressor topic.
os.environ["COMPRESSOR_TOPIC"] = "eventsourcing.compressor:ZlibCompressor"
```

1.5.10 Saving multiple aggregates

In many cases, it is both possible and very useful to save more than one aggregate in the same atomic transaction. The example below continues the example from the discussion of *namespaced IDs* in the previous section. The aggregate classes `Page` and `Index` are defined in that section.

We can define a simple wiki application, which creates named pages. Pages can be retrieved by name. Names can be changed and the pages can be retrieved by the new name.

```
class Wiki(Application):
    def create_page(self, name: str, body: str) -> None:
        page = Page.create(name, body)
        index = Index.create(page)
        self.save(page, index)

    def rename_page(self, name: str, new_name: str) -> None:
        page = self.get_page(name)
        page.update_name(new_name)
        index = Index.create(page)
        self.save(page, index)
        return page.body

    def get_page(self, name: str) -> Page:
        index_id = Index.create_id(name)
        index = self.repository.get(index_id)
        page_id = index.ref
        return self.repository.get(page_id)
```

Now let's construct the application object and create a new page (with a deliberate spelling mistake).

```
wiki = Wiki()

wiki.create_page(name="Erth", body="Lorem ipsum...")
```

We can use the page name to retrieve the body of the page.

```
assert wiki.get_page(name="Erth").body == "Lorem ipsum..."
```

We can also update the name of the page, and then retrieve the page using the new name.

```
wiki.rename_page(name="Erth", new_name="Earth")

assert wiki.get_page(name="Earth").body == "Lorem ipsum..."
```

The uniqueness constraint on the recording of stored domain event objects combined with the atomicity of recording domain events means that name collisions in the index will result in the wiki not being updated.

```
from eventsourcing.persistence import RecordConflictError

# Can't create another page using an existing name.
try:
    wiki.create_page(name="Earth", body="Neque porro quisquam...")
except RecordConflictError:
    pass
else:
    raise AssertionError("RecordConflictError not raised")

assert wiki.get_page(name="Earth").body == "Lorem ipsum..."

# Can't rename another page to an existing name.
wiki.create_page(name="Mars", body="Neque porro quisquam...")
try:
    wiki.rename_page(name="Mars", new_name="Earth")
except RecordConflictError:
    pass
else:
    raise AssertionError("RecordConflictError not raised")

assert wiki.get_page(name="Earth").body == "Lorem ipsum..."
assert wiki.get_page(name="Mars").body == "Neque porro quisquam..."
```

A more refined implementation might release old index objects when page names are changed so that they can be reused by other pages, or update the old index to point to the new index, so that redirects can be implemented.

1.5.11 Classes

class eventsourcing.application.**Repository**(*event_store: eventsourcing.persistence.EventStore[eventsourcing.domain.AggregateEvent][eve*
snapshot_store: Optional[eventsourcing.persistence.EventStore[eventsourcing.domain.Sna
= None)

Bases: typing.Generic

Reconstructs aggregates from events in an *EventStore*, possibly using snapshot store to avoid replaying all events.

__init__(*event_store: eventsourcing.persistence.EventStore[eventsourcing.domain.AggregateEvent][eventsourcing.domain.A*
snapshot_store: Optional[eventsourcing.persistence.EventStore[eventsourcing.domain.Snapshot][eventsourcing.do
= None)

Initialises repository with given event store (an *EventStore* for aggregate *AggregateEvent* objects) and optionally a snapshot store (an *EventStore* for aggregate *Snapshot* objects).

get(*aggregate_id: uuid.UUID*, *version: Optional[int] = None*) → TAggregate
Returns an *Aggregate* for given ID, optionally at the given version.

class eventsourcing.application.**Section**(*id: Optional[str]*, *items: List[eventsourcing.persistence.Notification]*, *next_id: Optional[str]*)
Bases: object

Frozen dataclass that represents a section from a *NotificationLog*. The items attribute contains a list of *Notification* objects. The id attribute is the section ID, two integers separated by a comma that described the first and last notification ID that are included in the section. The next_id attribute describes the section ID of the next section, and will be set if the section contains as many notifications are were requested.

Constructor arguments:

Parameters

- **id** (*Optional[str]*) – section ID of this section e.g. "1,10"
- **items** (*List[Notification]*) – a list of event notifications
- **next_id** (*Optional[str]*) – section ID of the following section

class eventsourcing.application.**NotificationLog**
Bases: abc.ABC

Abstract base class for notification logs.

__getitem__(*section_id: str*) → eventsourcing.application.Section
Returns a *Section* from a notification log.

select(*start: int*, *limit: int*) → List[eventsourcing.persistence.Notification]
Returns a list of *Notification* objects.

class eventsourcing.application.**LocalNotificationLog**(*recorder: eventsourcing.persistence.ApplicationRecorder*, *section_size: int = 10*)
Bases: *eventsourcing.application.NotificationLog*

Notification log that presents sections of event notifications retrieved from an *ApplicationRecorder*.

__init__(*recorder: eventsourcing.persistence.ApplicationRecorder*, *section_size: int = 10*)
Initialises a local notification object with given *ApplicationRecorder* and an optional section size.

Constructor arguments:

Parameters

- **recorder** (ApplicationRecorder) – application recorder from which event notifications will be selected
- **section_size** (*int*) – number of notifications to include in a section

__getitem__(*requested_section_id: str*) → eventsourcing.application.Section
Returns a *Section* of event notifications based on the requested section ID. The section ID of the returned section will describe the event notifications that are actually contained in the returned section, and may vary from the requested section ID if there are less notifications in the recorder than were requested, or if there are gaps in the sequence of recorded event notification.

select(*start: int*, *limit: int*) → List[eventsourcing.persistence.Notification]
Returns a list of *Notification* objects.

class eventsourcing.application.**Application**(*env: Optional[Mapping[KT, VT_co]] = None*)
Bases: abc.ABC, typing.Generic

Base class for event-sourced applications.

__init__(*env: Optional[Mapping[KT, VT_co]] = None*) → None
Initialises an application with an *InfrastructureFactory*, a *Mapper*, an *ApplicationRecorder*, an *EventStore*, a *Repository*, and a *LocalNotificationLog*.

construct_env(*env: Optional[Mapping[KT, VT_co]] = None*) → Mapping[KT, VT_co]
Constructs environment from which application will be configured.

construct_factory() → eventsourcing.persistence.InfrastructureFactory
Constructs an *InfrastructureFactory* for use by the application.

construct_mapper(*application_name: str = ''*) → eventsourcing.persistence.Mapper
Constructs a *Mapper* for use by the application.

construct_transcoder() → eventsourcing.persistence.Transcoder
Constructs a *Transcoder* for use by the application.

register_transcodings(*transcoder: eventsourcing.persistence.Transcoder*) → None
Registers *Transcoding* objects on given *JSONTranscoder*.

construct_recorder() → eventsourcing.persistence.ApplicationRecorder
Constructs an *ApplicationRecorder* for use by the application.

construct_event_store() → eventsourcing.persistence.EventStore[eventsourcing.domain.AggregateEvent][eventsourcin
Constructs an *EventStore* for use by the application to store and retrieve aggregate *AggregateEvent* objects.

construct_snapshot_store() → Optional[eventsourcing.persistence.EventStore[eventsourcing.domain.Snapshot][event
Constructs an *EventStore* for use by the application to store and retrieve aggregate *Snapshot* objects.

construct_repository() → eventsourcing.application.Repository[~TAggregate][TAggregate]
Constructs a *Repository* for use by the application.

construct_notification_log() → eventsourcing.application.LocalNotificationLog
Constructs a *LocalNotificationLog* for use by the application.

save(**aggregates*, ***kwargs*) → None
Collects pending events from given aggregates and puts them in the application's event store.

notify(*new_events: List[eventsourcing.domain.AggregateEvent]*) → None
Called after new domain events have been saved. This method on this class class doesn't actually do anything, but this method may be implemented by subclasses that need to take action when new domain events have been saved.

take_snapshot(*aggregate_id: uuid.UUID*, *version: Optional[int] = None*) → None
Takes a snapshot of the recorded state of the aggregate, and puts the snapshot in the snapshot store.

exception eventsourcing.application.**AggregateNotFound**
Bases: Exception

Raised when an *Aggregate* object is not found in a *Repository*.

class eventsourcing.cipher.**AESCipher**(*cipher_key: str*)
Bases: *eventsourcing.persistence.Cipher*

Cipher strategy that uses AES cipher in GCM mode.

static create_key(*num_bytes: int*) → str
Creates AES cipher key, with length num_bytes.

Parameters **num_bytes** – An int value, either 16, 24, or 32.

__init__(*cipher_key: str*)
Initialises AES cipher with `cipher_key`.

Parameters **cipher_key** (*str*) – 16, 24, or 32 bytes encoded as base64

encrypt(*plaintext: bytes*) → bytes
Return ciphertext for given plaintext.

decrypt(*ciphertext: bytes*) → bytes
Return plaintext for given ciphertext.

class eventsourcing.compressor.**ZlibCompressor**
Bases: *eventsourcing.persistence.Compressor*

compress(*data: bytes*) → bytes
Compress bytes using zlib.

decompress(*data: bytes*) → bytes
Decompress bytes using zlib.

1.6 persistence — Infrastructure

This module provides a cohesive mechanism for storing domain events.

The entire mechanism is encapsulated by the library's **event store** object class. An event store stores and retrieves domain events. The event store uses a mapper to convert domain events to stored events, and it uses a recorder to insert stored events in a datastore.

A **mapper** converts domain event objects of various types to stored event objects when domain events are stored in the event store. It also converts stored events objects back to domain event objects when domain events are retrieved from the event store. A mapper uses an extensible **transcoder** that can be set up with additional transcoding objects that serialise and deserialise particular types of object, such as Python's `UUID`, `datetime` and `Decimal` objects. A mapper may use a compressor to compress and decompress the state of stored event objects, and may use a cipher to encode and decode the state of stored event objects. If both a compressor and a cipher are being used by a mapper, the state of any stored event objects will be compressed and then encoded when storing domain events, and will be decoded and then decompressed when retrieving domain events.

A **recorder** inserts stored event objects in a datastore when domain events are stored in an event store, and selects stored events from a datastore when domain events are retrieved from an event store. Depending on the type of the recorder it may be possible to select the stored events as event notifications, and it may be possible atomically to record tracking records along with the stored events,

1.6.1 Transcoder

A transcoder is used by a *mapper* to serialise and deserialise the state of domain model event objects.

The library's *JSONTranscoder* class can be constructed without any arguments.

```
from eventsourcing.persistence import JSONTranscoder

transcoder = JSONTranscoder()
```

The `transcoder` object has methods *encode()* and *decode()* which are used to perform the serialisation and deserialisation. The serialised state is a Python `bytes` object.

```
data = transcoder.encode({"a": 1})
copy = transcoder.decode(data)
assert copy == {"a": 1}
```

The library's *JSONTranscoder* uses the Python `json` module. And so, by default, only the basic object types supported by that module can be encoded and decoded. The transcoder can be extended by registering transcodings for the other types of object used in your domain model's event objects. A transcoding will convert other types of object to a representation of the non-basic type of object that uses the basic types that are supported. The transcoder method *register()* is used to register individual transcodings with the transcoder.

1.6.2 Transcodings

In order to encode and decode non-basic types of object that are not supported by the transcoder by default, custom transcodings need to be defined in code and registered with the *transcoder* using the transcoder object's *register()* method. A transcoding will encode an instance of a non-basic type of object that cannot by default be encoded by the transcoder into a basic type of object that can be encoded by the transcoder, and will decode that representation into the original type of object. This makes it possible to transcode custom value objects, including custom types that contain custom types. The transcoder works recursively through the object and so included custom types do not need to be encoded by the transcoder, but will be converted subsequently.

The library includes a limited collection of custom transcoding objects. For example, the library's *UUIDAsHex* class transcodes a Python `UUID` objects as a hexadecimal string.

```
from uuid import uuid4

from eventsourcing.persistence import UUIDAsHex

transcoding = UUIDAsHex()

id1 = uuid4()
data = transcoding.encode(id1)
copy = transcoding.decode(data)
assert copy == id1
```

The library's *DatetimeAsISO* class transcodes Python `datetime` objects as ISO strings.

```
from datetime import datetime

from eventsourcing.persistence import (
    DatetimeAsISO,
)

transcoding = DatetimeAsISO()

datetime1 = datetime(2021, 12, 31, 23, 59, 59)
data = transcoding.encode(datetime1)
copy = transcoding.decode(data)
assert copy == datetime1
```

The library's *DecimalAsStr* class transcodes Python `Decimal` objects as decimal strings.

```
from decimal import Decimal

from eventsourcing.persistence import (
    DecimalAsStr,
```

(continues on next page)

(continued from previous page)

```
)

transcoding = DecimalAsStr()

decimal1 = Decimal("1.2345")
data = transcoding.encode(decimal1)
copy = transcoding.decode(data)
assert copy == decimal1
```

Transcodings are registered with the transcoder using the transcoder object's *register()* method.

```
transcoder.register(UUIDAsHex())
transcoder.register(DatetimeAsISO())
transcoder.register(DecimalAsStr())

data = transcoder.encode(id1)
copy = transcoder.decode(data)
assert copy == id1

data = transcoder.encode(datetime1)
copy = transcoder.decode(data)
assert copy == datetime1

data = transcoder.encode(decimal1)
copy = transcoder.decode(data)
assert copy == decimal1
```

Attempting to serialize an unsupported type will result in a Python `TypeError`.

