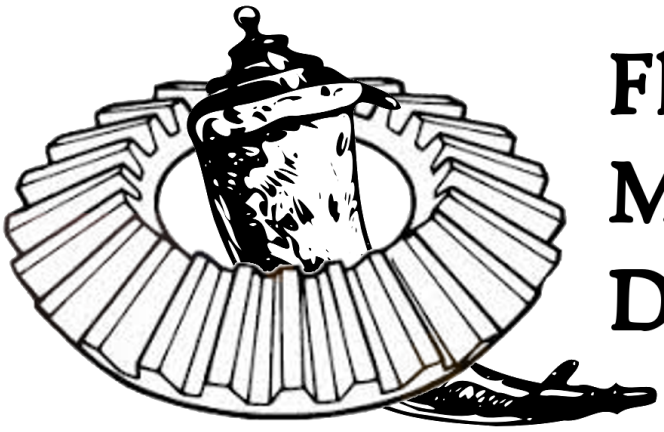
Flask-MonitoringDashboard Documentation

Release 3.1.1

Patrick Vogel, Bogdan Petre

May 31, 2022

Contents



Flask-Monitoring Dashboard

Automatically monitor the evolving performance of Flask/Python web services

The Flask Monitoring Dashboard is designed to easily monitor your Flask application.

CHAPTER 1

Functionality

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

- **Monitor the performance and utilization:** The Dashboard allows you to see which endpoints process a lot of requests and how fast. Additionally, it provides information about the evolving performance of an endpoint throughout different versions if you're using git.
- **Profile requests and endpoints:** The execution path of every request is tracked and stored into the database. This allows you to gain insight over which functions in your code take the most time to execute. Since all requests for an endpoint are also merged together, the Dashboard provides an overview of which functions are used in which endpoint.
- **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).
- **Collect additional information about your Flask-application:** Suppose you have an User-table and you want to know how many users are registered on your Flask-application. Then, you can run the following query: 'SELECT Count(*) FROM USERS;'. But this is just annoying to do regularly. Therefore, you can configure this in the Flask-MonitoringDashboard, which will provide you this information per day (or other time interval).

For more advanced documentation, have a look at the the detailed functionality page.

CHAPTER 2

User's Guide

If you are interested in the Flask-MonitoringDashboard, you can find more information in the links below:

2.1 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*.

2.1.1 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.

2.1.2 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 ENV/bin/activate
```

2.1.3 Installing the Flask-MonitoringDashboard Package

You can install the Flask-MonitoringDashboard using the command below:

```
pip install flask_monitoringdashboard
```

Alternatively, you can install the Flask-MonitoringDashboard from Github:

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

2.1.4 Setup the Flask-MonitoringDashboard

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)
```

2.1.5 Further configuration

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

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

2.2 Configuration

Once you have successfully installed the Flask-MonitoringDashboard using the instructions from the installation page, you can use the advanced features by correctly configuring the Dashboard.

2.2.1 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')
```

Your app then 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='FLASK_MONITORING_DASHBOARD_CONFIG')
```

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

2.2.2 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 properties can be configured:

```
[dashboard]
APP_VERSION=1.0
GIT=/<path to your project>/.git/
CUSTOM_LINK=dashboard
MONITOR_LEVEL=3
OUTLIER_DETECTION_CONSTANT=2.5
SAMPLING_PERIOD=20
ENABLE_LOGGING=True
BRAND_NAME=Flask Monitoring Dashboard
TITLE_NAME=Flask-MonitoringDashboard
DESCRIPTION=Automatically monitor the evolving performance of Flask/Python␣
↪web services
SHOW_LOGIN_BANNER=True
SHOW_LOGIN_FOOTER=True

[authentication]
USERNAME=admin
PASSWORD=admin
SECURITY_TOKEN=cc83733cb0af8b884ff6577086b87909

[database]
```

(continues on next page)

(continued from previous page)

```
TABLE_PREFIX=fmd
DATABASE=sqlite:////<path to your project>/dashboard.db

[visualization]
TIMEZONE=Europe/Amsterdam
COLORS={'main':'[0,97,255]',
        'static':'[255,153,0]'}
```

