university of groningen

faculty of science and engineering

University of Groningen

Bachelor thesis

A Dashboard for Automatic Monitoring Python Web Services

Patrick Vogel

First supervisor
Dr. Mircea Lungu

Second supervisor
Dr. Vasilios Andrikopoulos

August 2, 2017

Abstract

This bachelor thesis describes the problem of monitoring the performance of Flask-based Python web-services. For the web developer that wants to monitor the performance for their web-services, a solution is presented. The solution consists of an automatic monitoring dashboard, that can be installed in any existing web-service with Python and Flask. After the installation (installation requires 2 lines of code) an automatic monitoring service is ready to use. With several lines of extra configuration, the following features are supported:

1. Automatic version detection. The monitoring tool detects the active version of this VCS and combines this with the collected data.

2. Comparison of execution times across different users. Which users perform better or worse on certain versions of the system and how can this be improved? The monitoring dashboard creates graphs automatically, wherein it is easy to spot differences in execution times.

3. Automatic outlier detection. Whenever the execution time is larger than usual – an outlier –, the monitoring tool collects extra information about the requested data, such as the stack-trace of active threads, and CPU- and memory usage of the system. In order to reduce the overhead of the dashboard, logging extra information is only done for potential outliers.

Moreover, this thesis describes the design of the monitoring tool. After the development of the dashboard, it is deployed for a case study to validate its usefulness. The collected results have been used to analyze and improve the performance of that case study.

Contents

General information

Start date: April 17, 2017
First supervisor: Dr. Mircea Lungu (m.f.lungu@rug.nl)
Second supervisor: Dr. Vasilios Andrikopoulos (v.andrikopoulos@rug.nl)
Student: Patrick Vogel (p.p.vogel@student.rug.nl)
Project partner: Thijs Klooster (t.klooster.1@student.rug.nl

Chapter 1

Introduction

"The web is the only true object oriented system", says Alan kay, the inventor of object oriented programming. This because the service oriented architecture of the web embodies the early ideas of the OO paradigm much better than the way it has been implemented in the mainstream programming languages. Indeed, the web becomes increasingly a system of interconnected services.

1.1 Web frameworks

A web framework is a code library that makes a developer's life easier when building reliable, scalable and maintainable web applications. Web frameworks encapsulate what developers have learned over the past twenty years while programming sites and applications for the web. Frameworks make it easier to reuse code for common HTTP operations and to structure projects so other developers with knowledge of the framework can quickly build and maintain the application. [6]

Frameworks provide functionality in their code or through extensions to perform common operations required to run web applications. These common operations include:

1. URL routing
2. HTML, XML, JSON, and other output format templating
3. Database manipulation
4. Security against Cross-site request forgery (CSRF) and other attacks
5. Session storage and retrieval

1.1.1 Flask and Django

Flask and Django are two of the most popular web frameworks for Python. The biggest difference between Flask and Django is:

- Flask implements a bare-minimum and leaves the bells and whistles to add-ons or to the developer. Flask provides simplicity, flexibility and fine-grained control. It is unopinionated (it lets you decide how you want to implement things).

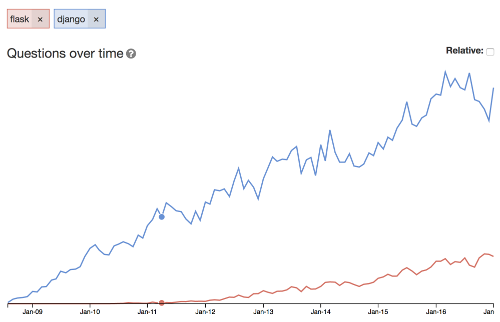


Figure 1.1: Number of questions for Flask and Django on StackOverflow

- Django follows a "batteries included" philosophy and gives you a lot more out of the box. Django provides an all-inclusive experience: you get an admin panel, database interfaces, an ORM, and directory structure for your apps and projects out of the box.

Django has been around for longer — it was first released in 2005, while Flask debuted in 2010 — and is more popular — in January 2017, there were 2631 StackOverflow questions about Django and 575 for Flask. Both frameworks are growing steadily in popularity, as can be seen by the number of StackOverflow questions about each in Figure 1.1. [4]

1.2 Flask

Since this bachelor project is about implementing a dashboard for Flask, a bit more information about Flask is provided and details are left out for Django.

Flask is called a micro framework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling, various open authentication technologies and several common framework related tools. Extensions are updated far more regularly than the core Flask program. [9]

Flask is easy to get started with as a beginner because there is little boilerplate code for getting a simple app up and running:

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

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

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

With the code snippet above, a simple web-service is created (line 1 and 2). Moreover, an endpoint is connected to the web-service (line 4).

The web-service can be started with the code expressed on the last line (line 9). Each Flask web-service contains the parts that are expressed in this code snippet, but are usually more complex[1] than this simple example.

1.3 Service Evolution

Software evolves over time. Due to new ideas or new requirements, new features have to be added, or existing features have to be updated. Since services consists of software, it is not different for web-services. In order to keep track of the changes, it is important for their maintainers to understand them.

The evolution of software is usually done is several versions. Popular Version Control Systems (VCS), like Git, make it easier to see the changes throughout different versions. However, VCS's are not designed for combining the different versions with the performance of web-services.

1.4 The problem

One of the problems of existing web-services, like Flask, is the fact that the developers of the web-service do not have information about which parts of their web-services are used the most and which are used the least or even not at all. In a Java system, for example, one can perform static analysis to discover such dependencies, but on the web, everything is dynamic, there is no equivalent way of analyzing the dependencies to a given web-service.

Thus, every one of those Flask projects faces one of the following options when confronted with the need of gathering insight into the execution time performance behavior of their implemented service:

1. Use a third party, heavyweight, professional service on a different server.
2. Implement their own analytics tool.
3. Live without analytics insight into their services.[2]

An example of a third party service is Google Analytics. It works by inserting a piece of JavaScript on the page that has to be monitored. [10] When requesting this page, the load time has increased, due to the overhead of this script. Another disadvantage of using third party services is that some requests are not being collected, since the script is usually blocked by tools such as Adblock. This leads to an incomplete overview of the performance of the web-service.

For projects which are done on a budget (e.g. research projects) the first and the second options are often not available due to time and financial constraints. To avoid these projects ending up in the third situation, this thesis presents a low-effort, lightweight service monitoring API for Flask and Python web-services.

[1]Web-services usually contain more than one endpoint.

[2]This is very real option: and is exactly what happened to the API that will be presented in this case study for many months.

1.5 Research question

Thus, to address the previous problem, the following research question is proposed:

How to design a system that can measure the performance of a Flask web service as it evolves over time, in such a way that the overhead of the service is minimal?

The overhead that is meant in the question above, consists of two parts. First, the overhead of starting to measure the performance should be minimal. Second, the overhead of the monitoring program should be minimal.

1.6 Article

Related to this thesis an article has been published [7]. The article shortly introduces the Dashboard. It is easy to read for people that want to gain interest in this topic. The article is a collaboration between two professors (Mircea Lungu and Vasilios Andrikopoulos) and two students (Thijs Klooster and Patrick Vogel).

Chapter 2

Related work

Several monitoring tools already exists for Python or web services. A few popular open source monitoring projects are listed below.

Sentry

Sentry is a Python project that focuses on monitoring real-time errors for various programming languages including Python. Thus, it can be used perfectly for monitoring Python web-services, like Flask. Sentry is focused at monitoring errors, but has no notion of errors throughout different software-versions. Whenever Sentry captured an error, the error is automatically sent to their website[1]. On that website it is possible to see all errors including a stack trace.

Graphite

Graphite is an enterprise-scale monitoring tool. It was originally designed and written by Chris Davis at Orbitz in 2006 as side project that ultimately grew to be a foundational monitoring tool. In 2008, Orbitz allowed Graphite to be released under the open source Apache 2.0 license. Since then Chris has continued to work on Graphite and has deployed it at other companies including Sears, where it serves as a pillar of the e-commerce monitoring system. [3] Graphite is focused on metrics, rather than the performance as a whole.

Flask_jsondash

Flask JSON-Dash is an open source project at Github [2]. With Flask JSON-Dash it is possible to build complex dashboards without any front-end code. It is easy to configure the project in an existing Flask-application. This project focuses on easily designing dashboard, rather than using them. Another disadvantage is that Flask JSON-dash has no notion of versioning. Thus, it is not possible to detect changes in the performance of execution times as the system evolves.

[1]`https://sentry.io/welcome/`

ELK Stack

The ELK Stack combines Elasticsearch, Logstash and Kibana.