```
from datetime import date

date1 = date(2021, 12, 31)
try:
    data = transcoder.encode(date1)
except TypeError as e:
    assert e.args[0] == (
        "Object of type <class 'datetime.date'> is not serializable. "
        "Please define and register a custom transcoding for this type."
    )
else:
    raise AssertionError("TypeError not raised")
```

Attempting to deserialize an unsupported type will also result in a Python `TypeError`.

```
try:
    JSONTranscoder().decode(data)
except TypeError as e:
    assert e.args[0] == (
        "Data serialized with name 'decimal_str' is not deserializable. "
        "Please register a custom transcoding for this type."
    )
else:
    raise AssertionError("TypeError not raised")
```

The library's abstract base class *Transcoding* can be subclassed to define custom transcodings for other object types. To define a custom transcoding, simply subclass this base class, assign to the class attribute `type` the class transcoded type, and assign a string to the class attribute `name`. Then define an *encode()* method that converts

an instance of that type to a representation that uses a basic type, and a *decode()* method that will convert that representation back to an instance of that type.

```
from eventsourcing.persistence import Transcoding
from typing import Union


class DateAsISO(Transcoding):
    type = date
    name = "date_iso"

    def encode(self, obj: date) -> str:
        return obj.isoformat()

    def decode(self, data: str) -> date:
        return date.fromisoformat(data)


transcoder.register(DateAsISO())

data = transcoder.encode(date1)
copy = transcoder.decode(data)
assert copy == date1
```

Please note, due to the way the Python `json` module works, it isn't currently possible to transcode subclasses of the basic Python types that are supported by default, such as `dict`, `list`, `tuple`, `str`, `int`, `float`, and `bool`. This behaviour also means an encoded `tuple` will be decoded as a `list`. This behaviour is coded in Python as C code, and can't be suspended without avoiding the use of this C code and thereby incurring a performance penalty in the transcoding of domain event objects.

```
data = transcoder.encode((1, 2, 3))
copy = transcoder.decode(data)
assert isinstance(copy, list)
assert copy == [1, 2, 3]
```

Custom or non-basic types that contain other custom or non-basic types can be supported in the transcoder by registering a transcoding for each non-basic type. The transcoding for the type which contains non-basic types must return an object that represents that type by involving the included non-basic objects, and this representation will be subsequently transcoded by the transcoder using the applicable transcoding for the included non-basic types. In the example below, `SimpleCustomValue` has a `UUID` and a `date` as its `id` and `data` attributes. The transcoding for `SimpleCustomValue` returns a Python `dict` that includes the non-basic `UUID` and `date` objects. The class `ComplexCustomValue` simply has a `ComplexCustomValue` object as its `value` attribute, and its transcoding simply returns that object.

```
from uuid import UUID


class SimpleCustomValue:
    def __init__(self, id: UUID, date: date):
        self.id = id
        self.date = date

    def __eq__(self, other):
        return (
            isinstance(other, SimpleCustomValue) and
            self.id == other.id and self.date == other.date
        )
```

(continues on next page)

(continued from previous page)

```
class ComplexCustomValue:
    def __init__(self, value: SimpleCustomValue):
        self.value = value

    def __eq__(self, other):
        return (
            isinstance(other, ComplexCustomValue) and
            self.value == other.value
        )


class SimpleCustomValueAsDict(Transcoding):
    type = SimpleCustomValue
    name = "simple_custom_value"

    def encode(self, obj: SimpleCustomValue) -> dict:
        return {"id": obj.id, "date": obj.date}

    def decode(self, data: dict) -> SimpleCustomValue:
        assert isinstance(data, dict)
        return SimpleCustomValue(**data)


class ComplexCustomValueAsDict(Transcoding):
    type = ComplexCustomValue
    name = "complex_custom_value"

    def encode(self, obj: ComplexCustomValue) -> SimpleCustomValue:
        return obj.value

    def decode(self, data: SimpleCustomValue) -> ComplexCustomValue:
        assert isinstance(data, SimpleCustomValue)
        return ComplexCustomValue(data)
```

The custom value object transcodings can be registered with the transcoder.

```
transcoder.register(SimpleCustomValueAsDict())
transcoder.register(ComplexCustomValueAsDict())
```

We can now transcode an instance of `ComplexCustomValueAsDict`.

```
obj1 = ComplexCustomValue(
    SimpleCustomValue(
        id=UUID("b2723fe2c01a40d2875ea3aac6a09ff5"),
        date=date(2000, 2, 20)
    )
)

data = transcoder.encode(obj1)
copy = transcoder.decode(data)
assert copy == obj1
```

As you can see from the bytes representation below, the transcoder puts the return value of each transcoding's `encode()` method in a Python `dict` that has two values `_data_` and `_type_`. The `_data_` value is the return value of the transcoding's `encode()` method, and the `_type_` value is the name of the transcoding. For this reason, it is necessary to avoid defining model objects to have a Python `dict` that has only two attributes `_data_`

and _type_, and avoid defining transcodings that return such a thing.

```
expected_data = (
    b'{"_type_": "complex_custom_value", "_data_": {"_type_": '
    b'"simple_custom_value", "_data_": {"id": {"_type_": '
    b'"uuid_hex", "_data_": "b2723fe2c01a40d2875ea3aac6a09ff5"},'
    b' "date": {"_type_": "date_iso", "_data_": "2000-02-20"}'
    b'}}}'
)
assert data == expected_data
```

1.6.3 Stored event objects

A stored event object is a common object type that can be used to represent domain event objects of different types. By using a common object for the representation of different types of domain events objects, the domain event objects can be stored and retrieved in a standard way.

The library's *StoredEvent* class is a Python frozen dataclass that can be used to hold information about a domain event object between it being serialised and being recorded in a datastore, and between it be retrieved from a datastore from an aggregate sequence and being deserialised as a domain event object.

```
from uuid import uuid4

from eventsourcing.persistence import StoredEvent

stored_event = StoredEvent(
    originator_id=uuid4(),
    originator_version=1,
    state="{}",
    topic="eventsourcing.model:DomainEvent",
)
```

1.6.4 Mapper

A mapper maps between domain event objects and stored event objects. It brings together a *transcoder*, and optionally a *cipher* and a *compressor*. It is used by an *event store*.

The library's *Mapper* class must be constructed with a *transcoder* object.

```
from eventsourcing.persistence import Mapper

mapper = Mapper(transcoder=transcoder)
```

The *from_domain_event()* method of the mapper object converts *DomainEvent* objects to *StoredEvent* objects.

```
from eventsourcing.domain import DomainEvent, TZINFO

domain_event1 = DomainEvent(
    originator_id = id1,
    originator_version = 1,
    timestamp = datetime.now(tz=TZINFO),
)
```

(continues on next page)

(continued from previous page)

```
stored_event1 = mapper.from_domain_event(domain_event1)
assert isinstance(stored_event1, StoredEvent)
```

The *to_domain_event()* method of the mapper object converts *StoredEvent* objects to *DomainEvent* objects.

```
assert mapper.to_domain_event(stored_event1) == domain_event1
```

1.6.5 Encryption

Using a cryptographic cipher with your mapper will make the state of your application encrypted "at rest" and "on the wire".

Without encryption, the state of the domain event will be visible in the recorded stored events in your database. For example, the timestamp of the domain event in the example above (domain_event1) is visible in the stored event (stored_event1).

```
assert domain_event1.timestamp.isoformat() in str(stored_event1.state)
```

The library's *AESCipher* class can be used to cryptographically encode and decode the state of stored events. It must be constructed with a cipher key. The class method *create_key()* can be used to generate a cipher key. The AES cipher key must be either 16, 24, or 32 bytes long. Please note, the same cipher key must be used to decrypt stored events as that which was used to encrypt stored events.

```
from eventsourcing.cipher import AESCipher

key = AESCipher.create_key(num_bytes=32) # 16, 24, or 32
cipher = AESCipher(cipher_key=key)

mapper = Mapper(
    transcoder=transcoder,
    cipher=cipher,
)

stored_event1 = mapper.from_domain_event(domain_event1)
assert isinstance(stored_event1, StoredEvent)
assert mapper.to_domain_event(stored_event1) == domain_event1
```

With encryption, the state of the domain event will not be visible in the stored event. This feature can be used to implement "application-level encryption" in an event-sourced application.

```
assert domain_event1.timestamp.isoformat() not in str(stored_event1.state)
```

The library's *AESCipher* class uses the AES cipher from the PyCryptodome library in GCM mode. AES is a very fast and secure symmetric block cipher, and is the de facto standard for symmetric encryption. Galois/Counter Mode (GCM) is a mode of operation for symmetric block ciphers that is designed to provide both data authenticity and confidentiality, and is widely adopted for its performance.

The mapper expects an instance of the abstract base class Cipher, and *AESCipher* implements this abstract base class, so if you want to use another cipher strategy simply implement the base class.

1.6.6 Compression

A compressor can be used to reduce the size of stored events.

The library's *ZlibCompressor* class can be used to compress and decompress the state of stored events. The size of the state of a compressed and encrypted stored event will be less than or equal to the size of the state of a stored event that is encrypted but not compressed.

```
from eventsourcing.compressor import ZlibCompressor

compressor = ZlibCompressor()

mapper = Mapper(
    transcoder=transcoder,
    cipher=cipher,
    compressor=compressor,
)

stored_event2 = mapper.from_domain_event(domain_event1)
assert mapper.to_domain_event(stored_event2) == domain_event1

assert len(stored_event2.state) <= len(stored_event1.state)
```

The library's *ZlibCompressor* class uses Python's `zlib` module.

The mapper expects an instance of the abstract base class `Compressor`, and *ZlibCompressor* implements this abstract base class, so if you want to use another compression strategy simply implement the base class.

1.6.7 Notification objects

Event notifications are used to propagate the state of an event sourced application in a reliable way. The stored events can be positioned in a "total order" by giving each a new domain event a notification ID that is higher that any previously recorded event. By recording the domain events atomically with their notification IDs, there will never be a domain event that is not available to be passed as a message across a network, and there will never be a message passed across a network that doesn't correspond to a recorded event. This solves the "dual writing" problem that occurs when separately a domain model is updated and then a message is put on a message queue.

The library's *Notification* class is a Python frozen dataclass that can be used to hold information about a domain event object when being transmitted as an item in a section of a *notification log*. It will be returned when selecting event notifications from a *recorder*, and presented in an application by a *notification log*.

```
from uuid import uuid4

from eventsourcing.persistence import Notification

stored_event = Notification(
    id=123,
    originator_id=uuid4(),
    originator_version=1,
    state="{}",
    topic="eventsourcing.model:DomainEvent",
)
```

1.6.8 Tracking objects

A tracking object can be used to encapsulate the position of an event notification in an upstream application's notification log. A tracking object can be passed into a process recorder along with new stored event objects, and recorded atomically with those objects. By ensuring the uniqueness of recorded tracking objects, we can ensure that a domain event notification is never processed twice. By recording the position of the last event notification that has been

processed, we can ensure to resume processing event notifications at the correct position. This constructs "exactly once" semantics when processing event notifications, by solving the "dual writing" problem that occurs when separately an event notification is consumed from a message queue with updates made to materialized view, and then an acknowledgement is sent back to the message queue.

The library's *Tracking* class is a Python frozen dataclass that can be used to hold the notification ID of a notification that has been processed.

```
from uuid import uuid4

from eventsourcing.persistence import Tracking

tracking = Tracking(
    notification_id=123,
    application_name="bounded_context1",
)
```

1.6.9 Recorder

A recorder adapts a database management system for the purpose of recording stored events. It is used by an *event store*.

The library's *Recorder* class is an abstract base for concrete recorder classes that will insert stored event objects in a particular datastore.

There are three flavours of recorder: "aggregate recorders" are the simplest and simply store domain events in aggregate sequences; "application recorders" extend aggregate recorders by storing domain events with a total order; "process recorders" extend application recorders by supporting the recording of domain events atomically with "tracking" objects that record the position in a total ordering of domain events that is being processed. The "aggregate recorder" can be used for storing snapshots.

The library includes in its *sqlite* module recorder classes for SQLite that use the Python `sqlite3` module, and in its *postgres* module recorders for PostgreSQL that use the third party `psycopg2` module.

Recorder classes are conveniently constructed by using an *infrastructure factory*. For illustrative purposes, the direct use of the library's SQLite recorders is shown below. The other persistence modules follow a similar naming scheme and pattern of use.

