PyDash Architecture Document

@ll@

[t]0.47**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

&

[t]0.47**Customers**

Patrick Vogel

Mircea Lungu

TA

Patrick Vogel

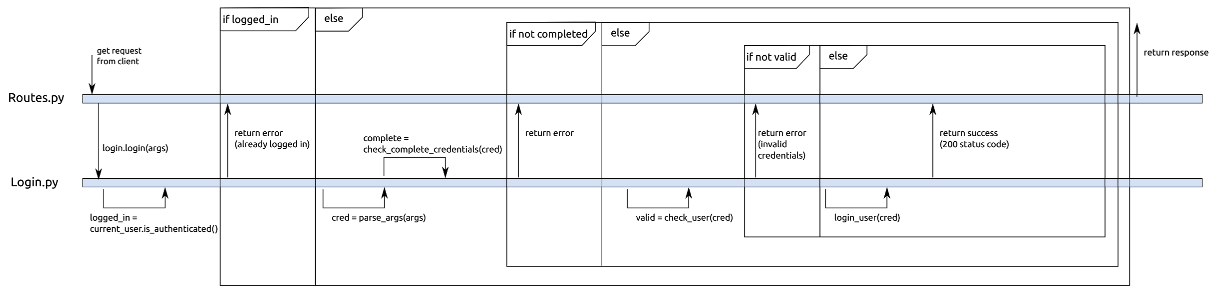
PyDash.io will be a platform for connecting existing Flask-monitoring dashboards to and monitor those dashboards as well. This document will describe the architecture and technologies used in the development of the PyDash.io platform. It will also provide a short overview of the API available.

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.

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.



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.

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

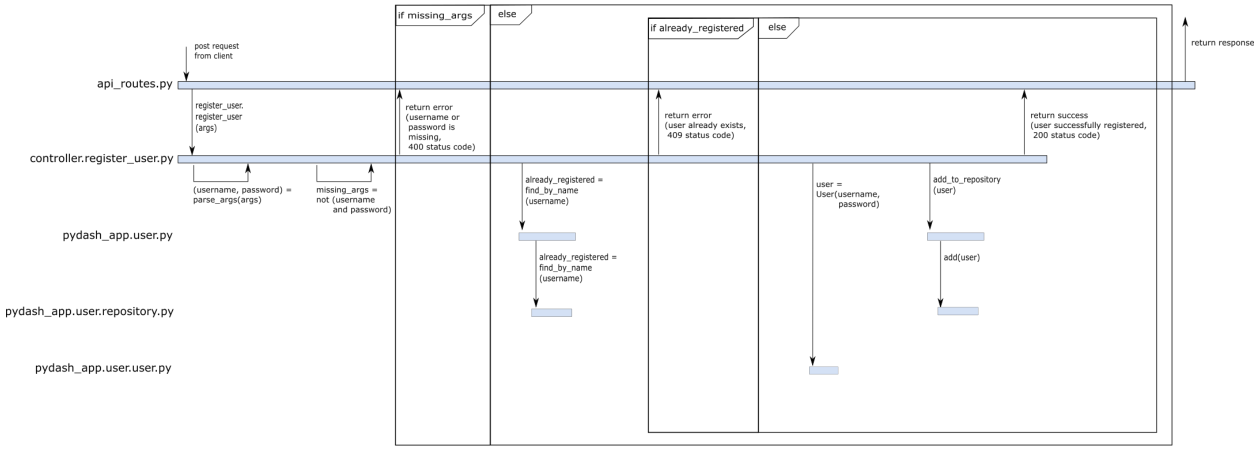
- An e-mail address is linked to the user

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.


if missing_args
else
if already_registered
else
post request from client
return response
api_routes.py
register_user.register_user (args)
return error (username or password is missing, 400 status code)
return error (user already exists, 409 status code)
return success (user successfully registered, 200 status code)
controller.register_user.py
(username, password) = parse_args(args)
missing_args = not (username and password)
already_registered = find_by_name (username)
user = User(username, password)
add_to_repository (user)
pydash_app.user.py
already_registered = find_by_name (username)
add(user)
pydash_app.user.repository.py
pydash_app.user.user.py

