



UNIVERSITY OF GRONINGEN

BACHELOR THESIS

Automatic Monitoring of Test Performance Evolution for Web Services

Author:
Thijs KLOOSTER

Supervisors:
Dr. Mircea LUNGU
Dr. Vasilios ANDRIKOPOULOS

*A thesis submitted in fulfillment of the requirements
for the degree of Bachelor of Computing Science*

in the

Software Engineering and Architecture Group
Department of Computing Science
Faculty of Science and Engineering

July 30, 2017

University of Groningen

Abstract

Faculty of Science and Engineering
Department of Computing Science

Bachelor of Computing Science

Automatic Monitoring of Test Performance Evolution for Web Services

by Thijs KLOOSTER

This thesis presents a novel library for Python-based Flask applications to gain insight into the evolving performance of a web service. As of yet, there is no library that supports the developer of a Flask web application to track its performance during the development lifetime of the project.

In order to track the performance, two solutions are possible. One is observing the evolution of unit test performance as the project evolves over time, the other is observing the evolution of live performance of a deployed service as this service evolves over time. This thesis presents and evaluates a system that enables the former: monitoring the evolution of the system via the monitoring of the test cases.

As a case study for testing the tool, a platform for vocabulary learning in a foreign language was utilized. It is a web application written in Python using Flask. Results of deploying the tool on this platform are discussed here as well.

Acknowledgements

I would like to thank Mircea Lungu and Vasilios Andrikopoulos for their support and assistance during this project. Without them, this project would not have been possible. Also, they co-authored a paper (Lungu et al., 2017) about the tool presented in this thesis, which will be published by VISSOFT, a conference on software visualization.

I would also like to thank Mircea Lungu, for his support in deploying the system presented in this thesis in the context of the Zeeguu API, to be used as a case study.

Lastly, I would like to thank Patrick Vogel, with whom I worked jointly on the implementation of the tool described in this thesis.

Contents

Abstract	1
Acknowledgements	2
1 Introduction	5
2 Background and Related Work	7
2.1 Web Services	7
2.1.1 Python web services	7
2.1.2 Flask	7
2.2 Service evolution	8
2.2.1 Git	8
2.2.2 TravisCI	9
2.3 Measuring service performance	10
2.3.1 Development environment	10
2.3.2 Production environment	10
2.4 Unit testing	10
2.4.1 Creating tests	11
2.4.2 Running tests	12
2.5 Related work: Software performance visualization	13
2.5.1 Visualizing parallelism	14
2.5.2 Visualizing project evolution	14
3 Implementation	16
3.1 The Dashboard	16
3.1.1 Solution design	16
3.1.2 Data visualization	18
3.1.3 Technology stack	19
3.1.4 Dashboard look	21
3.2 Dashboard usage	25
3.2.1 The Flask app	25
3.2.2 Configuration options	25
3.2.3 User variable	27
3.2.4 Binding	28
3.2.5 Dashboard routes	28
3.2.6 Travis integration	30
4 Evaluation	32
4.1 Case study	32
4.2 Results	33
4.2.1 Service utilization	33
4.2.2 Endpoint performance	35
4.2.3 User experience	37

4.2.4	Unit test performance	38
5	Conclusion and Future Work	40
5.1	Conclusion	40
5.2	Future Work	41
5.2.1	Case studies	41
5.2.2	Meta-dashboard	41
5.2.3	Error tracking	41
5.2.4	Flask core	42
A	Availability of Flask Dashboard	43
	Bibliography	44

Chapter 1

Introduction

This digital age is upon us. More and more people are browsing the Internet and with that, the number of web services is growing rapidly. Web services can be for example sources of information, sources of entertainment or social platforms. Of course, users of these services want to have the best possible user experience. This means that the aim of the developer should be to achieve the highest possible service quality. This includes, but is not limited to, the very important aspect of performance of the service. This thesis focuses on monitoring the performance of a web service throughout its development lifetime. The research question that this thesis will answer is:

"How to create a tool that allows the automatic performance monitoring of evolving unit tests for Flask services in a way that affects these services in the least amount possible?"