- Elasticsearch is a distributed, open source search and analytics engine based on Lucene. Elasticsearch is the most popular enterprise search engine followed by Apache Solr, also based on Lucene. Elasticsearch can be used to search all kinds of documents. [8]
- Logstash is an open source data collection, and transportation pipeline. With connectors to common infrastructure for easy integration, Logstash is designed to process a growing list of log, event, and unstructured data sources for distribution into a variety of outputs, including Elasticsearch.
- Kibana is an open source data visualization platform that allows the user to interact with the data. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster. Users can create bar, line and scatter plots, or pie charts and maps on top of large volumes of data. your data far and wide. [1]

The disadvantage of using the ELK stack is the same as for Sentry. It is focused at logging relevant information, rather than monitoring a web service. The ELK stack deals well with datasets that rapidly grow.

The ELK-stack is rather complicated to setup, since all three programs must be installed separately. Once the installation is complete, the programs must be configured to work with each other.

Chapter 3

Automatic Monitoring Dashboard

3.1 Binding the dashboard

After the research that has been done in the previous chapter, it is clear that there is no system that can measure the performance of a Flask web service as it evolves over time. In order to solve this problem, the Automatic Monitoring Dashboard has been developed. The dashboard consists of a drop-in Python library that allows developers to monitor their Flask based web-service. In order to achieve the simplicity of installing the dashboard, only two lines of code are required to start monitoring an existing web-service:[1]

```python
import dashboard
# app is the Flask application
dashboard.bind(app)
```

After binding to the service, the dashboard becomes available at `/dashboard`. In order to configure the dashboard at a different URL, an additional line of code is required. For all configuration options of the dashboard, see Appendix B.

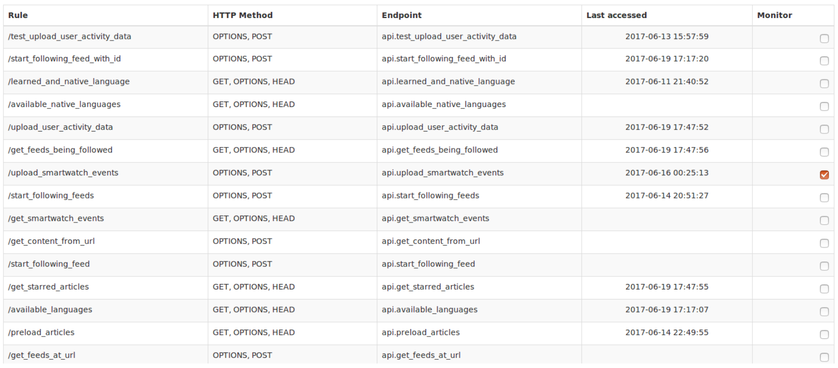
Whenever the web-service starts, the dashboard searches for all existing endpoints[2]. In order to provide a selection procedure for the endpoints that have to be monitored, an interface is rendered. An example of this interface can be found in Figure 3.1.

As can be seen from Figure 3.1, selecting and deselecting an endpoint is as easy as performing a single mouse click. Once an endpoint has been selected, the endpoint is wrapped into a monitoring function. Doing it this way, the wrapper will be executed whenever a request to that endpoint is made. The wrapper consists of measuring the performance of the execution time of processing the actual request. The details of this wrapper are explained in Section 4.

With this minimal set-up, several graphs are rendered to visualize the outcome of the measurements. First, the heatmap is presented to

[1]Before importing the dashboard, it must be installed. See Appendix B for more details.

[2]Endpoints are explained in the next chapter.



Rule	HTTP Method	Endpoint	Last accessed	Monitor
/test_upload_user_activity_data	OPTIONS, POST	api.test_upload_user_activity_data	2017-06-13 15:57:59	☐
/start_following_feed_with_id	OPTIONS, POST	api.start_following_feed_with_id	2017-06-19 17:17:20	☐
/learned_and_native_language	GET, OPTIONS, HEAD	api.learned_and_native_language	2017-06-11 21:40:52	☐
/available_native_languages	GET, OPTIONS, HEAD	api.available_native_languages		☐
/upload_user_activity_data	OPTIONS, POST	api.upload_user_activity_data	2017-06-19 17:47:52	☐
/get_feeds_being_followed	GET, OPTIONS, HEAD	api.get_feeds_being_followed	2017-06-19 17:47:56	☐
/upload_smartwatch_events	OPTIONS, POST	api.upload_smartwatch_events	2017-06-16 00:25:13	✓
/start_following_feeds	OPTIONS, POST	api.start_following_feeds	2017-06-14 20:51:27	☐
/get_smartwatch_events	GET, OPTIONS, HEAD	api.get_smartwatch_events		☐
/get_content_from_url	OPTIONS, POST	api.get_content_from_url		☐
/start_following_feed	OPTIONS, POST	api.start_following_feed		☐
/get_starred_articles	GET, OPTIONS, HEAD	api.get_starred_articles	2017-06-19 17:47:55	☐
/available_languages	GET, OPTIONS, HEAD	api.available_languages	2017-06-19 17:17:07	☐
/preload_articles	GET, OPTIONS, HEAD	api.preload_articles	2017-06-14 22:49:55	☐
/get_feeds_at_url	OPTIONS, POST	api.get_feeds_at_url		☐

Figure 3.1: Example of rules-table

detect any outliers in the number of requests. This heatmap answers questions like: During what hours does the endpoint get the most requests? Does the endpoint gets more requests during the week or in the weekend? And does the number of requests per day grow over time? For answering these questions, the monitoring tool shows a heatmap with on the y-axis every hour and the x-axis the last 30 days. Thus a column consists of 24 cells and the color of the cells represents whether the number of requests in that hour on that day is large (red) or small (blue) or any light-red, white, light-blue color in between. The heatmap is presented in Figure 3.2a.

In order to investigate the number of requests in the past two days of this endpoint, two additional graphs are created in Figure 3.2b and Figure 3.2c. The first graph (Figure 3.2b) is a vertical bar graph, which shows the maximum, average and minimum execution time per hour. In this graph it is easy to compare different hours with each other. Also, it is possible to detect whether the average execution time is due to one busy hour, or to requests with sequentially large execution times. The second graph is also a vertical bar graph, but shows the number of requests per hour. This is the same information as presented in the heatmap. However, the advantage of this graph is that it easier to compare different hours with each other. In a heatmap it is harder to see the relative differences (e.g. twice the number of requests in that hour, compared to another hour), while it is clearly visible in this graph.

3.2 Evolving systems

With the two lines of code given in the previous section, a simple dashboard is installed. However, one of the problems described in the previous section was to monitor the system as it evolves. In order to detect whether the system evolves, a configuration file can be set-up. Configuring the dashboard with this file can be done with the following lines of code:

```
import dashboard
dashboard.config.from_file('<path>/config.cfg')
dashboard.bind(app)
```

It is for example possible to tag the version of the system in the configuration file, this can be done with a file that looks like:

```
[dashboard]
APP_VERSION=1.0
```

Thus, the version of the active system is now 1.0, according to the configuration file. It might not be handy to update the configuration file, every time a change to the code has been made. Therefore, the functionality of the dashboard is to automate the detection of evolving systems. With this functionality, it is possible to detect whether the execution time of certain endpoints increases as newer versions of a system are deployed. An observation that was made during the development of the dashboard is that most professional projects are built with a Version Control System (VCS). In a VCS, the development of a software-project consists of multiple versions. Each version has a hash-code. The functionality of the dashboard consists of reading the hash-code of the active version as soon as the application starts. For configuring the dashboard with the functionality of automatically detecting versions, the following configuration file can be used:

```
[dashboard]
GIT=/.git/
```

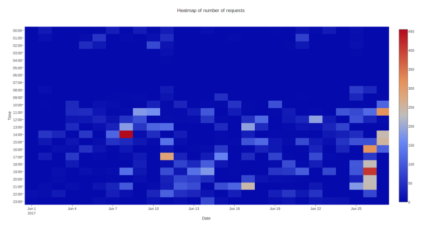
With the detection of different versions of the system, more graphs can be visualized. An example of such a graph is Figure 3.2d, which shows boxplots for all execution times for every version. The data in the boxplot are the execution times for that endpoint. Whenever an endpoint performs better since the release of a new version, it should be possible to detect in this graph. Also the release date and time of a version is presented, since the data could be less reliable if the version was exposed in a short time period.