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

Preconditions

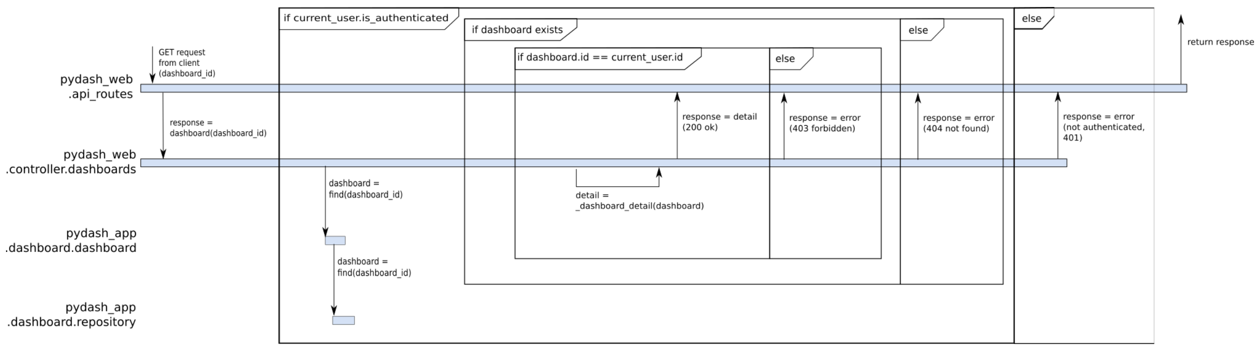
- The user is logged in
- The dashboard of which the data will be viewed is linked to the user

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


if current_user.is_authenticated
if dashboard exists
if dashboard.id == current_user.id
else
else
else
return response
GET request from client (dashboard_id)
pydash_web .api_routes
response = dashboard(dashboard_id)
pydash_web .controller.dashboards
response = detail (200 ok)
response = error (403 forbidden)
response = error (404 not found)
response = error (not authenticated, 401)
dashboard = find(dashboard_id)
detail = _dashboard_detail(dashboard)
pydash_app .dashboard.dashboard
dashboard = find(dashboard_id)
pydash_app .dashboard.repository

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.

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.

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:

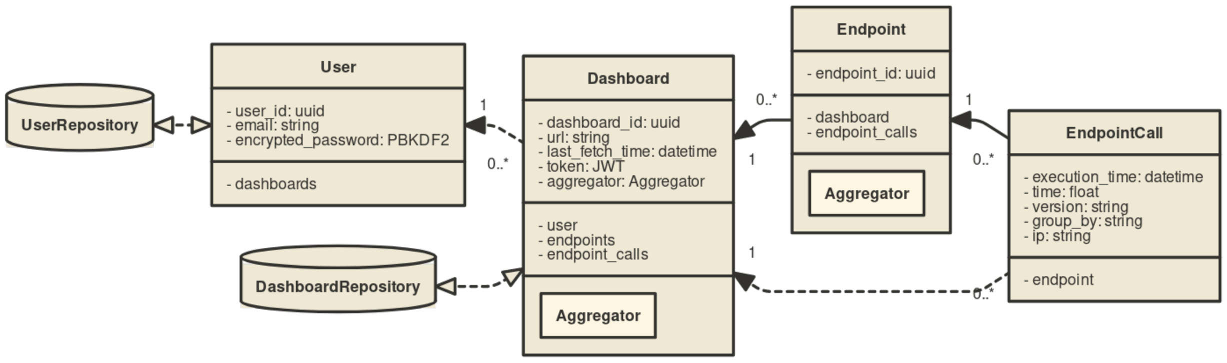
The Entity never knows how it itself is persisted,

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.

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.



