Flask-MonitoringDashboard Documentation

Release 1.13.0

Patrick Vogel, Thijs Klooster
Bogdan Petre

May 11, 2018

Contents

Automatically monitor the evolving performance of Flask/Python web services.

CHAPTER 1

What is the Flask-MonitoringDashboard?

The Flask Monitoring Dashboard is designed to easily monitor your existing Flask application. You can find a brief overview of the functionality *here*.

Or you can watch the video below:

CHAPTER 2

Functionality

The Flask Monitoring Dashboard is an extension that offers 4 main functionalities with little effort from the Flask developer:

- **Monitor the Flask application:** The Dashboard allows you to see which endpoints process a lot of request and how fast. Additionally, it provides information about the evolving performance of an endpoint throughout different versions if you're using git.
- **Monitor your test coverage:** The Dashboard allows you to find out which endpoints are covered by unit tests, allowing also for integration with Travis for automation purposes. For more information, see this file.
- **Collect extra information about outliers:** Outliers are requests that take much longer to process than regular requests. The Dashboard automatically detects that a request is an outlier and stores extra information about it (stack trace, request values, Request headers, Request environment).
- **Visualize the collected data in a number useful graphs:** The Dashboard is automatically added to your existing Flask application. You can view the results by default using the default endpoint (this can be configured to another route): dashboard.

For a more advanced documentation, take a look at the information on this page.

CHAPTER 3

User's Guide

If you are interested in the Flask Monitoring Dashboard, you can find more information in the links below:

3.1 Installation

This page provides an overview of installing the Flask Monitoring Dashboard. It starts from the very basic, but it is likely that you can directly go to *Installing the Flask Monitoring Dashboard Package*.

3.1.1 Install Python

You can check if you have Python installed by opening a terminal and execution the following command:

```
python --version
```

It should return something like `Python 3.6.3`, if not, you probably see something like `bash: python3: command not found`. In the former case, you're ok. In the latter, you can follow this link to install Python.

3.1.2 Installing a Virtual Environment (Optional)

Although you don't need a Virtual Environment, it is highly recommend. See this page to install a Virtual Environment.

Configuring the Virtual Environment (Optional)

If you have skipped the previous section, you can also skip this one (since it's optional). Once you've installed the Virtual Environment, you need to configure it. This can be done by the following command:

```
virtualenv ENV
```

Or using the following command for Python3:

```
virtualenv --python=python3 ENV
```

Activate the Virtual Environment (Optional)

This is the last part of the configuring the virtual environment. You should do this before you want to execute any python script/program. It is (again) one simple command:

```
source bin/activate
```

3.1.3 Installing the Flask Monitoring Dashboard Package

You can install the Flask Monitoring Dashboard using the command below:

```
pip install flask_monitoringdashboard
```

Alternatively, you can install the Flask Monitoring Dashboard from Github:

```
git clone https://github.com/flask-dashboard/Flask-MonitoringDashboard.git
cd Flask-MonitoringDashboard
python setup.py install
```

3.1.4 Setup the Flask Monitoring Dashboard

After you've successfully installed the package, you can use it in your code. Suppose that you've already a Flask application that looks like this:

```
from flask import Flask
app = Flask(__name__)

...

@app.route('/')
def index():
    return 'Hello World!'


if __name__ == '__main__':
  app.run(debug=True)
```

You can add the extension to your Flask application with only two lines of code:

```
...
import flask_monitoringdashboard as dashboard
dashboard.bind(app)
```

Together, it becomes:

```
from flask import Flask
import flask_monitoringdashboard as dashboard

app = Flask(__name__)
dashboard.bind(app)

...

@app.route('/')
def index():
    return 'Hello World!'

if __name__ == '__main__':
  app.run(debug=True)
```

3.1.5 Further configuration

You are now ready for using the Flask Monitoring Dashboard, and you can already view the Dashboard at: dashboard.

However, the Dashboard offers many functionality which has to be configured. This is explained on the configuration page.

3.2 Configuration

Once you have successfully installed the Flask Monitoring Dashboard with information from this page, you can use the advanced features by correctly configuring the Dashboard.

