PyDash Documentation

Release 0.4.0

The PyDash Team

Jun 18, 2018

CONTENTS:

1 pydash **1**

1.1 config module . . . 1
1.2 conftest module . . . 1
1.3 flask_monitoring_dashboard_client package . . . 1
1.4 periodic_tasks package . . . 2
1.5 pydash module . . . 6
1.6 pydash_app package . . . 6
1.7 pydash_database package . . . 26
1.8 pydash_logger package . . . 27
1.9 pydash_mail package . . . 27
1.10 pydash_web package . . . 28
1.11 tests package . . . 32

2 Indices and tables **35**

Python Module Index **37**

Index **39**

CHAPTER

ONE

PYDASH

1.1 config module

class config.**Config**

Bases: object

MAIL_DEFAULT_SENDER = ('PyDash', 'noreply.pydashtestmail@gmail.com')

MAIL_PASSWORD = None

MAIL_PORT = 587

MAIL_SERVER = 'smtp.googlemail.com'

MAIL_USERNAME = None

MAIL_USE_TLS = True

SECRET_KEY = 'rigorous-security-will-be-implemented-post-MVP'

1.2 conftest module

conftest.**clean_in_memory_database**(*_)

1.3 flask_monitoring_dashboard_client package

1.3.1 Module contents

Performs the remote requests to the flask-monitoring-dashboard.

The method names in this module 1:1 reflect the names of the flask-monitoring-dashboard API (but without the word 'JSON' in them, because conversion from JSON to Python dictionaries/lists is one of the thing this module handles for you.)

flask_monitoring_dashboard_client.**get_data**(*dashboard_url*, *dashboard_token*, *time_from=None*, *time_to=None*, *timeout=1*)

Get data from a deployed flask-monitoring-dashboard :param dashboard_url: The base URL for the deployed dashboard, without trailing slash :param dashboard_token: The secret token for the dashboard, used to decode the Json Web Token response :param time_from: An optional datetime indicating only data since that moment should be included :param time_to: An optional datetime indicating only data up to that point should be included; only valid if time_from is also specified :param timeout: Optional timeout to wait for a response from the dashboard :return: A dict containing all monitoring data, possibly limited to the given time range

flask_monitoring_dashboard_client.**get_details**(*dashboard_url*, *timeout=1*)
Get details from a deployed flask-monitoring-dashboard :param dashboard_url: The base URL for the deployed dashboard, without trailing slash :param timeout: Optional timeout to wait for a response from the dashboard :return: A dict containing details from the dashboard, or None if the request was unsuccessful

flask_monitoring_dashboard_client.**get_monitor_rules**(*dashboard_url*, *dashboard_token*, *timeout=1*)
Get monitor rules from a deployed flask-monitoring-dashboard :param dashboard_url: The base URL for the deployed dashboard, without trailing slash :param dashboard_token: The secret token for the dashboard, used to decode the Json Web Token response :param timeout: Optional timeout to wait for a response from the dashboard :return: A dict containing monitor rules of the dashboard, or None if the request was unsuccessful

1.4 periodic_tasks package

1.4.1 Submodules

1.4.2 periodic_tasks.pqdict_iter_upto_priority module

class periodic_tasks.pqdict_iter_upto_priority.**pqdict_iter_upto_priority**(*pqueue*, *priority*)
Bases: object

Wrapper around *pqdict* to implement an iterator that returns items up to the given *priority* (exclusive). The rest of the pqdict is kept unchanged.

Pqueue An instance of the *pqdict.pqdict* class.

Priority The threshold priority.

The comparison function that the pqueue itself uses is used to cutoff this iterator, so it will automatically work with both min-queues as wel as max-queues.

1.4.3 periodic_tasks.queue_nonblocking_iter module

class periodic_tasks.queue_nonblocking_iter.**queue_nonblocking_iter**(*queue*)
Bases: object

This iterator wraps the queue.Queue/multiprocessing.Queue objects, which provide both a blocking API and a non-blocking API that raises errors when attempting to retrieve an item while it is empty.

Since these queues exist on multiple threads/processes, checking for (non)emptyness before attempting an action is not good enough, because its state might change in-between.

So instead, we handle the *queue.Empty* that is raised when attempting to retrieve the next item from an emtpy queue.

1.4.4 periodic_tasks.task_scheduler module

Contains the meat of the task scheduling: The TaskScheduler class, and a couple of classes that it uses under the hood.

class periodic_tasks.task_scheduler.**TaskScheduler**(*granularity=0.1*, *pool_settings={}*)

Bases: object

Runs tasks in a process pool of subprocesses (See *multiprocessing.Pool*). The task scheduler itself, which passes tasks on to this process pool, runs its scheduling loop in a separate subprocess as well. This means that there is no computational overhead for the main process at runtime.

Internally, an indexable priority queue (c.f. the *pqdict* package) is used to keep track of the next tasks to run. This makes the scheduling loop quite efficient, because tasks are already ordered (so only the oldest task's desired execution moment needs to be compared to the current timestamp). Because the priority queue is indexed, adding and removing a task is also done in *O(log(n))*.

Adding/updating/removing tasks is possible by using the same name as used previously for the task. Names can be strings, but also any other hashable object, so referring to a task based on a tuple of strings + integers is also possible.

Tasks can be added/updated/removed at any time, including before the scheduler is started.

The scheduler will be started by calling the *start()* function. It will stop scheduling and tear down the spawned processes when calling the *stop()* function. This function will also (in most cases) be automatically called when the main process finishes execution.

add_background_task(*name*, *task*)

Adds a task to be run only once (and as soon as possible) to the scheduler.

Name An identifier to find this task again later (and e.g. remove or alter it). Can be any hashable (using a string or a tuple of strings/integers is common.)

(Calling this function again with the same name will override the earlier task). :target: A function (or other callable) that will perform this task's functionality.

add_periodic_task(*name*, *interval*, *task*, *run_at_start=False*)

Adds a task to be run periodically to the scheduler.

Name An identifier to find this task again later (and e.g. remove or alter it). Can be any hashable (using a string or a tuple of strings/integers is common.)

(Calling this function again with the same name will override the earlier task). :target: A function (or other callable) that will perform this task's functionality. :interval: A datetime.timedelta representing how frequently to run the given target. :run_at_start: If true, runs task right after it was added to the scheduler, rather than only after the first interval has passed.

remove_task(*name*)

Removes a task that was previously added from the scheduler. Will do nothing if there is no task with the given name.

Name The task with this name will be removed.

start()

Starts the scheduler scheduling loop on a separate process.

Should only be called once per scheduler.

```
>>> import periodic_tasks as pt
>>> ts = pt.TaskScheduler()
>>> ts.start()
>>> ts.start()
Traceback (most recent call last):
   ...
Exception
```

stop()

Stops the scheduler scheduling loop.

Should only be called once per scheduler, and only after *start()* was called. When the program exits suddenly, this function will (in most cases) automatically be called to clean up the scheduling process.

```
>>> import periodic_tasks as pt
>>> ts = pt.TaskScheduler()
>>> ts.stop()
Traceback (most recent call last):
   ...
Exception
```

1.4.5 Module contents

Allows for the running of tasks in the background, as well as periodically. Tasks can either be added to the *default_task_scheduler*, or multiple schedulers can be created.

Tasks are run in a process pool of subprocesses (See *multiprocessing.Pool*). The task scheduler itself, which passes tasks on to this process pool, runs its scheduling loop in a separate subprocess as well. This means that there is no computational overhead for the main process at runtime.

Internally, an indexable priority queue (c.f. the *pqdict* package) is used to keep track of the next tasks to run. This makes the scheduling loop quite efficient, because tasks are already ordered (so only the oldest task's desired execution moment needs to be compared to the current timestamp). Because the priority queue is indexed, adding and removing a task is also done in *O(log(n))*.

Adding/updating/removing tasks is possible by using the same name as used previously for the task. Names can be strings, but also any other hashable object, so referring to a task based on a tuple of strings + integers is also possible.

Tasks can be added/updated/removed at any time, including before the scheduler is started.

The scheduler will be started by calling the *start()* function. It will stop scheduling and tear down the spawned processes when calling the *stop()* function. This function will also (in most cases) be automatically called when the main process finishes execution.

Example code with default scheduler:

```
>>> import periodic_tasks as pt
>>> import datetime
>>> pt.start_default_scheduler()
>>> pt.add_periodic_task('foo', datetime.timedelta(seconds=3), pt.foo)
>>> pt.add_periodic_task('bar', datetime.timedelta(seconds=5), pt.bar)
>>> pt.add_background_task('baz', pt.baz)
>>> pt.add_periodic_task('bar', datetime.timedelta(seconds=1), pt.bar) # overrides␣
↪previous `bar` task with new settings
>>> pt.remove_task('foo')
>>> pt.default_task_scheduler.stop()
```

Example code with custom scheduler:

```
>>> import periodic_tasks as pt
>>> ts = pt.TaskScheduler()
>>> import datetime, time
>>> ts.start()
>>> ts.add_periodic_task('foo', datetime.timedelta(milliseconds=1), pt.foo)
>>> ts.add_periodic_task('bar', datetime.timedelta(milliseconds=5), pt.bar)
```

(continues on next page)

(continued from previous page)

```
>>> time.sleep(2)
>>> ts.stop()
```

periodic_tasks.**add_background_task**(*name*, *task*, *scheduler=<periodic_tasks.task_scheduler.TaskScheduler object>*)

Adds a task to be run only once (and as soon as possible) to the given *scheduler*, which defaults to the global *default_task_scheduler* that this module provides.

Name An identifier to find this task again later (and e.g. remove or alter it). Can be any hashable (using a string or a tuple of strings/integers is common.)

(Calling this function again with the same name will override the earlier task). :target: A function (or other callable) that will perform this task's functionality. :scheduler: Which TaskScheduler to run the task on. It defaults to the global *default_task_scheduler* that this module provides.

periodic_tasks.**add_periodic_task**(*name*, *interval*, *task*, *run_at_start=False*, *scheduler=<periodic_tasks.task_scheduler.TaskScheduler object>*)

Adds a task to be run periodically to the given *scheduler*, which defaults to the global *default_task_scheduler* that this module provides.

Name An identifier to find this task again later (and e.g. remove or alter it). Can be any hashable (using a string or a tuple of strings/integers is common.)

(Calling this function again with the same name will override the earlier task). :target: A function (or other callable) that will perform this task's functionality. :interval: A datetime.timedelta representing how frequently to run the given target. :run_at_start: If true, runs task right after it was added to the scheduler, rather than only after the first interval has passed. :scheduler: Which TaskScheduler to run the task on. It defaults to the global *default_task_scheduler* that this module provides.

periodic_tasks.**bar**()

periodic_tasks.**baz**()

periodic_tasks.**foo**()

periodic_tasks.**periodic_task**(*name*, *interval*, *run_at_start=False*, *scheduler=<periodic_tasks.task_scheduler.TaskScheduler object>*)

Function decorator to specify that the following function should be called periodically; It accepts the same arguments as *add_periodic_task* (with the *target* argument filled in by the function being decorated.)

Usage:

@periodic_task('qux', datetime.timedelta(seconds=2)) def qux():

print('qux')

@periodic_task('qux', datetime.timedelta(seconds=2), run_at_start=True, scheduler = your_scheduler) def qux():

print('qux')

periodic_tasks.**qux**()

periodic_tasks.**remove_task**(*name*, *scheduler=<periodic_tasks.task_scheduler.TaskScheduler object>*)

Removes a task that was previously added from the given *scheduler*, which defaults to the global *default_task_scheduler* that this module provides.. Will do nothing if there is no task with the given name.

Name The task with this name will be removed.

Scheduler Which TaskScheduler to remove the task from. It defaults to the global *default_task_scheduler* that this module provides.

periodic_tasks.**start_default_scheduler**()
Starts the default (global) scheduler that this module provides.

