Baroque Documentation

Release 0.0.3

Claudio Sparpaglione

May 05, 2017

Contents

Hi, welcome to Baroque's documentation!

Baroque is a convenient event broker and an extensible framework for building event-driven applications.

The best way to get started is to explore some real life scenarios in which Baroque may help you with:

- you want to keep in sync the values of two different properties of different software objects (as they change)
- you want to log each and only deletion in a database table
- you want to send a push notification to your devices every time an exception is raised in your super-critical production web applications
- you want a selected pool of persons in the marketing division of your company to get an e-mail whenever somebody places a post on your company's blog and that post contains specific words

...and so forth!

Baroque allows you to **build higher level abstractions on the fundamental messaging pattern** it implements: Publish-Subscribe

That's because you don't have to think about it - Baroque does the job for you and you're free to focus on building valuable software that leverages the pattern

Baroque behaviour can be easily configured through a YAML file.

Read on and get more details in the next sections

CHAPTER 1

Guides

How to use Baroque

Using Baroque is simple: you only need to declare what you want to happen whenever events of a specific type occurs. This concept can be further leveraged through topics, which basically are a convenient way for you to make something happen whenever multiple types of events occur: a topic is a "named manifest" of your subscription to the occurring of those event types.

Let's dive a bit deeper into Baroque gears...

Reactors

Reactors are objects that embed a Python function called "reaction": this function is "that something you wanted to happen"! You just have to provide that function, which can do literally anything you want, eg:

- change properties of one or more objects
- invoke other functions
- call an HTTP API
- spawna new worker thread
- put a message on a queue
- send an e-mail, SMS or push notification
- print something on the console
- write a row on a database table

Sky is the limit...

The only constraint that Baroque gives to reaction functions is that they must parametrically accept at least one positional argument: the triggering event. Baroque will pass in the event object whenever it executes the reaction function

When does the execution of the reaction happen? Whenever Baroque knows that an event of a certain type has been fired and that event types must result into the execution of that reactor.

Specifying the binding between Reactors and Event Types is the core operation when using Baroque, and it's up to you:

```
from baroque import Baroque, Reactor, MetricEventType

brq = Baroque()

# create a simple reactor from a reaction function
def greet(event):
    print("Hello world")
reactor = Reactor(greet)

# Tell Baroque to run your reactor whenever any event of type
# MetricEventType is published
brq.on(MetricEventType).run(reactor)
```

What if you want to execute your reaction function *only if* some conditions on the event are met? Don't worry: along with a reaction, a Reactor can embed a "condition" function. The condition is a standard Python function that you provide to the Reactor and must comply to the following:

- it gets one parameter: the Event object
- it returns a boolean value (*True* if the condition is met or *False* if it is not)

If you don't specify any condition when you create a Reactor object, no checks will be performed on the event that triggered the execution of the Reactor

Example:

```
from baroque import Reactor, Baroque, Event, GenericEventType

# Reaction function
def greet(event):
    print("Hello {}".format(event.payload.get("name", "world"))

# Condition function
def only_if_name_provided(event):
    return "name" in event.payload

reactor = Reactor(greet, only_if_name_provided)
brq = Baroque()
brq.on_any_event_run(reactor)


# The greeting is printed only if the triggering event contains a field
# named "name" in its payload...
brq.publish(Event(GenericEventType, payload={}))                # reaction is not run
brq.publish(Event(GenericEventType, payload={"name": "bob"}))    # reaction is run
```

For your convenience, Baroque offers a few out-of-the-box reactors types, available through a factory object:

```
from baroque import ReactorFactory

reactor = ReactorFactory.stdout          # mirror-print of event object on terminal
reactor = ReactorFactory.call_function   # invoke a function on an object instance
reactor = ReactorFactory.json_webhook    # HTTP POSTs some JSON to a URL
```

Events

Events are the core concept in Baroque. An event is an object that describes something that happened and that you want to notify to someone in order to allow something to happen in reaction to that.

At its bare minimum, an event is just a box of metadata defined by you and characterized by a specific event type: you can create and publish many different events of the same type.

The event type can either a valid instance of a subclass of the *EventType* class or the subclass.