3.3 Automatic User Detection

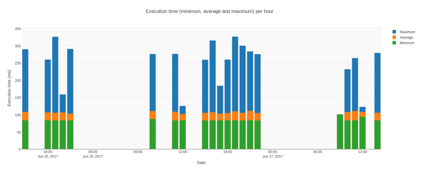
The last configuration option contains another main functionality of the dashboard. The performance of some endpoint could be different per user. Using this configuration it is possible to see which users of the application perform better than others. Which users get fast execution times, and which do not? To differentiate between the users, a custom function has to be set up. That function retrieves the session-id or username of a specific user that is requesting your site. Specifying the function is done using the following lines of code:

```python
import dashboard

def get_user_name():
    return session.get('username')

dashboard.config.get_group_by = get_user_name
dashboard.bind(app)
```

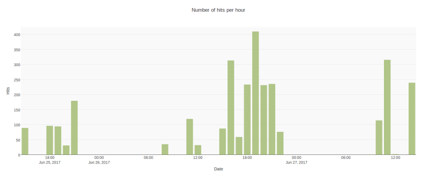
Allowing the dashboard to distinguish between different users helps in finding differences in execution times for those users. Suppose the application has an endpoint that iterates through all instances of a certain list that is being retrieved from the database, then that endpoint has a complexity of $\mathcal{O}(n)$. Suppose the length of the list is for user A 10 instances, and for user B 10,000 instances, then the application has



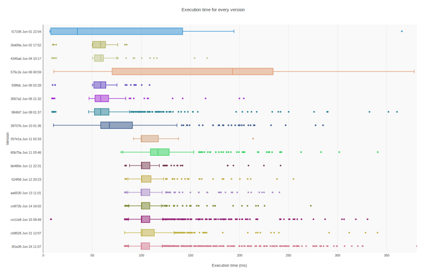
(a) Heatmap with the number of requests



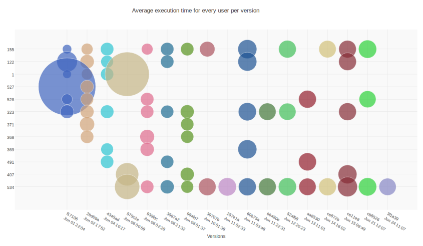
(b) Vertical bar plot with the execution time per hour



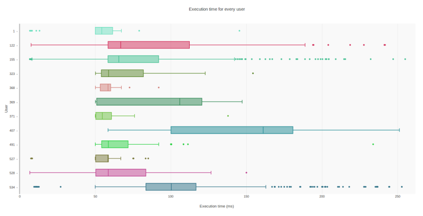
(c) Vertical bar plot with the number of requests per hour



(d) Horizontal boxplot with the execution time for every version



(e) Bubble plot with the execution time per user per version



(f) Horizontal boxplot with the execution time for some users

Figure 3.2: Graphs for the Dashboard

clearly different execution times for those users. In an endpoint with complexity $\mathcal{O}(1)$, no differences should be found at all.

With this functionality, new graphs can be visualized to present the outcome of the measurements. An example of such a graph is Figure 3.2e. This type of graph is a dot-graph, where a larger average execution time is presented by a larger dot. In this graph is it possible to detect whether larger execution times are due to some users, or due to some versions.

Another graph is presented in Figure 3.2f. This graph shows a number of horizontal boxplots, one for each user. The data in the boxplot are the execution times for that user for that endpoint. Thus, whenever an endpoint relies on user-specific data, the boxplots makes it possible to detect the differences.

Chapter 4

Architecture

The core functionality of the dashboard is to collect statistics about a selection of the endpoints. In order to let the maintainer of the dashboard choice which endpoints must be monitored, there are three options.

The first option is to statically add a python decorator to the endpoint that is followed. But the code of the user might not always be clearly written. Therefore it might be hard to find all endpoints.

Another option would be to write a Python-parser that searches through the code and adds a decorator to each endpoint. But parsing is an exhausting and complicated task, which would not be the best solution.

A third option was to use the built-in flask function to loop over all endpoints. The following code shows how this can be done.

```python
for rule in app.url_map.iter_rules():
    function = app.view_functions[rule.endpoint]
    # do something with the function
```

Since this is done dynamic, there is no need for the maintainer of the dashboard to change any existing code. Therefore, this option has the advantage of being easy-to-use.

4.1 Rule selection

As explained in the previous section, the decision was made to dynamically monitor the endpoints. In order to provide this functionality, there are two options.

The first option is to add a middleware to the Flask-application. The code for providing a middleware is the following:

```python
class MyMiddleWare(object):

    def __init__(self, app):
            self.app = app

    def __call__(self, environ, response):
            print('incoming requests')
            return self.app(environ, response)
```

```
app = Flask(__name__)
app.wsgi_app = MyMiddleWare(app.wsgi_app)
```

The disadvantage of the code above is, that every endpoint is exposed to the middleware. Thus every endpoint has an overhead, although small, even if it is not monitored at all.

Another solution is to dynamically add a decorator to the endpoints that are being monitored. An example of a decorator to measure the execution time of a certain endpoint is the following:

```
def track_performance(func, ...):
    @wraps(func)
    def wrapper(*args, **kwargs):
        ...
        time1 = time.time()
        result = func(*args, **kwargs)
        time2 = time.time()
        t = (time2-time1)*1000
        add_function_call(time=t, ...)
        ...
        return result
    return wrapper
```

Suppose, some project uses an endpoint `serve_index` that returns the index-page of a certain website. In order to collect statistics about this page, the following line of code dynamically adds the wrapper-function to the endpoint.

```
serve_index = track_performance(serve_index, ...)
```

However, adding decorators to rules that have to be monitored is a good idea, but then the dashboard must remove the decorator whenever the endpoint is deselected. Removing endpoints dynamically is a bit more complex. The dashboard solves this by adding the attribute 'original' to the decorator, which stores the original function. For clarification, the decorator is (partially) exposed below:

```
def track_performance(func, ...):
    @wraps(func)
    def wrapper(*args, **kwargs):

        time1 = time.time()
        result = func(*args, **kwargs)
        time2 = time.time()
        t = (time2-time1)*1000
        add_function_call(time=t, ...)

        return result
    wrapper.original = func
    return wrapper
```

The following code removes the decorator and restores the original function:

```
for rule in rules:
    original = getattr(user_app.view_functions
                [rule.endpoint], 'original', None)
    if original:
        user_app.view_functions[rule.endpoint] =
            original
```

Using the latter has the advantage of not having overhead for decorators that not have to be monitored. This option lead to the decision to make a page in the monitoring dashboard in which rules can be easily selected or deselected. This page provides a better overview and is easier for the developer and user of the dashboard. A possible table can be found in Figure 3.1. In this table it is easy to see which endpoints are being monitored and which are not.

4.2 Automatic outlier detection

In a number of graphs in the dashboard, it is possible to spot outliers. An example of such a graph can be found in Figure 5.6. In this figure, outliers are represented by dots. In case an outlier occurs, it is useful to discover possible reasons. Therefore, extra information is being collected. This information includes the current state of the server, like the amount of memory used on the server and the CPU-percentage used. Also, extra requests parameters are logged, like the headers, full URL, environment of the client and all GET-, POST-values of that request. And the current stack trace during the execution of the request is logged.

In order to log the current stack trace during the execution of the requests, an additional thread is created. Suppose a request arrives at time $t1$, then at that moment an extra thread is created. The purpose of the thread is to sleep N seconds, which is calculated by the average execution time for that endpoint, times the outlier-detection-constant, as explained in Appendix B. Thus, this moment can be expressed as $t2 = t1 + N$. At $t2$, the additional thread logs the entire stack trace (of all active threads).

Now, suppose that the request finished at $t3$. Then there are two possible cases:

- $t3 \leq t2$: The request is not an outlier and no extra information is logged.
- $t3 > t2$: By definition, the request is an outlier.

In the first case, the additional thread hasn't even started to collect the stack traces, and thus no overhead (apart from creating the thread) is used. In the second case, the additional thread logs the entire stack trace. This information is passed to the main thread, which stores this information, after the thread is finished processing the requests (this is at $t3$). In order to make it more clear, the information above is visualized in Figure 4.1. Note that this figure presents only the second case of the theory above.

An example of an outlier that is registered in the system, can be found in Figure 4.2. Note that the table is too large to fit in a single image, therefore the stack trace has been left out. The figure shows that the information is too large to fit in a single table. This is due to the fact that the data is unformatted.

