Panko Documentation

Release 9.0.0.0rc2.dev2

OpenStack Foundation

Sep 25, 2020

CONTENTS

The Panko project is an event storage service that provides the ability to store and querying event data generated by Ceilometer with potentially other sources.

Panko is a component of the Telemetry project.

This documentation offers information on how Panko works and how to contribute to the project.

CHAPTER ONE

OVERVIEW

1.1 Installing Panko

1.1.1 Installing development sandbox

Configuring devstack

1. Download devstack.
2. Create a `local.conf` file as input to devstack.
3. The panko services are not enabled by default, so they must be enabled in `local.conf` before running `stack.sh`.

   This example `local.conf` file shows all of the settings required for panko:

   ```
   [[local|localrc]]
   # Enable the Panko devstack plugin
   enable_plugin panko https://opendev.org/openstack/panko.git
   ```

1.1.2 Installing Manually

Storage Backend Installation

This step is a prerequisite for the collector and API services. You may use one of the listed database backends below to store Panko data.

MongoDB

Follow the instructions to install the MongoDB package for your operating system, then start the service. The required minimum version of MongoDB is 2.4.x. You will also need to have pymongo 2.4 installed

To use MongoDB as the storage backend, change the 'database' section in panko.conf as follows:

```
[database]
connection = mongodb://username:password@host:27017/panko
```

SQLalchemy-supported DBs

You may alternatively use any SQLAlchemy-supported DB such as *PostgreSQL* or *MySQL*.

To use MySQL as the storage backend, change the 'database' section in panko.conf as follows:

```
[database]
connection = mysql+pymysql://username:password@host/panko?charset=utf8
```

Installing the API Server

Note: The API server needs to be able to talk to keystone and panko's database. It is only required if you choose to store data in legacy database or if you inject new samples via REST API.

1. Clone the panko git repository to the server:

   ```
   $ cd /opt/stack
   $ git clone https://opendev.org/openstack/panko.git
   ```

2. As a user with `root` permissions or `sudo` privileges, run the panko installer:

   ```
   $ cd panko
   $ sudo python setup.py install
   ```

3. Create a service for panko in keystone:

   ```
   $ openstack service create event --name panko \
                                    --description "Panko Service"
   ```

4. Create an endpoint in keystone for panko:

   ```
   $ openstack endpoint create $PANKO_SERVICE \
                               --region RegionOne \
                               --publicurl "http://$SERVICE_HOST:8977" \
                               --adminurl "http://$SERVICE_HOST:8977" \
                               --internalurl "http://$SERVICE_HOST:8977"
   ```

   Note: PANKO_SERVICE is the id of the service created by the first command and SERVICE_HOST is the host where the Panko API is running. The default port value for panko API is 8977. If the port value has been customized, adjust accordingly.

5. Choose and start the API server.

   Panko includes the `panko-api` command. This can be used to run the API server. For smaller or proof-of-concept installations this is a reasonable choice. For larger installations it is strongly recommended to install the API server in a WSGI host such as mod_wsgi (see *Installing the API behind mod_wsgi*). Doing so will provide better performance and more options for making adjustments specific to the installation environment.

   If you are using the `panko-api` command it can be started as:

```
$ panko-api
```

Note: The development version of the API server logs to stderr, so you may want to run this step using a screen session or other tool for maintaining a long-running program in the background.

1.1.3 Installing the API behind mod_wsgi

Panko comes with a few example files for configuring the API service to run behind Apache with `mod_wsgi`.

app.wsgi

The file `panko/api/app.wsgi` sets up the V2 API WSGI application. The file is installed with the rest of the panko application code, and should not need to be modified.

1.1.4 Installing the API with uwsgi

Panko comes with a few example files for configuring the API service to run behind Apache with `mod_wsgi`.

app.wsgi

The file `panko/api/app.wsgi` sets up the V2 API WSGI application. The file is installed with the rest of the Panko application code, and should not need to be modified.

Example of uwsgi configuration file

Create panko-uwsgi.ini file:

```
[uwsgi]
http = 0.0.0.0:8041
wsgi-file = <path_to_panko>/panko/api/app.wsgi
plugins = python
# This is running standalone
master = true
# Set die-on-term & exit-on-reload so that uwsgi shuts down
exit-on-reload = true
die-on-term = true
# uwsgi recommends this to prevent thundering herd on accept.
thunder-lock = true
# Override the default size for headers from the 4k default. (mainly for␣
↪keystone token)
buffer-size = 65535
enable-threads = true
# Set the number of threads usually with the returns of command nproc
threads = 8
# Make sure the client doesn't try to re-use the connection.
```

(continues on next page)

(continued from previous page)

```
add-header = Connection: close
# Set uid and gip to an appropriate user on your server. In many
# installations ``panko`` will be correct.
uid = panko
gid = panko
```

Then start the uwsgi server:

```
uwsgi ./panko-uwsgi.ini
```

Or start in background with:

```
uwsgi -d ./panko-uwsgi.ini
```

Configuring with uwsgi-plugin-python on Debian/Ubuntu

Install the Python plugin for uwsgi:

```
apt-get install uwsgi-plugin-python
```

Run the server:

```
uwsgi_python --master --die-on-term --logto /var/log/panko/panko-api.log \
    --http-socket :8042 --wsgi-file /usr/share/panko-common/app.wsgi
```

1.2 Contribution Guide

In the Contribution Guide, you will find documented policies for developing with Panko. This includes the processes we use for bugs, contributor onboarding, core reviewer memberships, and other procedural items.