```
from eventsourcing.sqlite import SQLiteAggregateRecorder
from eventsourcing.sqlite import SQLiteApplicationRecorder
from eventsourcing.sqlite import SQLiteProcessRecorder
from eventsourcing.sqlite import SQLiteDatastore

datastore = SQLiteDatastore(db_name=":memory:")
aggregate_recorder = SQLiteAggregateRecorder(datastore, "snapshots")
aggregate_recorder.create_table()

application_recorder = SQLiteApplicationRecorder(datastore)
application_recorder.create_table()

datastore = SQLiteDatastore(db_name=":memory:")
process_recorder = SQLiteProcessRecorder(datastore)
process_recorder.create_table()
```

The library also includes in the *popo* module recorders that use "plain old Python objects", which simply keep stored events in a data structure in memory, and provides the fastest alternative for rapid development of event sourced applications (~4x faster than using SQLite, and ~20x faster than using PostgreSQL).

Recorders compatible with this version of the library for popular ORMs such as SQLAlchemy and Django, specialist event stores such as EventStoreDB and AxonDB, and NoSQL databases such as DynamoDB and MongoDB are forthcoming.

1.6.10 Event store

An event store provides a common interface for storing and retrieving domain event objects. It combines a *mapper* and a *recorder*, so that domain event objects can be converted to stored event objects and then stored event objects can be recorded in a datastore.

The library's *EventStore* class must be constructed with a *mapper* and a *recorder*.

The *EventStore* has an object method *put()* which can be used to store a list of new domain event objects. If any of these domain event objects conflict with any already existing domain event object (because they have the same aggregate ID and version number), an exception will be raised and none of the new events will be stored.

The *EventStore* has an object method *get()* which can be used to get a list of domain event objects. Only the `originator_id` argument is required, which is the ID of the aggregate for which existing events are wanted. The arguments `gt`, `lte`, `limit`, and `desc` condition the selection of events to be greater than a particular version number, less then or equal to a particular version number, limited in number, or selected in a descending fashion. The selection is by default ascending, unlimited, and otherwise unrestricted such that all the previously stored domain event objects for a particular aggregate will be returned in the order in which they were created.

```
from eventsourcing.persistence import EventStore

event_store = EventStore(
    mapper=mapper,
    recorder=application_recorder,
)

event_store.put([domain_event1])

domain_events = list(event_store.get(id1))
assert domain_events == [domain_event1]
```

1.6.11 Infrastructure factory

An infrastructure factory helps with the construction of the persistence infrastructure objects mentioned above. By reading and responding to particular environment variables, the persistence infrastructure of an event-sourced application can be easily *configured in different ways* at different times.

The library's *InfrastructureFactory* class is a base class for concrete infrastructure factories that help with the construction of persistence objects that use a particular database in a particular way.

The class method *construct()* will, by default, construct the library's "plain old Python objects" infrastructure *Factory*, which uses recorders that simply keep stored events in a data structure in memory (see *eventsourcing.popo*).

```
from eventsourcing.persistence import InfrastructureFactory

factory = InfrastructureFactory.construct()

recorder = factory.application_recorder()
mapper = factory.mapper(transcoder=transcoder)
event_store = factory.event_store(
```

(continues on next page)

(continued from previous page)

```
    mapper=mapper,
    recorder=recorder,
)

event_store.put([domain_event1])
stored_events = list(event_store.get(id1))
assert stored_events == [domain_event1]
```

The optional environment variables `COMPRESSOR_TOPIC`, `CIPHER_KEY`, and `CIPHER_TOPIC` may be used to enable compression and encryption of stored events when using POPO infrastructure.

1.6.12 SQLite

The module *`eventsourcing.sqlite`* supports storing events in SQLite.

The library's SQLite *`Factory`* uses various environment variables to control the construction and configuration of its persistence infrastructure.

The environment variable `SQLITE_DBNAME` is required to set the name of a database, normally a file path, but the special name `:memory:` can be used to create an in-memory database.

The optional environment variable `SQLITE_LOCK_TIMEOUT` may be used to adjust the SQLite timeout value. A file-based SQLite database will have its journal mode set to use write-ahead logging (WAL), which allows reading to proceed concurrently reading and writing. Writing is serialised with a lock. Setting this value to a positive number of seconds will cause attempts to lock the SQLite database for writing to timeout after that duration. By default this value is 5 (seconds).

The optional environment variables `COMPRESSOR_TOPIC`, `CIPHER_KEY`, and `CIPHER_TOPIC` may be used to enable compression and encryption of stored events.

The optional environment variable `CREATE_TABLE` may be control whether database tables are created. If the tables already exist, the `CREATE_TABLE` may be set to a "false" value (`"n"`, `"no"`, `"f"`, `"false"`, `"off"`, or `"0"`). This value is by default "true" which is normally okay because the tables are created only if they do not exist.

```
import os

os.environ["INFRASTRUCTURE_FACTORY"] = "eventsourcing.sqlite:Factory"
os.environ["SQLITE_DBNAME"] = ":memory:"
os.environ["SQLITE_LOCK_TIMEOUT"] = "10"


factory = InfrastructureFactory.construct()

recorder = factory.application_recorder()
mapper = factory.mapper(transcoder=transcoder)
event_store = factory.event_store(
    mapper=mapper,
    recorder=recorder,
)

event_store.put([domain_event1])
stored_events = list(event_store.get(id1))
assert stored_events == [domain_event1]
```

1.6.13 PostgreSQL

The module *eventsourcing.postgres* supports storing events in PostgresSQL.

The library's PostgreSQL *Factory* uses various environment variables to control the construction and configuration of its persistence infrastructure.

The environment variables POSTGRES_DBNAME, POSTGRES_HOST, POSTGRES_PORT, POSTGRES_USER, and POSTGRES_PASSWORD are required to set the name of a database, the database server's host name and port number, and the database user name and password.

The optional environment variable POSTGRES_CONN_MAX_AGE is used to control the length of time in seconds before a connection is closed. By default this value is not set, and connections will be reused indefinitely (or until an operational database error is encountered). If this value is set to a positive integer, the connection will be closed after this number of seconds from the time it was created, but only when the connection is idle. If this value if set to zero, each connection will only be used for one transaction. Setting this value to an empty string has the same effect as not setting this value. Setting this value to any other value will cause an environment error exception to be raised. If your database terminates idle connections after some time, you should set POSTGRES_CONN_MAX_AGE to a lower value, so that attempts are not made to use connections that have been terminated by the database server.

The optional environment variable POSTGRES_PRE_PING may be used to enable pessimistic disconnection handling. Setting this to a "true" value ("y", "yes", "t", "true", "on", or "1") means database connections will be checked that they are usable before executing statements, and database connections remade if the connection is not usable. This value is by default "false", meaning connections will not be checked before they are reused. Enabling this option will incur a small impact on performance.

The optional environment variable POSTGRES_LOCK_TIMEOUT may be used to enable a timeout on acquiring an 'EXCLUSIVE' mode table lock when inserting stored events. To avoid interleaving of inserts when writing events, an 'EXCLUSIVE' mode table lock is acquired when inserting events. This effectively serialises writing events. It prevents concurrent transactions interleaving inserts, which would potentially cause notification log readers that are tailing the application notification log to miss event notifications. Reading from the table can proceed concurrently with other readers and writers, since selecting acquires an 'ACCESS SHARE' lock which does not block and is not blocked by the 'EXCLUSIVE' lock. This issue of interleaving inserts by concurrent writers is not exhibited by SQLite, which supports concurrent readers when its journal mode is set to use write ahead logging. By default, this timeout has the value of 0 seconds, which means attempts to acquire the lock will not timeout. Setting this value to a positive integer number of seconds will cause attempt to obtain this lock to timeout after that duration has passed. The lock will be released when the transaction ends.

The optional environment variable POSTGRES_IDLE_IN_TRANSACTION_SESSION_TIMEOUT may be used to timeout sessions that are idle in a transaction. If a transaction cannot be ended for some reason, perhaps because the database server cannot be reached, the transaction may remain in an idle state and any locks will continue to be held. By timing out the session, transactions will be ended, locks will be released, and the connection slot will be freed. By default, this timeout has the value of 0 seconds, which means sessions in an idle transaction will not timeout. Setting this value to a positive integer number of seconds will cause sessions in an idle transaction to timeout after that duration has passed.

The optional environment variables COMPRESSOR_TOPIC, CIPHER_KEY, and CIPHER_TOPIC may be used to enable compression and encryption of stored events.

The optional environment variable CREATE_TABLE may be control whether database tables are created. If the tables already exist, the CREATE_TABLE may be set to a "false" value ("n", "no", "f", "false", "off", or "0"). This value is by default "true" which is normally okay because the tables are created only if they do not exist.

```
import os

os.environ["INFRASTRUCTURE_FACTORY"] = "eventsourcing.postgres:Factory"
os.environ["POSTGRES_DBNAME"] = "eventsourcing"
os.environ["POSTGRES_HOST"] = "127.0.0.1"
```

(continues on next page)

(continued from previous page)

```
os.environ["POSTGRES_PORT"] = "5432"
os.environ["POSTGRES_USER"] = "eventsourcing"
os.environ["POSTGRES_PASSWORD"] = "eventsourcing"
os.environ["POSTGRES_CONN_MAX_AGE"] = "10"
os.environ["POSTGRES_PRE_PING"] = "y"
os.environ["POSTGRES_LOCK_TIMEOUT"] = "5"
os.environ["POSTGRES_IDLE_IN_TRANSACTION_SESSION_TIMEOUT"] = "5"
```

```
factory = InfrastructureFactory.construct()

recorder = factory.application_recorder()
mapper = factory.mapper(transcoder=transcoder)
event_store = factory.event_store(
    mapper=mapper,
    recorder=recorder,
)