1.5 pydash module

1.6 pydash_app package

1.6.1 Subpackages

pydash_app.dashboard package

Subpackages

pydash_app.dashboard.aggregator package

Submodules

pydash_app.dashboard.aggregator.aggregator_group module

class pydash_app.dashboard.aggregator.aggregator_group.**AggregatorGroup**(*endpoint_calls=[]*)
Bases: persistent.Persistent

Maintains a powerset of dicts of aggregators, such that we can filter based on: - time - IP - FMD's group_by - etc.

Involved usage example: >>> from datetime import datetime >>> from pydash_app.dashboard.endpoint_call import EndpointCall >>> from pydash_app.dashboard.aggregator.aggregator_group import AggregatorGroup >>> ag = AggregatorGroup() >>> ec1 = EndpointCall("foo", 0.5, datetime.strptime("2018-04-25 15:29:23", "%Y-%m-%d %H:%M:%S"), "0.1", "None", "127.0.0.1") >>> ec2 = EndpointCall("foo", 0.5, datetime.strptime("2018-04-26 15:29:23", "%Y-%m-%d %H:%M:%S"), "0.1", "None", "127.0.0.1") >>> ec3 = EndpointCall("foo", 0.5, datetime.strptime("2018-04-25 15:29:23", "%Y-%m-%d %H:%M:%S"), "0.1", "None", "127.0.0.2") >>> ag.add_endpoint_call(ec1) >>> ag.add_endpoint_call(ec2) >>> ag.add_endpoint_call(ec3) >>> >>> # Filter by day ... a_day = ag.fetch_aggregator({'day':'2018-04-25'}) >>> a_day.as_dict()['total_visits'] == 2 True >>> >>> # Filter by week ... a_week = ag.fetch_aggregator({'week':'2018-W17'}) >>> a_week.as_dict()['total_visits'] == 3 True >>> >>> # Filter by day and ip ... a_day_ip = ag.fetch_aggregator({'day':'2018-04-25', 'ip':'127.0.0.1'}) >>> a_day_ip.as_dict()['total_visits'] == 1 True >>> >>> # No filtering (all endpoint calls are included in this aggregator) ... a_all = ag.fetch_aggregator({}) >>> a_all.as_dict()['total_visits'] == 3 True >>> >>> # Filter over a datetime range ... start_datetime = datetime(ec1.time.year, ec1.time.month, ec1.time.day) >>> end_datetime = datetime(ec2.time.year, ec2.time.month, ec2.time.day + 1) >>> a_all2 = ag.fetch_aggregator_daterange({}, start_datetime, end_datetime) >>> a_all2.as_dict()['total_visits'] == 3 True >>> a_all.as_dict() == a_all2.as_dict() True

add_endpoint_call(*endpoint_call*)
Adds the given endpoint call to the right aggregators within the group.

fetch_aggregator(*filter_dict={}*)
Filters the internal collection of aggregators and returns the right one depending on filter_dict. :param filter_dict: A dictionary containing property_name-value pairs to filter on.

This is in the gist of *{'day':'2018-05-20', 'ip':'127.0.0.1'}*

The current filter_names are:

- Time: * ‘year’ - e.g. ‘2018’ * ‘month’ - e.g. ‘2018-05’ * ‘week’ - e.g. ‘2018-W17’ * ‘day’ - e.g. ‘2018-05-20’ * ‘hour’ - e.g. ‘2018-05-20T20’ * ‘minute’ - e.g. ‘2018-05-20T20-10’

Note that for Time filter-values, the formatting is crucial.

- Version: * ‘version’ - e.g. ‘1.0.1’
- IP: * ‘ip’ - e.g. ‘127.0.0.1’
- Group-by: * ‘group_by’ - e.g. ‘None’

Note that when providing two filters of the same type, a ValueError is raised.

Returns An Aggregator instance that contains the right aggregated data for this query. Note that if an invalid value is given, a new (and empty) Aggregator is returned, due to the lazy addition.

fetch_aggregator_daterange(*filters*, *datetime_begin*, *datetime_end*)

Fetches an aggregator over the entire provided datetime range. :param filters: A dictionary that contains property_name-value pairs to filter on.

This is in the gist of {‘ip’: ‘127.0.0.1’, ‘version’: ‘1.0.1’} For the complete set of possible filters, see AggregatorGroup.fetch_aggregator. Note: may not contain time-based filters, for obvious reasons.

Parameters

- **datetime_begin** – A datetime object indicating the inclusive lower bound for the datetime range to aggregate over.
- **datetime_end** – A datetime object indicating the exclusive upper bound for the datetime range to aggregate over.

Returns An Aggregator object that contains the aggregated data over the entirety of the specified datetime range.

fetch_aggregator_inclusive_daterange(*filters*, *datetime_begin*, *datetime_end*, *granularity*)

Fetches an aggregator over the entire provided datetime range. :param filters: A dictionary that contains property_name-value pairs to filter on.

This is in the gist of {‘ip’: ‘127.0.0.1’, ‘version’: ‘1.0.1’} For the complete set of possible filters, see AggregatorGroup.fetch_aggregator. Note: May not contain time-based filters, for obvious reasons.

Parameters

- **datetime_begin** – A datetime object indicating the inclusive lower bound for the datetime range to aggregate over.
- **datetime_end** – A datetime object indicating the inclusive upper bound for the datetime range to aggregate over.
- **granularity** – A string denoting the granularity of the daterange.

Returns An Aggregator object that contains the aggregated data over the entirety of the specified datetime range.

fetch_aggregators_per_timeslice(*filters*, *timeslice*, *start_datetime*, *end_datetime*)

These datetimes are treated as inclusive boundaries of a datetime range (e.g. [start_datetime,

end_datetime]. Assumes start_datetime and end_datetime are both from utc. :param filters: A dictionary that contains property_name-value pairs to filter on.

This is in the gist of {'ip': '127.0.0.1', 'version': '1.0.1'} For the complete set of possible filters, see AggregatorGroup.fetch_aggregator. Note: May not contain time-based filters, for obvious reasons.

Parameters

- **timeslice** – A string denoting at what granularity the indicated datetime range should be split. The currently supported values for this are: 'year', 'month', 'week', 'day', 'hour' and 'minute'.
- **start_datetime** – A datetime object indicating the inclusive lower bound for the datetime range to aggregate over.
- **end_datetime** – A datetime object indicating the inclusive upper bound for the datetime range to aggregate over.

Returns A list of tuples consisting of a datetime string (formatted according to the ISO-8601 standard) and the corresponding aggregator, over the specified datetime range.

partition_funs = [<AggregatorPartitionFun field_name=year category=time >, <Aggregator

Note to our internal dev team: To add more partitions to filter on, a corresponding AggregatorPartitionFun class instance should be created (together with its corresponding '**partition_by_**' function) and added to the *partition_funs* list above.

partition_powerset = <generator object powerset_generator>

partitions_set = frozenset({frozenset({<AggregatorPartitionFun field_name=version cate

class pydash_app.dashboard.aggregator.aggregator_group.**AggregatorPartitionFun**(*field_name*, *category*, *fun*)

Bases: object

pydash_app.dashboard.aggregator.aggregator_group.**calc_endpoint_call_identifier**(*partition*, *endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_by_day_fun**(*endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_by_group_by_fun**(*endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_by_hour_fun**(*endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_by_ip_fun**(*endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_by_minute_fun**(*endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_by_month_fun**(*endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_by_version_fun**(*endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_by_week_fun**(*endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_by_year_fun**(*endpoint_call*)

pydash_app.dashboard.aggregator.aggregator_group.**partition_field_names**(*partition*)

pydash_app.dashboard.aggregator.aggregator_group.**powerset_generator**(*i*)

pydash_app.dashboard.aggregator.aggregator_group.**remove_duplicate_categories**(*partition_funs*)

pydash_app.dashboard.aggregator.statistics module

class pydash_app.dashboard.aggregator.statistics.**AverageExecutionTime**
Bases: *pydash_app.dashboard.aggregator.statistics.FloatStatisticABC*

Keeps track of the average execution time of all endpoints that have been appended to it. Rendered value is rounded to 3 decimal places by default.

add_together(*other*, *dependencies_self*, *dependencies_other*)
Should return a new statistic where the internals of self and other are added together.

dependencies = [<class 'pydash_app.dashboard.aggregator.statistics.TotalVisits'>, <cla

empty()

field_name()

perform_append(*endpoint_call*, *dependencies*)

should_be_rendered()
Note: implementing subclasses should add the @property decorator. There was some strange behaviour where without adding the decorator, subclasses implementing it as *return True* behaved normally, but those implementing it as *return False* still were treated as if it returned True. Adding the @property decorator fixed it.

class pydash_app.dashboard.aggregator.statistics.**ExecutionTimePercentileABC**
Bases: *pydash_app.dashboard.aggregator.statistics.FloatStatisticABC*

Abstract base class for execution time percentile statistics.

add_together(*other*, *dependencies_self*, *dependencies_other*)
Should return a new statistic where the internals of self and other are added together.

dependencies = [<class 'pydash_app.dashboard.aggregator.statistics.ExecutionTimeTDiges

empty()

percentile_nr

perform_append(*endpoint_call*, *dependencies*)

should_be_rendered()
Note: implementing subclasses should add the @property decorator. There was some strange behaviour where without adding the decorator, subclasses implementing it as *return True* behaved normally, but those implementing it as *return False* still were treated as if it returned True. Adding the @property decorator fixed it.

class pydash_app.dashboard.aggregator.statistics.**ExecutionTimeTDigest**
Bases: *pydash_app.dashboard.aggregator.statistics.Statistic*

Acts as the general execution time tdigest, from which its dependants take their data from. This class is supposed to be instantiated, but not rendered.

add_together(*other*, *dependencies_self*, *dependencies_other*)
Should return a new statistic where the internals of self and other are added together.

empty()

field_name()

perform_append(*endpoint_call*, *dependencies*)

should_be_rendered
Note: implementing subclasses should add the @property decorator. There was some strange behaviour where without adding the decorator, subclasses implementing it as *return True* behaved normally, but those

implementing it as *return False* still were treated as if it returned True. Adding the @property decorator fixed it.

class pydash_app.dashboard.aggregator.statistics.**FastestExecutionTime**
Bases: *pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC*

field_name()

percentile_nr()

class pydash_app.dashboard.aggregator.statistics.**FastestQuartileExecutionTime**
Bases: *pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC*

field_name()

percentile_nr()

class pydash_app.dashboard.aggregator.statistics.**FloatStatisticABC**
Bases: *pydash_app.dashboard.aggregator.statistics.Statistic*

The FloatStatisticABC is the abstract base class for statistics that render a single floating point number. It specifies the default amount of digits to round its rendered value to as 3. (E.g. 2.54, 123, 0.3, but not 0.123)

nr_of_digits

rendered_value()

class pydash_app.dashboard.aggregator.statistics.**MedianExecutionTime**
Bases: *pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC*

field_name()

percentile_nr()

class pydash_app.dashboard.aggregator.statistics.**NinetiethPercentileExecutionTime**
Bases: *pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC*

field_name()

percentile_nr()

class pydash_app.dashboard.aggregator.statistics.**NinetyNinthPercentileExecutionTime**
Bases: *pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC*

field_name()

percentile_nr()

class pydash_app.dashboard.aggregator.statistics.**SlowestExecutionTime**
Bases: *pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC*

field_name()

percentile_nr()

class pydash_app.dashboard.aggregator.statistics.**SlowestQuartileExecutionTime**
Bases: *pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC*

field_name()

percentile_nr()

class pydash_app.dashboard.aggregator.statistics.**Statistic**
Bases: persistent.Persistent, abc.ABC

classmethod add_to_collection(*collection*)
cls should only be a class instead of an instance.

add_together(*other*, *dependencies_self*, *dependencies_other*)
Should return a new statistic where the internals of self and other are added together.