As can be seen above, the configuration is split into 4 headers:

Dashboard

- **APP_VERSION:** The version of the application that you use. Updating the version allows seeing the changes in the execution time of requests over multiple versions.
- **GIT:** Since updating the version in the configuration-file when updating code isn't very convenient, another way is 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 specified value is the location to the git-folder. This is relative to the configuration-file.
- **CUSTOM_LINK:** The Dashboard can be visited at localhost:5000/{{CUSTOM_LINK}}.
- **MONITOR_LEVEL**: The level for monitoring your endpoints. The default value is 3. For more information, see the Overview page of the Dashboard.
- **OUTLIER_DETECTION_CONSTANT:** When the execution time is greater than $constant * average$, extra information is logged into the database. A default value for this variable is 2.5.
- **SAMPLING_PERIOD:** Time between two profiler-samples. The time must be specified in ms. If this value is not set, the profiler monitors continuously.
- **ENABLE_LOGGING:** Boolean if you want additional logs to be printed to the console. Default value is False.
- **BRAND_NAME:** The name displayed in the Dashboard Navbar. Default value is 'Flask Monitoring Dashboard'.
- **TITLE_NAME:** The name displayed in the browser tab. Default value is 'Flask-MonitoringDashboard'.
- **DESCRIPTION:** The text displayed in center of the Dashboard Navbar. Default value is 'Automatically monitor the evolving performance of Flask/Python web services'.
- **SHOW_LOGIN_BANNER:** Boolean if you want the login page to show the 'Flask Monitoring Dashboard' logo and title. Default value is True.
- **SHOW_LOGIN_FOOTER:** Boolean if you want the login page to show a link to the official documentation. Default value is True.

Authentication

- **USERNAME** and **PASSWORD:** Must be used for logging into the Dashboard. Both are required.
- **SECURITY_TOKEN:** The token that is used for exporting the data to other services. If you leave this unchanged, any service is able to retrieve the data from the database.

Database

- **TABLE_PREFIX:** A prefix to every table that the Flask-MonitoringDashboard uses, to ensure that there are no conflicts with the other tables, that are specified by the user of the dashboard.
- **DATABASE:** Suppose you have multiple projects that 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.

Visualization

- **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.