event_store.put([domain_event1])
stored_events = list(event_store.get(id1))
assert stored_events == [domain_event1]
```

1.6.14 Classes

class eventsourcing.persistence.**Transcoding**

Bases: abc.ABC

Abstract base class for custom transcodings.

type

Object type of transcoded object.

name

Name of transcoding.

encode(*obj: Any*) → Any

Encodes given object.

decode(*data: Any*) → Any

Decodes encoded object.

class eventsourcing.persistence.**Transcoder**

Bases: abc.ABC

Abstract base class for transcoders.

__init__() → None

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

register(*transcoding: eventsourcing.persistence.Transcoding*) → None

Registers given transcoding with the transcoder.

encode(*obj: Any*) → bytes

Encodes obj as bytes.

decode(*data: bytes*) → Any

Decodes obj from bytes.

class eventsourcing.persistence.**JSONTranscoder**

Bases: *eventsourcing.persistence.Transcoder*

Extensible transcoder that uses the Python `json` module.

__init__() → None
Initialize self. See help(type(self)) for accurate signature.

encode(*obj: Any*) → bytes
Encodes given object as a bytes array.

decode(*data: bytes*) → Any
Decodes bytes array as previously encoded object.

class eventsourcing.persistence.**UUIDAsHex**
Bases: *eventsourcing.persistence.Transcoding*

Transcoding that represents `UUID` objects as hex values.

type
alias of `uuid.UUID`

encode(*obj: uuid.UUID*) → str
Encodes given object.

decode(*data: str*) → uuid.UUID
Decodes encoded object.

class eventsourcing.persistence.**DecimalAsStr**
Bases: *eventsourcing.persistence.Transcoding*

Transcoding that represents `Decimal` objects as strings.

type
alias of `decimal.Decimal`

encode(*obj: decimal.Decimal*) → str
Encodes given object.

decode(*data: str*) → decimal.Decimal
Decodes encoded object.

class eventsourcing.persistence.**DatetimeAsISO**
Bases: *eventsourcing.persistence.Transcoding*

Transcoding that represents `datetime` objects as ISO strings.

type
alias of `datetime.datetime`

encode(*obj: datetime.datetime*) → str
Encodes given object.

decode(*data: str*) → datetime.datetime
Decodes encoded object.

class eventsourcing.persistence.**StoredEvent**(*originator_id: uuid.UUID*, *originator_version: int*, *topic: str*, *state: bytes*)
Bases: `object`

Frozen dataclass that represents *DomainEvent* objects, such as aggregate *Event* objects and *Snapshot* objects.

Constructor parameters:

Parameters

- **originator_id** (*UUID*) – ID of the originating aggregate

- **originator_version** (*int*) – version of the originating aggregate
- **topic** (*str*) – topic of the domain event object class
- **state** (*bytes*) – serialised state of the domain event object

class eventsourcing.persistence.**Compressor**
Bases: abc.ABC

Base class for compressors.

compress(*data: bytes*) → bytes
Compress bytes.

decompress(*data: bytes*) → bytes
Decompress bytes.

class eventsourcing.persistence.**Cipher**(*cipher_key: str*)
Bases: abc.ABC

Base class for ciphers.

__init__(*cipher_key: str*)
Initialises cipher with given key.

encrypt(*plaintext: bytes*) → bytes
Return ciphertext for given plaintext.

decrypt(*ciphertext: bytes*) → bytes
Return plaintext for given ciphertext.

class eventsourcing.persistence.**Mapper**(*transcoder: eventsourcing.persistence.Transcoder*, *compressor: Optional[eventsourcing.persistence.Compressor] = None*, *cipher: Optional[eventsourcing.persistence.Cipher] = None*)
Bases: typing.Generic

Converts between domain event objects and *StoredEvent* objects.

Uses a *Transcoder*, and optionally a cryptographic cipher and compressor.

__init__(*transcoder: eventsourcing.persistence.Transcoder*, *compressor: Optional[eventsourcing.persistence.Compressor] = None*, *cipher: Optional[eventsourcing.persistence.Cipher] = None*)
Initialize self. See help(type(self)) for accurate signature.

from_domain_event(*domain_event: TDomainEvent*) → eventsourcing.persistence.StoredEvent
Converts the given domain event to a *StoredEvent* object.

to_domain_event(*stored: eventsourcing.persistence.StoredEvent*) → TDomainEvent
Converts the given *StoredEvent* to a domain event object.

exception eventsourcing.persistence.**RecordConflictError**
Bases: Exception

Legacy exception, replaced with IntegrityError.

exception eventsourcing.persistence.**PersistenceError**
Bases: Exception

The base class of the other exceptions in this module.

Exception class names follow https://www.python.org/dev/peps/pep-0249/#exceptions

exception eventsourcing.persistence.**InterfaceError**
Bases: *eventsourcing.persistence.PersistenceError*

Exception raised for errors that are related to the database interface rather than the database itself.

exception eventsourcing.persistence.**DatabaseError**
Bases: *eventsourcing.persistence.PersistenceError*

Exception raised for errors that are related to the database.

exception eventsourcing.persistence.**DataError**
Bases: *eventsourcing.persistence.DatabaseError*

Exception raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc.

exception eventsourcing.persistence.**OperationalError**
Bases: *eventsourcing.persistence.DatabaseError*

Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc.

exception eventsourcing.persistence.**IntegrityError**
Bases: *eventsourcing.persistence.DatabaseError*, *eventsourcing.persistence.RecordConflictError*

Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails.

exception eventsourcing.persistence.**InternalError**
Bases: *eventsourcing.persistence.DatabaseError*

Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc.

exception eventsourcing.persistence.**ProgrammingError**
Bases: *eventsourcing.persistence.DatabaseError*

Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc.

exception eventsourcing.persistence.**NotSupportedError**
Bases: *eventsourcing.persistence.DatabaseError*

Exception raised in case a method or database API was used which is not supported by the database, e.g. calling the rollback() method on a connection that does not support transaction or has transactions turned off.

class eventsourcing.persistence.**Recorder**
Bases: abc.ABC

Abstract base class for stored event recorders.

class eventsourcing.persistence.**AggregateRecorder**
Bases: *eventsourcing.persistence.Recorder*

Abstract base class for recorders that record and retrieve stored events for domain model aggregates.

insert_events(*stored_events: List[eventsourcing.persistence.StoredEvent], **kwargs*) → None
Writes stored events into database.

select_events(*originator_id: uuid.UUID*, *gt: Optional[int] = None*, *lte: Optional[int] = None*, *desc: bool = False*, *limit: Optional[int] = None*) → List[eventsourcing.persistence.StoredEvent]
Reads stored events from database.

class eventsourcing.persistence.**Notification**(*originator_id: uuid.UUID, originator_version: int, topic: str, state: bytes, id: int*)

Bases: *eventsourcing.persistence.StoredEvent*

Frozen dataclass that represents domain event notifications.

class eventsourcing.persistence.**ApplicationRecorder**

Bases: *eventsourcing.persistence.AggregateRecorder*

Abstract base class for recorders that record and retrieve stored events for domain model aggregates.

Extends the behaviour of aggregate recorders by recording aggregate events in a total order that allows the stored events also to be retrieved as event notifications.

select_notifications(*start: int, limit: int*) → List[eventsourcing.persistence.Notification]

Returns a list of event notifications from 'start', limited by 'limit'.

max_notification_id() → int

Returns the maximum notification ID.

class eventsourcing.persistence.**ProcessRecorder**

Bases: *eventsourcing.persistence.ApplicationRecorder*

Abstract base class for recorders that record and retrieve stored events for domain model aggregates.

Extends the behaviour of applications recorders by recording aggregate events with tracking information that records the position of a processed event notification in a notification log.

max_tracking_id(*application_name: str*) → int

Returns the last recorded notification ID from given application.

class eventsourcing.persistence.**EventStore**(*mapper: eventsourcing.persistence.Mapper[~TDomainEvent][TDomainEvent], recorder: eventsourcing.persistence.AggregateRecorder*)

Bases: typing.Generic

Stores and retrieves domain events.

__init__(*mapper: eventsourcing.persistence.Mapper[~TDomainEvent][TDomainEvent], recorder: eventsourcing.persistence.AggregateRecorder*)

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

put(*events: List[TDomainEvent], **kwargs*) → None

Stores domain events in aggregate sequence.

get(*originator_id: uuid.UUID, gt: Optional[int] = None, lte: Optional[int] = None, desc: bool = False, limit: Optional[int] = None*) → Iterator[TDomainEvent]

Retrieves domain events from aggregate sequence.

class eventsourcing.persistence.**InfrastructureFactory**(*application_name: str, env: Mapping[KT, VT_co]*)

Bases: abc.ABC

Abstract base class for infrastructure factories.

classmethod construct(*application_name: str = '', env: Optional[Mapping[KT, VT_co]] = None*) → eventsourcing.persistence.InfrastructureFactory

Constructs concrete infrastructure factory for given named application. Reads and resolves infrastructure factory class topic from environment variable 'INFRASTRUCTURE_FACTORY'.

__init__(*application_name: str, env: Mapping[KT, VT_co]*)

Initialises infrastructure factory object with given application name.

getenv(*key: str*, *default: Optional[str] = None*, *application_name: str = ''*) → Optional[str]
Returns value of environment variable defined by given key.

mapper(*transcoder: eventsourcing.persistence.Transcoder*, *application_name: str = ''*) → eventsourcing.persistence.Mapper
Constructs a mapper.

cipher(*application_name: str*) → Optional[eventsourcing.persistence.Cipher]
Reads environment variables 'CIPHER_TOPIC' and 'CIPHER_KEY' to decide whether or not to construct a cipher.

compressor(*application_name: str*) → Optional[eventsourcing.persistence.Compressor]
Reads environment variable 'COMPRESSOR_TOPIC' to decide whether or not to construct a compressor.

static event_store(***kwargs*) → eventsourcing.persistence.EventStore
Constructs an event store.

aggregate_recorder(*purpose: str = 'events'*) → eventsourcing.persistence.AggregateRecorder
Constructs an aggregate recorder.

application_recorder() → eventsourcing.persistence.ApplicationRecorder
Constructs an application recorder.

process_recorder() → eventsourcing.persistence.ProcessRecorder
Constructs a process recorder.

is_snapshotting_enabled() → bool
Decides whether or not snapshotting is enabled by reading environment variable 'IS_SNAPSHOTTING_ENABLED'. Snapshotting is not enabled by default.

class eventsourcing.persistence.**Tracking**(*application_name: str*, *notification_id: int*)
Bases: object

Frozen dataclass representing the position of a domain event *Notification* in an application's notification log.

class eventsourcing.popo.**POPOAggregateRecorder**
Bases: *eventsourcing.persistence.AggregateRecorder*

__init__() → None
Initialize self. See help(type(self)) for accurate signature.

insert_events(*stored_events: List[eventsourcing.persistence.StoredEvent], **kwargs*) → None
Writes stored events into database.

select_events(*originator_id: uuid.UUID*, *gt: Optional[int] = None*, *lte: Optional[int] = None*, *desc: bool = False*, *limit: Optional[int] = None*) → List[eventsourcing.persistence.StoredEvent]
Reads stored events from database.

class eventsourcing.popo.**POPOApplicationRecorder**
Bases: *eventsourcing.persistence.ApplicationRecorder*, *eventsourcing.popo.POPOAggregateRecorder*

select_notifications(*start: int*, *limit: int*) → List[eventsourcing.persistence.Notification]
Returns a list of event notifications from 'start', limited by 'limit'.

max_notification_id() → int
Returns the maximum notification ID.

class eventsourcing.popo.**POPOProcessRecorder**
Bases: *eventsourcing.persistence.ProcessRecorder*, *eventsourcing.popo.POPOApplicationRecorder*

__init__() → None
Initialize self. See help(type(self)) for accurate signature.

max_tracking_id(*application_name: str*) → int
Returns the last recorded notification ID from given application.

class eventsourcing.popo.**Factory**(*application_name: str, env: Mapping[KT, VT_co]*)
Bases: *eventsourcing.persistence.InfrastructureFactory*

aggregate_recorder(*purpose: str = 'events'*) → eventsourcing.persistence.AggregateRecorder
Constructs an aggregate recorder.

application_recorder() → eventsourcing.persistence.ApplicationRecorder
Constructs an application recorder.

process_recorder() → eventsourcing.persistence.ProcessRecorder
Constructs a process recorder.

class eventsourcing.sqlite.**SQLiteAggregateRecorder**(*datastore: eventsourcing.sqlite.SQLiteDatastore, events_table_name: str = 'stored_events'*)
Bases: *eventsourcing.persistence.AggregateRecorder*

__init__(*datastore: eventsourcing.sqlite.SQLiteDatastore, events_table_name: str = 'stored_events'*)
Initialize self. See help(type(self)) for accurate signature.

insert_events(*stored_events: List[eventsourcing.persistence.StoredEvent], **kwargs*) → None
Writes stored events into database.

select_events(*originator_id: uuid.UUID, gt: Optional[int] = None, lte: Optional[int] = None, desc: bool = False, limit: Optional[int] = None*) → List[eventsourcing.persistence.StoredEvent]
Reads stored events from database.

class eventsourcing.sqlite.**SQLiteApplicationRecorder**(*datastore: eventsourcing.sqlite.SQLiteDatastore, events_table_name: str = 'stored_events'*)
Bases: *eventsourcing.sqlite.SQLiteAggregateRecorder*, *eventsourcing.persistence.ApplicationRecorder*

__init__(*datastore: eventsourcing.sqlite.SQLiteDatastore, events_table_name: str = 'stored_events'*)
Initialize self. See help(type(self)) for accurate signature.

select_notifications(*start: int, limit: int*) → List[eventsourcing.persistence.Notification]
Returns a list of event notifications from 'start', limited by 'limit'.

max_notification_id() → int
Returns the maximum notification ID.

class eventsourcing.sqlite.**SQLiteProcessRecorder**(*datastore: eventsourcing.sqlite.SQLiteDatastore, events_table_name: str = 'stored_events'*)
Bases: *eventsourcing.sqlite.SQLiteApplicationRecorder*, *eventsourcing.persistence.ProcessRecorder*

__init__(*datastore: eventsourcing.sqlite.SQLiteDatastore, events_table_name: str = 'stored_events'*)
Initialize self. See help(type(self)) for accurate signature.

max_tracking_id(*application_name: str*) → int
Returns the last recorded notification ID from given application.

class eventsourcing.sqlite.**Factory**(*application_name: str, env: Mapping[KT, VT_co]*)
Bases: *eventsourcing.persistence.InfrastructureFactory*

__init__(*application_name: str, env: Mapping[KT, VT_co]*)
Initialises infrastructure factory object with given application name.

aggregate_recorder(*purpose: str = 'events'*) → eventsourcing.persistence.AggregateRecorder
Constructs an aggregate recorder.

application_recorder() → eventsourcing.persistence.ApplicationRecorder
Constructs an application recorder.

process_recorder() → eventsourcing.persistence.ProcessRecorder
Constructs a process recorder.

class eventsourcing.postgres.**PostgresAggregateRecorder**(*datastore: eventsourcing.postgres.PostgresDatastore*, *events_table_name: str*)
Bases: *eventsourcing.persistence.AggregateRecorder*

__init__(*datastore: eventsourcing.postgres.PostgresDatastore*, *events_table_name: str*)
Initialize self. See help(type(self)) for accurate signature.

insert_events(*stored_events: List[eventsourcing.persistence.StoredEvent], **kwargs*) → None
Writes stored events into database.

select_events(*originator_id: uuid.UUID*, *gt: Optional[int] = None*, *lte: Optional[int] = None*, *desc: bool = False*, *limit: Optional[int] = None*) → List[eventsourcing.persistence.StoredEvent]
Reads stored events from database.

class eventsourcing.postgres.**PostgresApplicationRecorder**(*datastore: eventsourcing.postgres.PostgresDatastore*, *events_table_name: str = 'stored_events'*)
Bases: *eventsourcing.postgres.PostgresAggregateRecorder*, *eventsourcing.persistence.ApplicationRecorder*

__init__(*datastore: eventsourcing.postgres.PostgresDatastore*, *events_table_name: str = 'stored_events'*)
Initialize self. See help(type(self)) for accurate signature.

select_notifications(*start: int*, *limit: int*) → List[eventsourcing.persistence.Notification]
Returns a list of event notifications from 'start', limited by 'limit'.

max_notification_id() → int
Returns the maximum notification ID.

class eventsourcing.postgres.**PostgresProcessRecorder**(*datastore: eventsourcing.postgres.PostgresDatastore*, *events_table_name: str*, *tracking_table_name: str*)
Bases: *eventsourcing.postgres.PostgresApplicationRecorder*, *eventsourcing.persistence.ProcessRecorder*

__init__(*datastore: eventsourcing.postgres.PostgresDatastore*, *events_table_name: str*, *tracking_table_name: str*)
Initialize self. See help(type(self)) for accurate signature.

max_tracking_id(*application_name: str*) → int
Returns the last recorded notification ID from given application.

class eventsourcing.postgres.**Factory**(*application_name: str, env: Mapping[KT, VT_co]*)
Bases: *eventsourcing.persistence.InfrastructureFactory*

__init__(*application_name: str, env: Mapping[KT, VT_co]*)
Initialises infrastructure factory object with given application name.

aggregate_recorder(*purpose: str = 'events'*) → eventsourcing.persistence.AggregateRecorder
Constructs an aggregate recorder.

application_recorder() → eventsourcing.persistence.ApplicationRecorder
Constructs an application recorder.

process_recorder() → eventsourcing.persistence.ProcessRecorder
Constructs a process recorder.

1.7 system — Event-driven systems

This module shows how *event-sourced applications* can be combined to make an event driven system.

this page is under development — please check back soon

1.7.1 System of applications

The library's system class…

```
from eventsourcing.system import System
```

```
from dataclasses import dataclass
from uuid import uuid4

from eventsourcing.domain import Aggregate, AggregateCreated, AggregateEvent


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

    @classmethod
    def create(cls):
        return cls._create(
            event_class=cls.Created,
            id=uuid4(),
        )

    class Created(AggregateCreated):
        pass

    def make_it_so(self, what):
        self.trigger_event(self.SomethingHappened, what=what)

    class SomethingHappened(AggregateEvent):
        what: str

        def apply(self, world):
            world.history.append(self.what)
```

Now let's define an application...