append(*endpoint_call*, *dependencies*)

dependencies = []

empty()

classmethod field_name()

perform_append(*endpoint_call*, *dependencies*)

rendered_value()

should_be_rendered
Note: implementing subclasses should add the @property decorator. There was some strange behaviour where without adding the decorator, subclasses implementing it as *return True* behaved normally, but those implementing it as *return False* still were treated as if it returned True. Adding the @property decorator fixed it.

class pydash_app.dashboard.aggregator.statistics.**TotalExecutionTime**
Bases: *pydash_app.dashboard.aggregator.statistics.FloatStatisticABC*

add_together(*other*, *dependencies_self*, *dependencies_other*)
Should return a new statistic where the internals of self and other are added together.

empty()

field_name()

perform_append(*endpoint_call*, *dependencies*)

should_be_rendered()
Note: implementing subclasses should add the @property decorator. There was some strange behaviour where without adding the decorator, subclasses implementing it as *return True* behaved normally, but those implementing it as *return False* still were treated as if it returned True. Adding the @property decorator fixed it.

class pydash_app.dashboard.aggregator.statistics.**TotalVisits**
Bases: *pydash_app.dashboard.aggregator.statistics.Statistic*

add_together(*other*, *dependencies_self*, *dependencies_other*)
Should return a new statistic where the internals of self and other are added together.

empty()

field_name()

perform_append(*endpoint_call*, *dependencies*)

should_be_rendered()
Note: implementing subclasses should add the @property decorator. There was some strange behaviour where without adding the decorator, subclasses implementing it as *return True* behaved normally, but those implementing it as *return False* still were treated as if it returned True. Adding the @property decorator fixed it.

class pydash_app.dashboard.aggregator.statistics.**UniqueVisitorsAllTime**
Bases: *pydash_app.dashboard.aggregator.statistics.Statistic*

add_together(*other*, *dependencies_self*, *dependencies_other*)
Should return a new statistic where the internals of self and other are added together.

empty()

field_name()

perform_append(*endpoint_call*, *dependencies*)

rendered_value()

should_be_rendered()
Note: implementing subclasses should add the @property decorator. There was some strange behaviour where without adding the decorator, subclasses implementing it as *return True* behaved normally, but those implementing it as *return False* still were treated as if it returned True. Adding the @property decorator fixed it.

class pydash_app.dashboard.aggregator.statistics.**Versions**
Bases: *pydash_app.dashboard.aggregator.statistics.Statistic*

add_together(*other*, *dependencies_self*, *dependencies_other*)
Should return a new statistic where the internals of self and other are added together.

empty()

field_name()

perform_append(*endpoint_call*, *dependencies*)

rendered_value()

should_be_rendered()
Note: implementing subclasses should add the @property decorator. There was some strange behaviour where without adding the decorator, subclasses implementing it as *return True* behaved normally, but those implementing it as *return False* still were treated as if it returned True. Adding the @property decorator fixed it.

class pydash_app.dashboard.aggregator.statistics.**VisitsPerIP**
Bases: *pydash_app.dashboard.aggregator.statistics.Statistic*

add_together(*other*, *dependencies_self*, *dependencies_other*)
Should return a new statistic where the internals of self and other are added together.

empty()

field_name()

perform_append(*endpoint_call*, *dependencies*)

rendered_value()

should_be_rendered()
Note: implementing subclasses should add the @property decorator. There was some strange behaviour where without adding the decorator, subclasses implementing it as *return True* behaved normally, but those implementing it as *return False* still were treated as if it returned True. Adding the @property decorator fixed it.

pydash_app.dashboard.aggregator.statistics.**date_dict**(*dict*)

pydash_app.dashboard.aggregator.statistics.**reduce_precision**(*value*, *nr_of_digits*)
Reduces the precision of *value* based on the amount of non-zero digits before the decimal point and *nr_of_digits*.

Examples: >>> x = 2/3 >>> reduce_precision(x, 3) 0.67 >>> x = 1234.5678 >>> reduce_precision(x, 3) 1235

Module contents

class pydash_app.dashboard.aggregator.**Aggregator**(*endpoint_calls=[]*)
Bases: persistent.Persistent

Maintains aggregate data for either a dashboard or a single endpoint. This data is updated every time a new endpoint call is added.

add_endpoint_call(*endpoint_call*)
Add an endpoint call and update aggregated data :param endpoint_call: *EndpointCall* instance to add

as_dict()
Return aggregated data in a dict. Only includes statistics that should be rendered. :return: A dict containing several aggregated data points

contained_statistics_classes = OrderedSet([<class 'pydash_app.dashboard.aggregator.sta

statistic
alias of *pydash_app.dashboard.aggregator.statistics.Versions*

statistics_classes_with_dependencies = OrderedSet([<class 'pydash_app.dashboard.aggrega

pydash_app.dashboard.services package

Submodules

pydash_app.dashboard.services.fetching module

pydash_app.dashboard.services.fetching.**fetch_and_add_endpoint_calls**(*dashboard*)
Retrieve the latest endpoint calls of the given dashboard and add them to it. :param dashboard: The dashboard for which to update endpoint calls.

pydash_app.dashboard.services.fetching.**fetch_and_add_endpoints**(*dashboard*)
For a given dashboard, initialize it with the endpoints it has registered. Note that this will not add endpoint call data. :param dashboard: The dashboard to initialize with endpoints.

pydash_app.dashboard.services.fetching.**fetch_and_add_historic_endpoint_calls**(*dashboard*)
For a given dashboard, retrieve all historical endpoint calls and add them to it. :param dashboard: The dashboard to initialize with historical data.

pydash_app.dashboard.services.fetching.**fetch_and_update_historic_dashboard_info**(*dashboard_id*)
Updates the dashboard with the historic EndpointCall information that is fetched from the Dashboard's remote location.

pydash_app.dashboard.services.fetching.**fetch_and_update_new_dashboard_info**(*dashboard_id*)
Updates the dashboard with the new EndpointCall information that is fetched from the Dashboard's remote location.

pydash_app.dashboard.services.fetching.**schedule_all_periodic_dashboards_tasks**(*interval=datetime.*
3600),
sched-
uler=<periodic_ta
ob-
ject>)

Sets up all tasks that should be run periodically for each of the dashboards. (For now, that is only the EndpointCall fetching task.)

pydash_app.dashboard.services.fetching.**schedule_historic_dashboard_fetching**(*dashboard*, *scheduler=<periodic_tasks object>*)

Schedules the fetching of historic EndpointCall information as a background task. The periodic fetching of new EndpointCall information is scheduled as soon as this task completes.

pydash_app.dashboard.services.fetching.**schedule_periodic_dashboard_fetching**(*dashboard*, *interval=datetime.timedelta(0, 3600)*, *scheduler=<periodic_tasks object>*)

Schedules the periodic EndpointCall fetching task for this dashboard.

pydash_app.dashboard.services.seeding module

Fills the application with some preliminary dashboards to make it easier to test code in development and staging environments.

pydash_app.dashboard.services.seeding.**seed**()

For each user, stores some preliminary debug dashboards in the datastore, to be used during development.

Module contents

Contains services for the 'Dashboard' concern.

These are things that use or manipulate 'Dashboard' entities to perform tasks, where these tasks are either too complex to put in the Dashboard Entity, or where these are heavily interacting with outside logic that the business domain entity should not concern itself with directly.

pydash_app.dashboard.services.**is_valid_dashboard**(*url*)

Submodules

pydash_app.dashboard.endpoint module

class pydash_app.dashboard.endpoint.**Endpoint**(*name*, *is_monitored*)

Bases: persistent.Persistent

The Endpoint entity knows about: - Its own properties - The functionalities for Endpoint interactions with information from elsewhere.

It does not contain information on how to persistently store/load an endpoint, as currently endpoints only exist in combination with dashboard objects. If endpoints were to exist on their own, the *endpoint_repository* would handle their persistence.

add_endpoint_call(*call*)
Adds an EndpointCall to its internal collection of endpoint calls. :param call: The endpoint call to add.

aggregated_data(*filters={}*)
Returns aggregated data on this endpoint. :param filters: A dictionary containing property_name-value pairs to filter on. The keys are assumed to be strings.

This is in the gist of *{'day':'2018-05-20', 'ip':'127.0.0.1'}* Defaults to an empty dictionary.

The currently allowed filter_names are:

- Time: * 'year' - e.g. '2018' * 'month' - e.g. '2018-05' * 'week' - e.g. '2018-W17' * 'day' - e.g. '2018-05-20' * 'hour' - e.g. '2018-05-20T20' * 'minute' - e.g. '2018-05-20T20-10'

Note that for Time filter-values, the formatting is crucial.

- Version: * 'version' - e.g. '1.0.1'
- IP: * 'ip' - e.g. '127.0.0.1'
- Group-by: * 'group_by' - e.g. 'None'

Returns A dict containing aggregated data points.

aggregated_data_daterange(*start_date*, *end_date*, *granularity*, *filters={}*)
Returns the aggregated data on this endpoint over the specified daterange. :param start_date: A datetime object that is treated as the inclusive lower bound of the daterange. :param end_date: A datetime object that is treated as the inclusive upper bound of the daterange. :param granularity: A string denoting the granularity of the daterange. :param filters: A dictionary containing property_name-value pairs to filter on. The keys are assumed to be strings.

This is in the gist of *{'day':'2018-05-20', 'ip':'127.0.0.1'}* Defaults to an empty dictionary.

The currently allowed filter_names are:

- Version: * 'version' - e.g. '1.0.1'
- IP: * 'ip' - e.g. '127.0.0.1'
- Group-by: * 'group_by' - e.g. 'None'

Note that, contrary to *aggregated_data* method, Time based filters are not allowed.

Returns A dictionary with all aggregated statistics and their values.

get_id()

remove_endpoint_call(*call*)
Removes an EndpointCall from this endpoint's internal collection of endpoint calls. Raises a ValueError if no such call exists. Note: does not remove it from its aggregated dataset yet. :param call: The endpoint call to remove.

set_monitored(*is_monitored*)

statistic(*statistic*, *filters={}*)

statistic_per_timeslice(*statistic*, *timeslice*, *start_datetime*, *end_datetime*, *filters={}*)

pydash_app.dashboard.endpoint_call module

class pydash_app.dashboard.endpoint_call.**EndpointCall**(*endpoint*, *execution_time*, *time*, *version*, *group_by*, *ip*)

Bases: persistent.Persistent

An EndpointCall entity only serves to store JSON data pulled from the external dashboards.