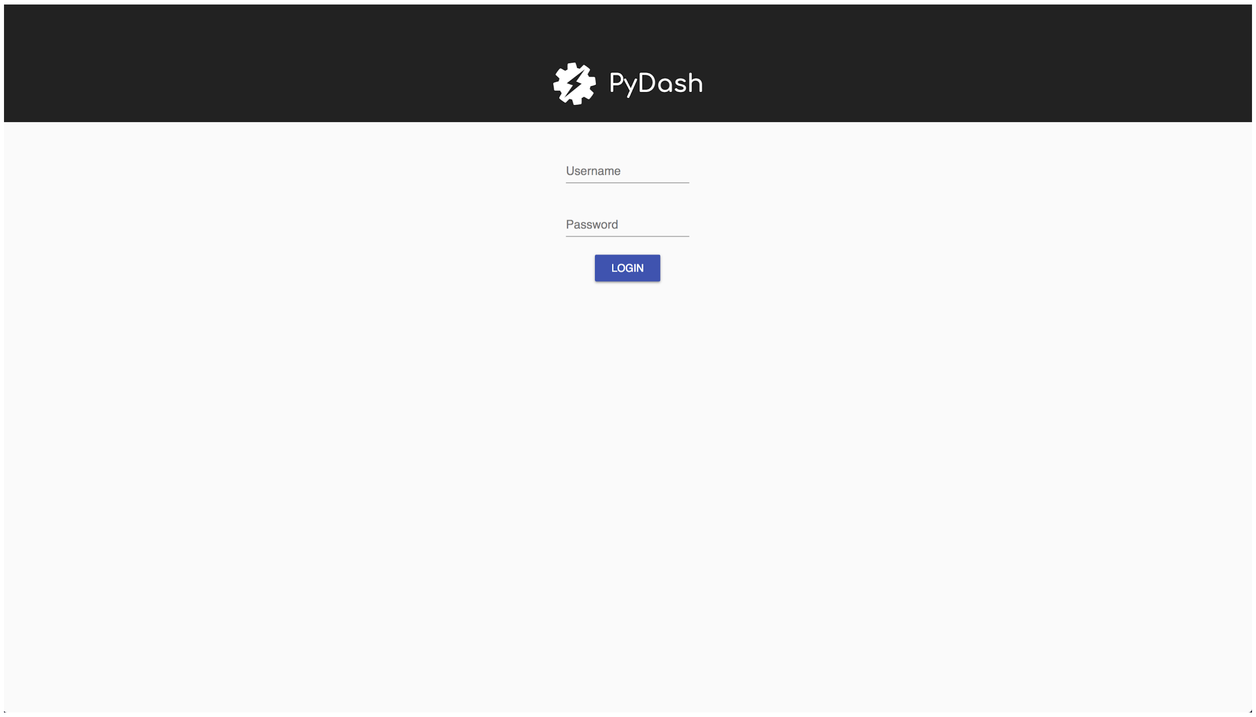
For the front-end part of the application, we will be creating a one-page application using the React.js framework, requesting all necessary information from the back-end using AJAX (Asynchonous Javascript and XML) calls.

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:

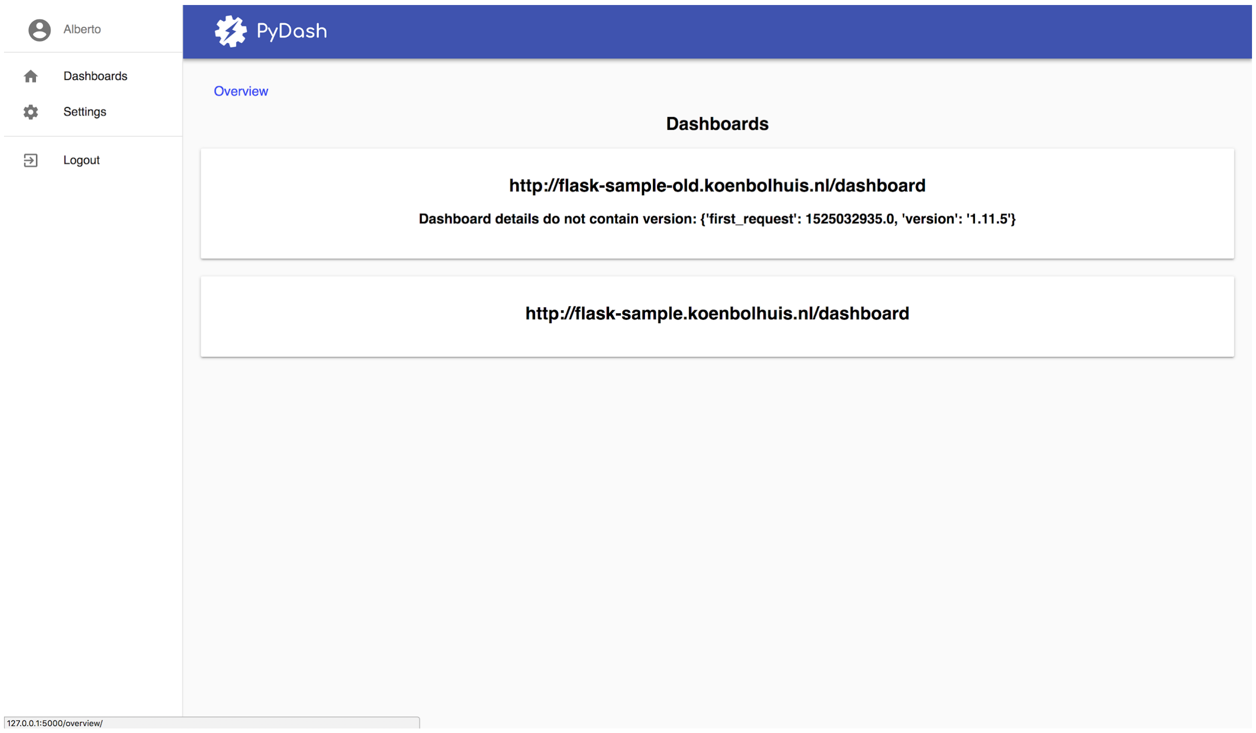
- The landing page/login page: When the users enter the website, this will be the first page they see. They will be able to login to an existing account.

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.



http://flask-sample-old.koenbolhuis.nl/dashboard

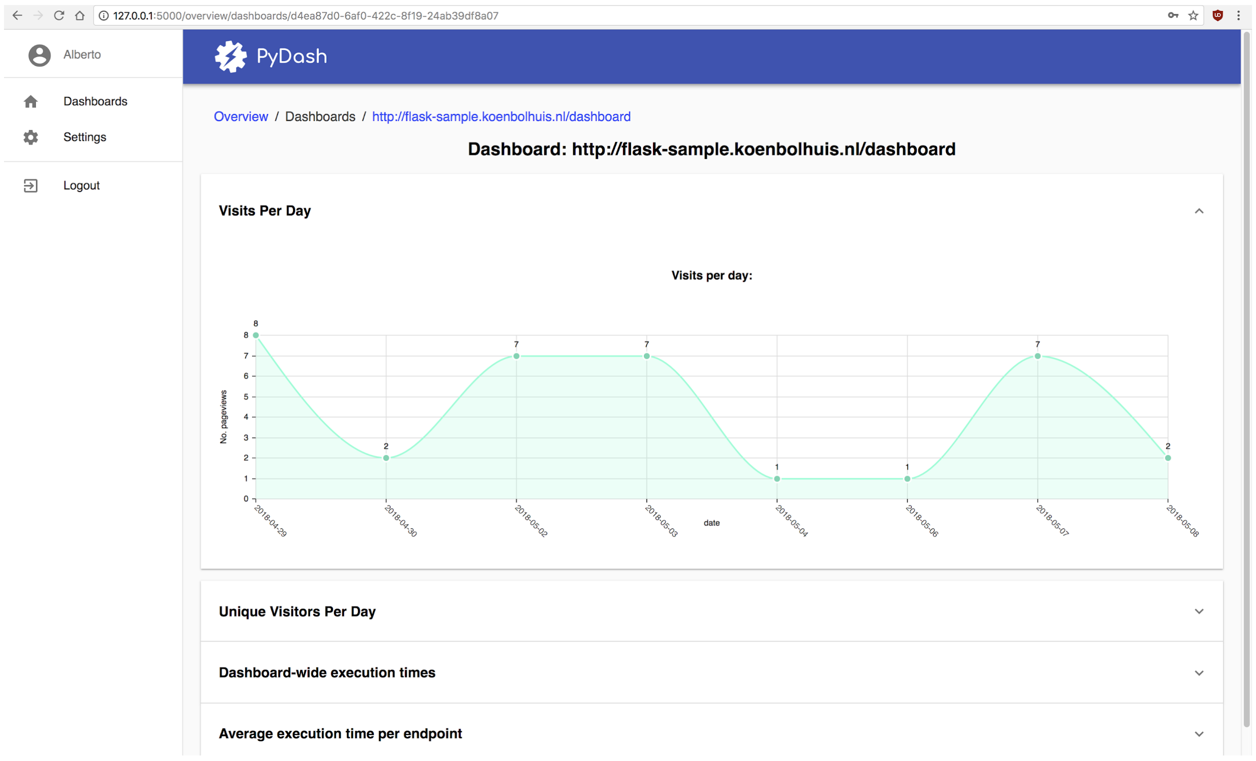
Dashboard details do not contain version: {'first_request': 1525032935.0, 'version': '1.11.5'}