```
from eventsourcing.application import Application


class WorldsApplication(Application):

    def create_world(self):
        world = World.create()
        self.save(world)
        return world.id

    def make_it_so(self, world_id, what):
        world = self.repository.get(world_id)
        world.make_it_so(what)
        self.save(world)

    def get_world_history(self, world_id):
        world = self.repository.get(world_id)
        return list(world.history)
```

Now let's define an analytics application...

```
from uuid import uuid5, NAMESPACE_URL

class Counter(Aggregate):
    def __init__(self, **kwargs):
        super(Counter, self).__init__(**kwargs)
        self.count = 0

    @classmethod
    def create_id(cls, name):
        return uuid5(NAMESPACE_URL, f'/counters/{name}')

    @classmethod
    def create(cls, name):
        return cls._create(
            event_class=cls.Created,
            id=cls.create_id(name),
        )

    class Created(AggregateCreated):
        pass

    def increment(self):
        self.trigger_event(self.Incremented)

    class Incremented(AggregateEvent):
        def apply(self, counter):
            counter.count += 1
```

```
from eventsourcing.application import AggregateNotFound
from eventsourcing.system import ProcessApplication
from eventsourcing.dispatch import singledispatchmethod


class Counters(ProcessApplication):
```

(continues on next page)

(continued from previous page)

```
    def policy(self, domain_event, process_event):
        pass

    @singledispatchmethod
    def policy(self, domain_event, process_event):
        """Default policy"""

    @policy.register(World.SomethingHappened)
    def _(self, domain_event, process_event):
        what = domain_event.what
        counter_id = Counter.create_id(what)
        try:
            counter = self.repository.get(counter_id)
        except AggregateNotFound:
            counter = Counter.create(what)
        counter.increment()
        process_event.save(counter)

    def get_count(self, what):
        counter_id = Counter.create_id(what)
        try:
            counter = self.repository.get(counter_id)
        except AggregateNotFound:
            return 0
        return counter.count
```

```
system = System(pipes=[[WorldsApplication, Counters]])
```

1.7.2 Single-threaded runner

```
from eventsourcing.system import SingleThreadedRunner

runner= SingleThreadedRunner(system)
runner.start()
worlds = runner.get(WorldsApplication)
counters = runner.get(Counters)

world_id1 = worlds.create_world()
world_id2 = worlds.create_world()
world_id3 = worlds.create_world()

assert counters.get_count('dinosaurs') == 0
assert counters.get_count('trucks') == 0
assert counters.get_count('internet') == 0

worlds.make_it_so(world_id1, 'dinosaurs')
worlds.make_it_so(world_id2, 'dinosaurs')
worlds.make_it_so(world_id3, 'dinosaurs')

assert counters.get_count('dinosaurs') == 3
assert counters.get_count('trucks') == 0
assert counters.get_count('internet') == 0

worlds.make_it_so(world_id1, 'trucks')
```

(continues on next page)

(continued from previous page)

```
worlds.make_it_so(world_id2, 'trucks')

assert counters.get_count('dinosaurs') == 3
assert counters.get_count('trucks') == 2
assert counters.get_count('internet') == 0

worlds.make_it_so(world_id1, 'internet')

assert counters.get_count('dinosaurs') == 3
assert counters.get_count('trucks') == 2
assert counters.get_count('internet') == 1
```

1.7.3 Multi-threaded runner

```
from eventsourcing.system import MultiThreadedRunner

runner= MultiThreadedRunner(system)
runner.start()
worlds = runner.get(WorldsApplication)
counters = runner.get(Counters)

world_id1 = worlds.create_world()
world_id2 = worlds.create_world()
world_id3 = worlds.create_world()

worlds.make_it_so(world_id1, 'dinosaurs')
worlds.make_it_so(world_id2, 'dinosaurs')
worlds.make_it_so(world_id3, 'dinosaurs')

worlds.make_it_so(world_id1, 'trucks')
worlds.make_it_so(world_id2, 'trucks')

worlds.make_it_so(world_id1, 'internet')

from time import sleep

sleep(0.01)

assert counters.get_count('dinosaurs') == 3
assert counters.get_count('trucks') == 2
assert counters.get_count('internet') == 1
```

...

1.7.4 Classes

class eventsourcing.system.**ProcessEvent**(*tracking: Optional[eventsourcing.persistence.Tracking] = None*)

Bases: object

Keeps together a *Tracking* object, which represents the position of a domain event notification in the notification log of a particular application, and the new domain events that result from processing that notification.

__init__(*tracking: Optional[eventsourcing.persistence.Tracking] = None*)
Initalises the process event with the given tracking object.

save(**aggregates*) → None
Collects pending domain events from the given aggregate.

class eventsourcing.system.**Follower**
Bases: *eventsourcing.application.Application*

Extends the *Application* class by using a process recorder as its application recorder, by keeping track of the applications it is following, and pulling and processing new domain event notifications through its *policy()* method.

__init__() → None
Initialises an application with an *InfrastructureFactory*, a *Mapper*, an *ApplicationRecorder*, an *EventStore*, a *Repository*, and a *LocalNotificationLog*.

construct_recorder() → eventsourcing.persistence.ProcessRecorder
Constructs and returns a *ProcessRecorder* for the application to use as its application recorder.

follow(*name: str*, *log: eventsourcing.application.NotificationLog*) → None
Constructs a notification log reader and a mapper for the named application, and adds them to its collection of readers.

pull_and_process(*name: str*) → None
Pulls and processes unseen domain event notifications from the notification log reader of the names application.

Converts received event notifications to domain event objects, and then calls the *policy()* with a new *ProcessEvent* object which contains a *Tracking* object that keeps track of the name of the application and the position in its notification log from which the domain event notification was pulled. The policy will save aggregates to the process event object, using its *save()* method, which collects pending domain events using the aggregates' *collect_events()* method, and the process event object will then be recorded by calling the *record()* method.

policy(*domain_event: eventsourcing.domain.AggregateEvent*, *process_event: eventsourcing.system.ProcessEvent*) → None
Abstract domain event processing policy method. Must be implemented by event processing applications. When processing the given domain event, event processing applications must use the *save()* method of the given process event object (instead of the application's *save()* method) to collect pending events from changed aggregates, so that the new domain events will be recorded atomically with tracking information about the position of the given domain event's notification.

record(*process_event: eventsourcing.system.ProcessEvent*) → None
Records given process event in the application's process recorder.

class eventsourcing.system.**Promptable**
Bases: abc.ABC

Abstract base class for "promptable" objects.

receive_prompt(*leader_name: str*) → None
Receives the name of leader that has new domain event notifications.

class eventsourcing.system.**Leader**
Bases: *eventsourcing.application.Application*

Extends the *Application* class by also being responsible for keeping track of followers, and prompting followers when there are new domain event notifications to be pulled and processed.

__init__() → None
Initialises an application with an *InfrastructureFactory*, a *Mapper*, an *ApplicationRecorder*, an *EventStore*, a *Repository*, and a *LocalNotificationLog*.

lead(*follower: eventsourcing.system.Promptable*) → None
Adds given follower to a list of followers.

notify(*new_events: List[eventsourcing.domain.AggregateEvent]*) → None
Extends the application *notify()* method by calling *prompt_followers()* whenever new events have just been saved.

prompt_followers() → None
Prompts followers by calling their *receive_prompt()* methods with the name of the application.

class eventsourcing.system.**ProcessApplication**
Bases: *eventsourcing.system.Leader*, *eventsourcing.system.Follower*, abc.ABC

Base class for event processing applications that are both "leaders" and followers".

class eventsourcing.system.**System**(*pipes: Iterable[Iterable[Type[eventsourcing.application.Application]]]*)
Bases: object

Defines a system of applications.

__init__(*pipes: Iterable[Iterable[Type[eventsourcing.application.Application]]]*)
Initialize self. See help(type(self)) for accurate signature.

class eventsourcing.system.**Runner**(*system: eventsourcing.system.System*)
Bases: abc.ABC

Abstract base class for system runners.

__init__(*system: eventsourcing.system.System*)
Initialize self. See help(type(self)) for accurate signature.

start() → None
Starts the runner.

stop() → None
Stops the runner.

get(*cls: Type[A]*) → A
Returns an application instance for given application class.

exception eventsourcing.system.**RunnerAlreadyStarted**
Bases: Exception

Raised when runner is already started.

class eventsourcing.system.**SingleThreadedRunner**(*system: eventsourcing.system.System*)
Bases: *eventsourcing.system.Runner*, *eventsourcing.system.Promptable*

Runs a *System* in a single thread. A single threaded runner is a runner, and so implements the *start()*, *stop()*, and *get()* methods. A single threaded runner is also a *Promptable* object, and implements the *receive_prompt()* method by collecting prompted names.

__init__(*system: eventsourcing.system.System*)
Initialises runner with the given *System*.

start() → None
Starts the runner. The applications are constructed, and setup to lead and follow each other, according to the system definition. The followers are setup to follow the applications they follow (have a notification log reader with the notification log of the leader), and their leaders are setup to lead the runner itself (send prompts).

receive_prompt(*leader_name: str*) → None
Receives prompt by appending name of leader to list of prompted names. Unless this method has previously been called but not yet returned, it will then proceed to forward the prompts received to its application by calling the application's *pull_and_process()* method for each prompted name.

stop() → None
Stops the runner.

get(*cls: Type[A]*) → A
Returns an application instance for given application class.

class eventsourcing.system.**MultiThreadedRunner**(*system: eventsourcing.system.System*)
Bases: *eventsourcing.system.Runner*

Runs a *System* with a *MultiThreadedRunnerThread* for each follower in the system definition. It is a runner, and so implements the *start()*, *stop()*, and *get()* methods.

__init__(*system: eventsourcing.system.System*)
Initialises runner with the given *System*.

start() → None
Starts the runner.

A multi-threaded runner thread is started for each 'follower' application in the system, and constructs an instance of each non-follower leader application in the system. The followers are then setup to follow the applications they follow (have a notification log reader with the notification log of the leader), and their leaders are setup to lead the follower's thead (send prompts).

stop() → None
Stops the runner.

get(*cls: Type[A]*) → A
Returns an application instance for given application class.

class eventsourcing.system.**MultiThreadedRunnerThread**(*app_class: Type[eventsourcing.system.Follower], is_stopping: threading.Event*)
Bases: *eventsourcing.system.Promptable*, threading.Thread

Runs one process application for a *MultiThreadedRunner*.

A multi-threaded runner thread is a *Promptable* object, and implements the *receive_prompt()* method by collecting prompted names and setting its threading event 'is_prompted'.

A multi-threaded runner thread is a Python threading.Thread object, and implements the thread's *run()* method by waiting until the 'is_prompted' event has been set and then calling its process application's *pull_and_process()* method once for each prompted name. It is expected that the process application will have been set up by the runner with a notification log reader from which event notifications will be pulled.

__init__(*app_class: Type[eventsourcing.system.Follower], is_stopping: threading.Event*)
Initialize self. See help(type(self)) for accurate signature.

run() → None
Begins by constructing an application instance from given application class and then loops forever until stopped. The loop blocks on waiting for the 'is_prompted' event to be set, then forwards the prompts already received to its application by calling the application's *pull_and_process()* method for each prompted name.

receive_prompt(*leader_name: str*) → None
Receives prompt by appending name of leader to list of prompted names.

class eventsourcing.system.**NotificationLogReader**(*notification_log: eventsourcing.application.NotificationLog*, *section_size: int = 10*)

Bases: object

Reads domain event notifications from a notification log.

__init__(*notification_log: eventsourcing.application.NotificationLog*, *section_size: int = 10*)
Initialises a reader with the given notification log, and optionally a section size integer which determines the requested number of domain event notifications in each section retrieved from the notification log.

read(*, *start: int*) → Iterator[eventsourcing.persistence.Notification]
Returns a generator that yields event notifications from the reader's notification log, starting from given start position (a notification ID).

This method traverses the linked list of sections presented by a notification log, and yields the individual event notifications that are contained in each section. When all the event notifications from a section have been yielded, the reader will retrieve the next section, and continue yielding event notification until all subsequent event notifications in the notification log from the start position have been yielded.

select(*, *start: int*) → Iterator[eventsourcing.persistence.Notification]
Returns a generator that yields event notifications from the reader's notification log, starting from given start position (a notification ID).

This method selects a limited list of notifications from a notification log and yields event notifications individually. When all the event notifications in the list are yielded, the reader will retrieve another list, and continue yielding event notification until all subsequent event notifications in the notification log from the start position have been yielded.

1.8 interface — Interface

this page is under development — please check back soon

1.8.1 Classes

class eventsourcing.interface.**NotificationLogInterface**
Bases: abc.ABC

Abstract base class for obtaining serialised sections of a notification log.

get_log_section(*section_id: str*) → str
Returns a serialised *Section* from a notification log.

get_notifications(*start: int*, *limit: int*) → str
Returns a serialised list of *Notification* objects from a notification log.

class eventsourcing.interface.**NotificationLogJSONService**(*app: TApplication*)
Bases: *eventsourcing.interface.NotificationLogInterface*, typing.Generic

Presents serialised sections of a notification log.

__init__(*app: TApplication*)
Initialises service with given application.

get_log_section(*section_id: str*) → str
Returns JSON serialised *Section* from a notification log.

get_notifications(*start: int*, *limit: int*) → str
Returns a serialised list of *Notification* objects from a notification log.

class eventsourcing.interface.**NotificationLogJSONClient**(*interface: eventsourcing.interface.NotificationLogInterface*)

Bases: *eventsourcing.application.NotificationLog*

Presents deserialized sections of a notification log.

__init__(*interface: eventsourcing.interface.NotificationLogInterface*)
Initialises log with a given interface.

__getitem__(*section_id: str*) → eventsourcing.application.Section
Returns a `Section` from a notification log.

select(*start: int*, *limit: int*) → List[eventsourcing.persistence.Notification]
Returns a list of *Notification* objects.

1.9 Examples

This library contains a few example applications and systems.

this page is under development — please check back soon

```
import unittest
```

1.9.1 Bank accounts

Test first...