As with the other entity classes, it does not concern itself with the implementation of its persistence, as it doesn't exist on its own. If this were the case, the *endpointcall_repository* would handle this concern.

```
>>> endpoint_call = EndpointCall("foo", 0.5, datetime.strptime("2018-04-25␣
↪15:29:23", "%Y-%m-%d %H:%M:%S"), "0.1", "None", "127.0.0.1")
>>> endpoint_call.as_dict()
{'endpoint': 'foo', 'execution_time': 0.5, 'time': datetime.datetime(2018, 4, 25,␣
↪15, 29, 23), 'version': '0.1', 'group_by': 'None', 'ip': '127.0.0.1'}
```

as_dict()

returns a dict containing the data of the EndpointCall

pydash_app.dashboard.entity module

Involved usage example:

```
>>> from pydash_app.dashboard.entity import Dashboard
>>> from pydash_app.user.entity import User
>>> from pydash_app.dashboard.endpoint import Endpoint
>>> from pydash_app.dashboard.endpoint_call import EndpointCall
>>> import uuid
>>> from datetime import datetime, timedelta
>>> user = User("Gandalf", "pass", 'some@email.com')
>>> d = Dashboard("http://foo.io", str(uuid.uuid4()), str(user.id))
>>> e1 = Endpoint("foo", True)
>>> e2 = Endpoint("bar", True)
>>> d.add_endpoint(e1)
>>> d.add_endpoint(e2)
>>> ec1 = EndpointCall("foo", 0.5, datetime.strptime("2018-04-25 15:29:23", "%Y-%m-%d
↪%H:%M:%S"), "0.1", "None", "127.0.0.1")
>>> ec2 = EndpointCall("foo", 0.1, datetime.strptime("2018-04-25 15:29:23", "%Y-%m-%d
↪%H:%M:%S"), "0.1", "None", "127.0.0.2")
>>> ec3 = EndpointCall("bar", 0.2, datetime.strptime("2018-04-25 15:29:23", "%Y-%m-%d
↪%H:%M:%S"), "0.1", "None", "127.0.0.1")
>>> ec4 = EndpointCall("bar", 0.2, datetime.strptime("2018-04-25 15:29:23", "%Y-%m-%d
↪%H:%M:%S") - timedelta(days=1), "0.1", "None", "127.0.0.1")
>>> ec5 = EndpointCall("bar", 0.2, datetime.strptime("2018-04-25 15:29:23", "%Y-%m-%d
↪%H:%M:%S") - timedelta(days=2), "0.1", "None", "127.0.0.1")
>>> d.add_endpoint_call(ec1)
>>> d.add_endpoint_call(ec2)
>>> d.add_endpoint_call(ec3)
>>> d.add_endpoint_call(ec4)
>>> d.add_endpoint_call(ec5)
>>> d.aggregated_data()
{'total_visits': 5, 'total_execution_time': 1.2, 'average_execution_time': 0.24,
↪'visits_per_ip': {'127.0.0.1': 4, '127.0.0.2': 1}, 'unique_visitors': 2, 'fastest_
↪measured_execution_time': 0.1, 'fastest_quartile_execution_time': 0.14, 'median_
↪execution_time': 0.2, 'slowest_quartile_execution_time': 0.39, 'ninetieth_
↪percentile_execution_time': 0.5, 'ninety-ninth_percentile_execution_time': 0.5,
↪'slowest_measured_execution_time': 0.5, 'versions': ['0.1']}
```

(continues on next page)

(continued from previous page)

```
>>> d.endpoints['foo'].aggregated_data()
{'total_visits': 2, 'total_execution_time': 0.6, 'average_execution_time': 0.3,
↪'visits_per_ip': {'127.0.0.1': 1, '127.0.0.2': 1}, 'unique_visitors': 2, 'fastest_
↪measured_execution_time': 0.1, 'fastest_quartile_execution_time': 0.1, 'median_
↪execution_time': 0.3, 'slowest_quartile_execution_time': 0.5, 'ninetieth_percentile_
↪execution_time': 0.5, 'ninety-ninth_percentile_execution_time': 0.5, 'slowest_
↪measured_execution_time': 0.5, 'versions': ['0.1']}
>>> d.endpoints['bar'].aggregated_data()
{'total_visits': 3, 'total_execution_time': 0.6, 'average_execution_time': 0.2,
↪'visits_per_ip': {'127.0.0.1': 3}, 'unique_visitors': 1, 'fastest_measured_
↪execution_time': 0.2, 'fastest_quartile_execution_time': 0.2, 'median_execution_time
↪': 0.2, 'slowest_quartile_execution_time': 0.2, 'ninetieth_percentile_execution_time
↪': 0.2, 'ninety-ninth_percentile_execution_time': 0.2, 'slowest_measured_execution_
↪time': 0.2, 'versions': ['0.1']}
```

class pydash_app.dashboard.entity.**Dashboard**(*url*, *token*, *user_id*, *name=None*)

Bases: persistent.Persistent

The Dashboard entity knows about: - Its own properties (id, url, user_id, endpoints, endpoint_calls and last_fetch_time) - The functionalities for Dashboard interactions with information from elsewhere.

It does not contain information on how to persistently store/load a dashboard. This task is handled by the *dashboard_repository*.

add_endpoint(*endpoint*)

Adds an endpoint to this dashboard's internal collection of endpoints. :param endpoint: The endpoint to add, expects an Endpoint object.

add_endpoint_call(*endpoint_call*)

Adds an endpoint call to the dashboard. Will register the corresponding endpoint to the dashboard if this has not been done yet.

Parameters **endpoint_call** – The endpoint call to add

aggregated_data(*filters={}*)

Returns aggregated data on this dashboard. :param filters: A dictionary containing property_name-value pairs to filter on. The keys are assumed to be strings.

This is in the gist of *{'day':'2018-05-20', 'ip':'127.0.0.1'}* Defaults to an empty dictionary.

The currently allowed filter_names are:

- Time: * 'year' - e.g. '2018' * 'month' - e.g. '2018-05' * 'week' - e.g. '2018-W17' * 'day' - e.g. '2018-05-20' * 'hour' - e.g. '2018-05-20T20' * 'minute' - e.g. '2018-05-20T20-10'

Note that for Time filter-values, the formatting is crucial.

- Version: * 'version' - e.g. '1.0.1'
- IP: * 'ip' - e.g. '127.0.0.1'
- Group-by: * 'group_by' - e.g. 'None'

Returns A dict containing aggregated data points.

aggregated_data_daterange(*start_date*, *end_date*, *granularity*, *filters={}*)

Returns the aggregated data on this dashboard over the specified daterange. :param start_date: A datetime object that is treated as the inclusive lower bound of the daterange. :param end_date: A datetime object that is treated as the inclusive upper bound of the daterange. :param granularity: A string denoting the

granularity of the daterange. :param filters: A dictionary containing property_name-value pairs to filter on. The keys are assumed to be strings.

This is in the gist of *{'day':'2018-05-20', 'ip':'127.0.0.1'}* Defaults to an empty dictionary.

The currently allowed filter_names are:

- Version: * 'version' - e.g. '1.0.1'
- IP: * 'ip' - e.g. '127.0.0.1'
- Group-by: * 'group_by' - e.g. 'None'

Note that, contrary to *aggregated_data* method, Time based filters are not allowed.

Returns A dictionary with all aggregated statistics and their values.

first_endpoint_call_time()

get_id()

remove_endpoint(*endpoint*)

Removes an endpoint from this dashboard's internal collection of endpoints.

Raises a ValueError if no such endpoint exists. :param endpoint: The endpoint to remove.

statistic(*statistic*, *filters={}*)

statistic_per_timeslice(*statistic*, *timeslice*, *start_datetime*, *end_datetime*, *filters={}*)

class pydash_app.dashboard.entity.**DashboardState**

Bases: enum.Enum

The DashboardState enum indicates the state in which a Dashboard can remain, regarding remote fetching:

- not_initialized indicates the dashboard is newly created and not initialized with Endpoints and historic EndpointCalls;
- initialized_endpoints indicates the dashboard has successfully initialized Endpoints, but not yet historical EndpointCalls;
- initialize_endpoints_failure indicates something went wrong while initializing Endpoints, which means initialization of Endpoints needs to be retried;
- initialized_endpoint_calls indicates the dashboard has successfully initialized historical EndpointCalls, and can start fetching new EndpointCalls in a periodic task;
- initialize_endpoint_calls_failure indicates something went wrong while initializing historical EndpointCalls, which means this needs to be retried;
- fetched_endpoint_calls indicates last time new EndpointCalls were fetched, it was done successfully;
- fetch_endpoint_calls_failure indicates something went wrong while fetching new EndpointCalls, which means this needs to be retried.

fetch_endpoint_calls_failure = 31

fetched_endpoint_calls = 30

initialize_endpoint_calls_failure = 21

initialize_endpoints_failure = 11

initialized_endpoint_calls = 20

initialized_endpoints = 10

```
not_initialized = 0
```

pydash_app.dashboard.repository module

This module handles the persistence of *Dashboard* entities:

It is an adapter of the actual persistence layer, to insulate the application from datastore-specific details.

It handles a subset of the following tasks (specifically, it only actually contains functions for the tasks the application needs in its current state!):

- Creating new entities of the specified type and finding them based on id.

```
>>> import pydash_app.dashboard.entity as dashboard
>>> import uuid
>>> dashboard = dashboard.Dashboard("", "", str(uuid.uuid4()))
>>> add(dashboard)
>>> found_dashboard = find(dashboard.get_id())
>>> found_dashboard.get_id() == dashboard.get_id()
True
```

- Asking for all dashboards is also possible!

```
>>> all()
<OOBTreeItems object at 0x...>
```

- Adding multiple instances of the same dashboard will return a KeyError or a DuplicateIndexError

TODO fix it so that it actually errors?? >>> import pydash_app.dashboard.entity as dashboard >>> import uuid >>> dashboard = dashboard.Dashboard(“”, “”, str(uuid.uuid4())) >>> add(dashboard) >>> add(dashboard)

- Persisting updated versions of existing entities.

```
>>> import pydash_app.dashboard.entity as dashboard
>>> import uuid
>>> dashboard = dashboard.Dashboard("", "", str(uuid.uuid4()))
>>> add(dashboard)
>>> dashboard.token = "newToken"
>>> update(dashboard)
>>> found_dashboard = find(dashboard.get_id())
>>> found_dashboard.token == dashboard.token
True
```

- Deleting entities from the persistence layer, note that find() will return a KeyError if no dashboard was found.

```
>>> delete(dashboard)
>>> found_dashboard = find(dashboard.get_id())
Traceback (most recent call last):
    ...
KeyError
```

- Deleting non-existent dashboards will result in a KeyError.

```
>>> import pydash_app.dashboard.entity as dashboard
>>> import uuid
>>> dashboard = dashboard.Dashboard("", "", str(uuid.uuid4()))
>>> add(dashboard)
>>> delete(dashboard)
```

(continues on next page)

(continued from previous page)

```
>>> delete(dashboard)
Traceback (most recent call last):
    ...
KeyError
```

pydash_app.dashboard.repository.**add**(*dashboard*)

pydash_app.dashboard.repository.**all**()

pydash_app.dashboard.repository.**clear_all**()

pydash_app.dashboard.repository.**delete**(*dashboard*)

pydash_app.dashboard.repository.**find**(*dashboard_id*)

pydash_app.dashboard.repository.**update**(*dashboard*)

Module contents

This module is the public interface (available to the web-application pydash_web) for interacting with Dashboards.

pydash_app.dashboard.**add_to_repository**(*dashboard*)

pydash_app.dashboard.**dashboards_of_user**(*user_id*)
Returns a list of Dashboard-entities that are connected to the given user. :param user_id: The UUID of the user whose dashboards we're requesting. :return: A list of Dashboard-entities.

pydash_app.dashboard.**find**(*dashboard_id*)
Returns a single Dashboard-entity with the given UUID or None if it could not be found. :param dashboard_id: UUID of the dashboard we hope to find. :return: The Dashboard-entity with the given UUID or raises an Exception if it could not be found.

pydash_app.dashboard.**find_verified_dashboard**(*dashboard_id*)
Verifies if a given dashboard_id is correct and if the current user has access to the dashboard. :param dashboard_id: The UUID of the dashboard to be validated. :return: True if the dashboard is valid, else False followed by the result and the http error code.

pydash_app.dashboard.**remove_from_repository**(*dashboard*)

pydash_app.user package

Subpackages

pydash_app.user.services package

Submodules

pydash_app.user.services.pruning module

Provides functionality to periodically remove all users that have not verified their account.

pydash_app.user.services.pruning.**schedule_periodic_pruning_task**(*interval=datetime.timedelta(1)*, *scheduler=<periodic_tasks.task_scheduler.T object>*)

pydash_app.user.services.seeding module

Fills the application with some preliminary users to make it easier to test code in development and staging environments.

pydash_app.user.services.seeding.**seed**()

Stores some preliminary debug users in the datastore, to be used during development.

```
>>> seed()
Adding user <User id=... name=Alberto>
Adding user <User id=... name=Arjan>
Adding user <User id=... name=JeroenO>
Adding user <User id=... name=JeroenL>
Adding user <User id=... name=Koen>
Adding user <User id=... name=Lars>
Adding user <User id=... name=Patrick>
Adding user <User id=... name=Tom>
Adding user <User id=... name=W-M>
Seeding of users is done!
>>> found_user = repository.find_by_name("Alberto")
>>> found_user.name == "Alberto"
True
```

Module contents

Contains services for the 'User' concern.

