PyDash Requirements 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

For implementing web services using Python, one of the most used solutions is called Flask. This however does not include a way to monitor which parts of the web service are used at what rate. A dashboard able to collect this data was developed by a group of computing science students of the University of Groningen. There are still a few shortcomings in this dashboard, most notably in the case of monitoring a large number of Flask apps.

To solve this, PyDash.io can be used. PyDash.io is a dashboard containing the information of multiple deployed Flask monitoring dashboards. The user (with the right credentials) now has a better overview of his deployed dashboards, and does not have to maintain multiple credentials for each one.

Administrators of web services/API's. This users are a potential target as it will be very interested to make possible that other companies, apart from RUG, make use of the system. It would be useful for them to be able to measure with the analytic system the number of hits in their website. The system can be used as a global measurement or more specific. This will also benefit companies as they will be able to measure a certain part of the website and view what customers are interested when entering in their pages. So, this will mean that they will be able to correct errors of their websites or even improve and make emphasis on what their customers like.

Developers. As well as for the Administrators of web services/API's, the Developers will be able to take advantages of the system to develop a dashboard. The dashboard will offer the developers to measure the effectiveness of their websites. Developers will be able to create a tool that will help them to improve their websites. So, as well as for Administrators, the system will help Developers to correct errors but also, to develop a effective system for their dashboards.

- Users need to be able to create an account.
- Users need to be able to log in to their account.
- Users need to be able to register flask-monitoring-dashboards.
- Users need to be able to see an overview of all their registered dashboards.
- The PyDash service needs to receive data from each registered dashboard in regular intervals, for example, on an hourly basis.
- For each registered dashboard: Functionality for users to change which endpoints are being monitored.
- For each registered dashboard: visualising (the performance of) said dashboard (similar to how a dashboard is currently visualised):
  - Overview of dashboard (all included endpoints):
    - Graph of summed endpoint executions (per hour)
    - A set of horizontal bar-plots of how many times Endpoints are executed per day
    - Execution time - all endpoints and versions:
      - Per version:
        Show a horizontal box-plot of the execution times (over all Endpoints) per version
      - Per endpoint:
        Show a horizontal box-plot of the execution times per Endpoint (over all versions)
- There should also be endpoint specific pages that show more detailed information:
  - Endpoint summary/overview:
    - Endpoint name
    - Added since app version (version nr.)
    - Date added to app (date & time)
    - Link to endpoint
    - Last accessed (date & time)
  - Execution time:
    - Per day:

            - Show a stacked-vertical-bar-plot with the average execution time per day.
        - Per user, per version:
          Show a dot-plot for the average execution time per user per version.
        - Per IP-address, per version:
          Show a dot-plot for the average execution time per IP-address per version.
        - Per version:
          Show a box-plot for the execution time per version.
        - Per user:
            - Show a box-plot for the execution time per user.
    - Hits per day:
        - Vertical-bar-plot:
          Show a vertical-bar-plot with the number of hits per day.
        - Heatmap:
          Show a heatmap (per hour) when the specific Endpoint was executed.
- For each registered dashboard visualising (the performance of) said dashboard:
    - Endpoint specific:
        - Debug info about potential outliers:
            - Date (date & time)
            - Execution time (in ms)
            - Request values
            - Request headers
            - Request environment
            - Request url
            - Cpu percent
            - Memory
            - Stacktrace
- A number of static tutorial pages, detailing how to use the flask-monitoring-dashboard.
- An administrator overview for PyDash.io, where an administrator can see visualizations on the usage of PyDash.io itself.
    - In order to do this, we deploy a flask-monitoring-dashboard on PyDash.io itself.
- Allow users to change scope of monitored endpoints in case there are many. e.g. hiding or showing endpoints or groups of endpoints.
- In the future, we might want to support other types of dashboards, such as, for example, a django-monitoring-dashboard.
- Allow users to enable/disable the actual monitoring of specific endpoints. ***Since the flask-monitoring-dashboard API does not yet provide any way to achieve this, we will not yet be able to implement this feature.***
- Add more visualizations.
    - We will most likely not add more visualizations than *flask-monitoring-dashboard* currently includes.