1.2.1 Contributing to Panko

Panko follows the same workflow as other OpenStack projects. To start contributing to Panko, please follow the workflow found here.

Project Hosting Details

Bug tracker https://bugs.launchpad.net/panko

Mailing list http://lists.openstack.org/cgi-bin/mailman/listinfo/openstack-dev (prefix subjects with `[Panko]` for faster responses)

Contribution Guide https://docs.openstack.org/panko/latest/contributor/index.html

Code Hosting https://opendev.org/openstack/panko/

Code Review https://review.opendev.org/#/q/status:open+project:openstack/panko,n,z

1.2.2 Running the Tests

Panko includes an extensive set of automated unit tests which are run through tox.

1. Install `tox`:

   ```
   $ sudo pip install tox
   ```

2. On Ubuntu install `mongodb` and `libmysqlclient-dev` packages:

   ```
   $ sudo apt-get install mongodb
   $ sudo apt-get install libmysqlclient-dev
   ```

   For Fedora20 there is no `libmysqlclient-dev` package, so youll need to install `mariadb-devel.x86-64` (or `mariadb-devel.i386`) instead:

   ```
   $ sudo yum install mongodb
   $ sudo yum install mariadb-devel.x86_64
   ```

3. Install the test dependencies:

   ```
   $ sudo pip install -r /opt/stack/panko/test-requirements.txt
   ```

4. Run the unit and code-style tests:

   ```
   $ cd /opt/stack/panko
   $ tox -e py27,pep8
   ```

   As tox is a wrapper around testr, it also accepts the same flags as testr. See the testr documentation for details about these additional flags.

   Use a double hyphen to pass options to testr. For example, to run only tests under tests/api/v2:

   ```
   $ tox -e py27 -- api.v2
   ```

   To debug tests (ie. break into pdb debugger), you can use ”debug” tox environment. Here’s an example, passing the name of a test since you’ll normally only want to run the test that hits your breakpoint:

   ```
   $ tox -e debug panko.tests.test_bin
   ```

   For reference, the `debug` tox environment implements the instructions here: https://wiki.openstack.org/wiki/Testr#Debugging_.28pdb.29_Tests

5. There is a growing suite of tests which use a tool called gabbi to test and validate the behavior of the Panko API. These tests are run when using the usual `py27` tox target but if desired they can be run by themselves:

   ```
   $ tox -e gabbi
   ```

   The YAML files used to drive the gabbi tests can be found in `panko/tests/functional/gabbi/gabbits`. If you are adding to or adjusting the API you should consider adding tests here.

See also:

- tox

1.2.3 Guru Meditation Reports

Panko contains a mechanism whereby developers and system administrators can generate a report about the state of a running Panko executable. This report is called a *Guru Meditation Report* (*GMR* for short).

Generating a GMR

A *GMR* can be generated by sending the *USR1* signal to any Panko process with support (see below). The *GMR* will then be outputted standard error for that particular process.

For example, suppose that `panko-api` has process id `8675`, and was run with `2>/var/log/panko/panko-api.log`. Then, `kill -USR1 8675` will trigger the Guru Meditation report to be printed to `/var/log/panko/panko-api.log`.

Structure of a GMR

The *GMR* is designed to be extensible; any particular executable may add its own sections. However, the base *GMR* consists of several sections:

Package Shows information about the package to which this process belongs, including version information

Threads Shows stack traces and thread ids for each of the threads within this process

Green Threads Shows stack traces for each of the green threads within this process (green threads don't have thread ids)

Configuration Lists all the configuration options currently accessible via the CONF object for the current process

Adding Support for GMRs to New Executables

Adding support for a *GMR* to a given executable is fairly easy.

First import the module (currently residing in oslo-incubator), as well as the Panko version module:

```
from oslo_reports import guru_meditation_report as gmr
from panko import version
```

Then, register any additional sections (optional):

```
TextGuruMeditation.register_section('Some Special Section',
                                    some_section_generator)
```

Finally (under main), before running the "main loop" of the executable (usually `service.server(server)` or something similar), register the *GMR* hook:

```
TextGuruMeditation.setup_autorun(version)
```

Extending the GMR

As mentioned above, additional sections can be added to the GMR for a particular executable. For more information, see the inline documentation about oslo.reports: oslo.reports

1.3 Web API

1.3.1 V2 Web API

Capabilities

The Capabilities API allows you to directly discover which functions from the V2 API functionality, including the selectable aggregate functions, are supported by the currently configured storage driver. A capabilities query returns a flattened dictionary of properties with associated boolean values - a 'False' or absent value means that the corresponding feature is not available in the backend.

GET /v2/capabilities

Returns a flattened dictionary of API capabilities.

Capabilities supported by the currently configured storage driver.

Return type `Capabilities`

class `panko.api.controllers.v2.capabilities.`**`Capabilities`**`(**kw)`

A representation of the API and storage capabilities.

Usually constrained by restrictions imposed by the storage driver.

api

A flattened dictionary of API capabilities

event_storage

A flattened dictionary of event storage capabilities

Events and Traits

GET /v2/event_types

Get all event types.

Return type list(`str`)

GET /v2/event_types/(*event_type*)

Unused API, will always return 404.

Parameters

- **event_type** (`str`) -- A event type

GET /v2/event_types/(*event_type*)**/traits**

Return all trait names for an event type.

Parameters

- **event_type** (`str`) -- Event type to filter traits by

Return type list(`TraitDescription`)

GET /v2/event_types/(*event_type*)**/traits/**
event_type Return all instances of a trait for an event type.