For example, this is an event of type *GenericEventType*, which is a subtype of *EventType*:

```
from baroque import Event, GenericEventType
event1 = Event(GenericEventType)
event2 = Event(GenericEventType())
```

An event has the following fields:

- a unique UUID
- an optional payload (*dict*) containing user-defined metadata
- an optional description
- an optional *set* of tags
- a publication status (*PUBLISHED* vs *UNPUBLISHED*)
- a creation timestamp
- an optional owner

In code:

```
event = Event(TweetEvent,
              payload=dict(tweet_id=12345678,
                           tweet_text="howdy this is a tweet"),
              description='My first tweet',
              owner='csparpa')
event.json()
event.md5()
event.id
event.owner
event.type
event.status
event.description
event.timestamp   # set to current timestamp with: event.touch()
event.payload
event.tags
event.tags.update(‘twitter’, ‘tweet’)
```

Any event can be dumped to JSON or can provide its own MD5 hash:

```
event.md5()
event.json()
```

Event Types

As stated before, each event must be identified by one event type. Event types are the way Baroque uses to:

- *convey events contents* in terms of data and structure, and *validate* them: this means that datastructures (eg. payload, sections of payload, whole event structure, etc.) carried by events of specific types can be validated so that events that claim to be of those types but do not carry well-formed data can be spot and handled with. Validation is enabled via JSON Schema.
- *convey events hierarchy*: you can create event types hierarchies

You can either define custom event types or use the ones that Baroque offers for your convenience, which you can find in module *baroque.defaults.eventtypes*

Let's start with the latter ones.

You might have no need to create any events hierarchy nor to specify what data your events carry: in this case, it's just OK to use a *GenericEventType*, which is a kind of "wildcard" event type that applies no schema validation on events and is not included in any event types hierarchy

```
from baroque import Event, GenericEventType
event = Event(GenericEventType)
```

The off-the-shelf event types include:

- *StateTransitionEventType* - models events fired on state machine transitions
- *DataOperationEventType* - models events fired on manipulation of data entities
- *MetricEventType* - models events fired on phenomena sampling or time-series variations

These event types apply schema validation to events: please refer to the code documentation to check out the expected format for data carried by these events.

In case you need to define your own event types, just subclass the base class *baroque.entities.eventtype.EventType* and provide the JSON schema you want events of your custom type to be validated against.

In example, let us imagine that we want to define events of type "BabyBornEventType" that must contain in their payload at least two information: the name of the baby and the baby's birth date:

```
from baroque import EventType

class BabyBornEventType(EventType):
    def __init__(self, owner=None):
        EventType.__init__(
            self,
            '''{
              "$schema": "http://json-schema.org/draft-04/schema#",
              "type": "object",
              "properties": {
                "payload": {
                  "type": "object",
                  "properties": {
                    "baby_name": {
                      "type": ["string"]
                    },
                    "birth_date": {
                      "type": ["string"]
                    }
                  },
                  "required": [
                    "baby_name",
                    "birth_date"
                  ]
                }
              },
              "required": [
                "payload"
              ]
            }''',
            description='A new baby is born',
            owner=owner)
```

Then if we instantiate events of type *BabyBornEventType*, they must conform to the JSON schema that we specified on the type:

```
from baroque import Event

# this event is valid
valid_event = Event(BabyBornEventType,
                    payload=dict(baby_name='Bob',
                                 birth_date='2017-04-19'))

# this event is not valid, as it does not carry the required data
invalid_event = Event(BabyBornEventType,
                      payload=dict(foo='bar'))
```

Invalid events can result in exceptions raised when trying to publish them: this depends on the library configuration (please see the relevant documentation section). By default, Baroque validates all events schema.

Please refer to JSON Schema specification for details about expressing events contents.

Topics

Topics are channels for notifying multiple event consumers at once that events of specific types have been published; they're a way to decouple producers of events from their consumers.

When you crete a topic you need to specify what event types it is bound to (passing in an iterable of either *EventType* instances or subclasses); a topic can be bound to one or more event types. Topic must have a name and can optionally have an owner, a description and a set of tags (strings) you can use later to search for the topic. Each topic also gets an unique ID:

```
from baroque import Topic
family_event_types = [ClaudioRelativesEventType(), ClaudioEventType()]
topic = Topic('my-family-events',
              family_event_types,
              description='all events about me and my family will be published here',
              owner='me',
              tags=['claudio', 'events'])
```

To make a topic useful, you must register it to the Baroque broker instance:

```
from baroque import Baroque
brq = Baroque()
brq.topics.register(topic)
```