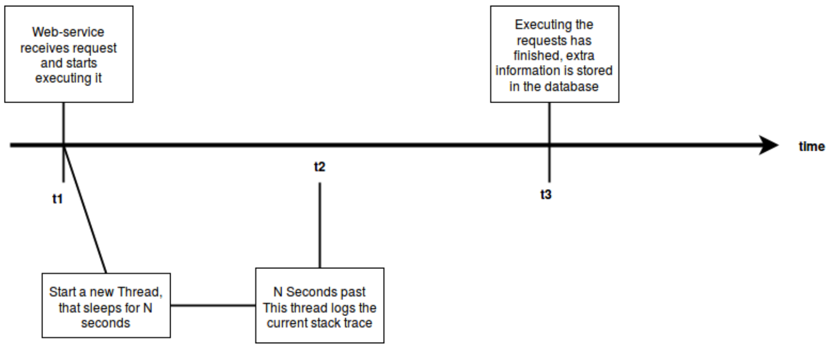
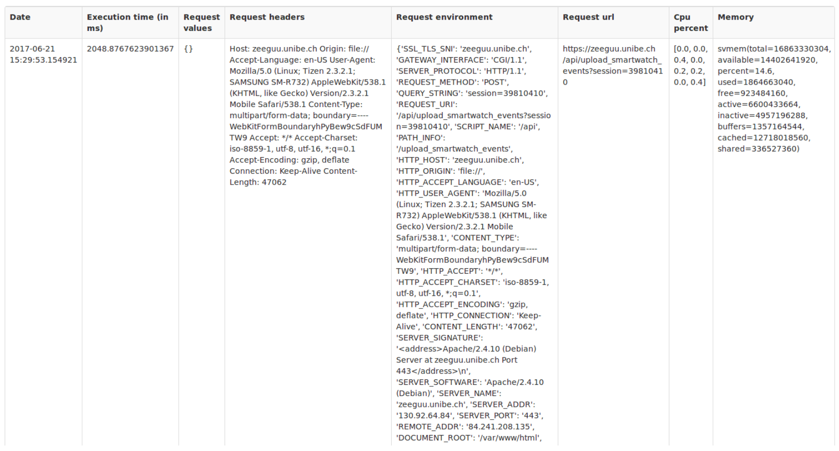

Web-service receives request and starts executing it
Executing the requests has finished, extra information is stored in the database
time
t1
t2
t3
Start a new Thread, that sleeps for N seconds
N Seconds past This thread logs the current stack trace

Figure 4.1: Timeline of processing a requests that is an outlier



Date	Execution time (in ms)	Request values	Request headers	Request environment	Request url	Cpu percent	Memory
2017-06-21 15:29:53.154921	2048.8767623901367	{}	Host: zeeguu.unibe.ch Origin: file:// Accept-Language: en-US User-Agent: Mozilla/5.0 (Linux; Tizen 2.3.2.1; SAMSUNG SM-R732) AppleWebKit/538.1 (KHTML, like Gecko) Version/2.3.2.1 Mobile Safari/538.1 Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryhPyBew9cSdFUMTW9 Accept: */* Accept-Charset: iso-8859-1, utf-8, utf-16, *;q=0.1 Accept-Encoding: gzip, deflate Connection: Keep-Alive Content-Length: 47062	{'SSL_TLS_SNI': 'zeeguu.unibe.ch', 'GATEWAY_INTERFACE': 'CGI/1.1', 'SERVER_PROTOCOL': 'HTTP/1.1', 'REQUEST_METHOD': 'POST', 'QUERY_STRING': 'session=39810410', 'REQUEST_URI': '/api/upload_smartwatch_events?session=39810410', 'SCRIPT_NAME': '/api', 'PATH_INFO': '/upload_smartwatch_events', 'HTTP_HOST': 'zeeguu.unibe.ch', 'HTTP_ORIGIN': 'file://', 'HTTP_ACCEPT_LANGUAGE': 'en-US', 'HTTP_USER_AGENT': 'Mozilla/5.0 (Linux; Tizen 2.3.2.1; SAMSUNG SM-R732) AppleWebKit/538.1 (KHTML, like Gecko) Version/2.3.2.1 Mobile Safari/538.1', 'CONTENT_TYPE': 'multipart/form-data; boundary=----WebKitFormBoundaryhPyBew9cSdFUMTW9', 'HTTP_ACCEPT': '*/*', 'HTTP_ACCEPT_CHARSET': 'iso-8859-1, utf-8, utf-16, *;q=0.1', 'HTTP_ACCEPT_ENCODING': 'gzip, deflate', 'HTTP_CONNECTION': 'Keep-Alive', 'CONTENT_LENGTH': '47062', 'SERVER_SIGNATURE': '<address>Apache/2.4.10 (Debian) Server at zeeguu.unibe.ch Port 443</address>\n', 'SERVER_SOFTWARE': 'Apache/2.4.10 (Debian)', 'SERVER_NAME': 'zeeguu.unibe.ch', 'SERVER_ADDR': '130.92.64.84', 'SERVER_PORT': '443', 'REMOTE_ADDR': '84.241.208.135', 'DOCUMENT_ROOT': '/var/www/html',	https://zeeguu.unibe.ch/api/upload_smartwatch_events?session=39810410	[0.0, 0.0, 0.4, 0.0, 0.2, 0.2, 0.0, 0.4]	svmem(total=16863330304, available=14402641920, percent=14.6, used=1864663040, free=923484160, active=6600433664, inactive=4957196288, buffers=1357164544, cached=12718018560, shared=336527360)

Figure 4.2: Detailed information about an outlier

4.3 Dependencies

In order to develop the automatic monitoring tool, several dependencies have been used. The dependencies are expressed in the setup file, which can be found in Appendix A. The usage of the dependencies can be found in the final source code[1]. The Flask-dependency is left out, since this dependency is described in Section 1.2.

4.3.1 SQLAlchemy

A database is used, since the automatic monitoring tool needs a database for storing the performance. In order to communicate with the database, the monitoring tool uses SQLalchemy, which is an Object Relational Mapper that gives application developers the full power and flexibility of SQL. It provides a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language. SQLAlchemy is used in the monitoring dashboard to communicate with the database. Although the dashboard has a default database type (SQLite), any other database type can also be configured. Without SQLAlchemy, this wouldn't be possible.

4.3.2 WTForms and Flask-WTF

The automatic monitoring tool uses HTML-forms to retrieve the input of the user, to configure itself. In order to do this, WTForms is used. WTForms is a library for generating HTML form fields. This allows the developer to maintain separation between code and presentation. With WTForms, it is not only possible to generate forms, but also to validate those forms. Although everything that WTForms does, can be done without, it makes life a little bit easier.

Flask-WTF is a simple integration of Flask and WTForms, including CSRF (Cross-Site Request Forgery), file upload, and reCAPTCHA (designed to establish that a computer user is human). Thus, Flask-WTF can be seen as an extension of WTForms.

4.3.3 Plotly

The automatic monitoring dashboard provides the output in multiple graphs. Another dependency, Plotly, is used to generate these graphs. Plotly is a visualization tool, which can be used for creating interactive, publication-quality graphs. All graphs in the dashboard are generated using Plotly. Plotly has a comprehensive library, supporting many graphs and many customizations.

4.3.4 ConfigParser

The dashboard uses a configuration file for loading several settings. In order to parse this file, another dependency comes in. The ConfigParser is a library for parsing configuration files. The configuration file consists of sections, led by a `[section]` header and followed by `name: value`-entries.

[1]Source code can be found at: `https://github.com/mircealungu/automatic-monitoring-dashboard`

4.3.5 PSUtil

The dashboard stores the current state of the server, whenever an outlier is detected. In order to retrieve the current state, another dependency is used. PSUtil (python system and process utilities) is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. PSUtil is used for logging extra information whenever an outlier is being detected in the dashboard.

4.3.6 ColorHash

The dashboard presents the same color for a single users across several graphs. In order to assign a color to a user, its name (order ID) is hashed. The dashboard uses ColorHash for converting a hash (or a regular string) into a color. Using this library, the dashboard generates the same color for every single endpoint, user, IP-Address or version.

4.3.7 Requests

The Continuous Integration Server sends the collected data to the dashboard. This sending is rather complicated without the Requests-library. Thus, the dashboard uses Requests, which allows the developer to send HTTP-requests, without the need for manual labor. Using this library, there is no need to form-encode your POST data.

Chapter 5

Case Study

For collecting the results, the dashboard has been deployed in the Zeeguu web-service. For more information about the Zeegue web-service, see Appendix C. The data collection started on May 30, 2017. For a period of 28 days[1], the dashboard processed $\sim$ 33,500 requests. The data of these requests is presented below.

5.1 Overview

Below is a quick overview of the rules that are monitored during the entire lifetime of the implemented dashboard and their intended purpose.