```
class TestBankAccounts(unittest.TestCase):
    def test(self) -> None:
        app = BankAccounts()

        # Check account not found error.
        with self.assertRaises(AccountNotFoundError):
            app.get_balance(uuid4())

        # Create an account.
        account_id1 = app.open_account(
            full_name="Alice",
            email_address="alice@example.com",
        )

        # Check balance.
        self.assertEqual(app.get_balance(account_id1), Decimal("0.00"))

        # Deposit funds.
        app.deposit_funds(
            credit_account_id=account_id1,
            amount=Decimal("200.00"),
        )

        # Check balance.
        self.assertEqual(app.get_balance(account_id1), Decimal("200.00"))

        # Withdraw funds.
        app.withdraw_funds(
            debit_account_id=account_id1,
```

(continues on next page)

(continued from previous page)

```
            amount=Decimal("50.00"),
        )

        # Check balance.
        self.assertEqual(app.get_balance(account_id1), Decimal("150.00"))

        # Fail to withdraw funds - insufficient funds.
        with self.assertRaises(InsufficientFundsError):
            app.withdraw_funds(
                debit_account_id=account_id1,
                amount=Decimal("151.00"),
            )

        # Check balance - should be unchanged.
        self.assertEqual(app.get_balance(account_id1), Decimal("150.00"))

        # Create another account.
        account_id2 = app.open_account(
            full_name="Bob",
            email_address="bob@example.com",
        )

        # Transfer funds.
        app.transfer_funds(
            debit_account_id=account_id1,
            credit_account_id=account_id2,
            amount=Decimal("100.00"),
        )

        # Check balances.
        self.assertEqual(app.get_balance(account_id1), Decimal("50.00"))
        self.assertEqual(app.get_balance(account_id2), Decimal("100.00"))

        # Fail to transfer funds - insufficient funds.
        with self.assertRaises(InsufficientFundsError):
            app.transfer_funds(
                debit_account_id=account_id1,
                credit_account_id=account_id2,
                amount=Decimal("1000.00"),
            )

        # Check balances - should be unchanged.
        self.assertEqual(app.get_balance(account_id1), Decimal("50.00"))
        self.assertEqual(app.get_balance(account_id2), Decimal("100.00"))

        # Close account.
        app.close_account(account_id1)

        # Fail to transfer funds - account closed.
        with self.assertRaises(AccountClosedError):
            app.transfer_funds(
                debit_account_id=account_id1,
                credit_account_id=account_id2,
                amount=Decimal("50.00"),
            )

        # Fail to transfer funds - account closed.
```

(continues on next page)

(continued from previous page)

```
        with self.assertRaises(AccountClosedError):
            app.transfer_funds(
                debit_account_id=account_id2,
                credit_account_id=account_id1,
                amount=Decimal("50.00"),
            )

        # Fail to withdraw funds - account closed.
        with self.assertRaises(AccountClosedError):
            app.withdraw_funds(
                debit_account_id=account_id1,
                amount=Decimal("1.00"),
            )

        # Fail to deposit funds - account closed.
        with self.assertRaises(AccountClosedError):
            app.deposit_funds(
                credit_account_id=account_id1,
                amount=Decimal("1000.00"),
            )

        # Check balance - should be unchanged.
        self.assertEqual(app.get_balance(account_id1), Decimal("50.00"))

        # Check overdraft limit.
        self.assertEqual(
            app.get_overdraft_limit(account_id2),
            Decimal("0.00"),
        )

        # Set overdraft limit.
        app.set_overdraft_limit(
            account_id=account_id2,
            overdraft_limit=Decimal("500.00"),
        )

        # Can't set negative overdraft limit.
        with self.assertRaises(AssertionError):
            app.set_overdraft_limit(
                account_id=account_id2,
                overdraft_limit=Decimal("-500.00"),
            )

        # Check overdraft limit.
        self.assertEqual(
            app.get_overdraft_limit(account_id2),
            Decimal("500.00"),
        )

        # Withdraw funds.
        app.withdraw_funds(
            debit_account_id=account_id2,
            amount=Decimal("500.00"),
        )

        # Check balance - should be overdrawn.
        self.assertEqual(
```

(continues on next page)

(continued from previous page)

```
            app.get_balance(account_id2),
            Decimal("-400.00"),
        )

        # Fail to withdraw funds - insufficient funds.
        with self.assertRaises(InsufficientFundsError):
            app.withdraw_funds(
                debit_account_id=account_id2,
                amount=Decimal("101.00"),
            )

        # Fail to set overdraft limit - account closed.
        with self.assertRaises(AccountClosedError):
            app.set_overdraft_limit(
                account_id=account_id1,
                overdraft_limit=Decimal("500.00"),
            )
```

The application class `BankAccounts`...

```
class BankAccounts(Application):
    def open_account(self, full_name: str, email_address: str) -> UUID:
        account = BankAccount.open(
            full_name=full_name,
            email_address=email_address,
        )
        self.save(account)
        return account.id

    def get_account(self, account_id: UUID) -> BankAccount:
        try:
            aggregate = self.repository.get(account_id)
        except AggregateNotFound:
            raise AccountNotFoundError(account_id)
        else:
            assert isinstance(aggregate, BankAccount)
            return aggregate

    def get_balance(self, account_id: UUID) -> Decimal:
        account = self.get_account(account_id)
        return account.balance

    def deposit_funds(self, credit_account_id: UUID, amount: Decimal) -> None:
        account = self.get_account(credit_account_id)
        account.append_transaction(amount)
        self.save(account)

    def withdraw_funds(self, debit_account_id: UUID, amount: Decimal) -> None:
        account = self.get_account(debit_account_id)
        account.append_transaction(-amount)
        self.save(account)

    def transfer_funds(
        self,
        debit_account_id: UUID,
        credit_account_id: UUID,
        amount: Decimal,
```

(continues on next page)

(continued from previous page)

```
    ) -> None:
        debit_account = self.get_account(debit_account_id)
        credit_account = self.get_account(credit_account_id)
        debit_account.append_transaction(-amount)
        credit_account.append_transaction(amount)
        self.save(debit_account, credit_account)

    def set_overdraft_limit(self, account_id: UUID, overdraft_limit: Decimal) -> None:
        account = self.get_account(account_id)
        account.set_overdraft_limit(overdraft_limit)
        self.save(account)

    def get_overdraft_limit(self, account_id: UUID) -> Decimal:
        account = self.get_account(account_id)
        return account.overdraft_limit

    def close_account(self, account_id: UUID) -> None:
        account = self.get_account(account_id)
        account.close()
        self.save(account)
```

```
class AccountNotFoundError(Exception):
    pass
```

The aggregate class `BankAccount`...

```
class BankAccount(Aggregate):
    def __init__(self, full_name: str, email_address: str):
        self.full_name = full_name
        self.email_address = email_address
        self.balance = Decimal("0.00")
        self.overdraft_limit = Decimal("0.00")
        self.is_closed = False

    @classmethod
    def open(cls, full_name: str, email_address: str) -> "BankAccount":
        return cls._create(
            cls.Opened,
            id=uuid4(),
            full_name=full_name,
            email_address=email_address,
        )

    class Opened(AggregateCreated):
        full_name: str
        email_address: str

    def append_transaction(
        self, amount: Decimal, transaction_id: Optional[UUID] = None
    ) -> None:
        self.check_account_is_not_closed()
        self.check_has_sufficient_funds(amount)
        self.trigger_event(
            self.TransactionAppended,
            amount=amount,
            transaction_id=transaction_id,
```

(continues on next page)

(continued from previous page)

```
        )

    def check_account_is_not_closed(self) -> None:
        if self.is_closed:
            raise AccountClosedError({"account_id": self.id})

    def check_has_sufficient_funds(self, amount: Decimal) -> None:
        if self.balance + amount < -self.overdraft_limit:
            raise InsufficientFundsError({"account_id": self.id})

    class TransactionAppended(AggregateEvent):
        amount: Decimal
        transaction_id: UUID

        def apply(self, aggregate: "BankAccount") -> None:
            aggregate.balance += self.amount

    def set_overdraft_limit(self, overdraft_limit: Decimal) -> None:
        assert overdraft_limit > Decimal("0.00")
        self.check_account_is_not_closed()
        self.trigger_event(
            self.OverdraftLimitSet,
            overdraft_limit=overdraft_limit,
        )

    class OverdraftLimitSet(AggregateEvent):
        overdraft_limit: Decimal

        def apply(self, aggregate: "BankAccount") -> None:
            aggregate.overdraft_limit = self.overdraft_limit

    def close(self) -> None:
        self.trigger_event(self.Closed)

    class Closed(AggregateEvent):
        def apply(self, aggregate: "BankAccount") -> None:
            aggregate.is_closed = True
```

```
class TransactionError(Exception):
    pass
```

```
class AccountClosedError(TransactionError):
    pass
```

```
class InsufficientFundsError(TransactionError):
    pass
```

Run the test. . .

```
suite = unittest.TestSuite()
suite.addTest(TestBankAccounts("test"))

runner = unittest.TextTestRunner()
result = runner.run(suite)

assert result.wasSuccessful()
```

1.9.2 Cargo shipping

Test first...

```
class TestBookingService(unittest.TestCase):
    def setUp(self) -> None:
        self.service = BookingService(BookingApplication())

    def test_admin_can_book_new_cargo(self) -> None:
        arrival_deadline = datetime.now(tz=TZINFO) + timedelta(weeks=3)

        cargo_id = self.service.book_new_cargo(
            origin="NLRTM",
            destination="USDAL",
            arrival_deadline=arrival_deadline,
        )

        cargo_details = self.service.get_cargo_details(cargo_id)
        self.assertTrue(cargo_details["id"])
        self.assertEqual(cargo_details["origin"], "NLRTM")
        self.assertEqual(cargo_details["destination"], "USDAL")

        self.service.change_destination(cargo_id, destination="AUMEL")
        cargo_details = self.service.get_cargo_details(cargo_id)
        self.assertEqual(cargo_details["destination"], "AUMEL")
        self.assertEqual(
            cargo_details["arrival_deadline"],
            arrival_deadline,
        )

    def test_scenario_cargo_from_hongkong_to_stockholm(
        self,
    ) -> None:
        # Test setup: A cargo should be shipped from
        # Hongkong to Stockholm, and it should arrive
        # in no more than two weeks.
        origin = "HONGKONG"
        destination = "STOCKHOLM"
        arrival_deadline = datetime.now(tz=TZINFO) + timedelta(weeks=2)

        # Use case 1: booking.

        # A new cargo is booked, and the unique tracking
        # id is assigned to the cargo.
        tracking_id = self.service.book_new_cargo(origin, destination, arrival_
↪deadline)

        # The tracking id can be used to lookup the cargo
        # in the repository.
        # Important: The cargo, and thus the domain model,
        # is responsible for determining the status of the
        # cargo, whether it is on the right track or not
        # and so on. This is core domain logic. Tracking
        # the cargo basically amounts to presenting
        # information extracted from the cargo aggregate
        # in a suitable way.
        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(
```

(continues on next page)

(continued from previous page)

```
            cargo_details["transport_status"],
            "NOT_RECEIVED",
        )
        self.assertEqual(cargo_details["routing_status"], "NOT_ROUTED")
        self.assertEqual(cargo_details["is_misdirected"], False)
        self.assertEqual(
            cargo_details["estimated_time_of_arrival"],
            None,
        )
        self.assertEqual(cargo_details["next_expected_activity"], None)

        # Use case 2: routing.
        #
        # A number of possible routes for this cargo is
        # requested and may be presented to the customer
        # in some way for him/her to choose from.
        # Selection could be affected by things like price
        # and time of delivery, but this test simply uses
        # an arbitrary selection to mimic that process.
        routes_details = self.service.request_possible_routes_for_cargo(tracking_id)
        route_details = select_preferred_itinerary(routes_details)

