PyDash Architecture Document

PyDash 2018
J. G. S. Overschie
T. W. E. Apol
A. Encinas-Rey Requena
K. Bolhuis
W. M. Wijnja
L. J. Doorenbos
J. Langhorst
A. Tilstra

Customers
Patrick Vogel
Mircea Lungu

TA
Patrick Vogel

June 24, 2018

Contents

Introduction

PyDash.io will be a platform for connecting existing Flask-monitoring dashboards to and monitor those dashboards as well. This document will describe the architecture and technologies used in the development of the PyDash.io platform. It will also provide a short overview of the API available.

There are several things this document focuses on, but they can be categorized in a handful of segments. Each will deal with a part of the design of PyDash.io. We will discuss the general architecture, tools used, use-cases, back-end, front-end, our database and the API we built for this piece of software.

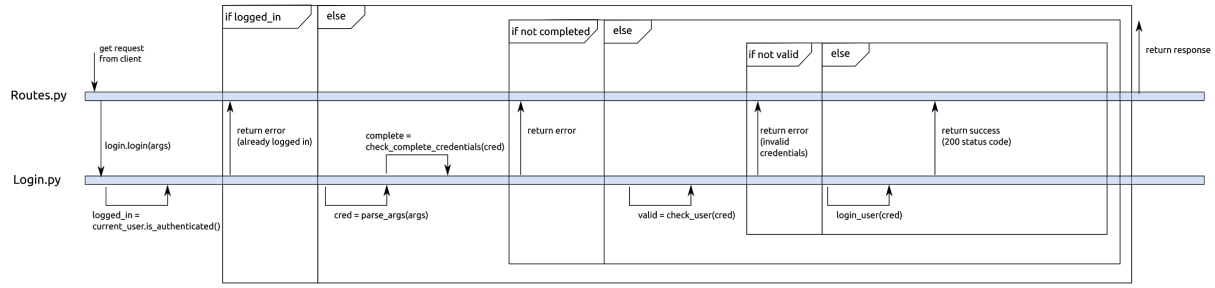
Architectural Overview

The PyDash.IO application will be split into a couple of separate parts:

- The Back-End part, which will be written using the Python programming language, and the Flask micro-web-framework.
- The Front-End part, which will be written as a Single-Page-Application using the React.JS interactive user interface framework.

These two parts will talk with each-other using a well-defined AJAX API that will be outlined later in this document.

Use-Cases



Logging in to PyDash

If a user wants to use PyDash, he or she needs to be logged in to an account. In order to log in the following steps have to be taken:

Preconditions

- The account with which you want to log in exists in the database and you know the correct credentials for said account

Postconditions

- The user is logged in to the correct account

Main success scenario

1. The user goes to the login page
2. The user fills in the name and corresponding password
3. The system successfully processes the request
4. The user is redirected to his own dashboard overview page.

Alternative flow

1. Incorrect credentials

(a) The user is notified he entered the wrong credentials

(b) Main success scenario continues at step 2.

Registering a user

If a user wants to use PyDash, he or she needs an account. To create such an account, a few steps have to be taken, which are described in this use case.

Preconditions

- None

Postconditions

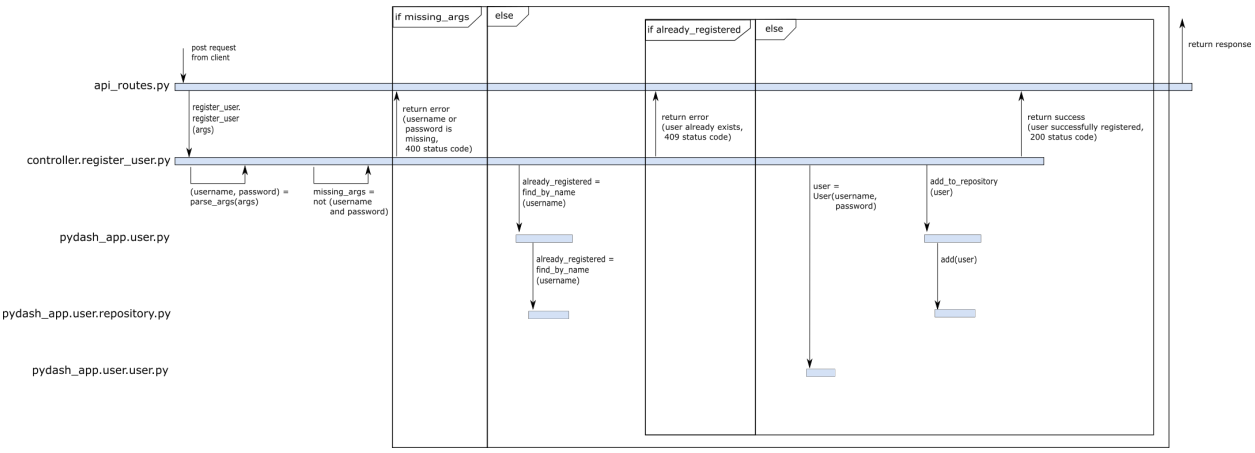
- The username-password combination is stored in the database
- An e-mail address is linked to the user

Main success scenario

1. The user requests to register an account
2. The user fills in the necessary details (name, password, email)
3. The system processes the request
4. A verification email is sent to the newly registered user
5. The user clicks the verification link in the email
6. The user is notified of his successful account creation

Alternative flow

1. Invalid combination, e.g. name is already taken.
   (a) The user is notified about what went wrong
   (b) Main success scenario continues at step 2.