Parameters

- **event_type** (`str`) -- Event type to filter traits by
- **trait_name** (`str`) -- Trait to return values for

Return type list(`Trait`)

GET /v2/events
Return all events matching the query filters.

Parameters

- **q** (list(`EventQuery`)) -- Filter arguments for which Events to return
- **limit** (`int`) -- Maximum number of samples to be returned.
- **sort** (list(`str`)) -- A pair of sort key and sort direction combined with ":"
- **marker** (`str`) -- The pagination query marker, message id of the last item viewed

Return type list(`Event`)

GET /v2/events/(*message_id*)
Return a single event with the given message id.

Parameters

- **message_id** (`str`) -- Message ID of the Event to be returned

Return type `Event`

class `panko.api.controllers.v2.events.`**Event**(***kw*)
A System event.

event_type
The type of the event

generated
The time the event occurred

message_id
The message ID for the notification

raw
The raw copy of notification

property traits
Event specific properties

class `panko.api.controllers.v2.events.`**Trait**(***kw*)
A Trait associated with an event.

name
The name of the trait

type
the type of the trait (string, integer, float or datetime)

value
the value of the trait

class panko.api.controllers.v2.events.**TraitDescription**(***kw*)
A description of a trait, with no associated value.

name
the name of the trait

type
the data type, defaults to string

Filtering Queries

class panko.api.controllers.v2.events.**EventQuery**(***kw*)
Query arguments for Event Queries.

field
Name of the field to filter on. Can be either a trait name or field of an event. 1) Use start_timestamp/end_timestamp to filter on *generated* field. 2) Specify the 'all_tenants=True' query parameter to get all events for all projects, this is only allowed by admin users.

type
the type of the trait filter, defaults to string

You can get API version list via request to endpoint root path. For example:

```
curl -H "X-AUTH-TOKEN: fa2ec18631f94039a5b9a8b4fe8f56ad" http://127.0.0.
↪1:8977
```

Sample response:

```
{
    "versions": {
        "values": [
            {
                "id": "v2",
                "links": [
                    {
                        "href": "http://127.0.0.1:8977/v2",
                        "rel": "self"
                    },
                    {
                        "href": "https://docs.openstack.org/",
                        "rel": "describedby",
                        "type": "text/html"
                    }
                ],
                "media-types": [
                    {
                        "base": "application/json",
                        "type": "application/vnd.openstack.telemetry-
↪v2+json"
                    },
                    {
```

(continues on next page)

(continued from previous page)

```
                        "base": "application/xml",
                        "type": "application/vnd.openstack.telemetry-v2+xml
↪"
                    }
                ],
                "status": "stable",
                "updated": "2013-02-13T00:00:00Z"
            }
        ]
    }
}
```

1.4 Source Code Index

1.4.1 panko

panko package

Subpackages

panko.api package

Subpackages

panko.api.controllers package

Subpackages

panko.api.controllers.v2 package

Submodules

panko.api.controllers.v2.base module

class panko.api.controllers.v2.base.**AdvEnum**(*name*, **args*, ***kwargs*)
Bases: wsme.types.wsproperty

Handle default and mandatory for wtypes.Enum.

class panko.api.controllers.v2.base.**Base**(***kw*)
Bases: wsme.types.DynamicBase

as_dict(*db_model*)

as_dict_from_keys(*keys*)

classmethod from_db_and_links(*m*, *links*)

classmethod from_db_model(*m*)

exception `panko.api.controllers.v2.base.`**`ClientSideError`**(*error*, *status_code=400*)
Bases: `wsme.exc.ClientSideError`

exception `panko.api.controllers.v2.base.`**`EntityNotFound`**(*entity*, *id*)
Bases: *`panko.api.controllers.v2.base.ClientSideError`*

class `panko.api.controllers.v2.base.`**`JsonType`**
Bases: `wsme.types.UserType`

A simple JSON type.

basetype
alias of `builtins.str`

name = 'json'

static validate(*value*)

exception `panko.api.controllers.v2.base.`**`ProjectNotAuthorized`**(*id*, *aspect='project'*)
Bases: *`panko.api.controllers.v2.base.ClientSideError`*

class `panko.api.controllers.v2.base.`**`Query`**(***kw*)
Bases: *`panko.api.controllers.v2.base.Base`*

Query filter.

as_dict()

field
The name of the field to test

get_op()

property op
The comparison operator. Defaults to 'eq'.

classmethod sample()

set_op(*value*)

type
The data type of value to compare against the stored data

value
The value to compare against the stored data

panko.api.controllers.v2.capabilities module

class `panko.api.controllers.v2.capabilities.`**`Capabilities`**(***kw*)
Bases: *`panko.api.controllers.v2.base.Base`*

A representation of the API and storage capabilities.

Usually constrained by restrictions imposed by the storage driver.

api
A flattened dictionary of API capabilities

event_storage
A flattened dictionary of event storage capabilities

classmethod sample()

class panko.api.controllers.v2.capabilities.**CapabilitiesController**(**args*, ***kwargs*)
Bases: `pecan.rest.RestController`

Manages capabilities queries.

get()
Returns a flattened dictionary of API capabilities.

Capabilities supported by the currently configured storage driver.

panko.api.controllers.v2.events module

class panko.api.controllers.v2.events.**Event**(***kw*)
Bases: *panko.api.controllers.v2.base.Base*

A System event.

event_type
The type of the event

generated
The time the event occurred

get_traits()

message_id
The message ID for the notification

raw
The raw copy of notification

classmethod sample()

set_traits(*traits*)

property traits
Event specific properties

class panko.api.controllers.v2.events.**EventQuery**(***kw*)
Bases: *panko.api.controllers.v2.base.Query*

Query arguments for Event Queries.

field
Name of the field to filter on. Can be either a trait name or field of an event. 1) Use start_timestamp/end_timestamp to filter on *generated* field. 2) Specify the 'all_tenants=True' query parameter to get all events for all projects, this is only allowed by admin users.

property op

classmethod sample()

type
the type of the trait filter, defaults to string

value

Complex type attribute definition.