- **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]'}
```

2.2.3 What have you configured?

We've shown a bunch of configuration settings, but what features are now supported in your Flask application and how are they related to the config options? Have a look at the detailed functionality page to find the answer.

2.3 Detailed Functionality

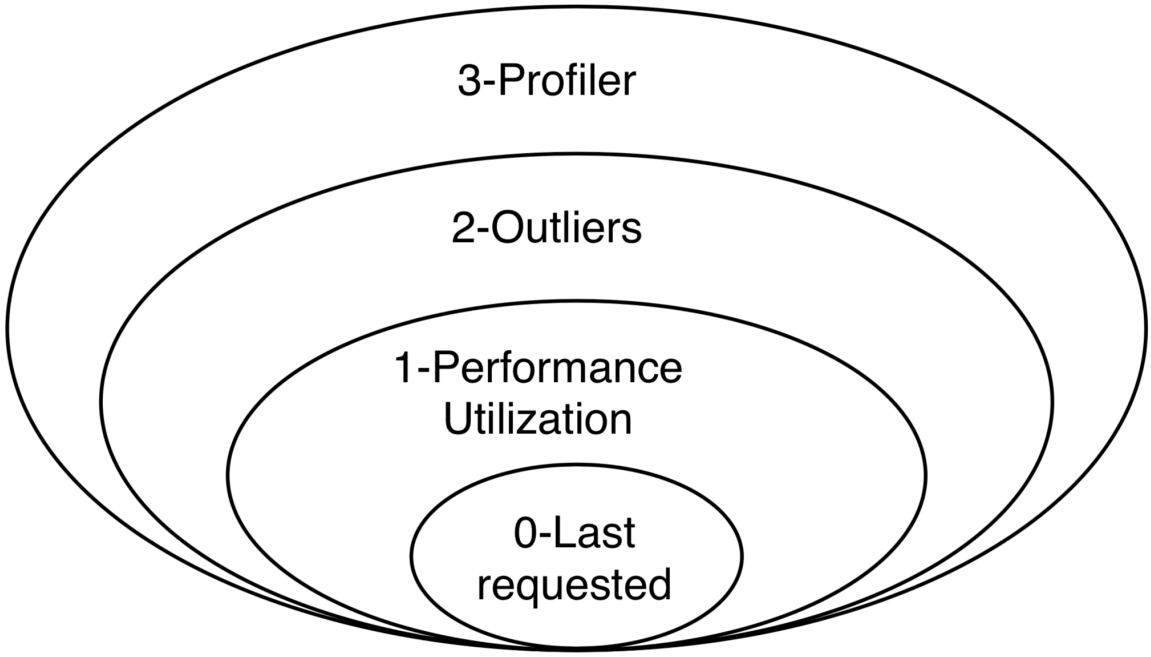
The functionality of the Dashboard is divided into two main components: data collection and data visualization. You can find detailed information about both components below.

2.3.1 1. Data Collection

The amount of data collected by the Dashboard varies for each endpoint of the monitored Flask application, depending on the monitoring level selected. To select the monitoring level of your endpoints, you have to do the following (assuming you have successfully configured the Dashboard as described in the configuration page):

1. Log into the Dashboard at: http://localhost:5000/dashboard/login
2. Go to the Overview tab in the left menu: http://localhost:5000/dashboard/overview
3. Select the endpoints that you want to monitor.
4. Select the desired monitoring level.

A summary of the monitoring levels is provided next. Note that every level keeps all the features of the level below, in addition to bringing its own new features, as represented in the diagram below.



Monitoring Level 0 - Disabled

When the monitoring level is set to 0, the Dashboard does not monitor anything about the performance of the endpoint. The only data that is stored is when the endpoint is last requested.

Monitoring Level 1 - Performance and Utilization Monitoring

When the monitoring level is set to 1, the Dashboard collects performance (as in response time) and utilization information for every request coming to that endpoint. The following data is recorded:

- **Duration:** the duration of processing that request.

- **Time_requested:** the timestamp of when the request is being made.
- **Version_requested:** the version of the Flask-application at the moment when the request arrived. This can either be retrieved via the *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')

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:

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

Monitoring Level 2 - Outliers

When the monitoring level is set to 2, the Dashboard collects extra information about slow requests.

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, a request is considered an outlier if its execution is a certain number of times longer than the average duration for requests coming to the same endpoint:

$duration_outlier > duration_average * constant$

Where $duration_average$ 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 about it, 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 the endpoint for which you want to see the Outlier information.
3. Go to the Outliers tab: http://localhost:5000/dashboard/endpoint/:endpoint_id:/outliers

Monitoring Level 3 - Profiler

When the monitoring level is set to 3, the Dashboard performs a statistical profiling of all the requests coming to that endpoint. What this means is that another thread will be launched in parallel with the one processing the request, it will periodically sample the processing thread, and will analyze its current stack trace. Using this information, the Dashboard will infer how long every function call inside the endpoint code takes to execute.

The profiler is one of the most powerful features of the Dashboard, pointing to where your optimization efforts should be directed, one level of abstraction lower than the performance monitoring of Level 1. To access this information, you have to:

1. Go to the Overview tab in the left menu: http://localhost:5000/dashboard/overview
2. Select an endpoint for which the monitoring level is or was at some point at least 2.

3. Go to the Profiler tab: http://localhost:5000/dashboard/endpoint/:endpoint_id:/profiler
4. Go to the Grouped Profiler tab: http://localhost:5000/dashboard/endpoint/:endpoint_id:/grouped-profiler

The Profiler tab shows all individual profiled requests of an endpoint in the form of a execution tree. Each code line is displayed along with its execution time and its share of the total execution time of the request.

The Grouped Profiler tab shows the merged execution of up to 100 most recent profiled requests of an endpoint. This is displayed both as a table and as a Sunburst graph. The table shows for each code line information about the Hits (i.e. how many times it has been executed), average execution time and standard deviation, and also total execution time.

2.3.2 2. Data Visualization

The Dashboard shows the collected data by means of two levels of abstraction: application-wide and endpoint-specific.

Application

Visualizations showing aggregated data of all the endpoints (with monitoring level at least 1) in the application can be found under the **Dashboard** menu:

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 was last requested, how often it is requested and what is the current monitoring level. Each endpoint can be clicked to access the Endpoint-specific visualizations.
2. **Hourly API Utilization:** 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. **Multi Version API Utilization**: 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. **Daily API Utilization:** 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. **API Performance:** 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. **Reporting:** A more experimental feature which aims to automatically detect and report changes in performance for various intervals (e.g. today vs. yesterday, this week vs. last week, etc).

Endpoint

For each endpoint in the Overview page, you can click on the endpoint to get more details. This provides the following information (all information below is specific for a single endpoint):

1. **Hourly API Utilization:** The same hourly load as explained in (2) above, but this time it is focused on the data of that particular endpoint only.
2. **User-Focused Multi-Version Performance:** 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.
3. **IP-Focused Multi-Version Performance:** The same type of plot as 'User-Focused Multi-Version Performance', but now that users are replaced by IP-addresses.
4. **Per-Version Performance:** 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.
5. **Per-User Performance:** 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.
6. **Profiler:** A tree with the execution path for all requests.
7. **Grouped Profiler:** A tree with the combined execution paths for all (<100) requests of this endpoint.
8. **Outliers:** The extra information collected on outlier requests.

2.3.3 Make it your own!

Just as no two applications are the same, we understand that monitoring requirements differ for every use case. While all the above visualizations are included by default in the FMD and answer a wide range of questions posed by the typical web application developer, you can also create your own custom visualizations tailored to your needs.

You might wish to know how the number of unique users, the size of your database, or the total number of endpoints have evolved over time. This is now easy to visualize using FMD.

An example of a custom graph is shown below. FMD will execute `on_the_minute()` every minute at the second 01 and the graph will appear in the **Custom graphs** menu.