Viewing the data of a certain dashboard

This use case describes the process of a user wanting to view the flask monitoring dashboard data of a certain site.

Preconditions

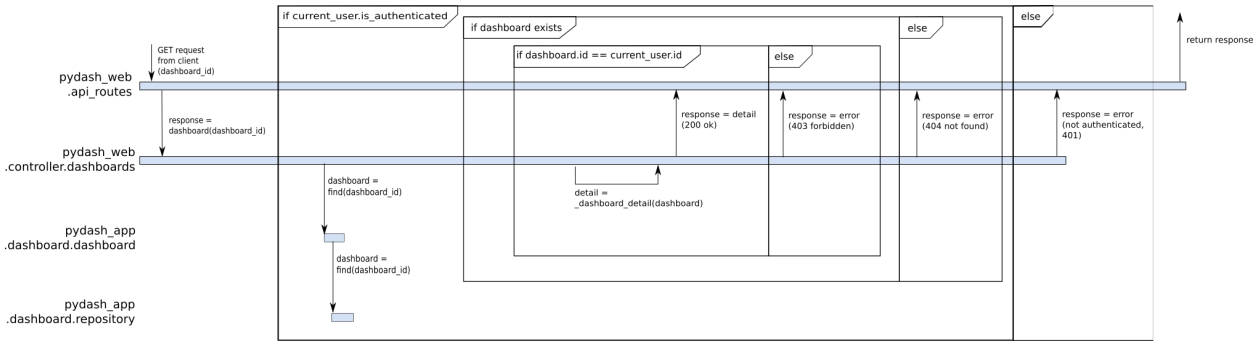
- The user is logged in
- The dashboard of which the data will be viewed is linked to the user

Postconditions

- The correct data of the correct dashboard is presented to the user in a readable fashion

Main success scenario

1. The user selects the dashboard for which he wants to see the data
2. Optionally the user filters out the endpoints he is not interested in
3. The system retrieves the correct dashboard and displays the data



Registering a dashboard

PyDash gathers data of multiple flask monitoring dashboards, but to do so they first have to be added by the user to his meta dashboard. This use case describes this process of adding a new dashboard to your overview.

Preconditions

- The user is logged in
- The user is in possession of the secret token associated with that dashboard

Postconditions

- The corresponding dashboard is added to the user's overview

Main success scenario

1. The user clicks the plus sign, the button for registering a new dashboard
2. The user fills in the url, the name he wants and the secret token associated with that dashboard
3. The system checks if the token and url combination is correct
4. The new dashboard is added to the overview page

Alternative flow

1. The token/url combination is incorrect
   (a) The user is notified that the given combination is wrong
   (b) Main success scenario continues at step 2.

Back-end Design

The Back-End part of the application is written in Python using the Flask micro-web-framework. The reason to use Flask here is to be able to use the *flask-monitoring-dashboard*, a Python library whose functionality PyDash.IO builds on top of, to be used for the *PyDash.IO* web-application itself as well.

The Back-End is split up using the well-known *Model-View-Controller* architecture pattern.

The *Model* contains all the actual application logic. Its implementation can be found in the *pydash_app* folder. The web-application is only a consumer of this package.

The *Controller* contains the dispatching logic to know how to respond to the different endpoints a visitor might request. The actual route-dispatching happens in *pydash_web/api_routes.py* , with the handling of each of the different routes being handled by its own dedicated module in the *pydash_web/controller/* folder.

The *View* part of the application currently consists of the *api_routes.py* file. This file serves as an interface between the front-end and back-end, thus being the view of the back-end with regards to the outside world.

Domain Driven Design

Inside the application logic (*pydash_app*), we use a simple variant of Domain Driven Design to split up our model's functionality in their respective concerns. For each data structure or finite-state-machine of interest, we create three parts:

1. A module of the entity name that contains the publicly available functions to interact with these entities.
2. A *Repository* module that knows how entities of this type are persisted: It exposes functionality to find certain entities of this type in the persistence layer, create new ones, update existing ones and possibly delete them. Only the actions that are actually relevant to the specific entities are modeled. The Repository is the only part that actually talks with the underlying persistence layer, and as such it can be considered an Adapter.
3. An *Entity* class which is a plain Python class with the properties of interest and the methods that make sense to prove and possibly manipulate this kind of entity.