- **api.report_exercise_outcome:** ($\sim$ 12,500 requests, average execution time of 99.9 ms) The purpose of this endpoint is to logs the performance of an exercise. Every such performance, has a source, and an outcome.
- **api.get_possible_translations:** ($\sim$ 8,000 requests, average execution time of 1.48 s) This returns a list of possible translations in a specific language for a specific word in another specific language.
- **api.bookmarks_to_study:** ($\sim$ 5,000 requests, average execution time of 1.08 s) This returns a number of bookmarks that are recommended for this user to study.
- **api.learned_language:** ($\sim$ 5,000 requests, average execution time of 5.8 ms) This returns the learned language of the user.
- **api.get_feed_items_with_metrics:** ($\sim$ 2,000 requests, average execution time of 11.5 s) Get a json-list of feed items for a given feed ID.
- **api.validate:** ($\sim$ 500 requests, average execution time of 5.7 ms) To test whether the session is valid.
- **api.studied_words:** ($\sim$ 400 requests, average execution time of 67.5 ms) Returns a list of the words that the user is currently studying.
- **api.upload_smartwatch_events:** ($\sim$ 400 requests, average execution time of 107 ms) This processes a smartwatch event.

[1]This section was written on June 27, 2017

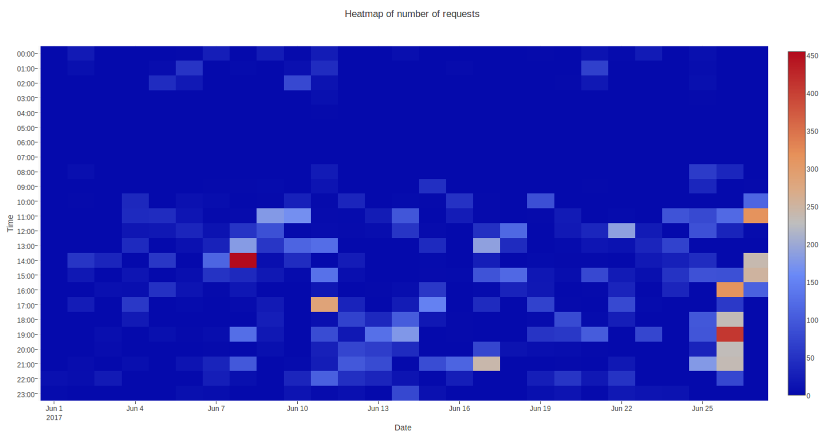


Figure 5.1: Heatmap with the number of requests for Report-exercise-outcome

From the overview above it is clear that there is a large difference in the number of requests per endpoint. It is too much to investigate all endpoints, but the most important endpoints are **api.report_exercise_outcome** since this has the most requests, and **api.get_feed_items_with_metrics**, since this endpoint has an enormous execution time.

With the two datasets, it is possible to present the data in various graphs. A distinguish is being made between graphs that present all data, and graphs that present data per only one endpoint. Every graph is explained below.

5.2 The Most Used Endpoint

The endpoint 'Report exercise outcome' has the largest number of requests in the past 28 days. This was the reason to investigate the endpoint and detect differences across versions and users. According to Figure 5.1, the web-service processed more requests in between 10:00 and 22:00 than outside this period. Another observation that is made is that the number of requests is increased in the past two days.

Figure 5.3 confirms that there are many requests in the past two days, with a maximum of $\sim$ 410 at June 26, between 19:00 and 20:00 hour. It could be that the server is overloaded at this time, which results in higher execution times. However, figure 5.2 shows that the minimum, average and maximum execution times do not varier much on June 26, between 19:00 and 20:00 hour, compared to other hours. From this figure we can conclude that the server is not overloaded.

Another graph that is useful is to detect differences in execution times per user per version. From Figure 5.4, the observation is made that there is not much difference in execution time per user across versions, since the size of the bubbles is approximately the same. However, this figure shows that the users with id 1 and 527 have larger execution times than other users. Since the execution times are normal in other versions for those users, there is no reason to worry about.

From Figure 5.4, it is possible to observe that there is not much vari-

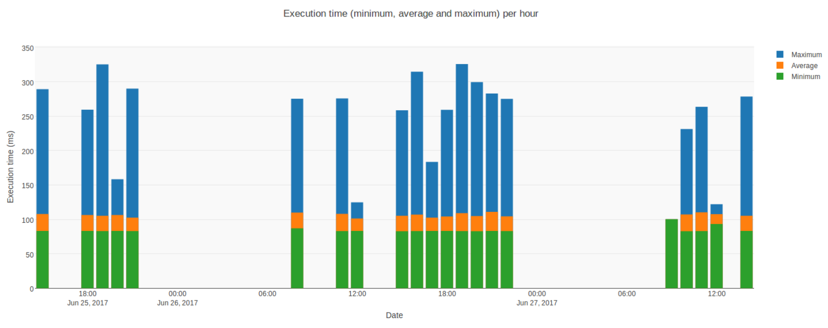


Figure 5.2: Vertical bar plot with the execution time per hour for Report-exercise-outcome

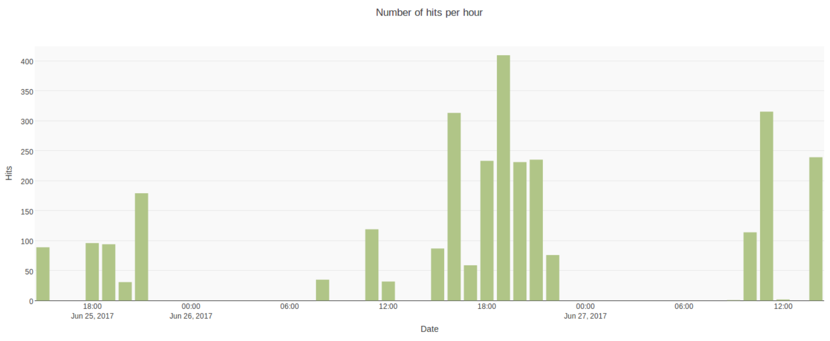


Figure 5.3: Vertical bar plot with the number of requests per hour for Report-exercise-outcome

ation in execution times across multiple versions, since the size of the bubbles don't seem to vary during the versions. According to Figure 5.5, the execution time has increased from version 257e1a on June 11, 2017 at 02:33 and later versions. This can be caused by a complexer function.

The last observation that is made about this endpoint is to detect difference in execution times across users. As concluded in Figure 5.4, there is no variation in the performance of execution times per user. To validate this observation Figure 5.6 is presented. This graph confirms that the execution times for users varies a lot, but it is impossible to conclude that some users have a better execution time than others.

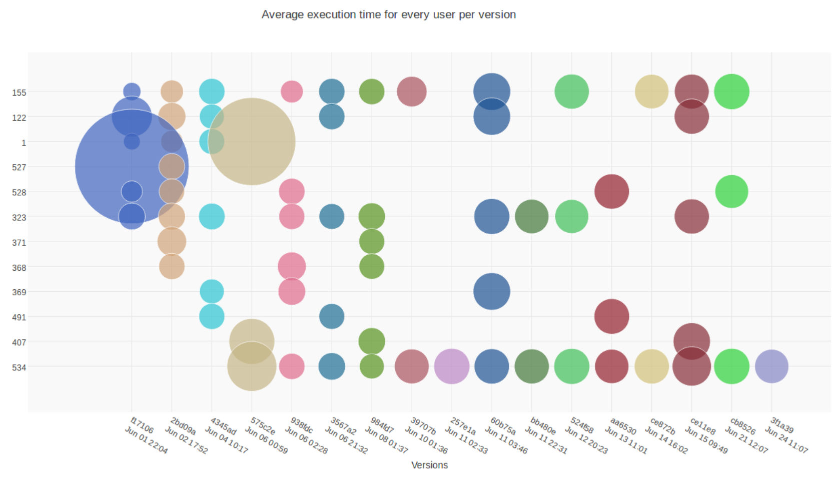


Figure 5.4: Bubble plot with the execution time per user per version for Report-exercise-outcome

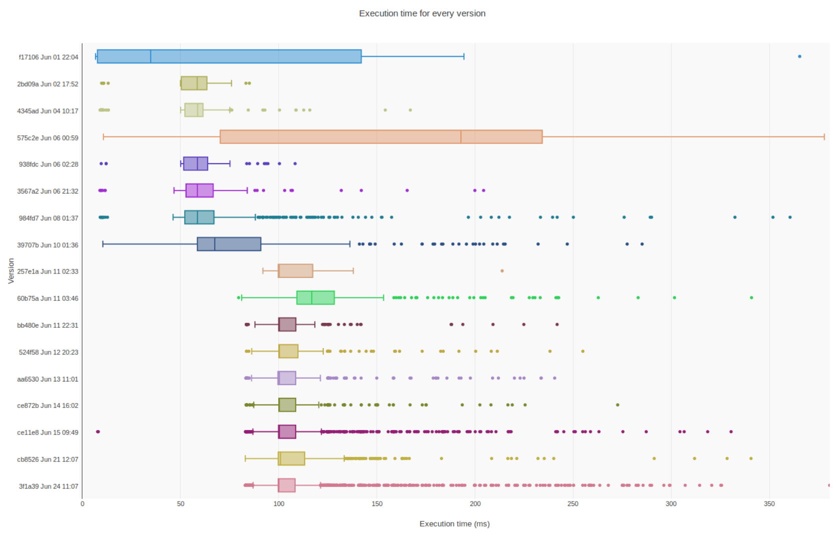


Figure 5.5: Horizontal boxplot with the execution time for every version for Report-exercise-outcome