http://flask-sample.koenbolhuis.nl/dashboard

127.0.0.1:5000/overview/

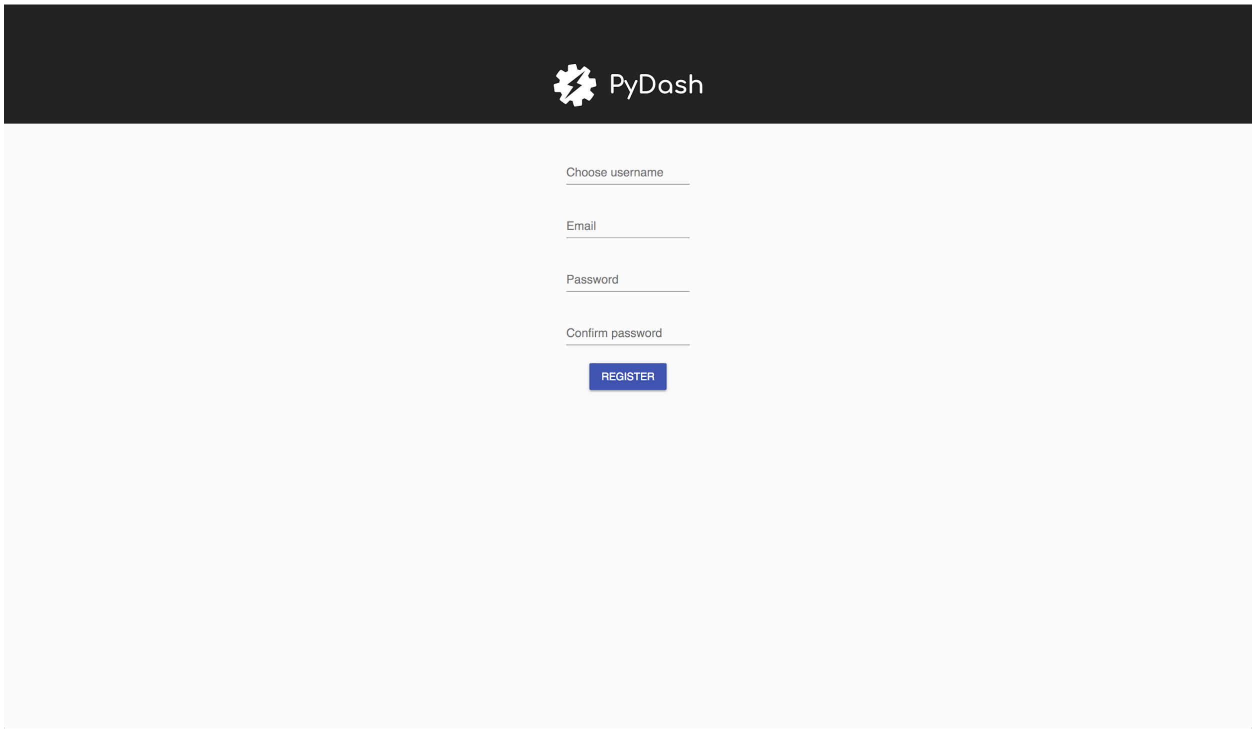
- The dashboard page (Here the users will be able to see all information coming in from the specified dashboard.