        # The cargo is then assigned to the selected
        # route, described by an itinerary.
        self.service.assign_route(tracking_id, route_details)

        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(
            cargo_details["transport_status"],
            "NOT_RECEIVED",
        )
        self.assertEqual(cargo_details["routing_status"], "ROUTED")
        self.assertEqual(cargo_details["is_misdirected"], False)
        self.assertTrue(cargo_details["estimated_time_of_arrival"])
        self.assertEqual(
            cargo_details["next_expected_activity"],
            ("RECEIVE", "HONGKONG"),
        )

        # Use case 3: handling

        # A handling event registration attempt will be
        # formed from parsing the data coming in as a
        # handling report either via the web service
        # interface or as an uploaded CSV file. The
        # handling event factory tries to create a
        # HandlingEvent from the attempt, and if the
        # factory decides that this is a plausible
        # handling event, it is stored. If the attempt
        # is invalid, for example if no cargo exists for
        # the specified tracking id, the attempt is
        # rejected.
        #
        # Handling begins: cargo is received in Hongkong.
        self.service.register_handling_event(tracking_id, None, "HONGKONG", "RECEIVE")
        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(cargo_details["transport_status"], "IN_PORT")
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(
            cargo_details["last_known_location"],
            "HONGKONG",
        )
        self.assertEqual(
            cargo_details["next_expected_activity"],
            ("LOAD", "HONGKONG", "V1"),
        )

        # Load onto voyage V1.
        self.service.register_handling_event(tracking_id, "V1", "HONGKONG", "LOAD")
        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(cargo_details["current_voyage_number"], "V1")
        self.assertEqual(
            cargo_details["last_known_location"],
            "HONGKONG",
        )
        self.assertEqual(
            cargo_details["transport_status"],
            "ONBOARD_CARRIER",
        )
        self.assertEqual(
            cargo_details["next_expected_activity"],
            ("UNLOAD", "NEWYORK", "V1"),
        )

        # Incorrectly unload in Tokyo.
        self.service.register_handling_event(tracking_id, "V1", "TOKYO", "UNLOAD")
        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(cargo_details["current_voyage_number"], None)
        self.assertEqual(cargo_details["last_known_location"], "TOKYO")
        self.assertEqual(cargo_details["transport_status"], "IN_PORT")
        self.assertEqual(cargo_details["is_misdirected"], True)
        self.assertEqual(cargo_details["next_expected_activity"], None)

        # Reroute.
        routes_details = self.service.request_possible_routes_for_cargo(tracking_id)
        route_details = select_preferred_itinerary(routes_details)
        self.service.assign_route(tracking_id, route_details)

        # Load in Tokyo.
        self.service.register_handling_event(tracking_id, "V3", "TOKYO", "LOAD")
        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(cargo_details["current_voyage_number"], "V3")
        self.assertEqual(cargo_details["last_known_location"], "TOKYO")
        self.assertEqual(
            cargo_details["transport_status"],
            "ONBOARD_CARRIER",
        )
        self.assertEqual(cargo_details["is_misdirected"], False)
        self.assertEqual(
            cargo_details["next_expected_activity"],
            ("UNLOAD", "HAMBURG", "V3"),
        )

        # Unload in Hamburg.
        self.service.register_handling_event(tracking_id, "V3", "HAMBURG", "UNLOAD")
```

(continues on next page)

(continued from previous page)

```
        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(cargo_details["current_voyage_number"], None)
        self.assertEqual(cargo_details["last_known_location"], "HAMBURG")
        self.assertEqual(cargo_details["transport_status"], "IN_PORT")
        self.assertEqual(cargo_details["is_misdirected"], False)
        self.assertEqual(
            cargo_details["next_expected_activity"],
            ("LOAD", "HAMBURG", "V4"),
        )

        # Load in Hamburg
        self.service.register_handling_event(tracking_id, "V4", "HAMBURG", "LOAD")
        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(cargo_details["current_voyage_number"], "V4")
        self.assertEqual(cargo_details["last_known_location"], "HAMBURG")
        self.assertEqual(
            cargo_details["transport_status"],
            "ONBOARD_CARRIER",
        )
        self.assertEqual(cargo_details["is_misdirected"], False)
        self.assertEqual(
            cargo_details["next_expected_activity"],
            ("UNLOAD", "STOCKHOLM", "V4"),
        )

        # Unload in Stockholm
        self.service.register_handling_event(tracking_id, "V4", "STOCKHOLM", "UNLOAD")
        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(cargo_details["current_voyage_number"], None)
        self.assertEqual(
            cargo_details["last_known_location"],
            "STOCKHOLM",
        )
        self.assertEqual(cargo_details["transport_status"], "IN_PORT")
        self.assertEqual(cargo_details["is_misdirected"], False)
        self.assertEqual(
            cargo_details["next_expected_activity"],
            ("CLAIM", "STOCKHOLM"),
        )

        # Finally, cargo is claimed in Stockholm.
        self.service.register_handling_event(tracking_id, None, "STOCKHOLM", "CLAIM")
        cargo_details = self.service.get_cargo_details(tracking_id)
        self.assertEqual(cargo_details["current_voyage_number"], None)
        self.assertEqual(
            cargo_details["last_known_location"],
            "STOCKHOLM",
        )
        self.assertEqual(cargo_details["transport_status"], "CLAIMED")
        self.assertEqual(cargo_details["is_misdirected"], False)
        self.assertEqual(cargo_details["next_expected_activity"], None)
```

Interface...

```
class BookingService(object):
    """
    Presents an application interface that uses
```

(continues on next page)

(continued from previous page)

```
    simple types of object (str, bool, datetime).
    """

    def __init__(self, app: BookingApplication):
        self.app = app

    def book_new_cargo(
        self,
        origin: str,
        destination: str,
        arrival_deadline: datetime,
    ) -> str:
        tracking_id = self.app.book_new_cargo(
            Location[origin],
            Location[destination],
            arrival_deadline,
        )
        return str(tracking_id)

    def get_cargo_details(self, tracking_id: str) -> CargoDetails:
        cargo = self.app.get_cargo(UUID(tracking_id))

        # Present 'next_expected_activity'.
        next_expected_activity: Optional[Union[Tuple[Any, Any], Tuple[Any, Any, Any]]]
        if cargo.next_expected_activity is None:
            next_expected_activity = None
        elif len(cargo.next_expected_activity) == 2:
            next_expected_activity = (
                cargo.next_expected_activity[0].value,
                cargo.next_expected_activity[1].value,
            )
        elif len(cargo.next_expected_activity) == 3:
            next_expected_activity = (
                cargo.next_expected_activity[0].value,
                cargo.next_expected_activity[1].value,
                cargo.next_expected_activity[2],
            )
        else:
            raise Exception(
                "Invalid next expected activity: {}".format(
                    cargo.next_expected_activity
                )
            )

        # Present 'last_known_location'.
        if cargo.last_known_location is None:
            last_known_location = None
        else:
            last_known_location = cargo.last_known_location.value

        # Present the cargo details.
        return {
            "id": str(cargo.id),
            "origin": cargo.origin.value,
            "destination": cargo.destination.value,
            "arrival_deadline": cargo.arrival_deadline,
            "transport_status": cargo.transport_status,
```

(continues on next page)

(continued from previous page)

```
            "routing_status": cargo.routing_status,
            "is_misdirected": cargo.is_misdirected,
            "estimated_time_of_arrival": cargo.estimated_time_of_arrival,
            "next_expected_activity": next_expected_activity,
            "last_known_location": last_known_location,
            "current_voyage_number": cargo.current_voyage_number,
        }

    def change_destination(self, tracking_id: str, destination: str) -> None:
        self.app.change_destination(UUID(tracking_id), Location[destination])

    def request_possible_routes_for_cargo(self, tracking_id: str) -> List[dict]:
        routes = self.app.request_possible_routes_for_cargo(UUID(tracking_id))
        return [self.dict_from_itinerary(route) for route in routes]

    def dict_from_itinerary(self, itinerary: Itinerary) -> ItineraryDetails:
        legs_details = []
        for leg in itinerary.legs:
            leg_details: LegDetails = {
                "origin": leg.origin,
                "destination": leg.destination,
                "voyage_number": leg.voyage_number,
            }
            legs_details.append(leg_details)
        route_details: ItineraryDetails = {
            "origin": itinerary.origin,
            "destination": itinerary.destination,
            "legs": legs_details,
        }
        return route_details

    def assign_route(
        self,
        tracking_id: str,
        route_details: ItineraryDetails,
    ) -> None:
        routes = self.app.request_possible_routes_for_cargo(UUID(tracking_id))
        for route in routes:
            if route_details == self.dict_from_itinerary(route):
                self.app.assign_route(UUID(tracking_id), route)

    def register_handling_event(
        self,
        tracking_id: str,
        voyage_number: Optional[str],
        location: str,
        handling_activity: str,
    ) -> None:
        self.app.register_handling_event(
            UUID(tracking_id),
            voyage_number,
            Location[location],
            HandlingActivity[handling_activity],
        )
```

```
def select_preferred_itinerary(
```

(continues on next page)

(continued from previous page)

```
    itineraries: List[ItineraryDetails],
) -> ItineraryDetails:
    return itineraries[0]
```

Application...

```
class BookingApplication(Application):
    def register_transcodings(self, transcoder: Transcoder) -> None:
        super(BookingApplication, self).register_transcodings(transcoder)
        transcoder.register(LocationAsName())
        transcoder.register(HandlingActivityAsName())
        transcoder.register(ItineraryAsDict())
        transcoder.register(LegAsDict())

    def book_new_cargo(
        self,
        origin: Location,
        destination: Location,
        arrival_deadline: datetime,
    ) -> UUID:
        cargo = Cargo.new_booking(origin, destination, arrival_deadline)
        self.save(cargo)
        return cargo.id

    def change_destination(self, tracking_id: UUID, destination: Location) -> None:
        cargo = self.get_cargo(tracking_id)
        cargo.change_destination(destination)
        self.save(cargo)

    def request_possible_routes_for_cargo(self, tracking_id: UUID) -> List[Itinerary]:
        cargo = self.get_cargo(tracking_id)
        from_location = (cargo.last_known_location or cargo.origin).value
        to_location = cargo.destination.value
        try:
            possible_routes = REGISTERED_ROUTES[(from_location, to_location)]
        except KeyError:
            raise Exception(
                "Can't find routes from {} to {}".format(from_location, to_location)
            )

        return possible_routes

    def assign_route(self, tracking_id: UUID, itinerary: Itinerary) -> None:
        cargo = self.get_cargo(tracking_id)
        cargo.assign_route(itinerary)
        self.save(cargo)

    def register_handling_event(
        self,
        tracking_id: UUID,
        voyage_number: Optional[str],
        location: Location,
        handing_activity: HandlingActivity,
    ) -> None:
        cargo = self.get_cargo(tracking_id)
        cargo.register_handling_event(
            tracking_id,
```

(continues on next page)

(continued from previous page)

```
            voyage_number,
            location,
            handing_activity,
        )
        self.save(cargo)

    def get_cargo(self, tracking_id: UUID) -> Cargo:
        cargo = self.repository.get(tracking_id)
        assert isinstance(cargo, Cargo)
        return cargo
```

```
class HandlingActivityAsName(Transcoding):
    type = HandlingActivity
    name = "handling_activity"

    def encode(self, obj: HandlingActivity) -> str:
        return obj.name

    def decode(self, data: str) -> HandlingActivity:
        assert isinstance(data, str)
        return HandlingActivity[data]
```

```
class ItineraryAsDict(Transcoding):
    type = Itinerary
    name = "itinerary"

    def encode(self, obj: Itinerary) -> dict:
        return obj.__dict__

    def decode(self, data: dict) -> Itinerary:
        assert isinstance(data, dict)
        return Itinerary(**data)
```

```
class LegAsDict(Transcoding):
    type = Leg
    name = "leg"

    def encode(self, obj: Leg) -> dict:
        return obj.__dict__

    def decode(self, data: dict) -> Leg:
        assert isinstance(data, dict)
        return Leg(**data)
```

```
class LocationAsName(Transcoding):
    type = Location
    name = "location"

    def encode(self, obj: Location) -> str:
        return obj.name

    def decode(self, data: str) -> Location:
        assert isinstance(data, str)
        return Location[data]
```

Domain model...