These are things that use or manipulate 'User' entities to perform tasks, where these tasks are either too complex to put in the User Entity, or where these are heavily interacting with outside logic that the business domain entity should not concern itself with directly.

Submodules

pydash_app.user.entity module

class pydash_app.user.entity.**User**(*name*, *password*, *mail*)

Bases: persistent.Persistent, flask_login.mixins.UserMixin

The User entity knows about:

- What properties a User has
- What functionality makes sense to have this User interact with information from elsewhere.

Per Domain Driven Design, it does _not_ contain information on how to persistently store/load a user! (That is instead handled by the *user_repository*).

The User entity checks its parameters on creation:

```
>>> User(42, 32, 11)
Traceback (most recent call last):
  ...
TypeError
```

check_password(*password*)

generate_new_verification_code()

get_id()

get_verification_code()
Returns this User's verification code or None if it has expired or this User has already been verified

get_verification_code_expiration_date()
Returns a datetime object of when this User's verification code is about to expire, or None if it has already expired or this User has already been verified

has_verification_code_expired()
Returns a boolean whether this User's verification code has expired, if it has one.

is_verified()

set_password(*password*)

pydash_app.user.repository module

This module handles the persistence of *User* entities:

It is an adapter of the actual persistence layer, to insulate the application from datastore-specific details.

It handles a subset of the following tasks (specifically, it only actually contains functions for the tasks the application needs in its current state!): - Creating new entities of the specified type - Finding them based on certain attributes - Persisting updated versions of existing entities. - Deleting entities from the persistence layer.

pydash_app.user.repository.**add**(*user*)
Adds the User-entity to the repository. Will raise a (KeyError, DuplicateIndexError) tuple on failure. :param user: The User-entity to add.

```
>>> list(all())
[]
>>> gandalf = User("Gandalf", "pass", 'some@email.com')
>>> dumbledore = User("Dumbledore", "secret", 'some@email.com')
>>> add(gandalf)
>>> add(dumbledore)
>>> sorted([user.name for user in all()])
['Dumbledore', 'Gandalf']
```

pydash_app.user.repository.**all**()
Returns a (lazy) collection of all users (in no guaranteed order).

```
>>> list(all())
[]
>>> gandalf = User("Gandalf", "pass", 'some@email.com')
>>> dumbledore = User("Dumbledore", "secret", 'some@email.com')
>>> add(gandalf)
>>> add(dumbledore)
>>> sorted([user.name for user in all()])
['Dumbledore', 'Gandalf']
>>> clear_all()
>>> sorted([user.name for user in all()])
[]
```

pydash_app.user.repository.**all_unverified**()
Returns a collection of all unverified users (in no guaranteed order).

pydash_app.user.repository.**clear_all**()
Flushes the database.

```
>>> gandalf = User("Gandalf", "pass", 'some@email.com')
>>> dumbledore = User("Dumbledore", "secret", 'some@email.com')
>>> add(gandalf)
>>> add(dumbledore)
>>> sorted([user.name for user in all()])
['Dumbledore', 'Gandalf']
>>> clear_all()
>>> list(all())
[]
```

pydash_app.user.repository.**delete_by_id**(*user_id*)

Removes the User-entity whose user_id is *user_id* from the repository. Will raise a KeyError if said user is not in the repository. Note that this might also occur when delete_by_id(user_id) is called in the middle of the deletion,

in a multiprocessing environment.

Parameters **user_id** – The ID of the User-entity to be removed. This can be either a UUID-entity or the corresponding string representation.

```
>>> gandalf = User("Gandalf", "pass", 'some@email.com')
>>> add(gandalf)
>>> find_by_name("Gandalf") == gandalf
True
>>> delete_by_id(gandalf.get_id())
>>> find_by_name("Gandalf") == gandalf
False
```

pydash_app.user.repository.**find**(*user_id*)

Finds a user in the database. :param user_id: UUID for the user to be retrieved. :return: User object or None if no user could be found.

pydash_app.user.repository.**find_by_name**(*name*)

Returns a single User-entity with the given *name*, or None if it could not be found.

name – Name of the user we hope to find.

pydash_app.user.repository.**find_by_verification_code**(*verification_code*)

Returns a single User-entity with the given *verification_code*, or None if it could not be found. The latter case might indicate that the user does not exist, or that the verification code has expired. :param verification_code: The verification code of the user we hope to find. Should be a pydash_app.user.verification_code.VerificationCode object.

pydash_app.user.repository.**update**(*user*)

Changes the user's information

```
>>> gandalf = User("GandalfTheGrey", "pass", 'some@email.com')
>>> add(gandalf)
>>> gandalf.name = "GandalfTheWhite"
>>> update(gandalf)
>>> find_by_name("GandalfTheGrey") == gandalf
False
>>> find_by_name("GandalfTheWhite") == gandalf
True
```

pydash_app.user.verification module

exception pydash_app.user.verification.**InvalidVerificationCodeError**
Bases: Exception

exception pydash_app.user.verification.**VerificationCodeExpiredError**
Bases: Exception

pydash_app.user.verification.**verify**(*verification_code*)
Attempts to verify a user with the provided verification code. This is intended as a one-time action per user after registration. :param verification_code: The verification code that should match the User-entity's verification code.

Can be a string or UUID object.

Returns Returns True if both verification codes are equal, returns False otherwise. Raises an InvalidVerificationCodeError when the provided verification code is invalid. Raises an VerificationCodeExpiredError when the provided verification code has expired.

pydash_app.user.verification_code module

class pydash_app.user.verification_code.**VerificationCode**(*expiration_time=datetime.timedelta(1)*)
Bases: object

A 'smart' randomly generated verification code that keeps track of whether it has expired. Default expiration time is 7 days.

is_expired()

Module contents

This module is the public interface (available to the web-application pydash_web) for interacting with Users.

Example Usage:

```
>>> gandalf = User("Gandalf", "pass", 'some@email.com')
>>> add_to_repository(gandalf)
...
>>> found_user = find(gandalf.id)
>>> found_user.name == "Gandalf"
True
```

You can also use a string-version of the ID to find the user again:

```
>>> found_user = find(str(gandalf.id))
>>> found_user.name == "Gandalf"
True
```

```
>>> found_user2 = find_by_name("Gandalf")
>>> found_user2 == found_user
True
>>> find_by_name("Dumbledore")
>>> # ^Returns nothing
>>> res_user = authenticate("Gandalf", "pass")
>>> res_user.name == "Gandalf"
```

(continues on next page)

(continued from previous page)

```
True
>>> authenticate("Gandalf", "youshallnot")
>>> # ^Returns nothing
>>> authenticate("Dumbledore", "secrets")
>>> # ^Returns nothing
```

pydash_app.user.**add_to_repository**(*user*)

Adds the given User-entity to the user_repository. Raises a KeyError if the user is already in the repository. :param user: The User-entity in question.

Adding the same user twice with the same name is not allowed:

```
>>> gandalf1 = User("Gandalf", "pass", 'some@email.com')
>>> add_to_repository(gandalf1)
>>> gandalf2 = User("Gandalf", "balrog", 'some@email.com')
>>> add_to_repository(gandalf2)
Traceback (most recent call last):
  ...
multi_indexed_collection.DuplicateIndexError
```

pydash_app.user.**authenticate**(*name*, *password*)

Attempts to authenticate the user with name *name* and password *password*.

If authentication fails (unknown user or incorrect password), returns None. Otherwise, returns the user object.

pydash_app.user.**check_password_requirements**(*password*)

pydash_app.user.**find**(*user_id*)

Returns a single User-entity with the given UUID or None if it could not be found.

user_id- UUID of the user we hope to find.

pydash_app.user.**find_by_name**(*name*)

Returns a single User-entity with the given *name*, or None if it could not be found.

name – Name of the user we hope to find.

pydash_app.user.**find_by_verification_code**(*verification_code*)

Returns a single User-entity with the given *verification_code*, or None if it could not be found. :param verification_code: The verification code of the user we hope to find.

pydash_app.user.**maybe_find_user**(*user_id*)

Returns the User entity, or *None* if it does not exist.

```
>>> user = User("Gandalf", "pass", 'some@email.com')
>>> add_to_repository(user)
...
>>> found_user = maybe_find_user(user.id)
>>> found_user.name == "Gandalf"
True
>>> import uuid
>>> unexistent_uuid = uuid.UUID('ced84534-7a55-440f-ad77-9912466fe022')
>>> unexistent_user = maybe_find_user(unexistent_uuid)
>>> unexistent_user == None
True
```

pydash_app.user.**remove_from_repository**(*user_id*)

Removes the User-entity whose user_id is *user_id* from the repository.

```
>>> gandalf1 = User("Gandalf", "pass", 'some@email.com')
>>> add_to_repository(gandalf1)
>>> remove_from_repository(gandalf1.get_id())
>>> found_user = find_by_name("Gandalf")
>>> found_user == None
True
```

Will raise a KeyError if said user is not in the repository.

```
>>> gandalf1 = User("Gandalf", "pass", 'some@email.com')
>>> add_to_repository(gandalf1)
>>> remove_from_repository(gandalf1.get_id())
>>> remove_from_repository(gandalf1.get_id())
Traceback (most recent call last):
  ...
KeyError
```

Parameters **user_id** – The ID of the User-entity to be removed. This can be either a UUID-entity or the corresponding string representation.

pydash_app.user.**verify**(*verification_code*)

Attempts to verify a user with the provided verification code. This is intended as a one-time action per user after registration. :param verification_code: The verification code that should match the User-entity's verification code.

Can be a string or UUID object.

Returns Returns True if both verification codes are equal, returns False otherwise. Raises an InvalidVerificationCodeError when the provided verification code is invalid. Raises an VerificationCodeExpiredError when the provided verification code has expired.

1.6.2 Module contents

The *pydash_app* package contains all business domain logic of the PyDash application: Everything that is not part of rendering a set of webpages.

pydash_app.**schedule_periodic_tasks**()

pydash_app.**seed_datastructures**()

pydash_app.**start_task_scheduler**()

pydash_app.**stop_task_scheduler**()

1.7 pydash_database package

1.7.1 Module contents

class pydash_database.**MultiIndexedPersistentCollection**(*properties*)

Bases: multi_indexed_collection.MultiIndexedCollection, persistent.Persistent

pydash_database.**database_connection**()

pydash_database.**database_root**()
Returns the ZEO database root object. Wraps a database connection; a new connection is initialized once on each multiprocessing.Process. (on all subsequent calls on this process, the connection is re-used.)

1.8 pydash_logger package

1.8.1 Submodules

1.8.2 pydash_logger.logger module

Logger object will log messages and errors to date-stamped '.log' files in the /logs directory of the project. Simply import the class and use it to log messages.

class pydash_logger.logger.**Logger**(*name='pydash_logger.logger'*)
Bases: object

debug(*msg*)
Takes a message and logs it at the logging.DEBUG level :param: msg: the message to be logged

error(*msg*)
Takes a message and logs it at the logging.ERROR level :param: msg: the message to be logged

info(*msg*)
Takes a message and logs it at the logging.INFO level :param: msg: the message to be logged

warning(*msg*)
Takes a message and logs it at the logging.WARN level :param: msg: the message to be logged

1.8.3 Module contents

1.9 pydash_mail package

1.9.1 Submodules

1.9.2 pydash_mail.templates module

Reads mail templates into memory and provides functions to format them.

pydash_mail.templates.**format_verification_mail_html**(*username*, *verification_url*, *expiration_date*)
Format an HTML verification mail. :param username: Username to use in the mail. :param verification_url: Verification link to use in the mail. :param expiration_date: Expiration date of the verification code. :return: The formatted HTML verification mail.

pydash_mail.templates.**format_verification_mail_plain**(*username*, *verification_url*, *expiration_date*)
Format a plaintext verification mail. :param username: Username to use in the mail. :param verification_url: Verification link to use in the mail. :param expiration_date: Expiration date of the verification code. :return: The formatted plaintext verification mail.