The dashboard page will look approximately like this.


127.0.0.1:5000/overview/dashboards/d4ea87d0-6af0-422c-8f19-24ab39df8a07
Alberto
Dashboards
Settings
Logout
PyDash
Overview / Dashboards / http://flask-sample.koenbolhuis.nl/dashboard
Dashboard: http://flask-sample.koenbolhuis.nl/dashboard
Visits Per Day
Visits per day:
No. pageviews
date
2018-04-29
2018-04-30
2018-05-02
2018-05-03
2018-05-04
2018-05-06
2018-05-07
2018-05-08
Unique Visitors Per Day
Dashboard-wide execution times
Average execution time per endpoint

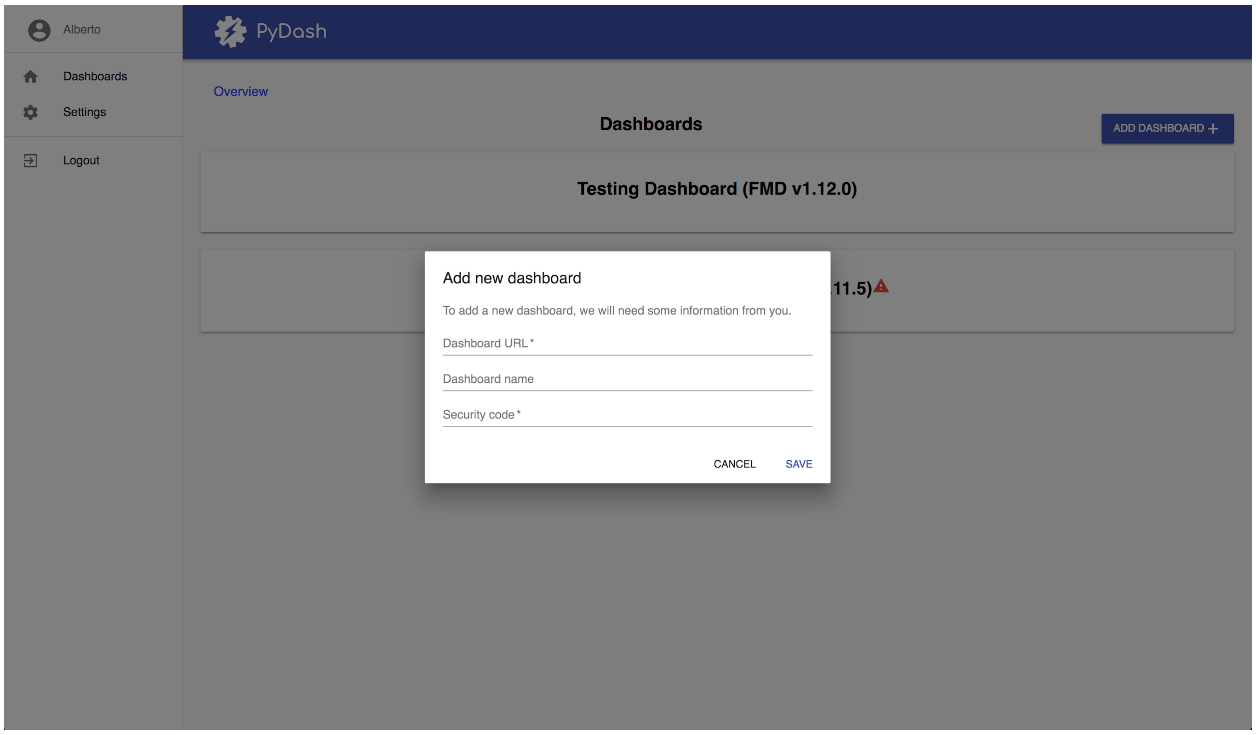
- The register page: Here the users will be able to create a new account

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.