Important to note is thus:

The Entity never knows how it itself is persisted,

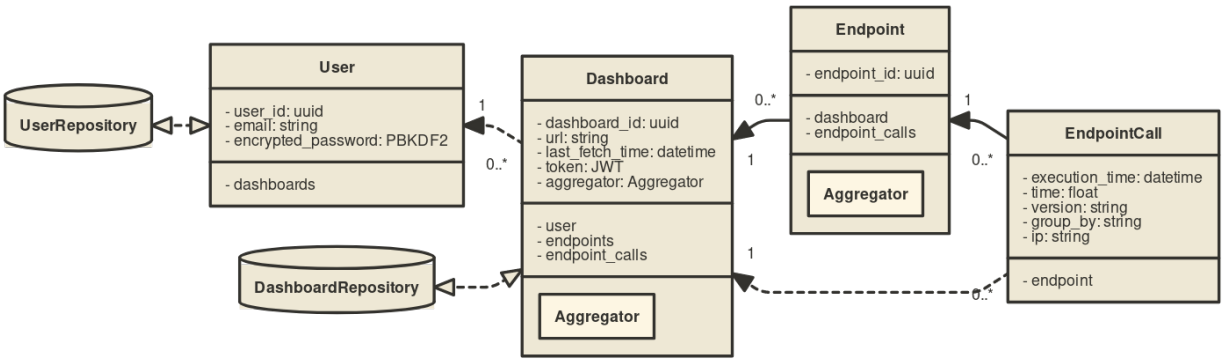
The Repository does not care about the internals of the Entity's logic.

Because of this, both are easy to change without needing to touch the other.

Database Design

We have decided on using the ZODB object-database as it provides a clear relationship between objects and elements in the database. ZODB in essence uses a large tree-like structure to store Python objects. Each class has its own branch which is a set of objects of that class. Each of these branches is called a repository.

To connect to ZODB we use ZEO. This is a tool that allows the database to be ran in a separate process to not poorly influence the performance of the web-app.



Front-end Design

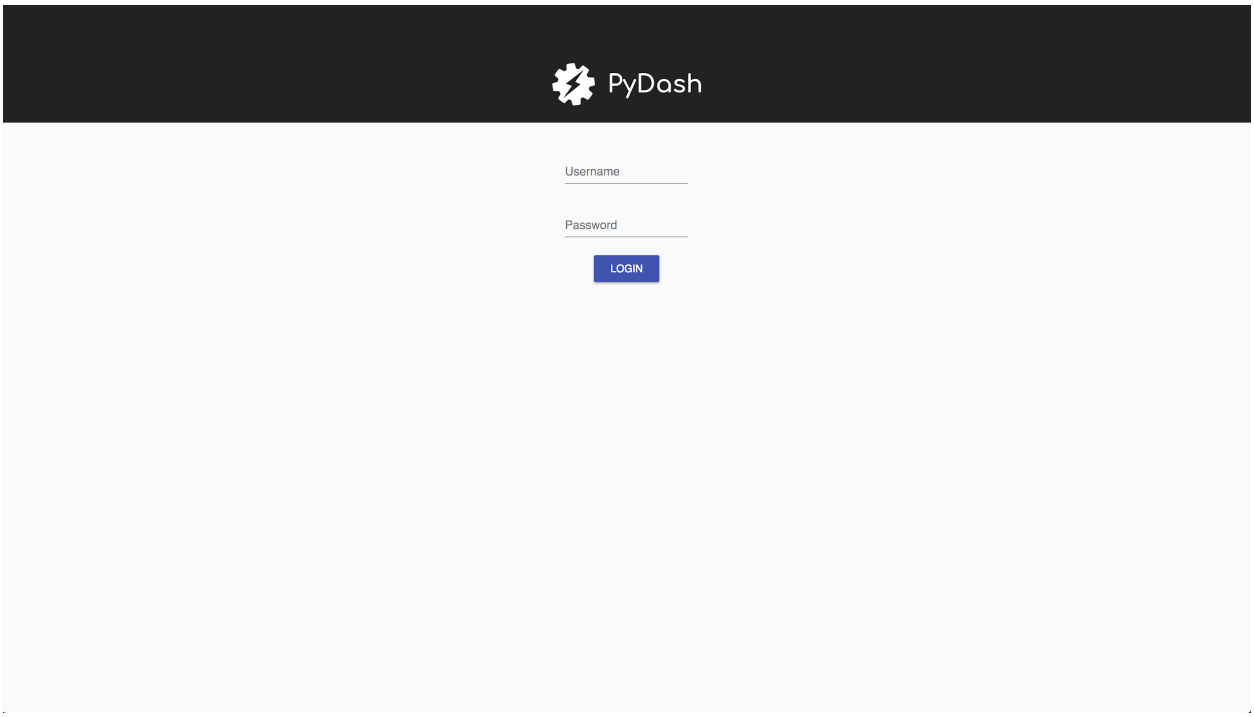
For the front-end part of the application, we will be creating a one-page application using the React.js framework, requesting all necessary information from the back-end using AJAX (Asynchonous Javascript and XML) calls.

For the User interface, we will be using the Material-UI framework, with the pages looking like this:

For the time being, there will be a few different pages:

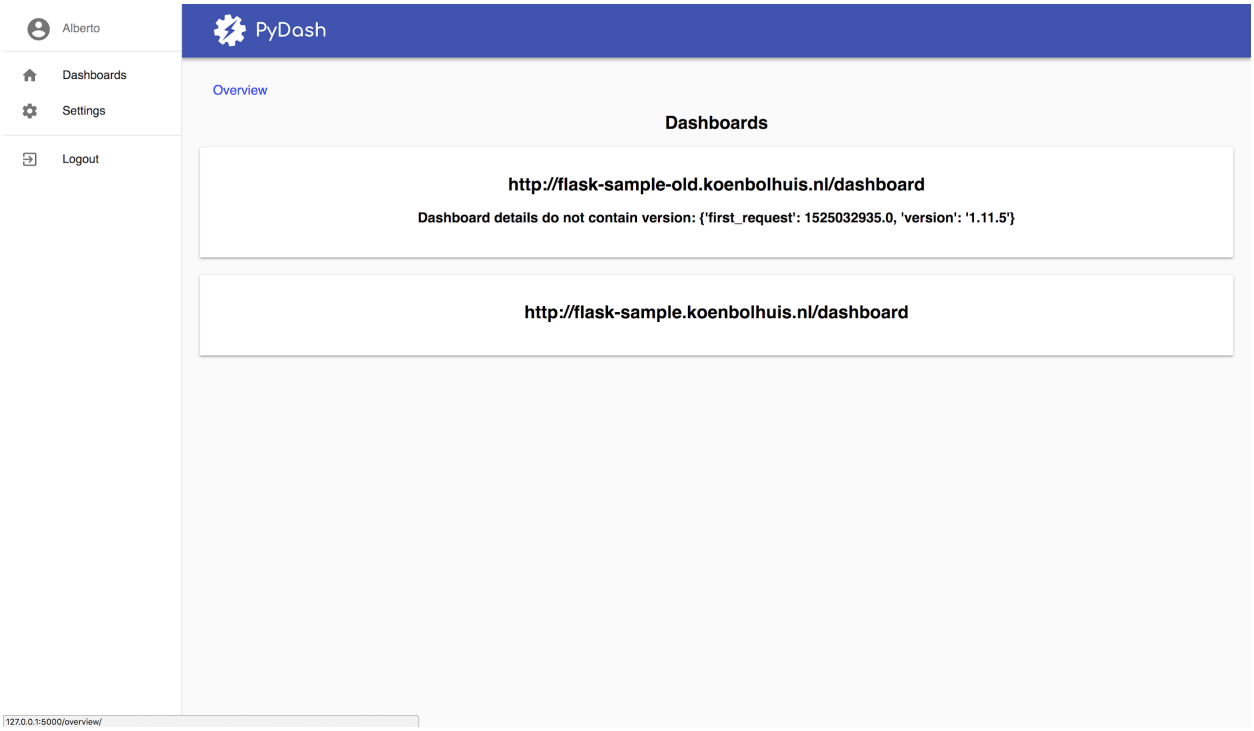
- The landing page/login page: When the users enter the website, this will be the first page they see. They will be able to login to an existing account.

The login page will look approximately like this.



- The overview page: Here the users will be able to see an overview of all dashboards they are monitoring.

The overview page will look approximately like this.

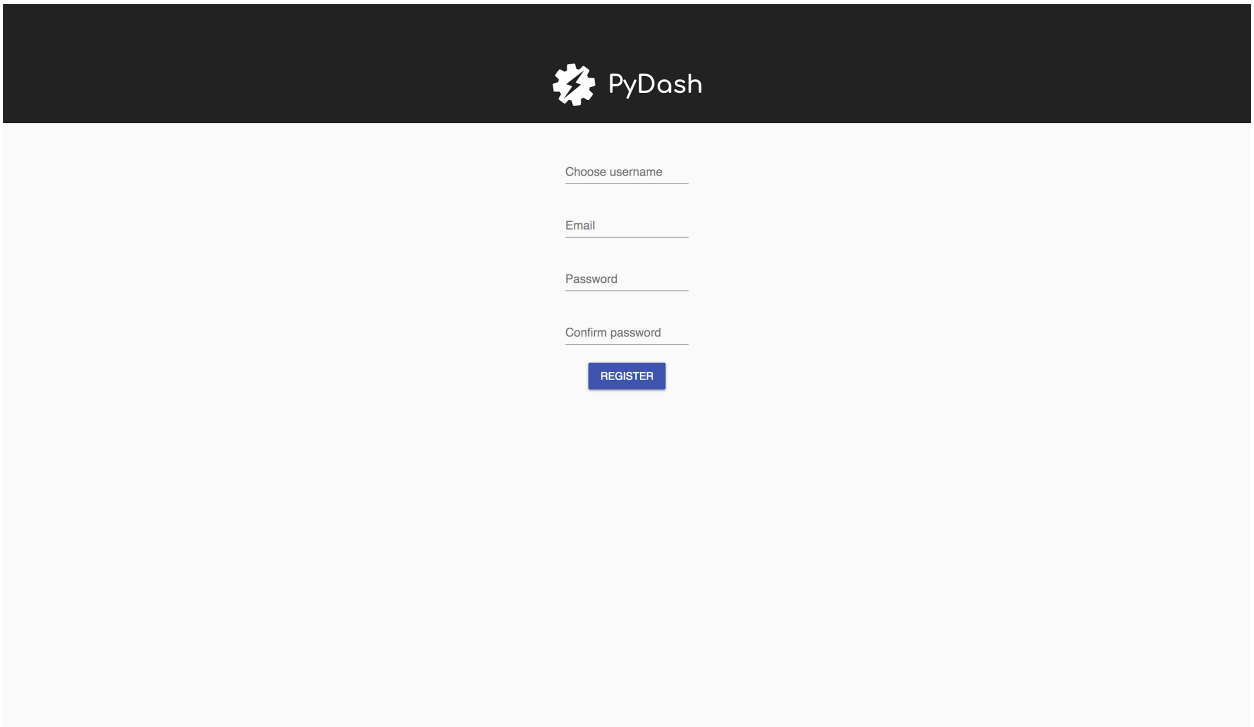


- The dashboard page (Here the users will be able to see all information coming in from the specified dashboard.