- Security + Privacy
  - We want to use an Secure Socket Layer connection, both to connect the users to the application, as well as to connect to the different instances of the *flask-monitoring-dashboard*. The reasoning behind this, is that when we do not do this, sensitive information might be shared or stolen with third parties.
  - We want to hash the user-account passwords in the application, to ensure that in the unfortunate event that PyDash.IO's information might be compromised, that the user accounts are relatively safe.
  - We want to prevent the resources used in the application to be susceptible to an *enumeration* attack: That is, authorization to access a certain resource will be checked before returning it, and inside URLs, identifiers that are non-incrementing integers will be used. This will stop people from both getting insight into the data we have stored by looking at the identifiers (for example how many dashboards we have) and using the URL syntax to access new endpoints they would otherwise not know about.
- Tooling
  - Usage of the programming language Python and the micro web framework Flask is required to build the PyDash.IO application. The reason behind this is to be able to use the *flask-monitoring-dashboard* to monitor the PyDash.IO web application itself at some point.
- Responsive Design: We want the application to function properly (do not have lacking functionality) and not contain obvious graphical bugs or oversights. We are going to test this by accessing our web tool on different systems. We want our tool to work on at least the following display sizes:
  - A Full-HD screen (1920 x 1080)px
  - A medium-sized computer or tablet screen (1280 x 768)px
  - A small portrait-mode smartphone screen (320 x 550)px
- Scalability
  - In the hopes that PyDash.IO will be used by a reasonable group of users at some point, we want to ensure that the application can handle at least fifty *flask-monitoring-dashboard* instances side-by-side without the application being hampered in its primary purpose, which is to have users browse the distilled information already extracted from those *flask-monitoring-dashboards*. An endpoint response is too slow for a user if it takes longer than 0.5 seconds to respond, measured in server execution time.
- Etiquette of performing external API calls
  - When calling into an external *flask-monitoring-dashboard*, it is mandatory to not stress (overload) that application too much. This means that we need to rate-limit the calls made to these external applications (Do not perform a request to the same application more often than once every three minutes), as well as ensure that we do not request too much data at one time from them.

Backend

- Fetching fixed
  - Due to concurrency issues and Wiebe-Marten not being available to help out, we did not manage to get periodic background fetching to work. For this reason, currently dashboard data is fetched everytime the dashboard page is loaded.
  - By Sprint #4, periodic fetching needs to be working.

- Better error handling when fetching
  - While fetching works at the moment, error handling needs to be improved since there are some edge cases where an error is raised and the application crashes.
- Account creation
  - An API route needs to be added to support account creation; this will put the new user in a "waiting for confirmation" state and generate a verification code.
  - An API route needs to be added for email verification based on a given code.
  - Sending emails, and the frontend UI, will be done in Sprint #5.
- Continuous Integration (CI) and deployment
  - Set up testing and continuous integration of the backend application.
  - Deploy the entire application to the server.
  - Arrange TLS/SSL.

Frontend

- Account creation
  - A register account button has to be added to the current login page which directs the user to the register account page.
  - The register account page has to be made, containing fields for name, e-mail, password and verify password which when all properly filled in will create the new account.
- Add more graphs
  - Visualizations have to be added to the individual dashboard pages and the individual endpoint pages, containing information about among others number of endpoint visits and last access time. In this sprint we will aim for at least three different graphs per dashboard page, namely a graph for visits per day, unique visitors per day, and endpoint popularity.
- Revise the overview page
  - The overview currently contains a graph and a name per dashboard, this has to be changed to containing just the name and a simple statistic such as total number of visits in the last week. Each tile is then a link to the individual page of said dashboard.
- Implement basis for adding and removing dashboards from the overview page
  - As this is a lot of work for just sprint 5, the basis can be built here. Think of adding the necessary buttons or designing the verification of the url and secret token.
- Implement endpoint specific page
  - Not only is it possible to view the data of a dashboard, there is also a page with statistics for every endpoint of that dashboard, accessed through the dashboard overview page. This endpoint page has to be created and the proper links have to be added to the dashboard pages. The design for the endpoint page can be found in the design folder of the google drive.
  - On this page multiple cards must be displayed. The top one when opened shows general data about the endpoint, and the other cards all consist of graphs.