A useful shortcut for creating topics *and* registering them on the broker is the following:

```
from baroque import Baroque
brq = Baroque()
family_event_types = [ClaudioRelativesEventType(), ClaudioEventType()]
topic = brq.topics.new('my-family-events',
                           family_event_types,
                           description='all events about me and my family will be␣
↪published here',
                           owner='me',
                           tags=['claudio', 'events'])
```

Event consumers *subscribe* to the topic by passing to the broker instance both a reference to that topic and the reactor object they want to be executed whenever *any* events of the types bound to the topic will be published on the broker:

```
from baroque import Baroque, ReactorFactory
brq = Baroque()
reactor = ReactorFactory.stdout
brq.on_topic_run(topic, reactor)
```

If the topic is not registered on the broker instance yet, this will be automatically registered. Baroque can be configured to raise an *UnregisteredTopicError* instead.

Subscribers can leverage Baroque topics search features to look for interesting topics:

```
from baroque import Baroque
brq = Baroque()
brq.topics.of('somebody')        # finds all topics owned by someone
brq.with_id('d3d5beb8')          # finds the topic with the specified ID
brq.with_name('my-topic')        # finds the topic with the specified name
brq.with_tags(['tag1', 'tag2'])  # finds all topics marked with the specified tags
```

Event producers that want their events to be published on a topic must do it via the broker; this will trigger execution of all reactors that were bound to the topic:

```
from baroque import Baroque, Event
brq = Baroque()

claudio_event = Event(ClaudioEventType())
brq.publish_on_topic(claudio_event, claudio_event)

cousin_event = Event(ClaudioRelativesEventType())
brq.publish_on_topic(cousin_event, claudio_event)
```

The Baroque broker

TBD

Samples of Baroque usage scenarios

TBD

Configuring Baroque

Baroque behaviour can be easily configured

Default configuration

The default configuration is stored in module:

```
baroque.defaults.config
```

as a Python dictionary and is loaded upon broker instantiation if no custom configuration file is provided as argument.

An actual YML version of the default version is stored in the root path of Baroque's installation (file: _baroque.yml_)

Custom configuration

Custom configurations can be provided as YML files, and can be loaded upon broker instantiation as follows:

```
from baroque import Baroque
brq = Baroque(configfile='path/to/file.yml')
```

Baroque validates the YML syntax, but performs no validation on the provided configuration switches (whether they make sense or not): the check is done in a lazy way - in other when the switches are actually used.

Configuration switches

Switches are grouped according to Baroque's data entities they impact:

- **Event Types**
    - *ignore_unregistered*: shall Baroque ignore upon events publication all the events with a type that is not registered? If not, then raise an exception [boolean]
    - *pre_registered*: this is the list of _EventType_ subclasses that are pre-registered on the broker right from the start, so that it is possible to publish on the broker events of those types without further hassle [list of str, each one being a dotted Python class path]
- **Events**
    - *validate_schema*: shall Baroque validate event type schema upon all events upon publishing? If not, raise an exception [boolean]
    - *persist*: Shall Baroque persist all published events on the provided persistence provider? [boolean]
    - *persistence_provider*: this is the class implementing events persistence. Must be a subtype of *baroque.backend.PersistenceBackend* asbtract class [str, dotted Python class path]
- **Reactors**
    - *propagate_exceptions*: shall Baroque bubble up exceptions raised by any reactor whenever they occur? If not, catch them silently [boolean]
- **Topics**
    - *register_on_binding*: shall Baroque register a previously unregistered topic whenever a reactor is bound to it? If not, raise an exception [boolean]

Example of YML config file contents

This is an example of a possible YML file contents:

```
eventtypes:
  ignore_unregistered: false
  pre_registered:
    - baroque.entities.eventtype.GenericEventType
    - baroque.entities.eventtype.StateTransitionEventType
    - baroque.entities.eventtype.DataOperationEventType
    - baroque.entities.eventtype.MetricEventType
events:
  validate_schema: true
  persist: false
  persistence_backend: baroque.persistence.inmemory.DictBackend
reactors:
  propagate_exceptions: true
topics:
  register_on_binding: true
```

Persisting events

TBD

CHAPTER 2

Baroque API documentation

baroque package

Subpackages

baroque.datastructures package

Submodules

baroque.datastructures.bags module

class `baroque.datastructures.bags.`**`EventTypesBag`**(*eventtypes=None*)

Bases: `object`

A type-aware collection of event types

Parameters **`eventtypes`** (*`collection, optional`*) – collection of `baroque.entities.eventtypes.EventType` items.