The dashboard page will look approximately like this.

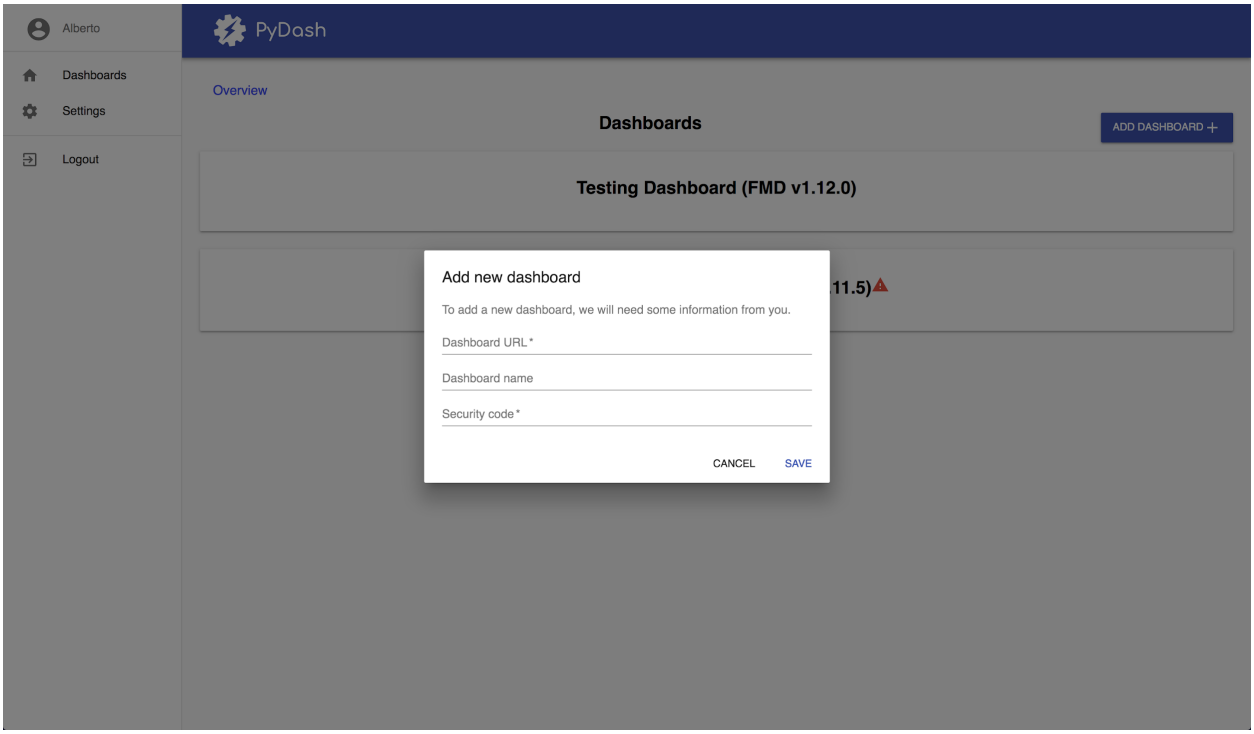
- The register page: Here the users will be able to create a new account

The registration page will look approximately like this.



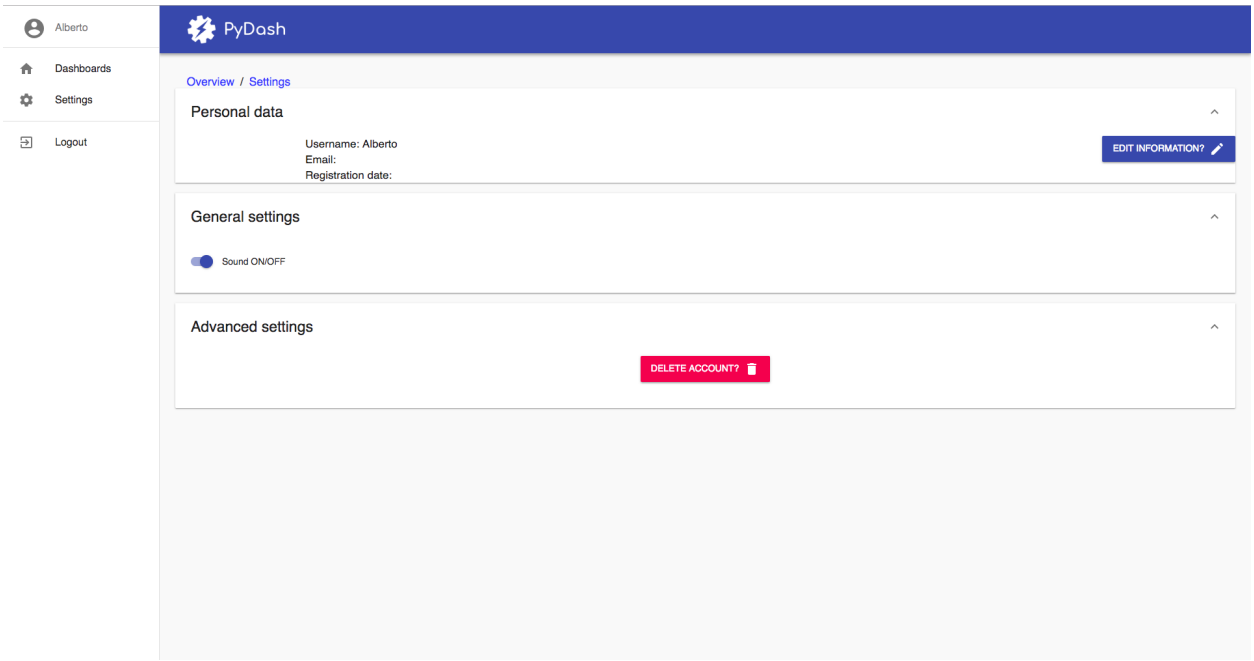
- Add dashboard: Here the users will be able to create a new dashboard by pressing a button located on the top right corner which opens a form to create the dashboard.