```
def on_the_minute():
    print(f"On the minute: {datetime.datetime.now()}")
    return int(random() * 100 // 10)


minute_schedule = {'second': 00}

dashboard.add_graph("On Half Minute", on_the_minute, "cron",␣
↪**minute_schedule)
```

Note the "cron" argument to the add graph. Just like in the case of the unix cron utility you can use more complex schedules. For example, if you want to collect the data every day at midnight you would use:

```
midnight_schedule = {'month':"*",
                     'day': "*",
                     'hour': 23,
                     'minute': 59,
                     'second': 00}
```

Besides cron, there's also the "interval" schedule type, which is exemplified in the following snippet:

```
def every_ten_seconds():
    print(f"every_ten_seconds!!! {datetime.datetime.now()}")
    return int(random() * 100 // 10)


every_ten_seconds_schedule = {'seconds': 10}

dashboard.add_graph("Every 10 Seconds", every_ten_seconds, "interval
↪", **every_ten_seconds_schedule)
```

Note that not all fields in the `schedule` dictionary are required, only the non-zero / non-star ones.

Also, note that in the "cron" graph types you use singular names (e.g. second) while in the "interval" you use plurals (e.g. seconds).

Finally, the implementation of the scheduler in the FMD is based on the appscheduler.schedulers.Background schedulers about which you can read more in the corresponding documentation page.

2.3.4 Need more information?

Check out the contact page to see how you can get in touch with us.

2.4 Known Issues

This page provides an overview of known bugs, workarounds, and limitations of the Flask Monitoring Dashboard.

2.4.1 Deploying with mod_wsgi (Apache)

The FMD relies on the `scipy` package for some of the statistical tests used in the *Reports* feature. This package is incompatible with `mod_wsgi` by default, causing the deployment to fail. This is a common issue [1] [2] and can be solved by setting the

