PyDash.io

Architectural Design 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

VERSION 2018.04.17

Table of contents

Table of contents **1**

Introduction **2**

Architectural Overview **2**

Use-Cases **2**

Registering a user 3

Viewing the data of a certain dashboard 3

Back-end Design **3**

Domain Driven Design 4

Database Design 4

Front-end Design **5**

Back-End Technology Stack 7

Front-End Technology stack 8

API Specification **11**

Customer Contact **11**

Meeting Log **11**

Changelog **12**

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.

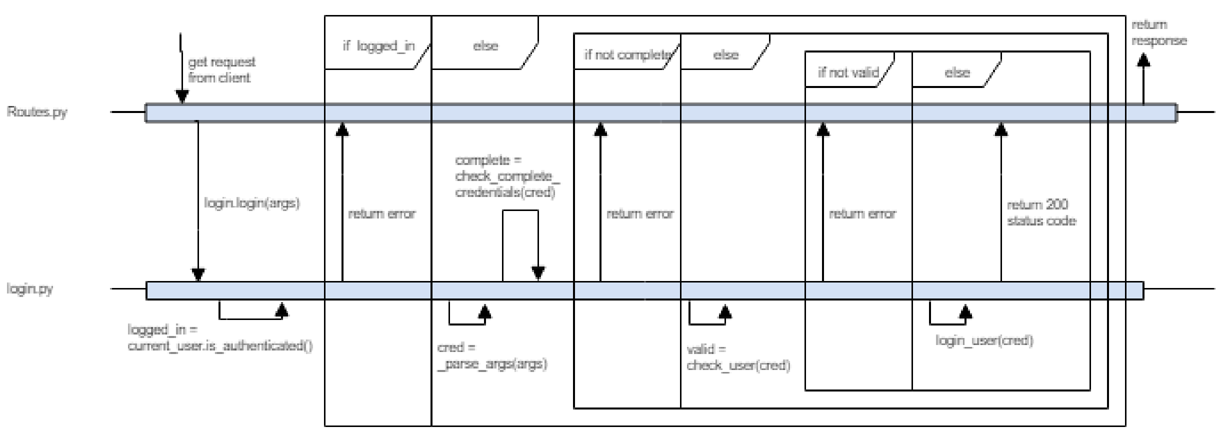
Architectural Overview

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 *(a future version of)* this document.

Use-Cases



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, e-mail)
3. The system processes the request
4. A verification e-mail is sent to the newly registered user
5. The user clicks the verification link in the e-mail
6. The user is notified of his successful account creation

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

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/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 *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:

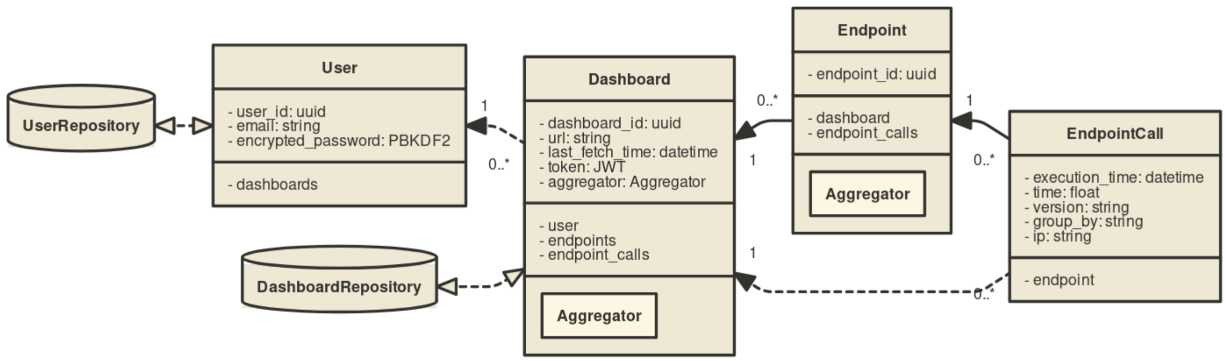
1. A module of the entity name that contains the publicly available functions to interact with these entities.
2. A *Repository* module that knows how entities of this type are persisted: It exposes functionality to find certain entities of this type in the persistence layer, create new ones, update 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 intended to wait as long as possible before committing to the choice of a specific persistence layer implementation, because it is one of the choices that has the largest (negative) impact on the flexibility of a software project.