`add`(*eventtypes*)

Adds a collection of eventtypes to this bag.

Parameters **`of`** (*`list`*) – the event types to be added

Raises *AssertionError* – when the supplied arg is not a collection or its items are not `baroque.entities.eventtypes.EventType` instances or `baroque.entities.eventtypes.EventType` subclasses

class `baroque.datastructures.bags.`**`ReactorsBag`**

Bases: `object`

A type-aware collection of reactors.

`count`()

Tells how many reactors are in this bag.

Returns int

`remove`(*reactor*)

Removes a reactor from this bag.

Parameters **`reactor`** (*`baroque.entities.reactor.Reactor`*) – the reactor to be removed

`remove_all`()

Removes all reactors from this bag.

`run`(*reactor*)

Adds a reactor to this bag.

Parameters **`reactor`** (*`baroque.entities.reactor.Reactor`*) – the reactor to be added

Raises *AssertionError* – when the supplied arg is not a *`baroque.entities.reactor.Reactor`* instance

`trigger`(*reactor*)

Alias for *baroque.datastructures.bags.ReactorBag.run* method

Parameters **`reactor`** (*`baroque.entities.reactor.Reactor`*) – the reactor to be added

Raises *AssertionError* – when the supplied arg is not a *`baroque.entities.reactor.Reactor`* instance

baroque.datastructures.counters module

class `baroque.datastructures.counters.`**`EventCounter`**

Bases: `object`

A counter of events.

`count`(*eventtype*)

Tells how many events have been counted of the specified type

Parameters **`eventtype`** (*`baroque.entities.eventtype.EventType`*) – the type of events to be counted

Returns int

`count_all`()

Tells how many events have been counted globally

Returns int

`increment_counting`(*event*)

Counts an event

Parameters **`event`** (*`baroque.entities.event.Event`*) – the event to be counted

baroque.datastructures.registries module

class `baroque.datastructures.registries.`**`EventTypesRegistry`**

Bases: `object`

Interface adapter to an event bag.

count()
Tells how many event types are registered on this registry

Returns int

register(*eventtype*)
Adds an event type to this registry.

Parameters **eventtype** (`baroque.entities.eventtypes.EventType` instance or *type* object) – the event type to be added

Raises *AssertionError* – when argument is not an `baroque.entities.eventtypes.EventType` instance or a *type* object

remove(*eventtype*)
Removes an event type from this registry

Parameters **eventtype** (`baroque.entities.eventtypes.EventType` instance or *type* object) – the event type to be removed

Raises *AssertionError* – when argument is not an `baroque.entities.eventtypes.EventType` instance or a *type* object

remove_all()
Removes all event types from this registry.

class `baroque.datastructures.registries.`**ReactorsRegistry**
Bases: `object`

A tracker for reaactors. Each reactor is intended to be executed when a certain type of events is fired: the reactors-eventtypes relationship is stored internally using a dict of bag datastructures.

Some reactors must be executed upon any event firing: these are stored internally into a “jolly bag”.

get_bag(*eventtype*)
Gives the reactors bag associated to the specified event type.

Parameters **eventtype** (*baroque.entities.eventtype.EventType*) – the associated event type

Returns *baroque.datastructures.bags.ReactorsBag*

Raises *AssertionError* – when the supplied event type is not a *baroque.entities.eventtype.EventType* instance or a *type* object

get_event_types_registry()
Gives the encapsulated event type registry

Returns `baroque.registries.EventTypeRegistry`

get_jolly_bag()
Gives the encapsulated bag that contains reactors to be executed upon any event firing.

Returns `baroque.registries.EventTypeRegistry`

get_or_create_bag(*eventtype*)
Gives the reactors bag associated to the specified event type, or creates one in case it does not exist yet.

Parameters **eventtype** (*baroque.entities.eventtype.EventType* instance or *type* object) – the associated event type

Returns *baroque.datastructures.bags.ReactorsBag*

Raises *AssertionError* – when the supplied event type is not a *baroque.entities.eventtype.EventType* instance or a *type* object

remove_all()
Clears the contents of all the encapsulated reactor bags.

to(*eventtype*)
Gives the encapsulated bag that contains reactors to be executed upon the firing of events of the supplied type.

Parameters **eventtype** (*baroque.entities.eventtype.EventType*) – the associated event type

Returns *baroque.datastructures.bags.ReactorsBag*

to_any_event()
Gives the encapsulated jolly bag, containing reactors to be executed upon the firing of any event.