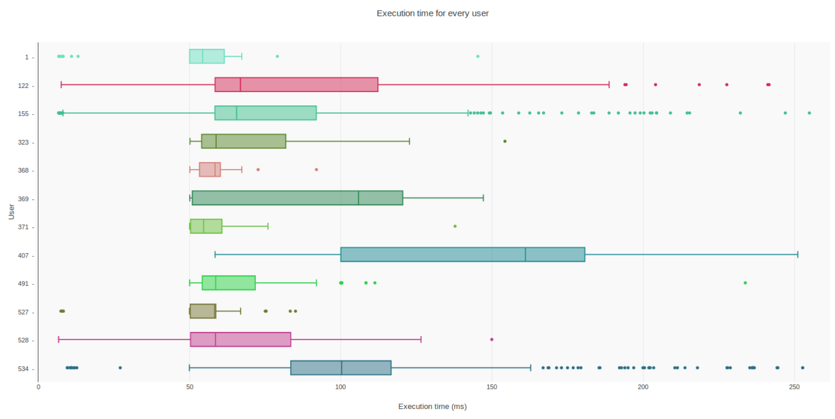


Figure 5.6: Horizontal boxplot with the execution time for some users for Report-exercise-outcome

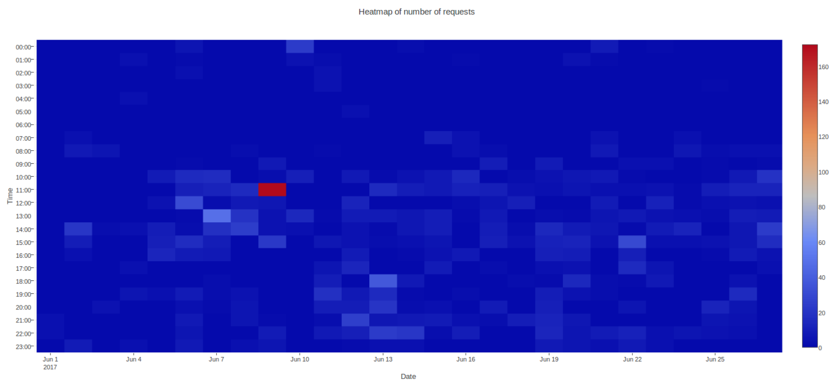


Figure 5.7: Heatmap with the number of requests for Get-feed-items-with-metrics

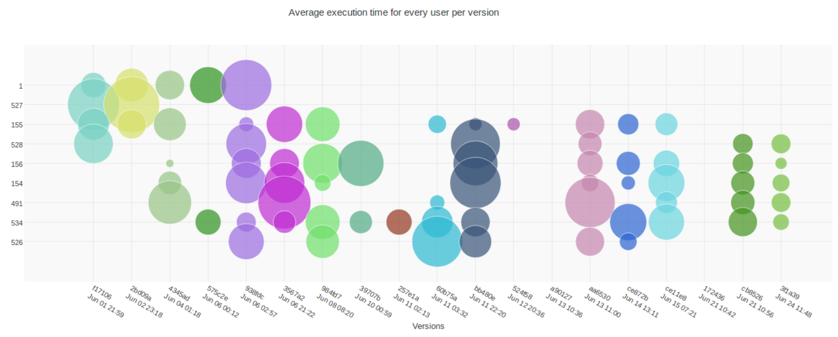


Figure 5.8: Bubble plot with the execution time per user per version for Get-feed-items-with-metrics

5.3 The Slowest Endpoint

Another interesting endpoint is **api.get_feed_items_with_metrics**, since it this endpoint has an average execution time of 11.5 seconds. More investigation is needed to explain whether this large average in execution times is due to some users, or that it is a complex endpoint. To start with, the heatmap for this endpoint is exposed in Figure 5.7. From this figure it is clear that one hour has a large number of requests, which is on June 9, 2017 between 11:00 and 12:00. Apart from this hour, the number of requests doesn't seem to be larger than 80 per hour. From this figure, we can conclude that the web-service is not overloaded for this endpoint.

Another interesting graph to expose is the average execution time per user per version of the application. This bubble graph is presented in Figure 5.8. From this figure it is clear that there is a lot of variation in execution times between different users. But from the same graph it can also be seen that the average execution time decreases in the most recent versions. Further plots have to be investigated to confirm or decline this.

The graph with horizontal box plots for the execution times per version is also interesting to see. This graph can be found in Figure 5.9. This graph can confirm whether the execution time in recent versions (lower in the figure) have been decreased compared to the execution times in older versions. From this figure it is not possible to detect those differences. However, the observation is made that version 172436 (re-

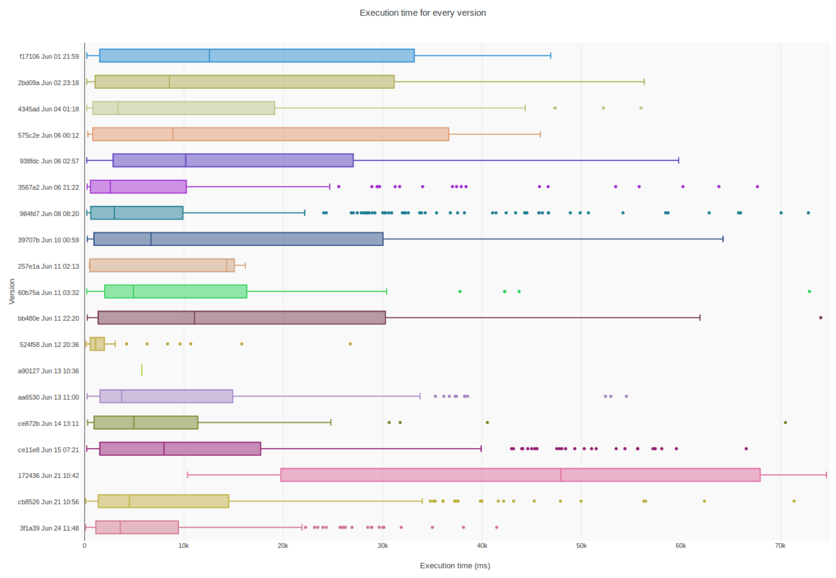


Figure 5.9: Horizontal boxplot with the execution time per version Get-feed-items-with-metrics

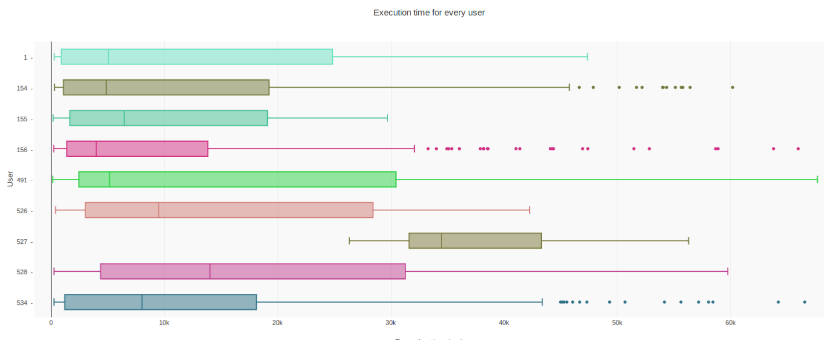


Figure 5.10: Horizontal boxplot with the execution time for a selection of users for Get-feed-items-with-metrics

leased at June 21, 2017 at 10:42) has a larger execution time than all other versions. This version can be omitted, since this version is only exposed for 14 minutes. Therefore the data in this version is not reliable.

Another observation that was made in Figure 5.8 was the fact that users have a lot of variation in execution times. Confirming this observation can be done with Figure 5.10. This figure shows horizontal boxplots with the execution time for every user. Observing this figure confirms that most users have a large variation in execution times. But for this selection of users it is not possible to confirm that some users perform better than others.