The add dashboard page will look approximately like this.



- The settings page: Here the users will be able to edit or erase their accounts by changing their username, password or email inside a dialog form. As well as that the user will be allowed to turn of the sounds. It consists on expandable panels.

The settings page will look approximately like this.



Back-End Technology Stack

In the Back-End of the Pydash.IO application, we have attempted to keep the number of moving parts as small as possible, because adding more things makes it increasingly difficult to set up the codebase on a new server and to maintain it all because of the mental overhead to understand a large group of tools at the same time.

This is the main reason we decided *not* to use an SQL database, but instead a Python Object Database, which allows us to just work directly with Python objects. It is also the reason we do not use the common library Celery for background-task-management (which requires an externally-running in-memory database like Redis) but instead created our own simple task scheduler that directly works inside Python 3.

- **Python 3:** The Programming Language which everything is built in. Python is an object-oriented programming language (with influences from functional programming) that has a large ecosystem of people providing free and open-source libraries to help with all kinds of tasks.
- **Flask:** The 'micro' web-framework built on top of Python, which makes it very clear to the developer what is going on (rather than doing all kinds of magic behind the scenes).
- **PyJWT** is used to work with JSON-WEB-Tokens which encrypt/decrypt the communication between the remote flask-monitoring-dashboards over a potentially unsafe (i.e. *http*) connection.
- **Zope Object Database** (exposed using ZEO): The database-layer we use to persist Python-objects in. We have created a custom indexable dictionary-like structure that allows us to easily search for certain objects on top of this (This has been split off in the Python package **multi-indexed-collection**). Using an Object Database means that we do not require to think about the peculiarities of an object-relational-mapping tool.
- **Custom Periodic Task scheduler:** A custom piece of code that schedules tasks using a pool of Subprocesses. This means that background- and periodic tasks will always run quickly without impacting the people that perform a request to the Flask application. The actual task scheduler is built in such a way that only the minimum of work is done to check what tasks should be run shortly. (internally, an indexable priority queue is used for this).

Front-End Technology stack

We use a lot of different technologies in the Front-End. Mainly, we follow some of the conventions used in the React build tool create-react-app. Technologies we use include:

webpack

Webpack gives us a nice way to pack up our application for production usage. It takes all your javascript modules, and recursively iterates through its dependencies, which enables us to exactly only include the modules that we actually use. This makes for no redundant code, and a nice packaged up single file with your application. It does the same for you with CSS. It also compresses images, along with some other nice stuff.

* Module-bundling
* Asset compression
* Code minification

LiveReload

LiveReload enables us to have a fast development cycle by automatically reloading the browser page upon saving a file from the project. This is done by efficiently monitoring the project filesystem folder.

* Fast development cycle

ESLint

ESLint scans our javascript for common flaws and warns you about them right in your text editor. Common flaws include warning about unused variables, unreachable code, messy assignments or weird constructs. It improves your code quality greatly overall. Aside from warnings in your editor it also warns you on the command line in the build process.

* Find errors early

JSX

JSX is a widely used precompiled dialect of Javascript in React.js. It allows mixed usage of HTML and JS in one file without too much syntactic overhead.

* Write HTML in JS
* Cleaner code

webpack

Webpack gives us a nice way to pack up our application for production usage. It takes all your javascript modules, and recursively iterates through its dependencies, which enables us to exactly only include the modules that we actually use. This makes for no redundant code, and a nice packaged up single file with your application. It does the same for you with CSS. It also compresses images, along with some other nice stuff.
* Module-bundling
* Asset compression
* Code minification

babel

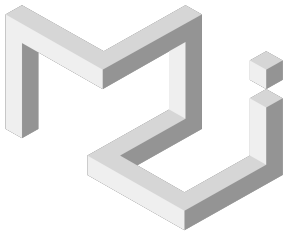
Babel allows us to write next-generation JavaScript! Nowadays most JS developers write JS using the ES6 (and higher) standards. Babel allows compiling this modern JS syntax to browser-compatible JavaScript.
* Modern JS

Service Worker

Create-react-app automatically hooks up a service-worker for you, which allows offline-usage and asset caching for your app.
* Cache
assets

github.com/axios

Axios is the AJAX library we use to make HTTP calls in React. For every piece of retrieved JSON data, we send an AJAX request to our backend using Axios.