Backend

- Make error messages more user friendly

  - Right now, the error messages are simply Python exceptions: we need to make this more user friendly.
- Implement email verification
  - The meat of the functionality is already in place (generating a code and the API route to verify it). We still need to send actual emails.
- Implement adding and removing dashboards
  - For example, add new routes to add and remove a dashboard.
  - Alternatively, use the PUT and DELETE HTTP methods to indicate intention.
- Implement user settings
  - Define user settings
  - Routes to retrieve, update and delete user settings need to be added.
  - Alternatively, this could be done using HTTP verbs on existing routes.
- Revise aggregator issue #105.3

Frontend

- Implement adding and removing dashboards from the overview page.
  - Addition will be done by clicking on a plus in the top right corner of the overview page. When clicked the user will be able to fill in the dashboard url, its name and the secret token associated with that dashboard. If correctly filled in, the dashboard will be added as a tile to the users overview page.
  - Removing is done by clicking on a three dot sign present in the bottom left corner of every dashboard page. When clicked, the user will be asked to confirm his action after which the dashboard is permanently removed from the overview page.
- Implement user settings
  - When clicking on the tab Settings, currently already present in the menu, the user will be redirected to the Settings page
  - The settings page has to be made, containing settings such as changing your password, the email account associated with the user, or account deletion.
- Add more graphs
  - As we will probably not be able to implement all the different visualisations we want during sprint 4, the rest or another part of them will be added during sprint 5. This sprint will be more focused on the individual endpoint page.
- Revise individual dashboard page
  - After adding functionalities as more visualisations and filters, we have to make sure the individual dashboard page still works as it should and presents its data in a intuitive and user friendly way. Maybe show summary in card header, or a separate card with a summary of all endpoints.
- Create individual endpoint page
  - The same as for the dashboard page goes for the individual endpoint pages.

Backend

- Extend user settings

  - Add the settings not added in the previous sprint, which exactly will be clear after sprint 5.
- Implement dashboard settings
  - Name
  - Token
  - URL
  - Deletion
- Update email verification
  - Removing old verification codes+users.
  - Regenerate verification codes on request.
- Heatmap API call

Frontend

- Add heatmap
- Implement endpoint monitoring filter(s)
  - Add a textbox that filters graphs
- Extend user settings
  - See backend task.
- Pages or text boxes for email verification

Backend

- Meta-monitoring
  - We need to have *flask-monitoring-dashboard* monitor PyDash itself.
  - In principle this is as simple as:
    * following the *flask-monitoring-dashboard* install instructions;
    * adding it to the PyDash project;
    * and connecting this dashboard to a PyDash user account.

Frontend

- Fill graphs with data.
- Add endpoint graphs
- Update front-end documentation
- Update front-end documentation
- Ensure user interface is responsive (works on all screen sizes)

Other

- Finalize documents
- Extract application documentation and add to documents, where possible.

This is a list of features that we would like to add, but ended up having too little time during the project time to do.

- Uptime checking

  - Ping the remote dashboard to see if it's still up.
  - Add some kind of status in the Dashboard object in the database.
  - Do this every five minutes.
- Switch Database (ZODB is not as nice as we'd hoped)
- Set up our own SMTP server, in order to not be dependant on Google's SMTP server with its limit of 100 free emails per day. This will also eliminate a possible point of attack on our email sending capabilities, which our user registration and user verification functionalities depend on.

Other

- Make video tutorial on PyDash
  - Explain to new users how the site works in a tutorial containing easy-to-follow steps and pictures for clarification.
- Design new logo.

We are set to use the API from the *flask-monitoring-dashboard*. We are going to use three endpoints, which are described as follows:

1. get_json_details

Returns, in JSON format, some information about the *flask-monitoring-dashboard* deployment. At the moment, the version is returned as well as the timestamp of the first request.

1. get_json_data/<time_from>/<time_to>

Returns, encoded using JSON Web Token with the dashboard's security token, a list of objects containing endpoint access data of the *flask-monitoring-dashboard*. Optionally, *time_from* and *time_to* UNIX timestamps can be provided to get all data since and up to this timestamp, respectively. The objects returned are structured as follows:

- endpoint <string>

The name of the endpoint.

- execution_time <float>

  The execution time of the endpoint in milliseconds.

- time <timestamp>

  When the endpoint has been triggered, returns a string containing the time (something like: '%Y-%m-%d %H:%M:%S.%f').

- version <string>

Version of the website.

- group_by <string>

Which user requested the endpoint.

- ip <string>

IP address of the client making the request.

1. get_json_monitor_rules

Returns, encoded using JSON Web Token with the dashboard's security token, a list of objects describing *monitoring rules* for all endpoints on the website. The objects are structured as follows:

- endpoint <string>

Name of the endpoint, corresponds to the endpoint above.

- last_accessed <timestamp>

When this endpoint was last accessed.

- monitor <boolean>

Describes whether the endpoint is monitored or not.

- time_added <timestamp>

Time the endpoint was added to the web service.

- version_added <string>

  Describes the version of the webservice at the time this endpoint was added. This is a git commit hash.

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.