1.9.3 Module contents

1.10 pydash_web package

1.10.1 Subpackages

pydash_web.controller package

Submodules

pydash_web.controller.change_dashboard_settings module

Handles changing dashboard settings.

pydash_web.controller.change_dashboard_settings.**change_dashboard_settings**(*dashboard_id*)

pydash_web.controller.change_password module

Manages changing of the user's password.

pydash_web.controller.change_password.**change_password**()

pydash_web.controller.change_settings module

Manages changing of user settings.

pydash_web.controller.change_settings.**change_settings**()

pydash_web.controller.dashboards module

Manages the lookup and returning of dashboard information for a certain user.

Currently only returns static mock data.

pydash_web.controller.dashboards.**check_allowed_statistics**(*statistic*)

pydash_web.controller.dashboards.**check_allowed_timeslices**(*timeslice*)

pydash_web.controller.dashboards.**dashboard**(*dashboard_id*)

Lists information of a single dashboard. :param dashboard_id: ID of the dashboard to retrieve information from. :return: The returned value consists of a tuple of dashboard information, together with a http status code. This route supports the following request arguments: - statistic: The name of the statistic of which aggregated information should be returned.

The currently supported statistics are:

- total_visits
- total_execution_time
- average_execution_time
- visits_per_ip
- unique_visitors

- fastest_measured_execution_time
- fastest_quartile_execution_time
- median_execution_time
- slowest_quartile_execution_time
- ninetieth_percentile_execution_time
- ninety-ninth_percentile_execution_time
- slowest_measured_execution_time

- **start_date, end_date: The start- and end dates of the datetime range in which the desired information lies.** Both start_date and end_date are inclusive resp. upper- and lower bounds of this datetime range. If start_date is not provided, it defaults to 1970-1-1. If end_date is not provided, it defaults to the current utc time.

  It is assumed both start_date and end_date are provided in utc time.

- **granularity: Since end_date is inclusive, a time granularity is required in order to determine how much time from** end_date on should be included as well. The possibilities here are: 'year', 'month', 'week', 'day', 'hour' and 'minute'. If granularity is not privided, it defaults to 'day'.
- **timeslice: Indicates the data should be returned as a series of points in time, each 'timeslice' long.** 'timeslice' overrules 'granularity' in terms of granularity.

If 'timeslice' is absent, a the returned information is a single value. When it is not, a dictionary is returned, containing datetime-value pairs, where 'datetime' is formatted to the granularity of 'timeslice'. (e.g. 'timeslice=day' will result in datetimes like '2018-05-29', while 'timeslice=minute' will result in datetimes like '2018-05-29T15:45')

Note that if the dashboard has not yet received any endpoint calls, it will simply return an empty dictionary.

pydash_web.controller.dashboards.**dashboards**()
Lists the dashboards of the current user. :return: A tuple containing:

- A list of dicts, containing dashboard details of the current user's dashboards. or A dict containing an error message describing the particular error.
- A corresponding HTML status code.

pydash_web.controller.dashboards.**handle_statistic_per_timeslice**(*dashboard*, *statistic*, *timeslice*, *start_datetime*, *end_datetime*)
These datetimes are treated as inclusive boundaries of a datetime range (e.g. [start_datetime, end_datetime]. Assumes start_timedate and end_timedate are both timezone aware, with timezone utc. :param dashboard: :param statistic: :param timeslice: :param start_datetime: :param end_datetime: :return: A dictionary consisting of a datetime string (key)(formatted according to the ISO-8601 standard)

and the corresponding statistic, over the specified datetime range.

pydash_web.controller.dashboards.**handle_statistic_without_timeslice**(*dashboard*, *statistic*, *start_datetime*, *end_datetime*, *granularity*)
These datetimes are treated as inclusive boundaries of a datetime range (e.g. [start_datetime, end_datetime]

:param dashboard: :param statistic: :param start_datetime: :param end_datetime: :param granularity: :return: The value of a single statistic over the specified datetime range.

pydash_web.controller.dashboards.**match_datetime_string_with_formats**(*datetime_string*)
Returns a datetime object of this datetime string if the provided string matched with one of the allowed formats. Otherwise, returns None and None.

pydash_web.controller.delete_dashboard module

Manages the deletion of a dashboard.

pydash_web.controller.delete_dashboard.**delete_dashboard**(*dashboard_id*)

pydash_web.controller.delete_user module

Manages deletion of a user.

pydash_web.controller.delete_user.**delete_user**()
Deletes the currently logged in user and all dashboards they own.

pydash_web.controller.execution_times_boxplots module

pydash_web.controller.execution_times_boxplots.**endpoint_execution_times_boxplots**(*dashboard_id*, *endpoint_name=*

pydash_web.controller.execution_times_per_version module

Handles requests for tdigest data of response times per version.

pydash_web.controller.execution_times_per_version.**execution_times_per_version**(*dashboard_id*, *endpoint_name=None*

pydash_web.controller.login module

Manages the logging in of a user into the application, and rejecting visitors that enter improper sign-in information or have not been verified yet.

pydash_web.controller.login.**login**()

pydash_web.controller.logout module

Allows a user to sign out again after finishing using the application

pydash_web.controller.logout.**logout**()

pydash_web.controller.register_dashboard module

pydash_web.controller.register_dashboard.**register_dashboard**()

pydash_web.controller.register_user module

Manages the registration of a new user.

pydash_web.controller.register_user.**register_user**()

pydash_web.controller.user_verification module

Manages the verification of a User.

pydash_web.controller.user_verification.**verify_user**()
Verifies the currently logged in User by comparing the given verification_code with the code assigned to the User. This is intended to be used only once, after the user has just registered their account in order to gain access to api-routes that have the *verification_required* decorator.

pydash_web.controller.utils module

The go-to place for general methods that can be used in multiple controller methods.

pydash_web.controller.utils.**execution_times**(*aggregator_group_container*, *filters={}*)

pydash_web.controller.visitor_heatmap module

pydash_web.controller.visitor_heatmap.**daterange**(*start_date*, *end_date*)

pydash_web.controller.visitor_heatmap.**get_hourly_data**(*dashboard*, *day*, *field*)

pydash_web.controller.visitor_heatmap.**visitor_heatmap**(*dashboard_id*, *field='total_visits'*)

Module contents

The controller contains one dispatching function per flask_webapp endpoint action.

1.10.2 Submodules

1.10.3 pydash_web.api module

Serves as a blueprint for the entire pydash_web package. url_for() calls within this package should prepend 'pydash_web.' to their input argument.

[e.g. url_for(login) becomes url_for(pydash_web.login)]

route decorators in this package should also use this blueprint object instead of the flask application object.

1.10.4 pydash_web.api_routes module

Contains the different routes (web endpoints) that the pydash_web flask application can respond to.

The actual implementation of each of the routes' dispatching logic is handled by the respective 'controller' function.

pydash_web.api_routes.**change_dashboard_settings**(*dashboard_id*)

pydash_web.api_routes.**change_password**()

pydash_web.api_routes.**change_settings**()

pydash_web.api_routes.**delete_dashboard**(*dashboard_id*)

pydash_web.api_routes.**delete_user**()

pydash_web.api_routes.**get_dashboard**(*dashboard_id*)

pydash_web.api_routes.**get_dashboards**()

pydash_web.api_routes.**get_endpoint_execution_times_boxplots**(*dashboard_id*)

pydash_web.api_routes.**get_execution_times_boxplot**(*dashboard_id*, *endpoint_name*)

pydash_web.api_routes.**get_execution_times_per_version_dashboard**(*dashboard_id*)

pydash_web.api_routes.**get_execution_times_per_version_endpoint**(*dashboard_id*, *endpoint_name*)

pydash_web.api_routes.**get_unique_visitor_heatmap**(*dashboard_id*)

pydash_web.api_routes.**get_visitor_heatmap**(*dashboard_id*)

pydash_web.api_routes.**login**()

pydash_web.api_routes.**logout**()

pydash_web.api_routes.**register_dashboard**()

pydash_web.api_routes.**register_user**()

pydash_web.api_routes.**verify_user**()

1.10.5 pydash_web.react_server module

pydash_web.react_server.**serve**(*path*)

1.10.6 Module contents

Entrypoint of *pydash_web*

Initializes a Flask web application, and loads the relevant configuration settings.

pydash_web.**load_user**(*user_id*)

pydash_web.**unauthorized**()

1.11 tests package

1.11.1 Submodules

1.11.2 tests.common_fixtures module

1.11.3 tests.conftest module

tests.conftest.**splinter_driver_kwargs**()
Webdriver kwargs.

tests.conftest.**testserver**(*request*)
Defines the testserver funcarg

1.11.4 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

c

config, 1
conftest, 1

f

flask_monitoring_dashboard_client, 1

p

periodic_tasks, 4
periodic_tasks.pqdict_iter_upto_priority, 2
periodic_tasks.queue_nonblocking_iter, 2
periodic_tasks.task_scheduler, 2
pydash, 6
pydash_app, 26
pydash_app.dashboard, 20
pydash_app.dashboard.aggregator, 13
pydash_app.dashboard.aggregator.aggregator_group, 6
pydash_app.dashboard.aggregator.statistics, 9
pydash_app.dashboard.endpoint, 14
pydash_app.dashboard.endpoint_call, 16
pydash_app.dashboard.entity, 16
pydash_app.dashboard.repository, 19
pydash_app.dashboard.services, 14
pydash_app.dashboard.services.fetching, 13
pydash_app.dashboard.services.seeding, 14
pydash_app.user, 24
pydash_app.user.entity, 21
pydash_app.user.repository, 22
pydash_app.user.services, 21
pydash_app.user.services.pruning, 20
pydash_app.user.services.seeding, 21
pydash_app.user.verification, 24
pydash_app.user.verification_code, 24
pydash_database, 26
pydash_logger, 27
pydash_logger.logger, 27
pydash_mail, 28
pydash_mail.templates, 27
pydash_web, 32
pydash_web.api, 31
pydash_web.api_routes, 31
pydash_web.controller, 31
pydash_web.controller.change_dashboard_settings, 28
pydash_web.controller.change_password, 28
pydash_web.controller.change_settings, 28
pydash_web.controller.dashboards, 28
pydash_web.controller.delete_dashboard, 30
pydash_web.controller.delete_user, 30
pydash_web.controller.execution_times_boxplots, 30
pydash_web.controller.execution_times_per_version, 30
pydash_web.controller.login, 30
pydash_web.controller.logout, 30
pydash_web.controller.register_dashboard, 30
pydash_web.controller.register_user, 31
pydash_web.controller.user_verification, 31
pydash_web.controller.utils, 31
pydash_web.controller.visitor_heatmap, 31
pydash_web.react_server, 32