3.2.1 Using a configuration file

You can use a single configuration file for all options below. This is explained in the following section. In order to configure the Dashboard with a configuration-file, you can use the following function:

```
dashboard.config.init_from(file='/<path to file>/config.cfg')
```

Thus, it becomes:

```
from flask import Flask
import flask_monitoringdashboard as dashboard
```

(continues on next page)

(continued from previous page)

```
app = Flask(__name__)
dashboard.config.init_from(file='/<path to file>/config.cfg')
# Make sure that you first configure the dashboard, before binding it to your␣
↪Flask application
dashboard.bind(app)
...

@app.route('/')
def index():
    return 'Hello World!'

if __name__ == '__main__':
  app.run(debug=True)
```

Instead of having a hard-coded string containing the location of the config file in the code above, it is also possible to define an environment variable that specifies the location of this config file. The line should then be:

```
dashboard.config.init_from(envvar='DASHBOARD_CONFIG')
```

This will configure the Dashboard based on the file provided in the environment variable called *DASHBOARD_CONFIG*.

3.2.2 The content of the configuration file

Once the setup is complete, a configuration file (e.g. 'config.cfg') should be set next to the python file that contains the entry point of the app. The following things can be configured:

```
[dashboard]
APP_VERSION=1.0
CUSTOM_LINK=dashboard
DATABASE=sqlite:////<path to your project>/dashboard.db
DEFAULT_MONITOR=True
TIMEZONE='Europe/Amsterdam'
USERNAME=admin
PASSWORD=admin
GUEST_USERNAME=guest
GUEST_PASSWORD=['dashboardguest!', 'second_pw!']
GIT=/<path to your project>/.git/
OUTLIER_DETECTION_CONSTANT=2.5
DASHBOARD_ENABLED=True
TEST_DIR=/<path to your project>/tests/
COLORS={'main':'[0,97,255]',
        'static':'[255,153,0]'}
```

This might look a bit overwhelming, but the following list explains everything in detail:

- **APP_VERSION:** The version of the application that you use. Updating the version helps in showing differences in execution times of a function over a period of time.

- **CUSTOM_LINK:** The Dashboard can be visited at localhost:5000/{{CUSTOM_LINK}}.
- **DATABASE:** Suppose you have multiple projects where you're working on and want to separate the results. Then you can specify different database_names, such that the result of each project is stored in its own database.
- **DEFAULT_MONITOR:** When this configuration is set to True, new endpoints are automatically monitored by the Dashboard.
- **TIMEZONE:** The timezone for converting a UTC timestamp to a local timestamp. For a list of all timezones, use the following:

  ```
  import pytz  # pip install pytz
  print(pytz.all_timezones)
  ```

  The dashboard saves the time of every request by default in a UTC-timestamp. However, if you want to display it in a local timestamp, you need this property.
- **USERNAME** and **PASSWORD:** Must be used for logging into the Dashboard. Thus both are required.
- **GUEST_USERNAME** and **GUEST_PASSWORD:** A guest can only see the results, but cannot configure/download any data.
- **GIT:** Since updating the version in the configuration-file when updating code isn't very useful, it is a better idea to provide the location of the git-folder. From the git-folder, The version is automatically retrieved by reading the commit-id (hashed value). The location is relative to the configuration-file.
- **OUTLIER_DETECTION_CONSTANT:** When the execution time is more than this $constant * average$, extra information is logged into the database. A default value for this variable is 2.5.
- **OUTLIERS_ENABLED:** Whether you want to collect information about outliers. If you set this to true, the expected overhead of the Dashboard is a bit larger, as you can find here.
- **TEST_DIR:** Specifies where the unit tests reside. This will show up in the configuration in the Dashboard.
- **COLORS:** The endpoints are automatically hashed into a color. However, if you want to specify a different color for an endpoint, you can set this variable. It must be a dictionary with the endpoint-name as a key, and a list of length 3 with the RGB-values. For example:

  ```
  COLORS={'main':'[0,97,255]',
          'static':'[255,153,0]'}
  ```

3.2.3 What have you configured?

A lot of configuration options, but you might wonder what functionality is now supported in your Flask application? Have a look at this file to find the answer.

3.3 Detailed Functionality

The functionality of the Dashboard is divided into 4 main components. You can find detailed information about every component below:

3.3.1 Endpoint Monitoring

The core functionality of the Dashboard is monitoring which Endpoints are heavily used and which are not. If you have successfully configured the Dashboard from this page, then you are ready to use it. In order to monitor a number of endpoints, you have to do the following:

1. Log into the Dashboard at: http://localhost:5000/dashboard/login
2. Go to the Rules-tab in the left menu: http://localhost:5000/dashboard/rules
3. Select the rules that you want to monitor.
4. Wait until a request to this endpoint is being made.
5. Go to the Dashboard Overview in the left menu: http://localhost:5000/measurements/overview

Collected data

For each request that is being to a monitored endpoint, the following data is recorded:

- **Execution time:** measured in ms.
- **Time:** the current timestamp of when the request is being made.
- **Version:** the version of the Flask-application. This can either be retrieved via the *CUSTOM_VERSION* value, or via the *GIT* value. If both are configured, the *GIT* value is used.
- **group_by:** An option to group the collected results. As most Flask applications have some kind of user management, this variable can be used to track the performance between different users. It is configured using the following command:

```
def get_user_id():
    return 1234  # replace with a function to retrieve the id of the
                 # user within a request.

dashboard.config.group_by = get_user_id
# Note that the function itself is passed, not the result of the␣
↪function.
```

  Thus, it becomes:

```
from flask import Flask
import flask_monitoringdashboard as dashboard

app = Flask(__name__)
dashboard.config.init_from(file='/<path to file>/config.cfg')
```

(continues on next page)

(continued from previous page)

```python

def get_user_id():
    return '1234'  # replace with a function to retrieve the id of the
                   # user within a request.

dashboard.config.group_by = get_user_id
dashboard.bind(app)

@app.route('/')
def index():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(debug=True)
```

The group_by-function must be a function that either returns a primitive (bool, bytes, float, int, str), or a function, or a tuple/list. Below is a list with a few valid examples:

Code	Result
dashboard.config.group_by = lambda: 3	3
dashboard.config.group_by = lambda: (‘User’, 3)	(User,3)
dashboard.config.group_by = lambda: lambda: 3	3
dashboard.config.group_by = (‘User’, lambda: 3)	(User,3)
dashboard.config.group_by = lambda: ‘username’	username
dashboard.config.group_by = lambda: [‘Username’, ‘username’]	(Username,username)
dashboard.config.group_by = lambda: [(‘User’, lambda: 3), (‘Username’, lambda: ‘username’)]	((User,3),(Username,username))

- **IP:** The IP-address from which the request is made. The IP is retrieved by the following code:

```python
from flask import request
print(request.environ['REMOTE_ADDR'])
```

Observations

Using the collected data, a number of observations can be made:

- Is there a difference in execution time between different versions of the application?
- Is there a difference in execution time between different users of the application?
- Is there a difference in execution time between different IP addresses? *As tracking the performance between different users requires more configuration, this can be a quick alternative.*
- On which moments of the day does the Flask application process the most requests?
- What are the users that produce the most requests?
- Do users experience different execution times in different version of the application?

3.3.2 Test-Coverage Monitoring

To enable Travis to run your unit tests and send the results to the Dashboard, two steps have to be taken:

1. The installation requirement for the Dashboard has to be added to the *setup.py* file of your app:

```
dependency_links=["https://github.com/flask-dashboard/Flask-
↪MonitoringDashboard/tarball/master#egg=flask_
↪monitoringdashboard"]

install_requires=('flask_monitoringdashboard')
```

2. In your *.travis.yml* file, one script command should be added:

```
python -m flask_monitoringdashboard.collect_performance \
--test_folder=./tests \
--times=5 \
--url=https://yourdomain.org/dashboard
```

   The *test_folder* argument specifies where the performance collection process can find the unit tests to use. The *times* argument (optional, default: 5) specifies how many times to run each of the unit tests. The *url* argument (optional) specifies where the Dashboard is that needs to receive the performance results. When the last argument is omitted, the performance testing will run, but without publishing the results.

3.3.3 Outliers

It is useful to investigate why certain requests take way longer to process than other requests. If this is the case, a request is seen as an outlier. Mathematically an outlier is determined if the execution of the request is longer than:

$> average * constant$

Where *average* is the average execution time per endpoint, and *constant* is given in the configuration by OUTLIER_DETECTION_CONSTANT (its default value is 2.5).

When a request is an outlier, the Dashboard stores more information, such as:

- The stack-trace in which it got stuck.
- The percentage of the CPU's that are in use.
- The current amount of memory that is used.
- Request values.
- Request headers.
- Request environment.

The data that is collected from outliers, can be seen by the following procedure:

1. Go to the Dashboard Overview: http://localhost:5000/measurements/overview

2. Click on the Details-button (on the right side) for which endpoint you want to see the Outlier information.
3. Go to the Outliers-tab: http://localhost:5000/dashboard/<endpoint-name>/main/outliers

3.3.4 Visualizations

There are a number of visualization generated to view the results that have been collected in (Endpoint-Monitoring) and (Test-Coverage Monitoring).

The main difference is that visualizations from (Endpoint-Monitoring) can be found in the menu 'Dashboard' (in the left menu), while visualizations from (Test-Coverage Monitoring) can be found in the menu 'Test Monitor' (below the 'Dashboard'-menu).