We have now 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

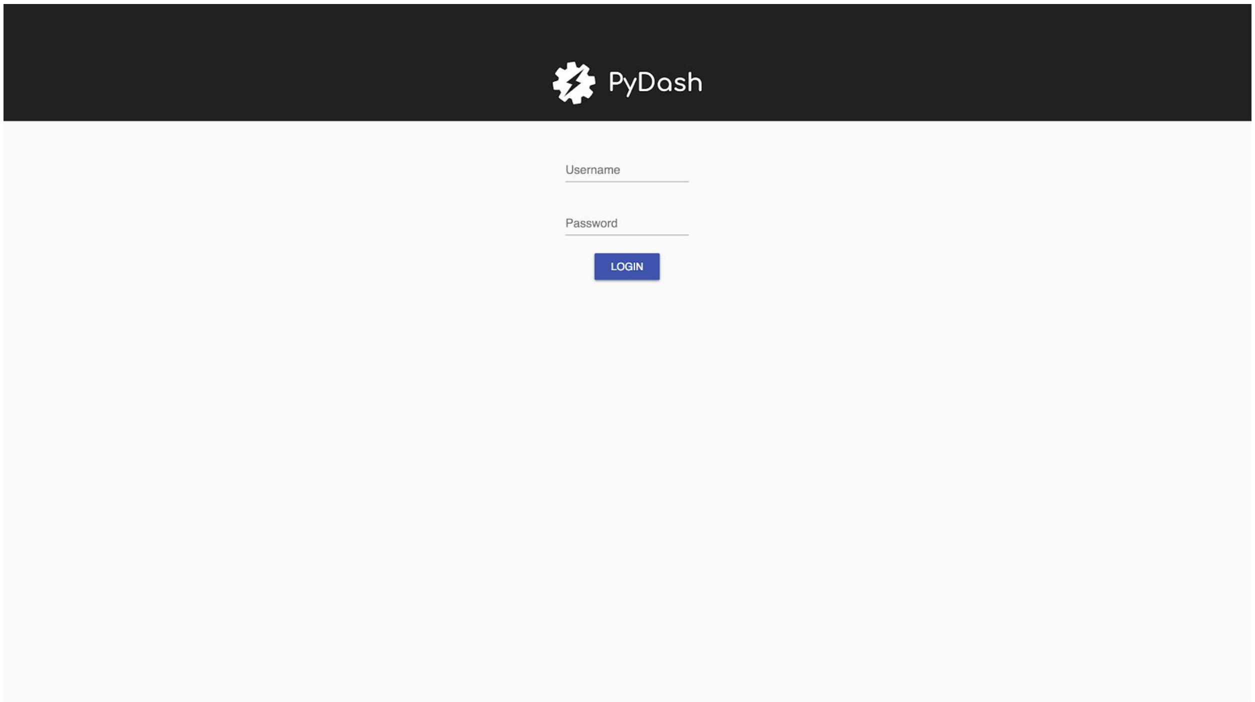
For the front-end part of the application, we will be creating a one-page application using the React.js framework, requesting all necessary information from the back-end using AJAX (Asynchonous Javascript and XML) calls.

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:

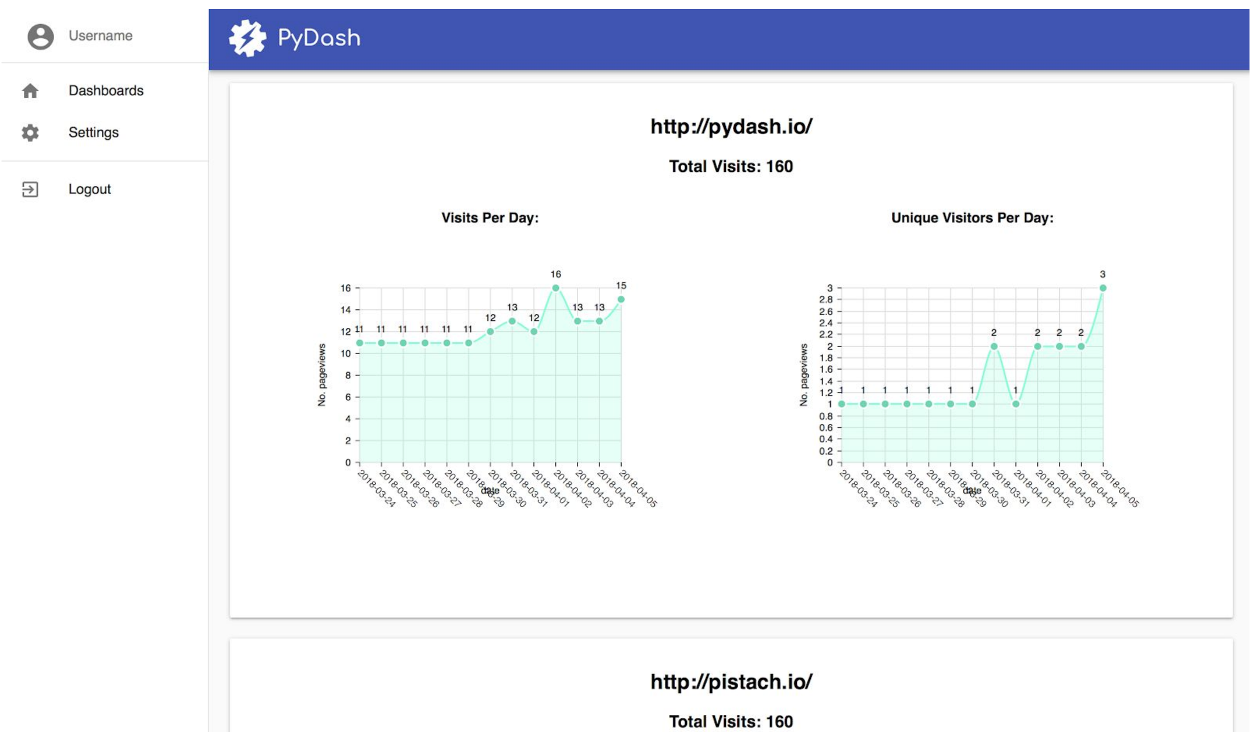
- The landing page/login page: When the users enter the website, this will be the first page they see. They will be able to login to an existing account.