Material-UI

Material-ui is the interface library we use. It is based upon the Material Design principles from Google, and has bindings for React.js. Bindings allow you to easily observe and bind Javascript variables to text in the view.
* No need to create UI components ourselves.

 webpack	Webpack gives us a nice way to pack up our application for production usage. It takes all your javascript modules, and recursively iterates through its dependencies, which enables us to exactly only include the modules that we actually use. This makes for no redundant code, and a nice packaged up single file with your application. It does the same for you with CSS. It also compresses images, along with some other nice stuff. * Module-bundling * Asset compression * Code minification
 NIVO	We use NIVO for displaying graphs in our statistics page. NIVO has an elaborate library of default graphs you can use. Customization is easy to do if wanted, NIVO's defaults are really good as well, though.

API Specification

*an * (asterisk) indicates that a certain parameter is mandatory.*

User

/api/login

Logs a user into the system.

- *name (string) - Name of the user to be logged in.
- *password (string) - Password for login in clear text.

Responses:

- 200 - User was logged in correctly.
- 400 - Missing username or password.
- 401 - Invalid username/password supplied.

/api/logout

Logs the current user out.

No parameters required

Responses:

- 200 - Successful operation.
- 401 - Returned if user was not logged in.

/api/user/register

Registers a user with the system.

- *name (string) - Name of the user to be registered.
- *password (string) - Password in clear text.

Responses:

- 200 - User successfully registered.
- 400 - Username/password missing.
- 409 - Username already registered.

/api/user/delete

Deletes the current user and all connected dashboards from the system.

- *password (string) - Password in clear text.

Responses:

- 200 - User successfully deleted.
- 400 - Password missing.
- 401 - Incorrect password provided.
- 500 - User not found.

/api/user/verify/{verification_code}

Verifies the user connected to the {verification_code}.

- *verification_code (string) - The verification token in clear text.

Responses:

- 200 - User successfully verified.
- 400 - Invalid or expired verification code.

/api/user/change_settings

Updates the settings for the current user.

- username (string) - New username
- play_sounds (boolean) - New sound setting

Responses:

- 200 - Settings successfully changed.
- 400 - Invalid settings or username already in use.

/api/user/change_password

Updates the password of the current user.

- *current_password (string) - The current password of the user.
- *new_password (string) - The new password of the user.

Responses:

- 200 - Password updated successfully.
- 400 - One of the passwords is missing.
- 401 - Current password invalid.

Dashboard

/api/dashboards

Returns the data for the dashboard overview page of the currently logged in user in a JSON format.

No parameters required.

Responses:

- 200 - Successful retrieval, even if no dashboards were found.

Example:

```
[
  {
    "id": "4242424242424242",
    "url": "http://pydash.io/",
    "endpoints": [
      {
        "name": "my.endpoint.name",
        "enabled": true
      }
    ]
  }
]
```

/api/dashboards/{dashboard_id}

Returns aggregated data for a particular dashboard in JSON format.

- *dashboard_id (string) - UUID of the dashboard to be retrieved.

Responses:

- 200 - Successful retrieval of data.

Example:

```
{
  "id": "4242424242424242",
  "url": "http://pydash.io/",
  "endpoints": [
    {
      "name": "my.endpoint.name",
      "enabled": true
    }
  ]
}
```

- 400 - Invalid UUID supplied.
- 403 - Current user is not allowed to view dashboard.
- 404 - Dashboard not found.

/api/dashboards/{dashboard_id}/statistic

Returns aggregated data for a particular dashboard and statistic in JSON format.

- *dashboard_id (string) - UUID of the dashboard to be retrieved.

Url query string parameters:

- *statistic (string) - name of the statistic of which aggregated information should be retrieved. As of yet, the following statistics are supported:
  - 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_percientile_execution_time
  - Slowest_measured_execution_time
- start_date, end_date (strings) - The start- and end dates of the datetime range in which the desired information lies. start_date and end_date are resp. The inclusive lower- and exclusive upper bounds of this datetime range.

  If start_date is not provided, it defaults to the timestamp of the dashboard's first endpoint call.

  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, as well as that they conform to the ISO-8601 date and time standard.

- timeslice (string) - Indicates the data should be returned as a series of points in time, each timeslice long.

The currently supported timeslices are: 'year', 'month', 'week', 'day', 'hour' and 'minute'.

- timeslice_is_static (boolean) - Indicates whether the timeslice should be 'static' (i.e. have a set place in the overarching timespan [e.g. W23, or the month of June]) or 'dynamic' (i.e. its start and end can be anything, but its length is set in stone)

Note that timeslice_is_static is mandatory when timeslice is provided.

Responses:

- 200 - Successful retrieval of data.

Example:

```
{
  "2018-04": 75.3,
  "2018-05": 63.6,
  "2018-06": 35.8
}
```

- 400 - 'statistic' url query string parameter is not provided.
- 400 - Invalid format of 'start_date' or 'end_date'.
- 400 - 'end_date' is earlier than 'start_date'.
- 400 - 'timeslice_is_static' is not provided when 'timeslice' is.
- 400 - 'timeslice_is_static' has an invalid value (as Flask passes it as a string to the python back-end).
- 400 - 'timeslice' and 'timeslice_is_static' combination is not supported.
- 403 - Current user is not allowed to view dashboard.
- 404 - Dashboard not found.