Dashboards

Testing Dashboard (FMD v1.12.0)

11.5)

Add new dashboard

To add a new dashboard, we will need some information from you.

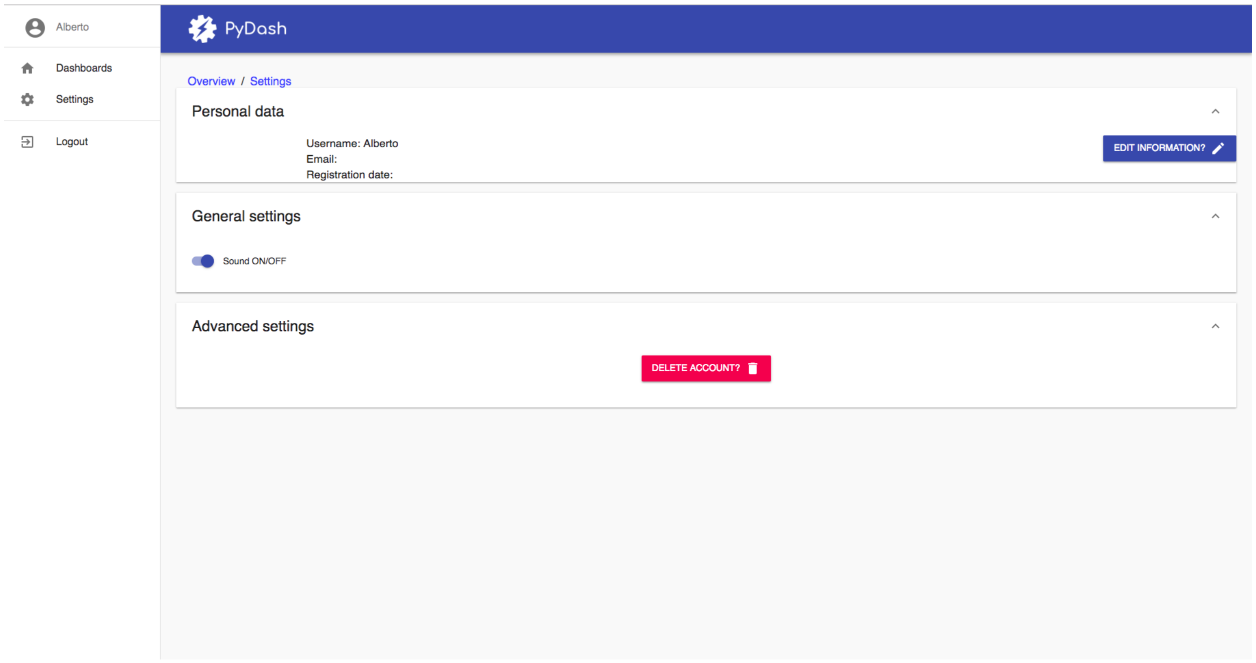
Dashboard URL *

Dashboard name

Security code *

- 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.



Personal data

Username: Alberto
Email:
Registration date:

General settings

Sound ON/OFF

Advanced settings

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).

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:

@ll@

[b]0.47

webpack

&

[b]0.47Webpack 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

[t]0.47

LiveReload

&

[t]0.47LiveReload 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

[t]0.47

ESLint

&

[t]0.47ESLint 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

[t]0.47

JSX

&

[t]0.47JSX 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

[t]0.47

babel

&

[t]0.47Babel 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

[t]0.47

Service Worker

&

[t]0.47Create-react-app automatically hooks up a service-worker for you, which allows offline-usage and asset caching for your app.

* Cache

assets

[t]0.47

github.com/axios

&

[t]0.47Axios 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.

[t]0.47

Material-UI

&

[t]0.47Material-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.

[t]0.47

NIVO

&

[t]0.47We 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.

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

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.

Logs the current user out.

No parameters required

Responses:

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

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.

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.

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.

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.

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.

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:

@l@

[t]0.97

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

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:

@l@

[t]0.97

```
{
  "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.

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:

@l@

[t]0.97

```
{

  "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.

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.

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.

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.

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

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

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.