```
class Cargo(Aggregate):
    """
    The Cargo aggregate is an event-sourced domain model aggregate that
    specifies the routing from origin to destination, and can track what
    happens to the cargo after it has been booked.
    """

    def __init__(
        self,
        origin: Location,
        destination: Location,
        arrival_deadline: datetime,
    ):
        self._origin: Location = origin
        self._destination: Location = destination
        self._arrival_deadline: datetime = arrival_deadline
        self._transport_status: str = "NOT_RECEIVED"
        self._routing_status: str = "NOT_ROUTED"
        self._is_misdirected: bool = False
        self._estimated_time_of_arrival: Optional[datetime] = None
        self._next_expected_activity: NextExpectedActivity = None
        self._route: Optional[Itinerary] = None
        self._last_known_location: Optional[Location] = None
        self._current_voyage_number: Optional[str] = None

    @property
    def origin(self) -> Location:
        return self._origin

    @property
    def destination(self) -> Location:
        return self._destination

    @property
    def arrival_deadline(self) -> datetime:
        return self._arrival_deadline

    @property
    def transport_status(self) -> str:
        return self._transport_status

    @property
    def routing_status(self) -> str:
        return self._routing_status

    @property
    def is_misdirected(self) -> bool:
        return self._is_misdirected

    @property
    def estimated_time_of_arrival(
        self,
    ) -> Optional[datetime]:
        return self._estimated_time_of_arrival

    @property
    def next_expected_activity(self) -> Optional[Tuple]:
```

(continues on next page)

(continued from previous page)

```
        return self._next_expected_activity

    @property
    def route(self) -> Optional[Itinerary]:
        return self._route

    @property
    def last_known_location(self) -> Optional[Location]:
        return self._last_known_location

    @property
    def current_voyage_number(self) -> Optional[str]:
        return self._current_voyage_number

    @classmethod
    def new_booking(
        cls,
        origin: Location,
        destination: Location,
        arrival_deadline: datetime,
    ) -> "Cargo":
        return cls._create(
            event_class=Cargo.BookingStarted,
            id=uuid4(),
            origin=origin,
            destination=destination,
            arrival_deadline=arrival_deadline,
        )

    class BookingStarted(AggregateCreated):
        origin: Location
        destination: Location
        arrival_deadline: datetime

    class Event(AggregateEvent["Cargo"]):
        def apply(self, aggregate: "Cargo") -> None:
            aggregate.apply(self)

    @singledispatchmethod
    def apply(self, event: "Cargo.Event") -> None:
        """
        Default aggregate projection.
        """

    def change_destination(self, destination: Location) -> None:
        self.trigger_event(
            self.DestinationChanged,
            destination=destination,
        )

    class DestinationChanged(Event):
        destination: Location

    @apply.register(DestinationChanged)
    def destination_changed(self, event: DestinationChanged) -> None:
        self._destination = event.destination
```

(continues on next page)

(continued from previous page)

```
    def assign_route(self, itinerary: Itinerary) -> None:
        self.trigger_event(self.RouteAssigned, route=itinerary)

    class RouteAssigned(Event):
        route: Itinerary

    @apply.register(RouteAssigned)
    def route_assigned(self, event: RouteAssigned) -> None:
        self._route = event.route
        self._routing_status = "ROUTED"
        self._estimated_time_of_arrival = datetime.now(tz=TZINFO) + timedelta(weeks=1)
        self._next_expected_activity = (
            HandlingActivity.RECEIVE,
            self.origin,
        )
        self._is_misdirected = False

    def register_handling_event(
        self,
        tracking_id: UUID,
        voyage_number: Optional[str],
        location: Location,
        handling_activity: HandlingActivity,
    ) -> None:
        self.trigger_event(
            self.HandlingEventRegistered,
            tracking_id=tracking_id,
            voyage_number=voyage_number,
            location=location,
            handling_activity=handling_activity,
        )

    class HandlingEventRegistered(Event):
        tracking_id: UUID
        voyage_number: str
        location: Location
        handling_activity: str

    @apply.register(HandlingEventRegistered)
    def handling_event_registered(self, event: HandlingEventRegistered) -> None:
        assert self.route is not None
        if event.handling_activity == HandlingActivity.RECEIVE:
            self._transport_status = "IN_PORT"
            self._last_known_location = event.location
            self._next_expected_activity = (
                HandlingActivity.LOAD,
                event.location,
                self.route.legs[0].voyage_number,
            )
        elif event.handling_activity == HandlingActivity.LOAD:
            self._transport_status = "ONBOARD_CARRIER"
            self._current_voyage_number = event.voyage_number
            for leg in self.route.legs:
                if leg.origin == event.location.value:
                    if leg.voyage_number == event.voyage_number:
                        self._next_expected_activity = (
                            HandlingActivity.UNLOAD,
```

(continues on next page)

(continued from previous page)

```
                            Location[leg.destination],
                            event.voyage_number,
                        )
                        break
            else:
                raise Exception(
                    "Can't find leg with origin={} and "
                    "voyage_number={}".format(
                        event.location,
                        event.voyage_number,
                    )
                )

        elif event.handling_activity == HandlingActivity.UNLOAD:
            self._current_voyage_number = None
            self._last_known_location = event.location
            self._transport_status = "IN_PORT"
            if event.location == self.destination:
                self._next_expected_activity = (
                    HandlingActivity.CLAIM,
                    event.location,
                )
            elif event.location.value in [leg.destination for leg in self.route.legs]:
                for i, leg in enumerate(self.route.legs):
                    if leg.voyage_number == event.voyage_number:
                        next_leg: Leg = self.route.legs[i + 1]
                        assert Location[next_leg.origin] == event.location
                        self._next_expected_activity = (
                            HandlingActivity.LOAD,
                            event.location,
                            next_leg.voyage_number,
                        )
                        break
            else:
                self._is_misdirected = True
                self._next_expected_activity = None

        elif event.handling_activity == HandlingActivity.CLAIM:
            self._next_expected_activity = None
            self._transport_status = "CLAIMED"

        else:
            raise Exception(
                "Unsupported handling event: {}".format(event.handling_activity)
            )
```

```
class HandlingActivity(Enum):
    RECEIVE = "RECEIVE"
    LOAD = "LOAD"
    UNLOAD = "UNLOAD"
    CLAIM = "CLAIM"
```

```
class Itinerary(object):
    """
    An itinerary along which cargo is shipped.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(
        self,
        origin: str,
        destination: str,
        legs: Tuple[Leg, ...],
    ):
        self.origin = origin
        self.destination = destination
        self.legs = legs
```

```
class Leg(object):
    """
    Leg of an itinerary.
    """

    def __init__(
        self,
        origin: str,
        destination: str,
        voyage_number: str,
    ):
        self.origin: str = origin
        self.destination: str = destination
        self.voyage_number: str = voyage_number
```

```
class Location(Enum):
    """
    Locations in the world.
    """

    HAMBURG = "HAMBURG"
    HONGKONG = "HONGKONG"
    NEWYORK = "NEWYORK"
    STOCKHOLM = "STOCKHOLM"
    TOKYO = "TOKYO"

    NLRTM = "NLRTM"
    USDAL = "USDAL"
    AUMEL = "AUMEL"
```

Run the test...

```
suite = unittest.TestSuite()
suite.addTest(TestBookingService("test_admin_can_book_new_cargo"))
suite.addTest(TestBookingService("test_scenario_cargo_from_hongkong_to_stockholm"))

runner = unittest.TextTestRunner()
result = runner.run(suite)

assert result.wasSuccessful()
```

1.10 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.

1.10.1 Version 9.x

Version 9.x series is a rewrite of the library that distills most of the best parts of the previous versions of the library into faster and simpler code. This version is recommended for new projects. It is not backwards-compatible with previous major versions. However the underlying principles are the same, and so conversion of code and stored events is very possible.

Version 9.1.3 (released 8 October 2021)

Added "trove classifier" for Python 3.10.

Version 9.1.2 (released 1 October 2021)

Clarified Postgres configuration options (POSTGRES_LOCK_TIMEOUT and POSTGRES_IDLE_IN_TRANSACTION_SESSION_TIMEOUT) require integer seconds. Added py.typed file (was missing since v9).

Version 9.1.1 (released 20 August 2021)

Changed PostgreSQL schema to use BIGSERIAL (was SERIAL) for notification IDs.

Version 9.1.0 (released 18 August 2021)

Added support for setting environment when constructing application. Added "eq" and "repr" methods on aggregate base class. Reinstated explicit definition of Aggregate.Created class. Added Invoice example, and Parking Lot example. Fixed bug when decorating property setter (use method argument name). Improved type annotations. Adjusted order of calling domain event mutate() and apply() methods, so apply() method is called first, in case exceptions are raised by apply() method so that the aggregate object can emerge unscathed whereas previously its version number and modified time would always be changed. Improved robustness of recorder classes, with more attention to connection state, closing connections on certain errors, retrying operations under certain conditions, and especially by changing the postgres recorders to obtain 'EXCLUSIVE' mode table lock when inserting events. Obtaining the table lock in PostgreSQL avoids interleaving of inserts between commits, which avoids event notifications from being committed with lower notification IDs than event notifications that have already been committed, and thereby prevents readers who are tailing the notification log of an application from missing event notifications for this reason. Added various environment variable options: for sqlite a lock timeout option; and for postgres a max connection age option which allows connections over a certain age to be closed when idle, a connection pre-ping option, a lock timeout option, and an option to timeout sessions idle in transaction so that locks can be released even if the database client has somehow ceased to continue its interactions with the server in a way that leave the session open. Improved the exception classes, to follow the standard Python DBAPI class names, and to encapsulate errors from drivers with library errors following this standard. Added methods to notification log and reader classes to allow notifications to be selected directly. Changed Follower class to select() rather than read() notifications. Supported defining initial version number of aggregates on aggregate class (with INITIAL_VERSION attribute).

Version 9.0.3 (released 17 May 2021)

Changed PostgreSQL queries to use transaction class context manager (transactions were started and not closed). Added possibility to specify a port for Postgres (thanks to Valentin Dion). Added **kwargs to Application.save() method signature, so other things can be passed down the stack. Fixed reference in installing.rst (thanks to Karl Heinrichmeyer). Made properties out of aggregate attributes: 'modified_on' and 'version'. Improved documentation.

Version 9.0.2 (released 16 April 2021)

Fixed issue with type hints in PyCharm v2021.1 for methods decorated with the @event decorator.

Version 9.0.1 (released 29 March 2021)

Improved documentation. Moved cipher base class to avoid importing cipher module.

Version 9.0.0 (released 13 March 2021)

First release of the distilled version of the library. Compared with previous versions, the code and documentation are much simpler. This version focuses directly on expressing the important concerns, without the variations and alternatives that had been accumulated over the past few years of learning and pathfinding.

The highlight is the new *declarative syntax* for event sourced domain models.

Dedicated persistence modules for SQLite and PostgresSQL have been introduced. Support for SQLAlchemy and Django, and other databases, has been removed. The plan is to support these in separate package distributions. The default "plain old Python object" infrastructure continues to exist, and now offers event storage and retrieval performance of around 20x the speed of using PostgreSQL and around 4x the speed of using SQLite in memory.

The event storage format is more efficient, because originator IDs and originator versions are removed from the stored event state before serialisation, and then reinstated on serialisation.

Rather than the using "INSERT SELECT MAX" SQL statements, database sequences are used to generate event notifications. This avoids table conflicts that sometimes caused exceptions and required retries when storing events. Although this leads to notification ID sequences that may have gaps, the use of sequences means there is still no risk of event notifications being inserted in the gaps after later event notifications have been processed, which was the motivation for using gapless sequences in previous versions. The notification log and log reader classes have been adjusted to support the possible existence of gaps in the notification log sequence.

The transcoder is more easily extensible, with the new style for defining and registering individual transcoding objects to support individual types of object that are not supported by default.

Domain event classes have been greatly simplified, with the deep hierarchy of entity and event classes removed in favour of the simple aggregate base class.

The repository class has been changed to provide a single get() method. It no longer supports the Python "indexing" square-bracket syntax, so that there is just one way to get an aggregate regardless of whether the requested version is specified or not.

Application configuration of persistence infrastructure is now driven by environment variables rather than constructor parameters, leading to a simpler interface for application object classes. The mechanism for storing aggregates has been simplified, so that aggregates are saved using the application "save" method. A new "notify" method has been added to the application class, to support applications that need to know when new events have just been recorded.

The mechanism by which aggregates published their events and a "persistence subscriber" subscribed and persisted published domain events has been completely removed, since aggregates that are saved always need some persistence

infrastructure to store the events, and it is the responsibility of the application to bring together the domain model and infrastructure, so that when an aggregate can be saved there is always an application.

Process application policy methods are now given a process event object and will use it to collect domain events, using its "save" method, which has the same method signature as the application "save" method. This allows policies to accumulate new events on the process event object in the order they were generated, whereas previously if new events were generated on one aggregate and then a second and then the first, the events of one aggregate would be stored first and the events of the second aggregate would be stored afterwards, leading to an incorrect ordering of the domain events in the notification log. The process event object existed in previous versions, was used to keep track of the position in a notification log of the event notification that was being processed by a policy, and continues to be used for that purpose.

The system runners have been reduced to the single-threaded and multi-threaded runners, with support for running with Ray and gRPC and so on removed (the plan being to support these in separate package distributions).

Altogether, these changes mean the core library now depends only on the PythonStandard Library, except for the optional extra dependencies on a cryptographic library (PyCryptodome) and a PostgresSQL driver (psycopg2), and the dependencies of development tools. Altogether, these changes make the test suite much faster to run (several seconds rather than several minutes for the previous version). These changes make the build time on CI services much quicker (around one minute, rather than nearly ten minutes for the previous version). And these changes make the library more approachable and fun for users and library developers. Test coverage has been increased to 100% line and branch coverage. Also mypy and flake8 checking is done.

The documentation has been rewritten to focus more on usage of the library code, and less on explaining surrounding concepts and considerations.

1.10.2 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 2021)

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.

1.10.3 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.

1.10.4 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)

1.10.5 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.

1.10.6 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)

1.10.7 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)

1.10.8 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)

1.10.9 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)

1.10.10 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)

CHAPTER 2

Modules Reference

- genindex
- modindex

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

T

U

V

Z