/api/dashboards/register

Registers a new dashboard using the given parameters and adds jobs for it to the scheduler.

- *name (string) - Name of the new dashboard.
- *url (string) - URL of the new dashboard.
- *token (string) - Security token for the new dashboard.

Responses:

- 200 - Dashboard successfully created.
- 400 - Missing or invalid values.

/api/dashboards/{dashboard_id}/delete

Deletes the dashboard with id {dashboard_id}.

- *dashboard_id (string) - UUID of the dashboard to be deleted.

Responses:

- 200 - Dashboard successfully deleted.
- 400 - Invalid dashboard_id.
- 403 - Current user not authorized to view dashboard.
- 404 - Dashboard not found.

/api/dashboards/{dashboard_id}/endpoint_boxplots

Returns the boxplot endpoint data from the dashboard with id {dashboard_id}.

- *dashboard_id (string) - UUID of the dashboard you want the data of.

Responses:

- 200 - Data successfully retrieved.
- 400 - Invalid dashboard_id.
- 403 - Current user not allowed to view dashboard.

Customer Contact

Patrick Vogel <p.p.vogel@student.rug.nl>

Mircea Lungu <m.f.lungu@rug.nl>

Meeting Log

For communication with the customer it was decided that we solely rely on using Slack. We therefore did not meet in person regarding issues the customer should be notified of. However, a summary of the decisions made is presented here:

- 11-03-2018: We requested sample data from the customer to use for testing purposes.
- 29-03-2018: We asked the customer to update the API of the FMD so we could fetch data in timeslices. This was done and the customer updated us of the addition.
- 12-04-2018: We asked how the get_json_details api call handled time. We were told it currently was broken but would be fixed in the next version.
- 27-04-2018: We asked the customer if we could get a server to host our development build on. We have been looking into this together for some time.
- 14-05-2018: The customer told us we could look into external server hosting solutions for which we will be reimbursed. They will also be getting us access to larger volumes of data.

Changelog

Date	Iteration	Changes	Author
2018-03-08	First delivery.	Initial document	*Shared Effort*
2018-04-05	Sprint 3	Updating front-end with Patrick's notes.	Alberto Encinas
2018-04-06	Sprint 3	Updated back-end info	Jeroen Langhorst
2018-04-12	Sprint 3	Back-end technology stack description, database schema design	Wiebe-Marten Wijnja
2018-04-16	Sprint 3	Add front-end tech stack.	Jeroen Overschie
2018-04-16	Sprint 3	Added use cases	Lars Doorenbos
2018-04-16	Sprint 3	Added sequence diagrams	Jeroen Langhorst
2018-04-23	Sprint 4	Updated and added use cases	Lars Doorenbos
2018-04-28	Sprint 4	Added API spec	Jeroen langhorst
2018-04-29	Sprint 4	Small textual improvements to the Frontend technology stack and the backend API	Koen Bolhuis
2018-04-29	Sprint 4	Updated use cases	Lars Doorenbos
2018-04-30	Sprint 4	Added a sequence diagram for the use case "Viewing the data of a certain dashboard"	Koen Bolhuis
2018-05-03	Sprint 5	Added password parameter to API spec for /api/user/delete	Koen Bolhuis
2018-05-08	Sprint 5	Added new screenshots of pages in the Front-End design	Alberto Encinas
2018-06-23	Sprint 7	Conversion to LaTeX	Wiebe-Marten Wijnja

PyDash Back-end Python Documentation

PyDash Documentation

Release 0.4.0

The PyDash Team

Jun 24, 2018

CONTENTS:

CHAPTER

ONE

PYDASH

1.1 flask_monitoring_dashboard_client package

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

1.2 periodic_tasks package

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)
>>> 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.

1.2.1 Submodules

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.

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.

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
```

1.3 pydash module

1.4 pydash_app package

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**()

1.4.1 Subpackages

pydash_app.dashboard package

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*)

Subpackages

pydash_app.dashboard.aggregator package

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

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=group_by cat

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

pydash_app.dashboard.services package

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.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*,
sched-
uler=<periodic_tasks
ob-
ject>)

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*,
in-
ter-
val=datetime.timedel
3600),
sched-
uler=<periodic_tasks
ob-
ject>)

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.

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']}
>>> 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']}
```

(continues on next page)

(continued from previous page)

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)
>>> 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*)

pydash_app.user package

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"
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.

Subpackages

pydash_app.user.services package

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.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
```

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()

1.5 pydash_database package

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.)

1.6 pydash_logger package

1.6.1 Submodules

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

1.7 pydash_mail package

1.7.1 Submodules

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.

1.8 pydash_web package

Entrypoint of *pydash_web*

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

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

pydash_web.**unauthorized**()

1.8.1 Subpackages

pydash_web.controller package

The controller contains one dispatching function per flask_webapp endpoint action.

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=N*

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'*)

1.8.2 Submodules

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.

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**()

pydash_web.react_server module

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

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

p

INDEX

G

H

I

L

M

N

P

Q

R

S

T

U

V

W