Example:

```
class MyComplexType(wsme.types.Base):
    optionalvalue = int
    mandatoryvalue = wsattr(int, mandatory=True)
    named_value = wsattr(int, name='named.value')
```

After inspection, the non-wsattr attributes will be replaced, and the above class will be equivalent to:

```
class MyComplexType(wsme.types.Base):
    optionalvalue = wsattr(int)
    mandatoryvalue = wsattr(int, mandatory=True)
```

class panko.api.controllers.v2.events.**EventTypesController**(**args, **kwargs*)

Bases: `pecan.rest.RestController`

Works on Event Types in the system.

get_all()

Get all event types.

get_one(*event_type*)

Unused API, will always return 404.

Parameters **event_type** -- A event type

traits = <panko.api.controllers.v2.events.TraitsController object>

class panko.api.controllers.v2.events.**EventsController**(**args, **kwargs*)

Bases: `pecan.rest.RestController`

Works on Events.

get_all(*q=None, limit=None, sort=None, marker=None*)

Return all events matching the query filters.

Parameters

- **q** -- Filter arguments for which Events to return
- **limit** -- Maximum number of samples to be returned.
- **sort** -- A pair of sort key and sort direction combined with ":"
- **marker** -- The pagination query marker, message id of the last item viewed

get_one(*message_id*)

Return a single event with the given message id.

Parameters **message_id** -- Message ID of the Event to be returned

class panko.api.controllers.v2.events.**Trait**(***kw*)

Bases: *panko.api.controllers.v2.base.Base*

A Trait associated with an event.

name
The name of the trait

classmethod sample()

type
the type of the trait (string, integer, float or datetime)

value
the value of the trait

class panko.api.controllers.v2.events.**TraitDescription**(***kw*)
Bases: *panko.api.controllers.v2.base.Base*

A description of a trait, with no associated value.

name
the name of the trait

classmethod sample()

type
the data type, defaults to string

class panko.api.controllers.v2.events.**TraitsController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

Works on Event Traits.

get_all(*event_type*)
Return all trait names for an event type.

Parameters **event_type** -- Event type to filter traits by

get_one(*event_type*, *trait_name*)
Return all instances of a trait for an event type.

Parameters

- **event_type** -- Event type to filter traits by
- **trait_name** -- Trait to return values for

panko.api.controllers.v2.root module

class panko.api.controllers.v2.root.**V2Controller**
Bases: object

Version 2 API controller root.

capabilities = <panko.api.controllers.v2.capabilities.CapabilitiesController

event_types = <panko.api.controllers.v2.events.EventTypesController object>

events = <panko.api.controllers.v2.events.EventsController object>

panko.api.controllers.v2.utils module

`panko.api.controllers.v2.utils.`**`get_auth_project`**(*on_behalf_of=None*)

`panko.api.controllers.v2.utils.`**`requires_admin`**(*func*)

`panko.api.controllers.v2.utils.`**`requires_context`**(*func*)

`panko.api.controllers.v2.utils.`**`set_pagination_options`**(*sort*, *limit*, *marker*, *api_model*)

Sets the options for pagination specifying query options

Arguments: sort -- List of sorting criteria. Each sorting option has to format <sort key>:<sort direction>

Valid sort keys: message_id, generated (SUPPORT_SORT_KEYS in panko/event/storage/models.py) Valid sort directions: asc (ascending), desc (descending) (SUPPORT_DIRS in panko/event/storage/models.py) This defaults to asc if unspecified (DEFAULT_DIR in panko/event/storage/models.py)

impl_sqlalchemy.py: (see _get_pagination_query) If sort list is empty, this defaults to ['generated:asc', 'message_id:asc'] (DEFAULT_SORT in panko/event/storage/models.py)

limit -- Integer specifying maximum number of values to return

If unspecified, this defaults to pecan.request.cfg.api.default_api_return_limit

marker -- If specified, assumed to be an integer and assumed to be the message id of the last object on the previous page of the results

api_model -- Specifies the class implementing the api model to use for this pagination. The class is expected to provide the following members:

SUPPORT_DIRS SUPPORT_SORT_KEYS DEFAULT_DIR DEFAULT_SORT PRIMARY_KEY

Module contents

Submodules

panko.api.controllers.root module

`class` `panko.api.controllers.root.`**`VersionsController`**

Bases: `object`

`index`()

`panko.api.controllers.root.`**`version_descriptor`**(*base_url*, *version*, *released_on*)

`panko.api.controllers.root.`**`version_url`**(*base_url*, *version_number*)

Module contents

Submodules

panko.api.app module

panko.api.app.**app_factory**(*global_config*, ***local_conf*)

panko.api.app.**build_wsgi_app**(*argv=None*)

panko.api.app.**load_app**(*conf*, *appname='panko+keystone'*)

panko.api.app.**setup_app**(*root*, *conf*)

panko.api.hooks module

class panko.api.hooks.**ConfigHook**(*conf*)

Bases: `pecan.hooks.PecanHook`

Attach the configuration object to the request.

That allows controllers to get it.

before(*state*)

Override this method to create a hook that gets called after routing, but before the request gets passed to your controller.