Returns *baroque.datastructures.bags.ReactorsBag*

class baroque.datastructures.registries.**TopicsRegistry**
Bases: object

A tracker for reactors to be executed upong event firing of events on specified topics: the reactors-topics relationship is stored internally using a dict

count()
Tells how many topics are registered.

Returns int

new(*name*, *eventtypes*, ***kwargs*)
Creates a new topic, adds it to the registry and returns it.

Parameters **name** (*str*) – name of the new topic

eventtypes (collection): the *baroque.entities.eventtype.EventType* objects that characterize the new topi
**kwargs: positional arguments for *Topic* instantiation

Returns *baroque.entities.topic.Topic*

of(*owner*)
Returns the topics belonging to the supplied owner

Parameters **owner** (*str*) – the topics owner

Returns *list* of *baroque.entities.topic.Topic* items

on_topic_run(*topic*, *reactor*)
Binds the specified reactor to event firing on the specified topic.

Parameters

- **topic** (*:obj:'baroque.entities.topic.Topic*) – the topic
- **reactor** (*baroque.entities.reactor.Reactor*) – the reactor

Raises *AssertionError* – when any of the supplied args is of wrong type

publish_on_topic(*event*, *topic*)

Publishes an event on a tracked topic, executing all the reactors bound to that topic.

Parameters

- **event** (*:obj:'baroque.entities.event.Event*) – the event to be published
- **topic** (*:obj:'baroque.entities.topic.Topic*) – the target topic

Raises *AssertionError* – when any of the supplied args is of wrong type

register(*topic*)
Adds a topic to the registry.

Parameters **topic** (*baroque.entities.topic.Topic*) – the topic to be added

remove(*topic*)
Removes a topic from the registry.

Parameters **topic** (*baroque.entities.topic.Topic*) – the topic to be removed

remove_all()
Clears all the topics from the registry.

with_id(*id*)
Returns the topic with the specified identifier

Parameters **id** (*str*) – the topic id

Returns *baroque.entities.topic.Topic*

with_name(*name*)
Returns the topic with the specified name (exact string matching)

Parameters **name** (*str*) – the topic name

Returns *baroque.entities.topic.Topic*

with_tags(*tags*)
Returns the topics marked by the specified tags.

Parameters **tags** (*set* of str items) – the tag set

Returns *list* of *baroque.entities.topic.Topic* items

Raises *AssertionError* – when the supplied tag set is not an iterable

Module contents

baroque.defaults package

Submodules

baroque.defaults.config module

Default Baroque configuration

baroque.defaults.events module

class baroque.defaults.events.**EventFactory**
Bases: object

A factory class that exposes methods to quickly create useful *baroque.entities.event.Event* instances

classmethod new(***kwargs*)
Factory method returning a generic type event.

Parameters ****kwargs** – positional arguments for *Event* instantiation

Returns `baroque.entities.event.Event`

baroque.defaults.eventtypes module

class `baroque.defaults.eventtypes.`**`DataOperationEventType`**(*owner=None*)

Bases: `baroque.entities.eventtype.EventType`

Describes events cast when some kind of operation is done on a piece of data. Suitable i.e. to track CRUD operations on DB tables or whole datastores. Details about the impacted data entity and the operation are conveyed in the event payload.

Parameters **`owner`** (*`str, optional`*) – ID of the owner of this event type.

class `baroque.defaults.eventtypes.`**`GenericEventType`**(*owner=None*)

Bases: `baroque.entities.eventtype.EventType`

Describes generic events with a free-form content.

Parameters **`owner`** (*`str, optional`*) – ID of the owner of this event type.

class `baroque.defaults.eventtypes.`**`MetricEventType`**(*owner=None*)

Bases: `baroque.entities.eventtype.EventType`

Describes events carrying metric data. Suitable i.e. to track values about measured physical quantities. The metric name and value are conveyed in the event payload.

Parameters **`owner`** (*`str, optional`*) – ID of the owner of this event type.

class `baroque.defaults.eventtypes.`**`StateTransitionEventType`**(*owner=None*)

Bases: `baroque.entities.eventtype.EventType`

Describes events cast when something changes its state. Suitable i.e. to track state machines changes. Old and new states are conveyed in the event payload, as well as the cause of the state transition.

