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 2018.03.13

Table of 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 *(a future version of)* this document.

Technology Stack

Use-Cases

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 a couple of *template* files (in the *pydash_web/templates/* folder). This is temporary: most of the actual presenter logic will end up being done by the Front-End part of the application.

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 intend 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.

Currently, the data we need for the Minimum Viable Product as required by the first one or two sprints is simple enough to not require a complex persistence layer.

If required at a later stage, we are considering using an Object Database rather than a Relational Database, because this would forego the requirement of an Object-Relational-Mapper, as well as keeping the non-Python requirements of our application as light as possible.

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 css framework.

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 overview page: Here the users will be able to see an overview of all dashboards they are monitoring.
- The dashboard page (one for each dashboard): Here the users will be able to see all information coming in from the specified dashboard.

We will likely add more pages in the future.

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.	None. Initial document	*Shared Effort*