Parameters **state** -- The Pecan `state` object for the current request.

class panko.api.hooks.**DBHook**(*conf*)

Bases: `pecan.hooks.PecanHook`

before(*state*)

Override this method to create a hook that gets called after routing, but before the request gets passed to your controller.

Parameters **state** -- The Pecan `state` object for the current request.

class panko.api.hooks.**TranslationHook**

Bases: `pecan.hooks.PecanHook`

after(*state*)

Override this method to create a hook that gets called after the request has been handled by the controller.

Parameters **state** -- The Pecan `state` object for the current request.

panko.api.middleware module

Middleware to replace the plain text message body of an error response with one formatted so the client can parse it.

Based on pecan.middleware.errordocument

class `panko.api.middleware.`**`ParsableErrorMiddleware`**(*app*)

Bases: `object`

Replace error body with something the client can parse.

static best_match_language(*accept_language*)

Determines best available locale from the Accept-Language header.

Returns the best language match or None if the 'Accept-Language' header was not available in the request.

panko.api.rbac module

Access Control Lists (ACL's) control access the API server.

`panko.api.rbac.`**`enforce`**(*policy_name*, *request*)

Return the user and project the request should be limited to.

Parameters

- **`request`** -- HTTP request
- **`policy_name`** -- the policy name to validate authz against.

`panko.api.rbac.`**`get_limited_to`**(*headers*)

Return the user and project the request should be limited to.

Parameters **`headers`** -- HTTP headers dictionary

Returns A tuple of (user, project), set to None if there's no limit on one of these.

`panko.api.rbac.`**`get_limited_to_project`**(*headers*)

Return the project the request should be limited to.

Parameters **`headers`** -- HTTP headers dictionary

Returns A project, or None if there's no limit on it.

`panko.api.rbac.`**`init`**()

`panko.api.rbac.`**`reset`**()

Module contents

panko.cmd package

Submodules

panko.cmd.storage module

`panko.cmd.storage.`**`dbsync`**`()`

`panko.cmd.storage.`**`expirer`**`()`

Module contents

panko.conf package

Submodules

panko.conf.defaults module

`panko.conf.defaults.`**`set_cors_middleware_defaults`**`()`
Update default configuration options for oslo.middleware.

Module contents

panko.policies package

Submodules

panko.policies.base module

`panko.policies.base.`**`list_rules`**`()`

panko.policies.segregation module

`panko.policies.segregation.`**`list_rules`**`()`

panko.policies.telemetry module

`panko.policies.telemetry.`**`list_rules`**`()`

Module contents

`panko.policies.`**`list_policies`**`()`

panko.publisher package

Submodules

panko.publisher.database module

`class` `panko.publisher.database.`**`DatabasePublisher`**`(`*ceilo_conf*, *parsed_url*`)`

Bases: `object`

Publisher class for recording event data into database.

The publisher class which records each event into a database configured in Ceilometer configuration file.

To enable this publisher, the following section needs to be present in panko.conf file

[database] connection = mysql+pymysql://panko:password@127.0.0.1/panko?charset=utf8

Then, panko:// should be added to Ceilometer's event_pipeline.yaml

`publish_events``(`*events*`)`

Module contents

panko.storage package

Subpackages

panko.storage.mongo package

Submodules

panko.storage.mongo.utils module

Common functions for MongoDB backend

`class` `panko.storage.mongo.utils.`**`ConnectionPool`**

Bases: `object`

`connect``(`*url*, *max_retries*, *retry_interval*`)`

class panko.storage.mongo.utils.**CursorProxy**(*cursor*, *max_retry*, *retry_interval*)
Bases: `pymongo.cursor.Cursor`

class panko.storage.mongo.utils.**MongoProxy**(*conn*, *max_retries*, *retry_interval*)
Bases: `object`

create_index(*keys*, *name=None*, **args*, ***kwargs*)

find(**args*, ***kwargs*)

panko.storage.mongo.utils.**make_events_query_from_filter**(*event_filter*)
Return start and stop row for filtering and a query.

Query is based on the selected parameter.

Parameters **event_filter** -- storage.EventFilter object.

panko.storage.mongo.utils.**make_timestamp_range**(*start*, *end*, *start_timestamp_op=None*, *end_timestamp_op=None*)
Create the query document to find timestamps within that range.

This is done by given two possible datetimes and their operations. By default, using $gte for the lower bound and $lt for the upper bound.

Module contents

panko.storage.sqlalchemy package

Submodules

panko.storage.sqlalchemy.models module

SQLAlchemy models for Panko data.

class panko.storage.sqlalchemy.models.**Event**(*message_id*, *event_type*, *generated*, *raw*)
Bases: `sqlalchemy.ext.declarative.api.Base`

event_type

event_type_id

generated

id

message_id

raw

class panko.storage.sqlalchemy.models.**EventType**(*event_type*)
Bases: `sqlalchemy.ext.declarative.api.Base`

Types of event records.

desc

id

class panko.storage.sqlalchemy.models.**JSONEncodedDict**(**args*, ***kwargs*)

Bases: sqlalchemy.sql.type_api.TypeDecorator

Represents an immutable structure as a json-encoded string.

impl

alias of sqlalchemy.sql.sqltypes.Text

static process_bind_param(*value*, *dialect*)

Receive a bound parameter value to be converted.

Subclasses override this method to return the value that should be passed along to the underlying TypeEngine object, and from there to the DBAPI execute() method.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

This operation should be designed with the reverse operation in mind, which would be the process_result_value method of this class.

Parameters

- **value** -- Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** -- the Dialect in use.

static process_result_value(*value*, *dialect*)

Receive a result-row column value to be converted.