The 'Dashboard'-menu contains the following content:

1. **Overview:** A table with the all the endpoints that are being monitored (or have been monitored in the past). This table provides information about when the endpoint is last being requested, how often it is requested and what the median execution time is. Furthermore, it has a 'Details' button on the right. This is explained further in (6).
2. **Hourly load:** This graph provides information for each hour of the day of how often the endpoint is being requested. In this graph it is possible to detect popular hours during the day.
3. **Version Usage**: This graph provides information about the distribution of the utilization of the requests per version. That is, how often (in percentages) is a certain endpoint requested in a certain version.
4. **Requests per endpoint:** This graph provides a row of information per day. In this graph, you can find whether the total number of requests grows over days.
5. **Time per endpoint:** This graph provides a row of information per endpoint. In that row, you can find all the requests for that endpoint. This provides information whether certain endpoints perform better (in terms of execution time) than other endpoints.
6. For each endpoint, there is a 'Details'-button (alternatively, you can click on the row itself). This provides the following information (thus, all information below is specific for a single endpoint):
   - **Hourly load:** The same hourly load as explained in (2), but this time it is focused on the data of that particular endpoint only.
   - **Time per version per user:** A circle plot with the average execution time per user per version. Thus, this graph consists of 3 dimensions (execution time, users, versions). A larger circle represents a higher execution time.
   - **Time per version per ip:** The same type of plot as 'Time per version per user', but now that users are replaced by IP-addresses.
   - **Time per version:** A horizontal box plot with the execution times for a specific version. This graph is equivalent to (4.), but now it is focused on the data of that particular endpoint only.
   - **Time per user:** A horizontal box plot with the execution time per user. In this graph, it is possible to detect if there is a difference in the execution time between users.

- **Outliers:** See Section (Outliers) above.

3.3.5 Need more information?

See the contact page to see how you can contribute on the project. Furthermore you can request this page for questions, bugs, or other information.

CHAPTER 4

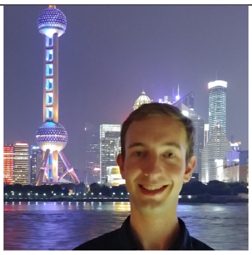
Developer information

4.1 Contact

This page provides information about how to ask a question, or post an issue.

4.1.1 Developing-team

Currently, the team consists of three active developers:

	**Patrick Vogel** Core developer **E-mail:** patrickvogel@live.nl.
	Thijs Klooster Test Monitor Specialist
	Bogdan Petre Outlier Specialist

4.1.2 Found a bug?

Post an issue on Github. You can use the template below for the right formatting:

Issue Template

- Expected Behavior

  Tell us what should happen

- Current Behavior

  Tell us what happens instead of the expected behavior

- Possible Solution

  Not obligatory, but suggest a fix/reason for the bug

- Steps to Reproduce

  Provide a link to a live example, or an unambiguous set of steps to reproduce this bug. Include code to reproduce, if relevant

  1.
  2.
  3.
  4.

- Context (Environment)

  How has this issue affected you? What are you trying to accomplish? Providing context helps us come up with a solution that is most useful in the real world

- Detailed Description

  Provide a detailed description of the change or addition you are proposing

- Possible Implementation

  Not obligatory, but suggest an idea for implementing addition or change

4.2 Developing

This page provides information about contributing to the Flask Monitoring Dashboard. Furthermore, a number of useful tools for improving the quality of the code are discussed.

4.2.1 Implementation

The Dashboard is implemented in the following 6 folders: core, database, static, templates, test and views. Together this forms a Model-View-Controller-pattern:

- **Model**: The model consists of the database-code. To be more specific, it is defined in 'Flask-MonitoringDashboard/database.__init__.py'.
- **View**: The view is a combination of the following three folders:
  - **static**: contains some CSS and JS files.
  - **templates**: contains the HTML files for rendering the Dashboard. The HTML files are rendered using the *Jinja2 templating language*. Jinja2 allows a HTML-template to inherit from another HTML- template. The hierarchy of all templates is:

```
fmd_base.html
├──fmd_dashboard/overview.html
```

(continues on next page)

(continued from previous page)

```
│    └──fmd_dashboard/graph.html
│       └──fmd_dashboard/graph-details.html
│          └──fmd_dashboard/outliers.html
├──fmd_testmonitor/testmonitor.html
├──fmd_testmonitor/testresult.html
├──fmd_config.html
├──fmd_login.html
└──fmd_urles.html
fmd_export-data.html
```