Parameters **`owner`** (*`str, optional`*) – ID of the owner of this event type.

baroque.defaults.reactors module

class `baroque.defaults.reactors.`**`ReactorFactory`**

Bases: `object`

A factory class that exposes methods to quickly create useful `baroque.entities.reactor.Reactor` instances

classmethod **`call_function`**(*obj*, *function_name*, **args*, ***kwargs*)

Factory method returning a reactor that calls a method on an object.

Parameters

- **`obj`** (*`object`*) – the target object
- **`function_name`** (*`function`*) – the function to be invoked on the object

Returns `baroque.entities.reactor.Reactor`

classmethod **`json_webhook`**(*url*, *payload*, *query_params=None*, *headers=None*)

Factory method returning a reactor that POSTs arbitrary JSON data to a webhook, along with the specified HTTP headers.

Parameters

- **`url`** (*`str`*) – the webhook URL

- **`payload`** (*`dict`*) – payload data dict to be dumped to JSON and sent
- **`query_params`** (*`dict`*) – dict of query parameters
- **`headers`** (*`dict`*) – dict of headers

Returns A dict containing the response HTTP status code (int) and payload (str), ie: `{'status': 200, 'payload': None}`

classmethod `log_event` (*logger*, *loglevel*)

Factory method returning a reactor that logs on a logger at a specified loglevel.

Parameters

- **`logger`** (`logging.Logger`) – the logger object
- **`loglevel`** (*`int`*) – the logging level

Returns *`baroque.entities.reactor.Reactor`*

classmethod `stdout` ()

Factory method returning a reactor that prints events to stdout.

Returns *`baroque.entities.reactor.Reactor`*

Module contents

baroque.entities package

Submodules

baroque.entities.event module

class `baroque.entities.event.`**`Event`** (*eventtype*, *payload=None*, *description=None*, *owner=None*)

Bases: `object`

An event that can be published.

Parameters

- **`eventtype`** (*`baroque.entities.eventtype.EventType`* instance or *type* object) – the type of the event
- **`payload`** (*`dict, optional`*) – the content of this event
- **`description`** (*`str, optional`*) – the description of this event
- **`owner`** (*`str, optional`*) – the owner of this event

`json` ()

Dumps this object to a JSON string.

Returns str

`md5` ()

Returns the MD5 hash of this object.

Returns str

`set_published` ()

Sets the status of this event to published.

set_unpublished()
Sets the status of this event to unpublished.

touch()
Sets the current time as timestamp of this event

class baroque.entities.event.**EventStatus**
Bases: object

Represents the binary state of events publication: `published` or `unpublished`

PUBLISHED = 'published'

UNPUBLISHED = 'unpublished'

baroque.entities.eventtype module

class baroque.entities.eventtype.**EventType**(*jsonschema*, *description=None*, *owner=None*)
Bases: object

The type of an event, describing its semantics and content.

Parameters

- **jsonschema** (*str*) – the JSON schema string describing the content of the events having this type
- **description** (*str, optional*) – the description of this event type
- **owner** (*str, optional*) – the owner of this event type

json()
Dumps this object to a JSON string.

Returns str

md5()
Returns the MD5 hash of this object.

Returns str

static validate(*evt*, *evttype*)
Validates the content of an event against the JSON schema of its type.

Parameters

- **evt** (*baroque.entities.event.Event*) – the event to be validated
- **evttype** (*baroque.entities.eventtype.EventType*) – the type of
- **event that needs to be validated** (*the*) –

Returns `True` if validation is OK, `False` otherwise

baroque.entities.reactor module

class baroque.entities.reactor.**Reactor**(*reaction*, *condition=None*)
Bases: object

An action to be executed whenever some type of events are published, with an optional condition to be satisfied satisfied. If a condition is set, this is checked out and if the outcome is `True` then the action is executed. If no condition is set, then the action is always executed.

Parameters

- **`reaction`** (*`function`*) – the action to be executed
- **`condition`** (*`function, optional`*) – the boolean condition to be satisfied

Raises *AssertionError* – when the supplied reaction is *None* or is not a callable, or (when supplied) when the condition is not a callable

`count_reactions``()`
Gives the number of times this reactor's action has been executed

Returns int

`last_event_reacted``()`
Gives the ID of the last event this reactor reacted on

Returns str

`last_reacted_on``()`
Gives the timestamp of the last time when the reactor's action has been executed

Returns str if reactor reacted at least once, `None` otherwise

`only_if``(`*condition*`)`
Sets the boolean condition for this reactor.