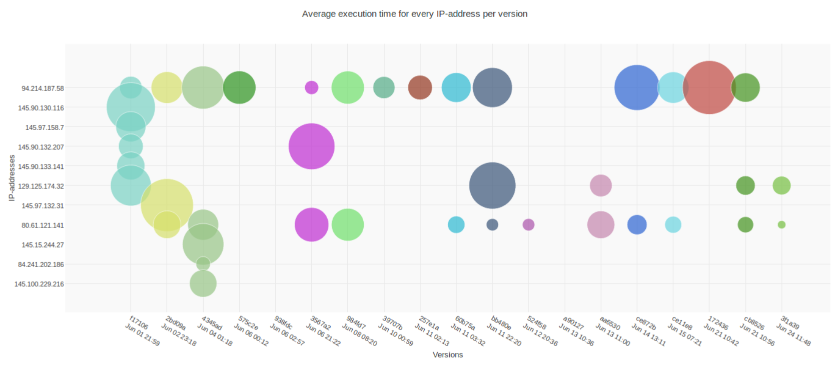


Figure 5.11: Bubble plot with the execution time per IP-address per version for Get-feed-items-with-metrics

The graph with the execution time per IP-address is interesting as well. This graph is exposed in Figure 5.11. This figure should show much similarities with Figure 5.8. It shows also a dot-graph, but this time with the average execution time per IP-address per version. Assuming that most users will visit the web-service from the same IP-address, should results in the same graph as the previous graph. An advantage of this graph, is that it doesn't require a setup function as the previous graph does, but this data can be less reliable, since multiple users can log in from the same IP-address, or a single user can use multiple IP-addresses.

As concluded in the previous paragraph, this figure should show much similarities with Figure 5.8. However, this doesn't seem the case. There are several options for this observation. The first is that only a selection of users and a selection of IP-addresses is exposed. It can be the case that the users exposed in Figure 5.8 are different that the users in Figure 5.11. Another option for the observation is that the users of this web-service uses many IP-addresses. Or that an IP-address is used by multiple users. From the selection presented in the graphs, it is impossible to validate one of the options.

Chapter 6

Conclusion

The goal of this bachelor project is to investigate the possibility to monitor the impact of the system evolution on the quality of the service, and in particular, on the performance for the deployed service. As explained in Section 1, there is no dedicated solution for monitoring the performance of Python web-applications. This thesis presents a solution by developing an automatic monitoring dashboard, that can be implemented in any web-service that uses Python and Flask.

This thesis describes exhaustively which frameworks and programming languages are used to develop the automatic monitoring dashboard. A lot of different programming techniques are used to develop the dashboard. All those techniques are also described in detail in the sections above.

The automatic monitoring dashboard is created in such a way, that it is possible to configure the dashboard to the needs of the maintainer. Once the configuration is successful, the dashboard is extended to the existing Flask application. This has been done to make the dashboard as easy-to-use as possible.

The dashboard has been deployed in the Zeeguu web-service for 28 days. In those days, $\sim$ 33,000 requests have been collected to investigate the impact on the performance. All data is visualized in various graphs. Those graphs allow the maintainer of the dashboard to detect differences in execution times across various versions. With the same approach it is possible to detect differences in execution times across various users.

With the visualization of the graphs, it is possible to monitor the impact of the system evolution of the quality of the service. Various results have been presented to observe differences in various execution times. The results conclude that, with the automatic monitoring dashboard, is it possible to visualize the impact on the performance of the system as the system evolves.

Chapter 7

Future work

As concluded in the previous section, the automatic monitoring dashboard succeeds in monitoring the performance of the system as it evolves. However, several improvements can be made.

The dashboard provides various graphs which shows the execution time across per users across several versions. Since most web-services are used by many users, the collected data may be too large to fit in a single graph. Currently, the size of the graph grows as the number of users grow. However, a graph in which more than 100 users are presented during 20 different versions, results in a bubble graph with 2,000 entries. It is not very useful to provide such a graph, since it impossible to maintain the overview. Further research can be made by providing an automatic selection procedure to reduce the entries in the graph, without losing monitoring information about possible outliers.

Another improvement in the field of the outliers has to be made. Currently the information that is presented is unformatted. Therefore, the information is not useful (yet) for detecting reasons for slow execution times of this outlier. An example of the data that is collected is presented in Figure 4.2. Further development is required to obtain a better overview of possible outliers and the reasons for having an execution time that is worse, compared to other requests.

Lastly, each dashboard is bound to the existing web-service. However, it is not rare for a company to have multiple web-services. In order to obtain the overview of all dashboards, a meta-dashboard can be set up. Such a meta-dashboard, possibly on a third party server, can be used to maintain a list of existing monitoring dashboards. With the meta-dashboard only one log-in procedure is required and easily shows which web-services are used the most.

Acknowledgements

In the first place, I want to thank Mircea Lungu, for assigning this bachelor project to Thijs and me. The process of developing the automatic monitoring dashboard was nice and attractive to do, since I really liked the project. Although the entire bachelor project took more than 10 weeks, the weeks flew over. I also want to thank Mircea for his enthusiasm and assistance in the meetings that we have to discuss the bachelor project.

Another person that I would really want to thank is Vasilios. His enthusiasm and experience contributes to a much better automatic monitoring dashboard. Vasilios has contributed to the project with a lot of professional feedback and he has provided useful tips in order to improve the final result. Also his positive feedback for the bachelor presentation has lead to this great result.

Last but not least, I want to thank Thijs for this contributions to the dashboard. Thijs implemented the testing-procedure for gathering results on the performance of unit-tests. His contributions to the dashboard has improved the design and the way of presenting the data in the graphs. Not only were his contributions professional, his personal ideas about the project in general were also great.

Bibliography

[1] Damyan Bogoev. How to monitor your flask application. `https://damyanon.net/flask-series-monitoring/`, 2015. [Online; accessed 20-June-2017].

[2] christabor. Flask jsondash. `https://github.com/christabor/flask_jsondash`, 2017. [Online; accessed 21-July-2017].

[3] Chris Davis. What graphite is and is not. `https://graphite.readthedocs.io/en/latest/overview.html`, 2008. [Online; accessed 2-June-2017].

[4] Gareth Dwyer. Flask vs. django: Why flask might be better. `https://www.codementor.io/garethdwyer/flask-vs-django-why-flask-might-be-better-4xs7mdf8v`, 2017. [Online; accessed 4-June-2017].

[5] Mircea F. Lungu. Bootstrapping an ubiquitous monitoring ecosystem for accelerating vocabulary acquisition. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, ECSAW '16, pages 28:1–28:4, New York, NY, USA, 2016. ACM.

[6] Matt Makai. Web frameworks. `https://www.fullstackpython.com/web-frameworks.html`, 2002. [Online; accessed 4-June-2017].

[7] Patrick Vogel, Thijs Klooster, Vasilios Andrikopoulos, and Mircea Lungu. A low-effort analytics platform for visualizing evolving flask-based python web services. In *Proceedings of the 5th IEEE Working Conference on Software Visualization (VISSOFT'17)*, September 2017.

[8] Wikipedia. Elasticsearch. `https://en.wikipedia.org/wiki/Elasticsearch`, 2017. [Online; accessed 10-July-2017].

[9] Wikipedia. Flask (web framework). `https://en.wikipedia.org/wiki/Flask_(web_framework)`, 2017. [Online; accessed 4-June-2017].

[10] Wikipedia. Google analytics. `https://en.wikipedia.org/wiki/Google_Analytics`, 2017. [Online; accessed 3-July-2017].

Appendix A

Source code setup-script

```
import setuptools

setuptools.setup(
    name="flask_monitoring_dashboard",
    version="1.8",
    packages=setuptools.find_packages(),
    include_package_data=True,
    zip_safe=False,
    url='https://github.com/mircealungu/
        automatic-monitoring-dashboard',
    author="Patrick Vogel & Thijs Klooster",
    author_email="p.p.vogel@student.rug.nl",
    description="A dashboard for automatic
        monitoring of python web-services",
    install_requires=[
        # for monitoring the web-service
    'flask>=0.9',
        # for database support
    'sqlalchemy>=1.1.9',
        # for generating forms
    'wtforms>=2.1',
        # also for generating forms
    'flask_wtf',
        # for generating graphs
    'plotly',
        # for parsing the config-file
    'configparser',
        # for logging extra CPU-info
    'psutil',
        # for hashing a string into a color
    'colorhash',
        # for submitting unit test results
    'requests']
)
```

Appendix B

Installation and configuration

One of the main nonfunctional requirements of the bachelor project is, that the monitoring tool should be 'easy to use'. Python supports a feature to achieve this simplicity. This can be done by creating a setup script. With the setup script, the dashboard can be installed without much effort. More information on installing and using the project can be found at section B.2.

The main purpose of the setup script is to describe the module distribution to the Distutils[1], so that the various commands that operate on the modules do the right thing. Our setup script can be found in Appendix A.

Most setup scripts only contain a call to the setup function. Arguments to this function are the name of our project, version, which packages to export, data files, authors, contact information to the authors, descriptions and other dependencies. The dashboard exports all markup-files, HTML-templates, JavaScript-functions and other files for designing our dashboard.