  * **fmd_base.html**: For rendering the container of the Dashboard, and load all required CSS and JS scripts.
  * **fmd_config.html**: For rendering the Configuration-page.
  * **fmd_login.html**: For rendering the Login-page.
  * **fmd_urles.html**: For rendering the Rules-page.
  * **fmd_dashboard/overview.html**: For rendering the Overview-page.
  * **fmd_dashboard/graph.html**: For rendering the following graphs:
    · Hourly load
    · Version Usage
    · Requests per endpoint
    · Time per endpoint
  * **fmd_dashboard/graph-details.html**: For rendering the following graphs:
    · Hourly load
    · Time per version per user
    · Time per version per ip
    · Time per version
    · Time per user
  * **fmd_dashboard/outliers.html**: For rendering the Outlier-page.
  * **fmd_testmonitor/testmonitor.html**: For rendering the Testmonitor-page.
  * **fmd_testmonitor/testresult.html**: For rendering the results of the Testmonitor.

– **views**: Contains all Flask route-functions that the Dashboard defines.

• **Controller**: The Controller contains all Dashboard Logic. It is defined in the **core**-folder.

4.2.2 Tools

The following tools are used for helping the development of the Dashboard:

- **Branches**: The Dashboard uses the following branches:
  - **Master**: This is the branch that will ensure a working version of the Dashboard. It is shown as the default branch on Github. The Master branch will approximately be updated every week. Every push to the master will be combined with a new version that is released in PyPi. This branch is also used to compute the Code coverage and build the documentation. In case of a PR from development into master, take care of the following two things:
    1. The version must be updated in flask_monitoringdashboard/constants.json
    2. The changelog should be updated in docs/changelog.rst
  - **Development**: This branch contains the current working version of the Dashboard. This branch contains the most recent version of the Dashboard, but there might be a few bugs in this version.
  - **Feature branch**: This branch is specific per feature, and will be removed after the corresponding feature has been merged into the development branch. It is recommended to often merge development into this branch, to keep the feature branch up to date.
- **Unit testing**: The code is tested before a Pull Request is accepted. If you want to run the unit tests locally, you can use the following command:

  Use this command while being in the root of the Flask-MonitoringDashboard folder.

  ```
  python setup.py test
  ```

- **Travis**: Travis CI is a hosted, distributed continuous integration service used to build and test software projects hosted at GitHub. The Dashboard uses Travis to ensure that all tests are passed before a change in the code reaches the Master branch.
- **Documentation**: The documentation is generated using Sphinx and hosted on ReadTheDocs. If you want to build the documentation locally, you can use the following commands:

  Use the commands while being in the docs folder (Flask-MonitoringDashboard/docs).

  ```
  pip install -r requirements.txt
  make html
  ```

  The generated html files can be found in the following folder: Flask-MonitoringDashboard/docs/build.

  Using the make command, you can build more, than only HTML-files. For a list of all possible options, use the following command:

  ```
  make help
  ```

4.2.3 Versions

The Dashboard uses Semantic-versioning. Therefore, it is specified in a **Major** . **Minor** . **Patch** -format:

- **Major**: Increased when the Dashboard contains incompatible API changes with the previous version.
- **Minor**: Increased when the Dashboard has new functionality in a backwards-compatible manner.
- **Patch**: Increased when a bug-fix is made.

4.3 TODO List

All things that can improved in Flask Monitoring Dashboard are listed below.

4.3.1 Features to be implemented

- Combine multiple queries into a single query. This will improve the performance of multiple pages.

4.3.2 Work in progress

- Improving the outlier functionality
- Improving the Test monitor

4.4 Change Log

All notable changes to this project will be documented in this file. This project adheres to Semantic Versioning. Please note that the changes before version 1.10.0 have not been documented.

4.4.1 v1.13.0

Changed

- Added boxplot of CPU loads
- Updated naming scheme of all graphs
- Implemented two configuration options: the local timezone and the option to automatically monitor new endpoints
- Updated the Test-Monitoring initialization
- Updated Database support for MySQL

4.4.2 v1.12.0

Changed

- Removed two graphs: hits per hour and execution time per hour
- New template design
- Refactored backhand of the code
- Updated Bootstrap 3.0 to 4.0
- Setup of Code coverage

4.4.3 v1.11.0

Changed

- Added new graph: Version usage
- Added column (Hits in past 7 days) in Measurements Overview
- Fixed bug with configuration
- Changed rows and column in outlier-table
- Added TODO List
- Updated functionality to retrieve the stacktrace of an Outlier
- Fixed bug with white colors from the config option

4.4.4 v1.10.0

Changed

- Added security for automatic endpoint-data retrieval.
- Added test for export_data-endpoints
- Added MIT License.
- Added documentation