Parameters **`condition`** (*`function`*) – the boolean condition to be satisfied

`react``(`*event*`)`
Execute the action of this reactor.

Note: the condition of this reactor is out of the scope of this method (please see method :obj:`react_conditionally()`)

Parameters

- **`reaction`** (*`function`*) – the action to be executed
- **`condition`** (*`function, optional`*) – the boolean condition to be satisfied

`react_conditionally``(`*event*`)`
First checks if the condition is satisfied, then based on the outcome executes the action.

Parameters **`event`** (*`baroque.entities.event.Event`*) – the triggering event

baroque.entities.topic module

class `baroque.entities.topic.`**`Topic`**`(`*name, eventtypes, description=None, owner=None, tags=None*`)`
Bases: `object`

A distribution channel where events of specific types can be published and can be seen by subscribers of the topic. Topic subscribers will attach a reactor to the topic, which will be fired whenever any event of the types that are supported by the topic is published on the topic itself.

Parameters

- **`name`** (*`str`*) – the name of this topic

- **eventtypes** (*collection*) – the *baroque.entities.eventtype.EventType* objects that characterize this topic
- **description** (*str, optional*) – a description of this topic
- **owner** (*str, optional*) – the owner of this topic
- **tags** (*set, optional*) – the *set* of tags that describe this topic

Raises *AssertionError* – name or tags are *None* or have a wrong type

eventtypes
baroque.datastructures.bags.EventTypesBag – bag containing the event types of this topic

json()
Dumps this object to a JSON string.

Returns str

md5()
Returns the MD5 hash of this object.

Returns str

touch()
Sets the current time as timestamp of this topic

Module contents

baroque.exceptions package

Submodules

baroque.exceptions.configuration module

exception baroque.exceptions.configuration.**ConfigurationNotFoundError**
Bases: Exception

Raised when configuration source file is not available

exception baroque.exceptions.configuration.**ConfigurationParseError**
Bases: Exception

Raised on failures in parsing configuration data

baroque.exceptions.eventtypes module

exception baroque.exceptions.eventtypes.**InvalidEventSchemaError**
Bases: Exception

Raised when the event validation against its eventtype JSON schema fails

exception baroque.exceptions.eventtypes.**UnregisteredEventTypeError**
Bases: Exception

Raised when attempting to publish on the broker events of an unregistered type

baroque.exceptions.topics module

exception `baroque.exceptions.topics.`**`UnregisteredTopicError`**

Bases: `Exception`

Raised when attempting to publish events on a topic that is not registered on the broker

Module contents

baroque.persistence package

Submodules

baroque.persistence.backend module

class `baroque.persistence.backend.`**`PersistenceBackend`**

Bases: `object`

`create`(*event*)

Persists an event.

Parameters **`event`** (*`baroque.entities.event.Event`*) – the event to be persisted

`delete`(*event_id*)

Deletes an event.

Parameters **`event_id`** (*`str`*) – the identifier of the event to be deleted

`read`(*event_id*)

Loads an event.

Parameters **`event_id`** (*`str`*) – the identifier of the event to be loaded

Returns *`baroque.entities.event.Event`*

`update`(*event*)

Updates the event.

Parameters **`event`** (*`baroque.entities.event.Event`*) – the event to be updated

baroque.persistence.inmemory module

class `baroque.persistence.inmemory.`**`DictBackend`**

Bases: *`baroque.persistence.backend.PersistenceBackend`*

An in-memory `baroque.persistence.backend.ConfigurationBackend:` implementation backed by Python dict

`clear`()

Clears all the key-value pairs of this collection-like object.

`create`(*event*)

`delete`(*event_id*)

`keys`()

Gives the key set of this collection-like object.

Returns set

read(*event_id*)

update(*event*)

values()

Gives the value set of this collection-like object.

Returns set

Module contents

baroque.utils package

Submodules

baroque.utils.configreader module

Utility functions for handling with Baroque config datastructure

baroque.utils.configreader.**read_config_or_default**(*path_to_file*)

Loads configuration data from the supplied file or returns the default Baroque configuration.

Parameters **path_to_file** (*str, optional*) – Path to the configuration file.

Returns The configuration dictionary

Return type dict

Raises

- (*baroque.exceptions.configuration.ConfigurationNotFoundError*
- when the supplied filepath is not a regular file
- (*baroque.exceptions.configuration.ConfigurationParseError*
- when the supplied file cannot be parsed

baroque.utils.configreader.**readconfig**(*path_to_file*)

Loads configuration data from the supplied file.