t

tests, 33
tests.common_fixtures, 32
tests.conftest, 32

INDEX

A

add() (in module pydash_app.dashboard.repository), 20
add() (in module pydash_app.user.repository), 22
add_background_task() (in module periodic_tasks), 5
add_background_task() (periodic_tasks.task_scheduler.TaskScheduler method), 3
add_endpoint() (pydash_app.dashboard.entity.Dashboard method), 17
add_endpoint_call() (pydash_app.dashboard.aggregator.Aggregator method), 13
add_endpoint_call() (pydash_app.dashboard.aggregator.aggregator_group.AggregatorGroup method), 6
add_endpoint_call() (pydash_app.dashboard.endpoint.Endpoint method), 14
add_endpoint_call() (pydash_app.dashboard.entity.Dashboard method), 17
add_periodic_task() (in module periodic_tasks), 5
add_periodic_task() (periodic_tasks.task_scheduler.TaskScheduler method), 3
add_to_collection() (pydash_app.dashboard.aggregator.statistics.Statistic class method), 11
add_to_repository() (in module pydash_app.dashboard), 20
add_to_repository() (in module pydash_app.user), 25
add_together() (pydash_app.dashboard.aggregator.statistics.AverageExecutionTime method), 9
add_together() (pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC method), 9
add_together() (pydash_app.dashboard.aggregator.statistics.ExecutionTimeTDigest method), 9
add_together() (pydash_app.dashboard.aggregator.statistics.Statistic method), 11
add_together() (pydash_app.dashboard.aggregator.statistics.TotalExecutionTime method), 11
add_together() (pydash_app.dashboard.aggregator.statistics.TotalVisits method), 11
add_together() (pydash_app.dashboard.aggregator.statistics.UniqueVisitors method), 12
add_together() (pydash_app.dashboard.aggregator.statistics.Versions method), 12
add_together() (pydash_app.dashboard.aggregator.statistics.VisitsPerIP method), 12
aggregated_data() (pydash_app.dashboard.endpoint.Endpoint method), 15
aggregated_data() (pydash_app.dashboard.entity.Dashboard method), 17
aggregated_data_daterange() (pydash_app.dashboard.endpoint.Endpoint method), 15
aggregated_data_daterange() (pydash_app.dashboard.entity.Dashboard method), 17
Aggregator (class in pydash_app.dashboard.aggregator), 13
AggregatorGroup (class in pydash_app.dashboard.aggregator.aggregator_group), 6
AggregatorPartitionFun (class in pydash_app.dashboard.aggregator.aggregator_group), 8
all() (in module pydash_app.dashboard.repository), 20
all() (in module pydash_app.user.repository), 22
all_unverified() (in module pydash_app.user.repository), 22
append() (pydash_app.dashboard.aggregator.statistics.Statistic method), 11
as_dict() (pydash_app.dashboard.aggregator.Aggregator method), 13
as_dict() (pydash_app.dashboard.endpoint_call.EndpointCall method), 16
authenticate() (in module pydash_app.user), 25
AverageExecutionTime (class in pydash_app.dashboard.aggregator.statistics), 9

B

bar() (in module periodic_tasks), 5

baz() (in module periodic_tasks), 5

C

calc_endpoint_call_identifier() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
change_dashboard_settings() (in module pydash_web.api_routes), 31
change_dashboard_settings() (in module pydash_web.controller.change_dashboard_settings), 28
change_password() (in module pydash_web.api_routes), 31
change_password() (in module pydash_web.controller.change_password), 28
change_settings() (in module pydash_web.api_routes), 32
change_settings() (in module pydash_web.controller.change_settings), 28
check_allowed_statistics() (in module pydash_web.controller.dashboards), 28
check_allowed_timeslices() (in module pydash_web.controller.dashboards), 28
check_password() (pydash_app.user.entity.User method), 21
check_password_requirements() (in module pydash_app.user), 25
clean_in_memory_database() (in module conftest), 1
clear_all() (in module pydash_app.dashboard.repository), 20
clear_all() (in module pydash_app.user.repository), 22
Config (class in config), 1
config (module), 1
conftest (module), 1
contained_statistics_classes (pydash_app.dashboard.aggregator.Aggregator attribute), 13

D

Dashboard (class in pydash_app.dashboard.entity), 17
dashboard() (in module pydash_web.controller.dashboards), 28
dashboards() (in module pydash_web.controller.dashboards), 29
dashboards_of_user() (in module pydash_app.dashboard), 20
DashboardState (class in pydash_app.dashboard.entity), 18
database_connection() (in module pydash_database), 26
database_root() (in module pydash_database), 26
date_dict() (in module pydash_app.dashboard.aggregator.statistics), 12
daterange() (in module pydash_web.controller.visitor_heatmap), 31
debug() (pydash_logger.logger.Logger method), 27
delete() (in module pydash_app.dashboard.repository), 20
delete_by_id() (in module pydash_app.user.repository), 23
delete_dashboard() (in module pydash_web.api_routes), 32
delete_dashboard() (in module pydash_web.controller.delete_dashboard), 30
delete_user() (in module pydash_web.api_routes), 32
delete_user() (in module pydash_web.controller.delete_user), 30
dependencies (pydash_app.dashboard.aggregator.statistics.AverageExecutio attribute), 9
dependencies (pydash_app.dashboard.aggregator.statistics.ExecutionTimeP attribute), 9
dependencies (pydash_app.dashboard.aggregator.statistics.Statistic attribute), 11

E

empty() (pydash_app.dashboard.aggregator.statistics.AverageExecutionTim method), 9
empty() (pydash_app.dashboard.aggregator.statistics.ExecutionTimePercent method), 9
empty() (pydash_app.dashboard.aggregator.statistics.ExecutionTimeTDiges method), 9
empty() (pydash_app.dashboard.aggregator.statistics.Statistic method), 11
empty() (pydash_app.dashboard.aggregator.statistics.TotalExecutionTime method), 11
empty() (pydash_app.dashboard.aggregator.statistics.TotalVisits method), 11
empty() (pydash_app.dashboard.aggregator.statistics.UniqueVisitorsAllTim method), 12
empty() (pydash_app.dashboard.aggregator.statistics.Versions method), 12
empty() (pydash_app.dashboard.aggregator.statistics.VisitsPerIP method), 12
Endpoint (class in pydash_app.dashboard.endpoint), 14
endpoint_execution_times_boxplots() (in module pydash_web.controller.execution_times_boxplots), 30
EndpointCall (class in pydash_app.dashboard.endpoint_call), 16
error() (pydash_logger.logger.Logger method), 27
execution_times() (in module pydash_web.controller.utils), 31
execution_times_per_version() (in module pydash_web.controller.execution_times_per_version), 30
ExecutionTimePercentileABC (class in pydash_app.dashboard.aggregator.statistics), 9
ExecutionTimeTDigest (class in pydash_app.dashboard.aggregator.statistics),

9

F

FastestExecutionTime (class in pydash_app.dashboard.aggregator.statistics), 10
FastestQuartileExecutionTime (class in pydash_app.dashboard.aggregator.statistics), 10
fetch_aggregator() (pydash_app.dashboard.aggregator.aggregator_group.AggregatorGroup method), 6
fetch_aggregator_daterange() (pydash_app.dashboard.aggregator.aggregator_group.AggregatorGroup method), 7
fetch_aggregator_inclusive_daterange() (pydash_app.dashboard.aggregator.aggregator_group.AggregatorGroup method), 7
fetch_aggregators_per_timeslice() (pydash_app.dashboard.aggregator.aggregator_group.AggregatorGroup method), 7
fetch_and_add_endpoint_calls() (in module pydash_app.dashboard.services.fetching), 13
fetch_and_add_endpoints() (in module pydash_app.dashboard.services.fetching), 13
fetch_and_add_historic_endpoint_calls() (in module pydash_app.dashboard.services.fetching), 13
fetch_and_update_historic_dashboard_info() (in module pydash_app.dashboard.services.fetching), 13
fetch_and_update_new_dashboard_info() (in module pydash_app.dashboard.services.fetching), 13
fetch_endpoint_calls_failure (pydash_app.dashboard.entity.DashboardState attribute), 18
fetched_endpoint_calls (pydash_app.dashboard.entity.DashboardState attribute), 18
field_name() (pydash_app.dashboard.aggregator.statistics.AverageExecutionTime method), 9
field_name() (pydash_app.dashboard.aggregator.statistics.ExecutionTimeTDigest method), 9
field_name() (pydash_app.dashboard.aggregator.statistics.FastestExecutionTime method), 10
field_name() (pydash_app.dashboard.aggregator.statistics.FastestQuartileExecutionTime method), 10
field_name() (pydash_app.dashboard.aggregator.statistics.MedianExecutionTime method), 10
field_name() (pydash_app.dashboard.aggregator.statistics.NinetiethPercentileExecutionTime method), 10
field_name() (pydash_app.dashboard.aggregator.statistics.NinetyNinthPercentileExecutionTime method), 10
field_name() (pydash_app.dashboard.aggregator.statistics.SlowestExecutionTime method), 10
field_name() (pydash_app.dashboard.aggregator.statistics.SlowestQuartileE method), 10
field_name() (pydash_app.dashboard.aggregator.statistics.Statistic class method), 11
field_name() (pydash_app.dashboard.aggregator.statistics.TotalExecutionTi method), 11
field_name() (pydash_app.dashboard.aggregator.statistics.TotalVisits method), 11
field_name() (pydash_app.dashboard.aggregator.statistics.UniqueVisitorsAl method), 12
field_name() (pydash_app.dashboard.aggregator.statistics.Versions method), 12
field_name() (pydash_app.dashboard.aggregator.statistics.VisitsPerIP method), 12
find() (in module pydash_app.dashboard), 20
find() (in module pydash_app.dashboard.repository), 20
find() (in module pydash_app.user), 25
find() (in module pydash_app.user.repository), 23
find_by_name() (in module pydash_app.user), 25
find_by_name() (in module pydash_app.user.repository), 23
find_by_verification_code() (in module pydash_app.user), 25
find_by_verification_code() (in module pydash_app.user.repository), 23
find_verified_dashboard() (in module pydash_app.dashboard), 20
first_endpoint_call_time() (pydash_app.dashboard.entity.Dashboard method), 18
flask_monitoring_dashboard_client (module), 1
FloatStatisticABC (class in pydash_app.dashboard.aggregator.statistics), 10
foo() (in module periodic_tasks), 5
format_verification_mail_html() (in module pydash_mail.templates), 27
format_verification_mail_plain() (in module pydash_mail.templates), 27

G

generate_new_verification_code() (pydash_app.user.entity.User method), 21
get_dashboard() (in module pydash_web.api_routes), 32
get_dashboards() (in module pydash_web.api_routes), 32
get_data() (in module flask_monitoring_dashboard_client), 1
get_details() (in module flask_monitoring_dashboard_client), 1
get_endpoint_execution_times_boxplots() (in module pydash_web.api_routes), 32
get_execution_times_boxplot() (in module pydash_web.api_routes), 32

get_execution_times_per_version_dashboard() (in module pydash_web.api_routes), 32
get_execution_times_per_version_endpoint() (in module pydash_web.api_routes), 32
get_hourly_data() (in module pydash_web.controller.visitor_heatmap), 31
get_id() (pydash_app.dashboard.endpoint.Endpoint method), 15
get_id() (pydash_app.dashboard.entity.Dashboard method), 18
get_id() (pydash_app.user.entity.User method), 21
get_monitor_rules() (in module flask_monitoring_dashboard_client), 2
get_unique_visitor_heatmap() (in module pydash_web.api_routes), 32
get_verification_code() (pydash_app.user.entity.User method), 22
get_verification_code_expiration_date() (pydash_app.user.entity.User method), 22
get_visitor_heatmap() (in module pydash_web.api_routes), 32

H

handle_statistic_per_timeslice() (in module pydash_web.controller.dashboards), 29
handle_statistic_without_timeslice() (in module pydash_web.controller.dashboards), 29
has_verification_code_expired() (pydash_app.user.entity.User method), 22

I

info() (pydash_logger.logger.Logger method), 27
initialize_endpoint_calls_failure (pydash_app.dashboard.entity.DashboardState attribute), 18
initialize_endpoints_failure (pydash_app.dashboard.entity.DashboardState attribute), 18
initialized_endpoint_calls (pydash_app.dashboard.entity.DashboardState attribute), 18
initialized_endpoints (pydash_app.dashboard.entity.DashboardState attribute), 18
InvalidVerificationCodeError, 24
is_expired() (pydash_app.user.verification_code.VerificationCode method), 24
is_valid_dashboard() (in module pydash_app.dashboard.services), 14
is_verified() (pydash_app.user.entity.User method), 22