Subclasses should implement this method to operate on data fetched from the database.

Subclasses override this method to return the value that should be passed back to the application, given a value that is already processed by the underlying TypeEngine object, originally from the DBAPI cursor method fetchone() or similar.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** -- Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** -- the Dialect in use.

This operation should be designed to be reversible by the "process_bind_param" method of this class.

class panko.storage.sqlalchemy.models.**PankoBase**

Bases: object

Base class for Panko Models.

update(*values*)

Make the model object behave like a dict.

class panko.storage.sqlalchemy.models.**PreciseTimestamp**(**args*, ***kwargs*)

Bases: sqlalchemy.sql.type_api.TypeDecorator

Represents a timestamp precise to the microsecond.

impl
alias of `sqlalchemy.sql.sqltypes.DateTime`

load_dialect_impl(*dialect*)
Return a `TypeEngine` object corresponding to a dialect.

This is an end-user override hook that can be used to provide differing types depending on the given dialect. It is used by the `TypeDecorator` implementation of `type_engine()` to help determine what type should ultimately be returned for a given `TypeDecorator`.

By default returns `self.impl`.

static process_bind_param(*value*, *dialect*)
Receive a bound parameter value to be converted.

Subclasses override this method to return the value that should be passed along to the underlying `TypeEngine` object, and from there to the DBAPI `execute()` method.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

This operation should be designed with the reverse operation in mind, which would be the process_result_value method of this class.

Parameters

- **value** -- Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **dialect** -- the `Dialect` in use.

static process_result_value(*value*, *dialect*)
Receive a result-row column value to be converted.

Subclasses should implement this method to operate on data fetched from the database.

Subclasses override this method to return the value that should be passed back to the application, given a value that is already processed by the underlying `TypeEngine` object, originally from the DBAPI cursor method `fetchone()` or similar.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** -- Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **dialect** -- the `Dialect` in use.

This operation should be designed to be reversible by the "process_bind_param" method of this class.

class `panko.storage.sqlalchemy.models.`**TraitDatetime**(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

Event datetime traits.

event_id

key

value

class `panko.storage.sqlalchemy.models.`**TraitFloat**(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

Event float traits.

event_id

key

value

class `panko.storage.sqlalchemy.models.`**TraitInt**(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

Event integer traits.

event_id

key

value

class `panko.storage.sqlalchemy.models.`**TraitText**(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

Event text traits.

event_id

key

value

Module contents

Submodules

panko.storage.base module

Base classes for storage engines

class `panko.storage.base.`**Connection**(*conf*)
Bases: `object`

Base class for event storage system connections.

CAPABILITIES = {'events': {'query': {'simple': False}}}

STORAGE_CAPABILITIES = {'storage': {'production_ready': False}}

static clear()
Clear database.

static clear_expired_data(*ttl*, *max_count=None*)
Clear expired data from the backend storage system.

Clearing occurs according to the time-to-live. :param ttl: Number of seconds to keep records for. :param max_count: Number of records to delete.

classmethod get_capabilities()
Return an dictionary with the capabilities of each driver.

static get_event_types()
Return all event types as an iterable of strings.

static get_events(*event_filter*, *pagination=None*)
Return an iterable of model.Event objects.

classmethod get_storage_capabilities()
Return a dictionary representing the performance capabilities.

This is needed to evaluate the performance of each driver.

static get_trait_types(*event_type*)
Return a dictionary containing the name and data type of the trait.

Only trait types for the provided event_type are returned. :param event_type: the type of the Event

static get_traits(*event_type*, *trait_type=None*)
Return all trait instances associated with an event_type.

If trait_type is specified, only return instances of that trait type. :param event_type: the type of the Event to filter by :param trait_type: the name of the Trait to filter by

static record_events(*events*)
Write the events to the backend storage system.

Parameters **events** -- a list of model.Event objects.

static upgrade()
Migrate the database to *version* or the most recent version.

class panko.storage.base.**Model**(***kwds*)
Bases: object

Base class for storage API models.

as_dict()

panko.storage.impl_elasticsearch module

class panko.storage.impl_elasticsearch.**Connection**(*url*, *conf*)
Bases: *panko.storage.base.Connection*

Put the event data into an ElasticSearch db.

Events in ElasticSearch are indexed by day and stored by event_type. An example document:

```
{"_index":"events_2014-10-21",
 "_type":"event_type0",
 "_id":"dc90e464-65ab-4a5d-bf66-ecb956b5d779",
 "_score":1.0,
 "_source":{"timestamp": "2014-10-21T20:02:09.274797"
            "traits": {"id4_0": "2014-10-21T20:02:09.274797",
                       "id3_0": 0.7510790937279408,
                       "id2_0": 5,
                       "id1_0": "18c97ba1-3b74-441a-b948-a702a30cbce2
↪"}
```

(continues on next page)

(continued from previous page)

```
                }
}
```

CAPABILITIES = {'events': {'query': {'simple': True}}}

STORAGE_CAPABILITIES = {'storage': {'production_ready': True}}

get_event_types()
Return all event types as an iterable of strings.

get_events(*event_filter*, *pagination=None*)
Return an iterable of model.Event objects.

get_trait_types(*event_type*)
Return a dictionary containing the name and data type of the trait.

Only trait types for the provided event_type are returned. :param event_type: the type of the Event

get_traits(*event_type*, *trait_type=None*)
Return all trait instances associated with an event_type.

If trait_type is specified, only return instances of that trait type. :param event_type: the type of the Event to filter by :param trait_type: the name of the Trait to filter by

record_events(*events*)
Write the events to the backend storage system.