The login page will look approximately like this.



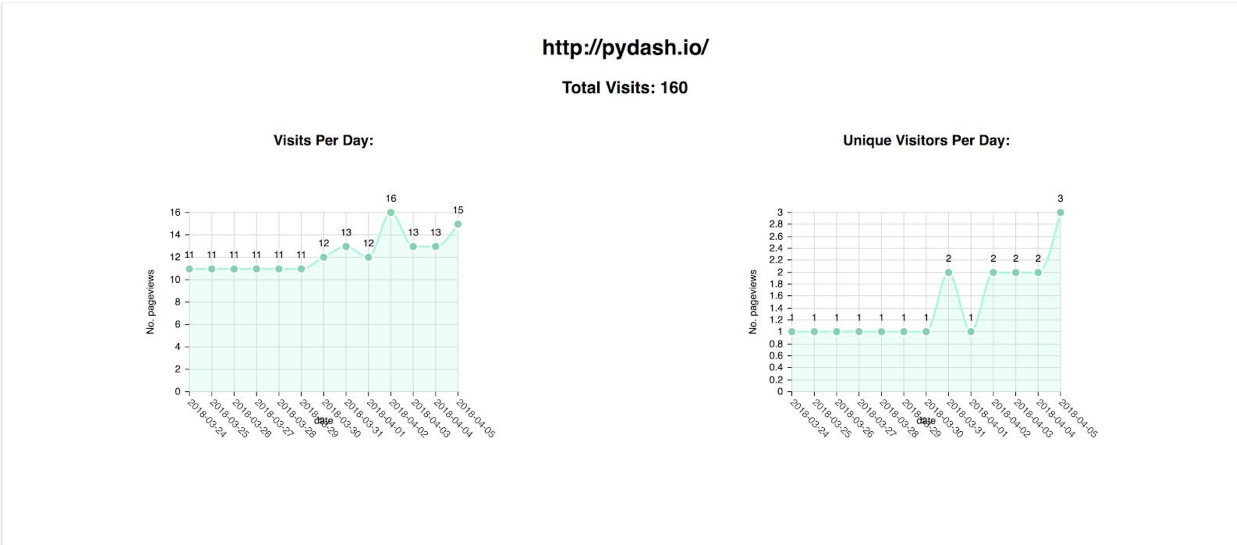
● The overview page: Here the users will be able to see an overview of all dashboards they are monitoring.

The overview page will look approximately like this.



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

The dashboard page will look approximately like this.



We will likely add more pages in the future, like “creating an account page” or “create a dashboard” page.

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).
    - **Flask-REST+** is used to make it easier to create a JSON-based REST API that the front-end can properly use.

    - **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 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

babel

Babel allows us to write next-generation javascript! Nowadays most javascript developers write in this new version of 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 a 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.

We use NIVO for displaying graphs in our statistics page. NIVO has a elaborate library of default graphs you

NIVO	can use. Customization is easy to do if wanted, NIVO's defaults are really good as well, though.

That's it for the frontend technology stack!

API Specification

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.

Changelog

Date	Iteration	Changes	Author
2018-03-08	First delivery.	Initial document	*Shared Effort*
2018-04-05	Sprint 3	Updating front-end with Patrick 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