L

load_user() (in module pydash_web), 32
Logger (class in pydash_logger.logger), 27
login() (in module pydash_web.api_routes), 32
login() (in module pydash_web.controller.login), 30
logout() (in module pydash_web.api_routes), 32
logout() (in module pydash_web.controller.logout), 30

M

MAIL_DEFAULT_SENDER (config.Config attribute), 1
MAIL_PASSWORD (config.Config attribute), 1
MAIL_PORT (config.Config attribute), 1
MAIL_SERVER (config.Config attribute), 1
MAIL_USE_TLS (config.Config attribute), 1
MAIL_USERNAME (config.Config attribute), 1
match_datetime_string_with_formats() (in module pydash_web.controller.dashboards), 30
maybe_find_user() (in module pydash_app.user), 25
MedianExecutionTime (class in pydash_app.dashboard.aggregator.statistics), 10
MultiIndexedPersistentCollection (class in pydash_database), 26

N

NinetiethPercentileExecutionTime (class in pydash_app.dashboard.aggregator.statistics), 10
NinetyNinthPercentileExecutionTime (class in pydash_app.dashboard.aggregator.statistics), 10
not_initialized (pydash_app.dashboard.entity.DashboardState attribute), 18
nr_of_digits (pydash_app.dashboard.aggregator.statistics.FloatStatisticABC attribute), 10

P

partition_by_day_fun() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
partition_by_group_by_fun() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
partition_by_hour_fun() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
partition_by_ip_fun() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
partition_by_minute_fun() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
partition_by_month_fun() (in module pydash_app.dashboard.aggregator.aggregator_group), 8

partition_by_version_fun() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
partition_by_week_fun() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
partition_by_year_fun() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
partition_field_names() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
partition_funs (pydash_app.dashboard.aggregator.aggregator_group.AggregatorGroup attribute), 8
partition_powerset (pydash_app.dashboard.aggregator.aggregator_group.AggregatorGroup attribute), 8
partitions_set (pydash_app.dashboard.aggregator.aggregator_group.AggregatorGroup attribute), 8
percentile_nr (pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC attribute), 9
percentile_nr() (pydash_app.dashboard.aggregator.statistics.FastestExecutionTime method), 10
percentile_nr() (pydash_app.dashboard.aggregator.statistics.FastestQuartileExecutionTime method), 10
percentile_nr() (pydash_app.dashboard.aggregator.statistics.MedianExecutionTime method), 10
percentile_nr() (pydash_app.dashboard.aggregator.statistics.NinetiethPercentileExecutionTime method), 10
percentile_nr() (pydash_app.dashboard.aggregator.statistics.NinetyNinthPercentileExecutionTime method), 10
percentile_nr() (pydash_app.dashboard.aggregator.statistics.SlowestExecutionTime method), 10
percentile_nr() (pydash_app.dashboard.aggregator.statistics.SlowestQuartileExecutionTime method), 10
perform_append() (pydash_app.dashboard.aggregator.statistics.AverageExecutionTime method), 9
perform_append() (pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentileABC method), 9
perform_append() (pydash_app.dashboard.aggregator.statistics.ExecutionTimeTDigest method), 9
perform_append() (pydash_app.dashboard.aggregator.statistics.Statistic method), 11
perform_append() (pydash_app.dashboard.aggregator.statistics.TotalExecutionTime method), 11
perform_append() (pydash_app.dashboard.aggregator.statistics.TotalVisits method), 11
perform_append() (pydash_app.dashboard.aggregator.statistics.UniqueVisitorsAllTime method), 12
perform_append() (pydash_app.dashboard.aggregator.statistics.VisitsPerIP method), 12
perform_append() (pydash_app.dashboard.aggregator.statistics.VisitsPerIP method), 12
periodic_task() (in module periodic_tasks), 5
periodic_tasks (module), 4
periodic_tasks.pqdict_iter_upto_priority (module), 2
periodic_tasks.queue_nonblocking_iter (module), 2
periodic_tasks.task_scheduler (module), 2
powerset_generator() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
pqdict_iter_upto_priority (class in periodic_tasks.pqdict_iter_upto_priority), 2
pydash (module), 6
pydash_app (module), 26
pydash_app.dashboard (module), 20
pydash_app.dashboard.aggregator (module), 13
pydash_app.dashboard.aggregator.aggregator_group (module), 6
pydash_app.dashboard.aggregator.statistics (module), 9
pydash_app.dashboard.endpoint (module), 14
pydash_app.dashboard.endpoint_call (module), 16
pydash_app.dashboard.entity (module), 16
pydash_app.dashboard.repository (module), 19
pydash_app.dashboard.services (module), 14
pydash_app.dashboard.services.fetching (module), 13
pydash_app.dashboard.services.seeding (module), 14
pydash_app.user (module), 24
pydash_app.user.entity (module), 21
pydash_app.user.repository (module), 22
pydash_app.user.services (module), 21
pydash_app.user.services.seeding (module), 21
pydash_app.user.verification_code (module), 24
pydash_app.user.verification_code (module), 24
pydash_exclusion (module), 26
pydash_logger (module), 27
pydash_logger.logger (module), 27
pydash_mail (module), 28
pydash_mail.templates (module), 27
pydash_web (module), 32
pydash_web.api (module), 31
pydash_web.api_routes (module), 31
pydash_web.controller (module), 31
pydash_web.controller.change_dashboard_settings (module), 28
pydash_web.controller.change_password (module), 28
pydash_web.controller.change_settings (module), 28
pydash_web.controller.dashboards (module), 28
pydash_web.controller.delete_dashboard (module), 30
pydash_web.controller.delete_user (module), 30
pydash_web.controller.execution_times_boxplots (module), 30
pydash_web.controller.execution_times_per_version (module), 30
pydash_web.controller.login (module), 30
pydash_web.controller.logout (module), 30
pydash_web.controller.register_dashboard (module), 30

pydash_web.controller.register_user (module), 31
pydash_web.controller.user_verification (module), 31
pydash_web.controller.utils (module), 31
pydash_web.controller.visitor_heatmap (module), 31
pydash_web.react_server (module), 32

Q

queue_nonblocking_iter (class in periodic_tasks.queue_nonblocking_iter), 2
qux() (in module periodic_tasks), 5

R

reduce_precision() (in module pydash_app.dashboard.aggregator.statistics), 12
register_dashboard() (in module pydash_web.api_routes), 32
register_dashboard() (in module pydash_web.controller.register_dashboard), 30
register_user() (in module pydash_web.api_routes), 32
register_user() (in module pydash_web.controller.register_user), 31
remove_duplicate_categories() (in module pydash_app.dashboard.aggregator.aggregator_group), 8
remove_endpoint() (pydash_app.dashboard.entity.Dashboard method), 18
remove_endpoint_call() (pydash_app.dashboard.endpoint.Endpoint method), 15
remove_from_repository() (in module pydash_app.dashboard), 20
remove_from_repository() (in module pydash_app.user), 25
remove_task() (in module periodic_tasks), 5
remove_task() (periodic_tasks.task_scheduler.TaskScheduler method), 3
rendered_value() (pydash_app.dashboard.aggregator.statistics.FloatStatisticABC method), 10
rendered_value() (pydash_app.dashboard.aggregator.statistics.Statistic method), 11
rendered_value() (pydash_app.dashboard.aggregator.statistics.UniqueVisitorsAllTime method), 12
rendered_value() (pydash_app.dashboard.aggregator.statistics.Versions method), 12
rendered_value() (pydash_app.dashboard.aggregator.statistics.VisitsPerIP method), 12

S

schedule_all_periodic_dashboards_tasks() (in module pydash_app.dashboard.services.fetching), 13
schedule_historic_dashboard_fetching() (in module pydash_app.dashboard.services.fetching), 14
schedule_periodic_dashboard_fetching() (in module pydash_app.dashboard.services.fetching), 14
schedule_periodic_pruning_task() (in module pydash_app.user.services.pruning), 20
schedule_periodic_tasks() (in module pydash_app), 26
SECRET_KEY (config.Config attribute), 1
seed() (in module pydash_app.dashboard.services.seeding), 14
seed() (in module pydash_app.user.services.seeding), 21
seed_datastructures() (in module pydash_app), 26
serve() (in module pydash_web.react_server), 32
set_monitored() (pydash_app.dashboard.endpoint.Endpoint method), 15
set_password() (pydash_app.user.entity.User method), 22
should_be_rendered (pydash_app.dashboard.aggregator.statistics.ExecutionTimeTDigest attribute), 9
should_be_rendered (pydash_app.dashboard.aggregator.statistics.Statistic attribute), 11
should_be_rendered() (pydash_app.dashboard.aggregator.statistics.AverageExecutionTime method), 9
should_be_rendered() (pydash_app.dashboard.aggregator.statistics.ExecutionTimePercentil method), 9
should_be_rendered() (pydash_app.dashboard.aggregator.statistics.TotalExecutionTime method), 11
should_be_rendered() (pydash_app.dashboard.aggregator.statistics.TotalVisits method), 11
should_be_rendered() (pydash_app.dashboard.aggregator.statistics.UniqueVisitorsAllTime method), 12
should_be_rendered() (pydash_app.dashboard.aggregator.statistics.Versions method), 12
should_be_rendered() (pydash_app.dashboard.aggregator.statistics.VisitsPerIP method), 12
SlowestExecutionTime (class in pydash_app.dashboard.aggregator.statistics), 10
SlowestQuartileExecutionTime (class in pydash_app.dashboard.aggregator.statistics), 10
splinter_driver_kwargs() (in module tests.conftest), 32
start() (periodic_tasks.task_scheduler.TaskScheduler method), 3
start_default_scheduler() (in module periodic_tasks), 6
start_task_scheduler() (in module pydash_app), 26

Statistic (class in pydash_app.dashboard.aggregator.statistics), 11
statistic (pydash_app.dashboard.aggregator.Aggregator attribute), 13
statistic() (pydash_app.dashboard.endpoint.Endpoint method), 15
statistic() (pydash_app.dashboard.entity.Dashboard method), 18
statistic_per_timeslice() (pydash_app.dashboard.endpoint.Endpoint method), 15
statistic_per_timeslice() (pydash_app.dashboard.entity.Dashboard method), 18
statistics_classes_with_dependencies (pydash_app.dashboard.aggregator.Aggregator attribute), 13
stop() (periodic_tasks.task_scheduler.TaskScheduler method), 3
stop_task_scheduler() (in module pydash_app), 26

T

TaskScheduler (class in periodic_tasks.task_scheduler), 2
tests (module), 33
tests.common_fixtures (module), 32
tests.conftest (module), 32
testserver() (in module tests.conftest), 33
TotalExecutionTime (class in pydash_app.dashboard.aggregator.statistics), 11
TotalVisits (class in pydash_app.dashboard.aggregator.statistics), 11

U

unauthorized() (in module pydash_web), 32
UniqueVisitorsAllTime (class in pydash_app.dashboard.aggregator.statistics), 11
update() (in module pydash_app.dashboard.repository), 20
update() (in module pydash_app.user.repository), 23
User (class in pydash_app.user.entity), 21

V

VerificationCode (class in pydash_app.user.verification_code), 24
VerificationCodeExpiredError, 24
verify() (in module pydash_app.user), 26
verify() (in module pydash_app.user.verification), 24
verify_user() (in module pydash_web.api_routes), 32
verify_user() (in module pydash_web.controller.user_verification), 31
Versions (class in pydash_app.dashboard.aggregator.statistics), 12
visitor_heatmap() (in module pydash_web.controller.visitor_heatmap), 31
VisitsPerIP (class in pydash_app.dashboard.aggregator.statistics), 12

W

warning() (pydash_logger.logger.Logger method), 27