```
WSGIApplicationGroup %{GLOBAL}
```

directive in your WSGI configuration file, as described in the linked posts.

CHAPTER 3

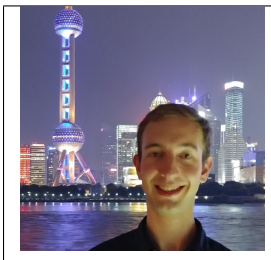
Developer information

3.1 Contact

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

3.1.1 Developing-team

Currently, the team consists of four active developers:

	**Patrick Vogel** Project Leader **E-mail:** patrickvogel@live.nl.
	Bogdan Petre Core Developer
	**Thijs Klooster** Test Monitor Specialist
	Amedeo Bussi Mongodb expert

3.1.2 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

3.2 Developing

Fixing a bug, implementing a new feature, or just improving the quality of the code, we always appreciate contributions! We understand that getting accustomed to a new project takes some time and effort, but we're trying to make this process as smooth and intuitive as possible.

Whereas until now, we've discussed FMD from the point of view of the user, this page shows FMD from the point of view of the developer. We explain the architecture of the project, take a look at the main components, and then present some useful tools that we use during development.

3.2.1 Getting started

In order to get started with the environment of the FMD, run the following script

```
. ./config/install.sh
```

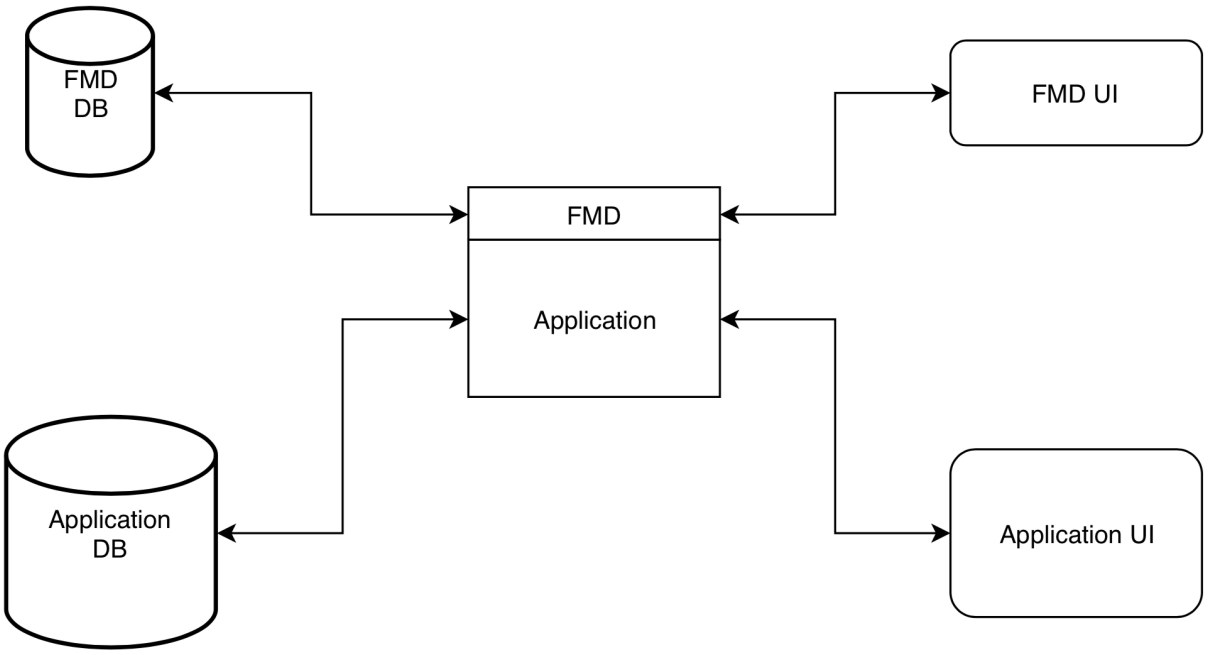
Note the '. ' before the script. This will activate your virtual environment. This script is responsible for setting up the project, and installing the following pre-commit hooks:

- Black: for automatic formatting the code. Note that we use a width-length of 100 characters.
- Flake8: for checking style error.
- Auto increase the Python version. This can either be a major, minor, or patch version increase. For more info, see the Versions-section below.

3.2.2 Architecture

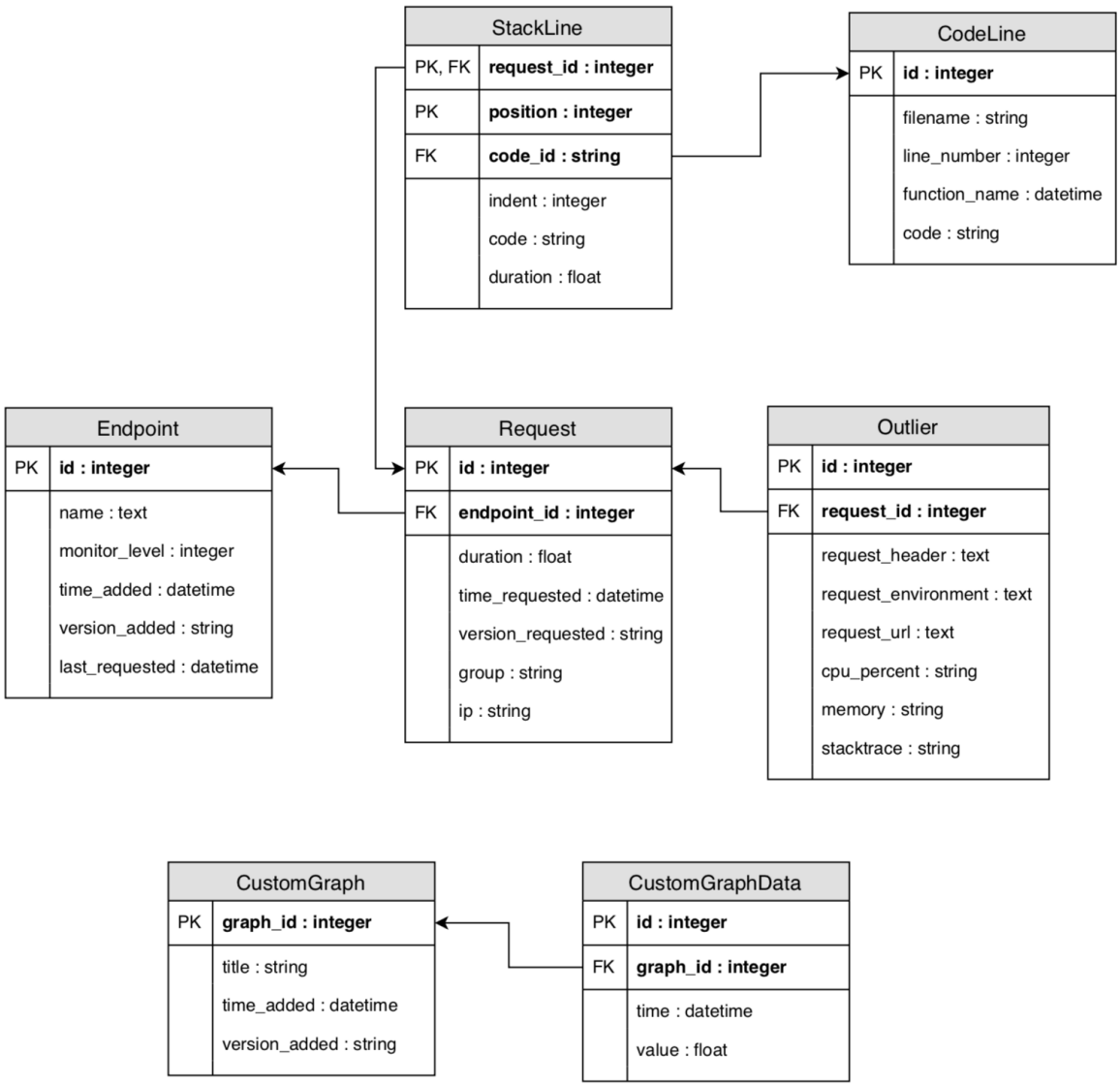
From an architectural perspective, the Flask Monitoring Dashboard uses the Layers pattern. There are 3 layers: data layer, logic layer, presentation layer. Each of these is discussed in detail in this section.

The diagram below shows how FMD interacts with the monitored application. We assumed that the application uses a database and a web interface, but these components are not mandatory. Also, the FMD DB can be the same as the Application DB.



Data layer

This layer is responsible for the data collected by FMD about the monitored application. The database schema is shown below.


StackLine
PK, FK
request_id : integer
PK
position : integer
FK
code_id : string
indent : integer
code : string
duration : float
CodeLine
PK
id : integer
filename : string
line_number : integer
function_name : datetime
code : string
Endpoint
PK
id : integer
name : text
monitor_level : integer
time_added : datetime
version_added : string
last_requested : datetime
Request
PK
id : integer
FK
endpoint_id : integer
duration : float
time_requested : datetime
version_requested : string
group : string
ip : string
Outlier
PK
id : integer
FK
request_id : integer
request_header : text
request_environment : text
request_url : text
cpu_percent : string
memory : string
stacktrace : string