Parameters **events** -- a list of model.Event objects.

upgrade()
Migrate the database to *version* or the most recent version.

panko.storage.impl_log module

class panko.storage.impl_log.**Connection**(*conf*)
Bases: *panko.storage.base.Connection*

Log event data.

static clear_expired_data(*ttl*, *max_count*)
Clear expired data from the backend storage system.

Clearing occurs according to the time-to-live.

Parameters

- **ttl** -- Number of seconds to keep records for.
- **max_count** -- Number of records to delete.

panko.storage.impl_mongodb module

MongoDB storage backend

class panko.storage.impl_mongodb.**Connection**(*url*, *conf*)

Bases: *panko.storage.pymongo_base.Connection*

Put the event data into a MongoDB database.

CONNECTION_POOL = <panko.storage.mongo.utils.ConnectionPool object>

clear()

Clear database.

clear_expired_data(*ttl*, *max_count=None*)

Clear expired data from the backend storage system.

Clearing occurs according to the time-to-live.

Parameters

- **ttl** -- Number of seconds to keep records for.
- **max_count** -- Number of records to delete (not used for MongoDB).

static update_ttl(*ttl*, *ttl_index_name*, *index_field*, *coll*)

Update or create time_to_live indexes.

Parameters

- **ttl** -- time to live in seconds.
- **ttl_index_name** -- name of the index we want to update or create.
- **index_field** -- field with the index that we need to update.
- **coll** -- collection which indexes need to be updated.

upgrade()

Migrate the database to *version* or the most recent version.

panko.storage.impl_sqlalchemy module

SQLAlchemy storage backend.

class panko.storage.impl_sqlalchemy.**Connection**(*url*, *conf*)

Bases: *panko.storage.base.Connection*

Put the event data into a SQLAlchemy database.

Tables:

```
- EventType
  - event definition
  - { id: event type id
      desc: description of event
      }
- Event
  - event data
  - { id: event id
```

(continues on next page)

(continued from previous page)

```
      message_id: message id
      generated = timestamp of event
      event_type_id = event type -> eventtype.id
      }
- TraitInt
  - int trait value
  - { event_id: event -> event.id
      key: trait name
      value: integer value
      }
- TraitDatetime
  - datetime trait value
  - { event_id: event -> event.id
      key: trait name
      value: datetime value
      }
- TraitText
  - text trait value
  - { event_id: event -> event.id
      key: trait name
      value: text value
      }
- TraitFloat
  - float trait value
  - { event_id: event -> event.id
      key: trait name
      value: float value
      }
```

CAPABILITIES = {'events': {'query': {'simple': True}}}

STORAGE_CAPABILITIES = {'storage': {'production_ready': True}}

clear()

Clear database.

clear_expired_data(*ttl*, *max_count*)

Clear expired data from the backend storage system.

Clearing occurs according to the time-to-live.

Parameters

- **ttl** -- Number of seconds to keep records for.
- **max_count** -- Number of records to delete.

static dress_url(*url*)

get_event_types()

Return all event types as an iterable of strings.

get_events(*event_filter*, *pagination=None*)

Return an iterable of model.Event objects.

Parameters

- **event_filter** -- EventFilter instance
- **pagination** -- Pagination parameters.

get_trait_types(*event_type*)
Return a dictionary containing the name and data type of the trait.

Only trait types for the provided event_type are returned. :param event_type: the type of the Event

get_traits(*event_type*, *trait_type=None*)
Return all trait instances associated with an event_type.

If trait_type is specified, only return instances of that trait type. :param event_type: the type of the Event to filter by :param trait_type: the name of the Trait to filter by

record_events(*event_models*)
Write the events to SQL database via sqlalchemy.

Parameters **event_models** -- a list of model.Event objects.

upgrade()
Migrate the database to *version* or the most recent version.

panko.storage.models module

Model classes for use in the events storage API.

class panko.storage.models.**Event**(*message_id*, *event_type*, *generated*, *traits*, *raw*)
Bases: *panko.storage.base.Model*

A raw event from the source system. Events have Traits.

Metrics will be derived from one or more Events.

DEFAULT_DIR = 'asc'

DEFAULT_SORT = [('generated', 'asc'), ('message_id', 'asc')]

DUPLICATE = 1

INCOMPATIBLE_TRAIT = 3

PRIMARY_KEY = 'message_id'

SUPPORT_DIRS = ('asc', 'desc')

SUPPORT_SORT_KEYS = ('message_id', 'generated')

UNKNOWN_PROBLEM = 2

append_trait(*trait_model*)

serialize()

class panko.storage.models.**Trait**(*name*, *dtype*, *value*)
Bases: *panko.storage.base.Model*

A Trait is a key/value pair of data on an Event.

The value is variant record of basic data types (int, date, float, etc).

DATETIME_TYPE = 4

FLOAT_TYPE = 3

INT_TYPE = 2

NONE_TYPE = 0

TEXT_TYPE = 1

classmethod convert_value(*trait_type*, *value*)

classmethod get_name_by_type(*type_id*)

classmethod get_type_by_name(*type_name*)

get_type_name()

classmethod get_type_names()

serialize()

type_names = {0: 'none', 1: 'string', 2: 'integer', 3: 'float', 4: 'dat

panko.storage.models.**serialize_dt**(*value*)

Serializes parameter if it is datetime.

panko.storage.pymongo_base module

Common functions for MongoDB backend

class panko.storage.pymongo_base.**Connection**(*conf*)

Bases: *panko.storage.base.Connection*

Base event Connection class for MongoDB driver.