Parameters **path_to_file** (*str, optional*) – Path to the configuration file.

Returns The configuration dictionary

Return type dict

Raises

- (*baroque.exceptions.configuration.ConfigurationNotFoundError*
- when the supplied filepath is not a regular file
- (*baroque.exceptions.configuration.ConfigurationParseError*
- when the supplied file cannot be parsed

baroque.utils.importer module

Utility functions for handling imports

`baroque.utils.importer.`**`class_from_dotted_path`**`(`*dotted_path*`)`

Loads a Python class from the supplied Python dot-separated class path. The class must be visible according to the PYTHONPATH variable contents.

Example

```
"package.subpackage.module.MyClass" --> MyClass
```

Parameters **`dotted_path`** (*`str`*) – the dot-separated path of the class

Returns a `type` object

baroque.utils.timestamp module

`baroque.utils.timestamp.`**`TIME_FORMAT`** **= '%Y-%m-%dT%H:%M:%SZ'**

str – ISO-8601 time format used for timestamp printing

`baroque.utils.timestamp.`**`stringify`**`(`*timestamp*`)`

Turns a timestamp into its ISO-8601 string representation.

Note: refer to the *TIME_FORMAT* template string

Parameters **`timestamp`** (`datetime.datetime`) – the timestamp to be stringified

Returns The ISO-8601 time formatted string

Return type str

`baroque.utils.timestamp.`**`utc_now`**`()`

Gives the current UTC time-aware timestamp.

Returns The UTC timestamp

Return type *datetime.datetime*

Module contents

Submodules

baroque.baroque module

class `baroque.baroque.`**`Baroque`**`(`*configfile=None*`)`

Bases: `object`

The Baroque event broker class.

Note: When no configuration file is specified, the default configuration is loaded.

Parameters **`configfile`** (*`str, optional`*) – Path to the configuration YML file.

Raises

- *baroque.exceptions.configuration.ConfigurationNotFoundError* – when the supplied filepath is not a regular file
- *baroque.exceptions.configuration.ConfigurationParseError* – when the supplied file cannot be parsed

configuration
`dict` – the configuration for this broker instance

events
baroque.datastructures.counters.EventCounter – counter of events published on this broker instance so far

eventtypes
baroque.datastructures.registries.EventTypesRegistry – registry of event types registered on this broker instance

fire (*event*)
Alias for *baroque.baroque.Baroque.publish()* method

on (*eventtype*)
Registers an event type on the broker.

Parameters **eventtype** (*baroque.entities.eventtype.EventType*) – the event type to be registered

Returns *baroque.datastructures.bags.ReactorsBag*

on_any_event_run (*reactor*)
Subscribes a reactor on the broker to be run upon any event firing.

Parameters **reactor** (*baroque.entities.reactor.Reactor*) – the reactor to be subscribed

Returns `baroque.datastructures.reactor.Reactor`

on_any_event_trigger (*reactor*)
Alias for *baroque.baroque.Baroque.on_any_event_run()* method

on_topic_run (*topic*, *reactor*)
Attaches a reactor on a topic registered on the broker.

Parameters

- **topic** (*baroque.entities.topic.Topic*) – the topic to which the reactor must be attached
- **reactor** (*baroque.entities.reactor.Reactor*) – the reactor to be attached to the topic

publish (*event*)
Publishes an event on the broker.

Note: This is a template-method

Parameters **event** (*baroque.entities.event.Event*) – the event to be published

publish_on_topic (*event*, *topic*)
Publishes an event on a specified topic registered on the broker.

Note: This is a template-method

Parameters

- **event** (*baroque.entities.event.Event*) – the event to be published on the topic
- **topic** (*baroque.entities.topic.Topic*) – the topic on which the event must be published

Raises *baroque.exceptions.topics.UnregisteredTopicError*: when trying to publish events on a topic that is not registered on the broker

reactors
`baroque.datastructures.registries.ReactorRegistry` – registry of reactors subscribed to this broker instance

reset()
Resets the reactors register and the published events counter of this broker instance.

topics
baroque.datastructures.registries.TopicsRegistry – registry of topics registered on this broker instance

version
`tuple` – version tuple for this broker instance

baroque.constants module

Baroque costant values

Module contents

Export of Baroque main classes

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index