CustomGraph
PK
graph_id : integer
title : string
time_added : datetime
version_added : string
CustomGraphData
PK
id : integer
FK
graph_id : integer
time : datetime
value : float

The Data layer is technology-agnostic: you can use any RDBMS system you like, as long as it is supported by SQLAlchemy, the Object Relational Mapper used by FMD! We mostly use SQLite for development purposes, but regularly test FMD with MySQL and PostgreSQL.

Logic layer

This layer is responsible for implementing all the features of FMD, storing and retrieving the collected data to and from the Data layer, and providing a REST API to be used by the Presentation layer. The Logic layer is written in Python and contains the following 4 directories: controllers, core, database, views.

- **database:** contains all the functions that access the Data layer. No function from outside this directory should make queries to the database directly.
- **views:** contains the REST API. The Presentation layer uses the REST API to get the data that it has to show in the web interface.
- **controllers:** contains the business logic that transforms the objects from the database into objects that can be used by the Presentation layer. It represents the link between **database** and **views**.
- **core:** this is where the magic of FMD happens. Measuring the performance of monitored endpoints, profiling requests, and detecting outliers are all implemented in this directory.

Presentation layer

This layer is responsible for showing the data collected by FMD in a user-friendly web interface. It is written using AngularJS 1.7.5 framework and Jinja2 templating language, and contains 2 directories: static and templates.

- **templates:** only contains 2 Jinja2 templates. They represent the entry point for the web interface and request all the Javascript files required.
- **static:** contains the JavaScript, HTML, and CSS files. The code follows the Model-View-Controller pattern. The main components of this directory are described below:
  - app.js: defines the app and implements the routing mechanism. For each route, a JS controller and HTML template are specified.
  - controllers: JS files that make calls to the REST API of FMD and implement the logic of how the data is visualized.
  - services: JS files that contain cross-controller logic.
  - directives.js: file that declares custom HTML tags.
  - filters.js: contains functions used for nicely formatting the data.
  - pages: HTML files for building the web interface.

3.2.3 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.
- **Heroku deployment**: The following branches are automatically deployed to Heroku. This is useful for quickly testing, without running any code locally.
  - **Master**: The master branch is deployed at: https://fmd-master.herokuapp.com.
  - **Development**: The development is deployed at: https://fmd-development.herokuapp.com.
  - **Pull requests**: Pull requests are also automatically build with a unique URL.
- **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 from the root of Flask-MonitoringDashboard directory:

```
python setup.py test
```

  All the tests are in the **test** directory and follow the naming convetion `test_*.py`.
- **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
```

- **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.

3.3 Migration

3.3.1 Migrating from 1.X.X to 2.0.0

Version 2.0.0 offers a lot more functionality, including Request- and Endpoint-profiling.

There are two migrations that you have to do, before you can use version 2.0.0.

1. **Migrate the database:** Since version 2.0.0 has a different database scheme, the Flask-MonitoringDashboard cannot automatically migrate to this version.

   We have created a script for you that can achieve this. It migrates the data in the existing database into a new database, without removing the existing database.

   You can find the migration script here.

   If you want to run this script, you need to be aware of the following:

   - If you already have version 1.X.X of the Flask-MonitoringDashboard installed, first update to 2.0.0 before running this script. You can update a package by:

     ```
     pip install flask_monitoringdashboard --upgrade
     ```

   - set **OLD_DB_URL** on line 16, such that it points to your existing database.
   - set **NEW_DB_URL** on line 17, to a new database name version. Note that they can't be the same.
   - Once you have migrated your database, you have to update the database location in your configuration-file.

2. **Migrate the configuration file**: You also have to update the configuration file completely, since we've re factored this to make it more clear. The main difference is that several properties have been re factored to a new header-name.

   For an example of a new configuration file, see this configuration file.

3.3.2 Migrating from 2.X.X to 3.0.0

Version 3.0.0 adds functionality for tracking return status codes for each endpoint.

This requires a minimal change to the database: adding the 'status_code' (INT) field to the Request table.

You can add the field by hand, or you can run the corresponding migration script:

3.4 TODO List

All things that can be improved in Flask-MonitoringDashboard are listed below.

3.4.1 Features to be implemented

- create a memory usage graph for the outliers, similar to the cpu usage one
- allow for endpoint purge and/or deletion
- allow for merging of endpoints
- improve code readability through comments

3.4.2 Work in progress

3.5 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.

3.5.1 v3.1.0

Changed

- Added support for Python 3.8
- Started using pytest instead of python's unittest
- Started using Webpack for frontend packaging
- Improved reports
- Moved to Github Actions from Travis for CI/CD
- Improved docstrings
- Various bug fixes

3.5.2 v3.0.9

Changed

- Fixed upgrade message bug
- Fixed Heroku deployment

3.5.3 v3.0.8

Changed

- Fixed the changelog; functionality is the same as 3.0.7 :)

3.5.4 v3.0.7

Changed

- Added a first version of the Reporting functionality
- Improved usability of the overview table
- Fixed the issue with some table columns being sorted as text as opposed to numbers
- A few other bug fixes

3.5.5 v3.0.6

Changed

- Removed profiler feature from monitoring level 2
- Added outlier detection feature to monitoring level 3
- Configurable profiler sampling period, with 5 ms default
- Implemented an in-memory cache for performance improvements

3.5.6 v3.0.0

Changed

- Tracking also status codes
- Display times as numbers to make them sortable
- Add leading slash to blueprint paths
- Added status codes with corresponding views

3.5.7 v2.1.1

Changed

- Default monitoring level is now 1
- Fixed bug causing config file not being parsed
- Monitoring level can be set from the 'detail' section
- Improved README

3.5.8 v2.1.0

Changed

- Frontend is now using AngularJS
- Removed TestMonitor
- Added Custom graphs
- Fixed Issue #206
- Added support for Python 3.7
- Updated documentation
- Updated unit tests

3.5.9 v2.0.7

Changed

- Fixed Issue #174
- Fixed issue with profiler not going into code
- Implemented a Sunburst visualization of the Grouped Profiler
- Improved test coverage
- Improved python-doc
- Added functionality to download the outlier data
- Dropped support for Python 3.3 and 3.4

3.5.10 v2.0.0

Changed

- Added a configuration option to prefix a table in the database
- Optimize queries, such that viewing data is faster
- Updated database scheme
- Implemented functionality to customize time window of graphs
- Implemented a profiler for Request profiling
- Implemented a profiler for Endpoint profiling
- Refactored current code, which improves readability
- Refactoring of Test-Monitoring page
- Identify testRun by Travis build number

3.5.11 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

3.5.12 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

3.5.13 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

3.5.14 v1.10.0

Changed

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

- Added documentation