CAPABILITIES = {'events': {'query': {'simple': True}}}

STORAGE_CAPABILITIES = {'storage': {'production_ready': True}}

get_event_types()

Return all event types as an iter of strings.

get_events(*event_filter*, *pagination=None*)

Return an iter of models.Event objects.

Parameters

- **event_filter** -- storage.EventFilter object, consists of filters for events that are stored in database.
- **pagination** -- Pagination parameters.

get_trait_types(*event_type*)

Return a dictionary containing the name and data type of the trait.

Only trait types for the provided event_type are returned.

Parameters **event_type** -- the type of the Event.

get_traits(*event_type*, *trait_name=None*)

Return all trait instances associated with an event_type.

If trait_type is specified, only return instances of that trait type.

Parameters

- **event_type** -- the type of the Event to filter by

- **trait_name** -- the name of the Trait to filter by

record_events (*event_models*)
Write the events to database.

Parameters **event_models** -- a list of models.Event objects.

Module contents

Storage backend management

class panko.storage.**EventFilter** (*start_timestamp=None*, *end_timestamp=None*, *event_type=None*, *message_id=None*, *traits_filter=None*, *admin_proj=None*)

Bases: object

Properties for building an Event query.

Parameters

- **start_timestamp** -- UTC start datetime (mandatory)
- **end_timestamp** -- UTC end datetime (mandatory)
- **event_type** -- the name of the event. None for all.
- **message_id** -- the message_id of the event. None for all.
- **admin_proj** -- the project_id of admin role. None if non-admin user.
- **traits_filter** -- the trait filter dicts, all of which are optional. This parameter is a list of dictionaries that specify trait values:

```
{'key': <key>,
'string': <value>,
'integer': <value>,
'datetime': <value>,
'float': <value>,
'op': <eq, lt, le, ne, gt or ge> }
```

exception panko.storage.**InvalidMarker**
Bases: Exception

Invalid pagination marker parameters

exception panko.storage.**StorageBadAggregate**
Bases: Exception

Error raised when an aggregate is unacceptable to storage backend.

code = 400

exception panko.storage.**StorageBadVersion**
Bases: Exception

Error raised when the storage backend version is not good enough.

exception panko.storage.**StorageUnknownWriteError**
Bases: Exception

Error raised when an unknown error occurs while recording.

`panko.storage.`**`get_connection`**`(`*url*, *conf*`)`
: Return an open connection to the database.

`panko.storage.`**`get_connection_from_config`**`(`*conf*`)`

Submodules

panko.i18n module

oslo.i18n integration module.

See https://docs.openstack.org/oslo.i18n/latest/user/usage.html

`panko.i18n.`**`get_available_languages`**`()`

`panko.i18n.`**`translate`**`(`*value*, *user_locale*`)`

panko.opts module

`panko.opts.`**`list_opts`**`()`

panko.profiler module

`class` `panko.profiler.`**`WsgiMiddleware`**`(`*application*, ***kwargs*`)`
: Bases: `object`

  `classmethod factory``(`*global_conf*, ***local_conf*`)`

`panko.profiler.`**`setup`**`(`*conf*`)`

`panko.profiler.`**`trace_cls`**`(`*name*, ***kwargs*`)`
: Wrap the OSprofiler trace_cls.

  Wrap the OSprofiler trace_cls decorator so that it will not try to patch the class unless OSprofiler is present.

  Parameters

  - **`name`** -- The name of action. For example, wsgi, rpc, db, ...
  - **`kwargs`** -- Any other keyword args used by profiler.trace_cls

panko.service module

`panko.service.`**`prepare_service`**`(`*argv=None*, *config_files=None*, *share=False*`)`

panko.utils module

Utilities and helper functions.

`panko.utils.`**`decimal_to_dt`**`(`*dec*`)`
Return a datetime from Decimal unixtime format.

`panko.utils.`**`decode_unicode`**`(`*input*`)`
Decode the unicode of the message, and encode it into utf-8.

`panko.utils.`**`dt_to_decimal`**`(`*utc*`)`
Datetime to Decimal.

Some databases don't store microseconds in datetime so we always store as Decimal unixtime.

`panko.utils.`**`recursive_keypairs`**`(`*d*, *separator=':'*`)`
Generator that produces sequence of keypairs for nested dictionaries.

`panko.utils.`**`sanitize_timestamp`**`(`*timestamp*`)`
Return a naive utc datetime object.

`panko.utils.`**`update_nested`**`(`*original_dict*, *updates*`)`
Updates the leaf nodes in a nest dict.

Updates occur without replacing entire sub-dicts.

panko.version module

Module contents

`exception` `panko.`**`NotImplementedError`**
Bases: *`NotImplementedError`*

`code = 501`

CHAPTER

TWO

SAMPLE CONFIGURATION FILES

2.1 Panko Sample Policy

The following is a sample panko policy file that has been auto-generated from default policy values in code. If you're using the default policies, then the maintenance of this file is not necessary, and it should not be copied into a deployment. Doing so will result in duplicate policy definitions. It is here to help explain which policy operations protect specific panko APIs, but it is not suggested to copy and paste into a deployment unless you're planning on providing a different policy for an operation that is not the default.

The sample policy file can also be viewed in `file form`.

```
#"context_is_admin": "role:admin"

# Return the user and project the requestshould be limited to
# GET  /v2/events
# GET  /v2/events/{message_id}
#"segregation": "rule:context_is_admin"

# Return all events matching the query filters.
# GET  /v2/events
#"telemetry:events:index": ""

# Return a single event with the given message id.
# GET  /v2/events/{message_id}
#"telemetry:events:show": ""
```