B.1 Virtual Environment

One of the arguments in the setup script is a list with dependencies of the project[2]. Each entry in the list consists of the name of the dependency and possible a lower- and an upper version.

Now suppose that a developer has two projects which both uses a list of dependencies. It could be that project A uses dependency A with a minimum version of 2.0, while project B uses dependency A with a maximum version of 1.9. Switching from project A to B requires to downgrade dependency A and switching back requires to upgrade dependency A. Such a conflict can be clumsy for the developer of project A and B.

Python offers a solution to resolve this type of conflict by using a Virtual Environment. A Virtual Environment is a tool to keep the dependencies required by different projects in separate places, by creating

[1]This is the standard tool for packaging in Python.
[2]A description of all dependencies can be found in section 4.3.

virtual Python environments for them. A Virtual Environment can be created using *virtualenv*. This module is by default installed in Python 3, but on other versions it can also be installed using the following command:

```
$ pip install virtualenv
```

Once it is installed, you can create a new virtual environment using the command:

```
$ mkdir my—environment
$ cd my—environment
$ virtualenv ENV
```

The commands above creates a virtual environment with Python 2 as the interpreter. Since the use of Python 2 is discouraged, the following commands create a Python 3 virtual environment:

```
$ mkdir my—environment
$ cd my—environment
$ virtualenv —p python3 ENV
```

The last line (in both examples above) creates a directory 'ENV' inside the directory 'my-directory' and creates the following:

- *ENV/lib/* and *ENV/include/* are created, containing supporting library files for a new virtualenv python. Packages installed in this environment will live under *ENV/lib/pythonX.X/site-packages/*.

- *ENV/bin* is created, where executables live - noticeably a new python. Thus running a script with *#! /path/to/ENV/bin/python* would run that script under this virtualenv's python.

- The crucial packages *pip* and *setuptools* are installed, which allow other packages to be easily installed to the environment. This associated pip can be run from *ENV/bin/pip*.

Once the virtual environment is created, it can be activated using the following command:

```
$ source ENV/bin/activate
```

Now, the virtual environment is up and running. If you want to shut it down, you can use the following command:

```
$ deactivate
```

B.2 Downloading

Once the virtual environment is created and running, the monitoring tool can be downloaded from Github manually, or automatically using the following command:

```
$ git clone
https://github.com/mircealungu/automatic—monitoring—
dashboard.git
```

The command creates the folder 'automatic-monitoring-dashboard' including the setup script.

Another useful command is to install the monitoring tool via PyPI, the Python Package Index. Using PyPI, the monitoring dashboard can be downloaded and installed using one command:

```
$ pip install flask_monitoring_dashboard
```

B.3 Installing

Installing the monitoring tool (that is downloaded via Github) can be done using the command:

```
$ python setup.py install
```

After the installation completed, the installed project can be viewed in the folder:
'ENV/local/lib/pythonX.X/site-packages'. A nice feature is to install the project in development-mode, using the command:

```
$ python setup.py develop
```

This command installs all dependencies, but not the project itself. Instead, it creates a symbolic-link to the source packages. This allows the developer to continuously modify the project, without installing the project every time a change has been made.

B.4 Configuration

Once the project has been successfully installed, it can be used in any Flask application. The project achieves simplicity by decoupling the monitoring tool as much as possible from the existing Flask application. The implementation of the monitoring tool requires only two lines of code. The following script binds the dashboard to the flask application:

```
from flask import Flask
import dashboard

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

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

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

The dashboard adds several pages that can be visited using the URL `http://localhost:5000/dashboard`. Flask allows to have only one endpoint per URL. So, suppose an existing project already has an URL `/dashboard`, then that endpoint cannot be visited and thus would make an existing project a bit less productive. This, and other possible conflicts can be resolved by configuring the dashboard by using a configuration file. The setup (including the configuration) requires now three lines of code:

```
from flask import Flask
import dashboard
```

```
dashboard.config.from_file('<path>/config.cfg')

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

B.4.1 The configuration in more detail

A possible implementation for the configuration-file could be:

```
[dashboard]
DATABASE=sqlite:///flask-dashboard.db
CUSTOM_LINK=dashboard
APP_VERSION=1.0
```

At least the header `[dashboard]` is mandatory and there are a number of variables that can be added below this header:

- **DATABASE:** The value of this variable is the database location for the dashboard. The monitoring tool supports all major database types. Specifying the database location provides another advantage. Suppose that a project which uses the dashboard has two stages. A development-stage, for modifying the project and (possibly) adding new features. And a production-stage, where it is used by the intended users. The developer of the project want to separate the data collected from both environments. Then, the developer has to use two database-files, one for each stage. If no variable is specified, then the default database location is:

  ```
  DATABASE=sqlite:///flask-dashboard.db
  ```

- **CUSTOM_LINK:** Suppose an existing project has a web-service that uses many routings. One of those routings is `/dashboard`. Since this URL is also used by the dashboard, the dashboard can not be used. However, for this purpose, it is possible to configure every page from the dashboard to a different URL. This can be done by specifying a **CUSTOM_LINK**-variable. If no variable is specified, then the dashboard can be found at `/dashboard`.

- **APP_VERSION:** Suppose a developer want to measure if the execution time of a certain endpoint changes over different versions as the developer manipulates the code of a project. Then it is useful to tag a version to the project by configuring this variable. Allowing to tag versions to a project is one of the main functionality of the dashboard. If no version is specified, then the default version is set to '1.0'.

  It might be a bit clumsy to update the version-tag every time when the code has been manipulated. This is one of the observations that is made during the development of the dashboard. Therefore, the dashboard contains another functionality to specify the version of the project. It is possible to configure the git version of the project by location to the root of the git-directory. This can be done by configuring the following variable:

  ```
  GIT=/.git/
  ```

The version of the project is updated during the first request. Whenever a developer updates the project, commits the changes, and restarts the application, a new version is automatically detected.

A disadvantage of using versions in general is that whenever the version is updated (either by using the **APP_VERSION**-variable or the **GIT**-variable), not every endpoint might be manipulated. The graphs that are being rendered do not keep this in mind and show the execution time for every version, whether the code is manipulated or not. In case of the latter, the graphs shouldn't produce different results.

- **USERNAME** and **PASSWORD:** An username and password can be specified to login into the dashboard. Without configuring the username and password, the user of the dashboard must still login before having access to the dashboard. In that case, the default values for the username and password are both 'admin'. Since the username and password are stored in plain text, the configuration file must not be published, otherwise everyone has can login into the dashboard.

  Two similar variables are **GUEST_USERNAME** and **GUEST_PASSWORD**. These variables are configured to provide a guest to login into the dashboard. A guest can only see the graphs, but cannot download the exported data, or change which rules are monitored. By default, the username and password for guests are 'guest' and 'guest_password'.

- **OUTLIER_DETECTION_CONSTANT:** Another functionality of the dashboard is to provide extra information whenever an outlier in the graph is detected. This information includes: request-values, request-headers, request-environment and request-URL But also the CPU percent, memory and the stack-trace.[3]. Whenever the application has received a number of requests for that endpoint, the dashboard computes the average execution time for that endpoint (the average is updated whenever a new request is processed). When the execution time is more than this constant times the average, the dashboard sees this request as an outlier and extra information is logged. Outlier detection can help in investigating why the request took more time to process. The default value for the outlier_detection_constant is 2.5.

[3]Those three values are calculated at the moment of processing the request

Appendix C

Case study

Zeeguu is a research project designed to speed and fun up vocabulary learning in a new language. According to [5], the approach is designed around three fundamental principles:

1. Unlike most traditional learning materials based on pre-defined, standardized and let's admit it: boring texts, we think that you should read only the stuff that you love: eBooks, blogs, articles and news in the desired language. In each of these contexts and on any chosen device, you should have translations of unfamiliar terms "at the touch of a fingertip".

2. Words that you have once translated, will be available to you always (read as "from any of your devices") and they will go into your own, private dictionary. The information about the context where you encountered a given new word will also be synchronized among the applications and devices. There is no doubt we understand and remember so much better words in context.

3. Based on the texts you read, the words you translate, and lists of word frequencies, Zeeguu will use machine learning to build a model of your current vocabulary in the target language. Applying this model and using spaced repetition, you will receive personalized exercises that will help you learn, practice and retain the new words, ranked according to their importance.

Zeeguu is the perfect test-case for the project, because it is build with Flask and the number of users is not too large. This keeps the data that is being collected reasonable small. In a small dataset it is easier to remain the overview. On the other hand the dataset is large enough to make statistical conclusions about the service performance during the lifetime. This is one of the major research questions for this thesis.