There are two ways of monitoring the impact of the system evolution on the performance of the service. One is observing the evolution of unit test performance as the project evolves over time, the other is observing the evolution of live performance of a deployed service as this service evolves over time. This thesis presents and evaluates a system that enables the former: monitoring the evolution of the system by monitoring its unit tests. The thesis by Patrick Vogel presents and evaluates a system that enables the latter: monitoring the evolution of the system by monitoring its live deployment (Vogel, 2017).

Python has become one of the most popular programming languages lately. In fact, the TIOBE Index¹ shows Python as the fourth most popular language as of June 2017. Python is also a very popular language in the programming of web applications. There is a library for Python called Flask, which is quite popular and free-to-use. It allows the developer to create Python web services with very little effort. The tool presented in this research is aimed at these web applications. The tool is implemented using Python and Flask as well, in order to easily extend the monitored service with a monitoring dashboard.

This thesis contains background information and related work in Chapter 2. Then, it talks about the implementation of the tool created during this research in Chapter 3. It continues talking about the evaluation of the created tool in Chapter 4, by detailing the results of a case study that has been done in this research. Lastly, a conclusion is given along with possible future work in Chapter 5.

¹TIOBE programming community index is a measure of popularity of programming languages (<https://www.tiobe.com/tiobe-index/>)

The work presented in the thesis by Vogel (2017) in combination with the work presented in this thesis resulted also in the publication of a paper written by Lungu et al., 2017. This paper will be published by VISSOFT², a conference on software visualization. The paper presents Flask Dashboard (the tool presented in the thesis by Vogel and this one), a plug-in for Python-based Flask web applications that monitors them and provides their developers with information about the performance of these applications. The paper also talks about a case study in which the plug-in has been tested, and that based on the results of these tests, the plug-in has been improved.

²<http://vissoft17.dcc.uchile.cl/>

Chapter 2

Background and Related Work

2.1 Web Services

The biggest part of the web consists of services that entertain users, allow their users to gather information, or act as a social platform. All of the aforementioned possibilities need some kind of server, which enables users to find these services on the web and actually use them. Web services are defined by W3C¹ as a software system designed to support interoperable machine-to-machine interaction over a network (Haas, 2004).

These services use the Hypertext Transfer Protocol (HTTP) for communication between the service requester and the service provider. Data that is being exchanged can be for example XML or JSON, or HTML in combination with CSS and JavaScript, which a browser of a client (the requester) will render into the requested web page.

The architecture that is being used more and more for achieving this is Representational State Transfer (REST). This architecture consists of a set of predefined stateless operations, some of which correspond to the HTTP verbs `GET`, `PUT`, `POST`, `DELETE`. These enable the client to access and/or manipulate the data on the server. (Fielding et al., 1999)

2.1.1 Python web services

Python has been around for quite some time now, as it was first released in 1991. This programming language emphasizes code readability by using white space to delimit code blocks. It has a syntax that allows the programmer to say more with less lines of code than other languages would need to express the same thing.

Creating web applications in Python is made relatively simple also due to the number of web frameworks available for this language. These include Django², Pyramid³ and Flask⁴. All of these frameworks aim to improve the simplicity of setting up a Python-based web service.

2.1.2 Flask

Flask is called a micro web framework, since it is a very lightweight and minimalistic framework for creating Python-based web applications. It is based on Werkzeug⁵ and Jinja 2⁶. Werkzeug is a Web Server Gateway Interface (WSGI) utility library. WSGI has been adopted as the standard for Python web application development.

¹World Wide Web Consortium (<https://www.w3.org/>)

²<https://www.djangoproject.com/>

³<https://trypyramid.com/>

⁴<http://flask.pocoo.org/>

⁵Werkzeug WSGI library (<http://werkzeug.pocoo.org/>)

⁶Jinja 2 template engine (<http://jinja.pocoo.org/>)

Jinja 2 is a full featured template engine for Python. To create a simple "Hello World" web application using Flask, one only needs five lines of Python code (Ronacher, 2010):

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

@app.route("/")
def hello():
    return "Hello World!"
```

When running this code, this live "Hello World" application will be available by default on `localhost:5000/`. It will only have one page, which would be the index page (`/`) of the application. It will render the plain text "Hello World" on the page in your browser.

For the implementation of the tool presented in this research, Flask is used to create a dashboard to let the developer interact with the tool in a user-friendly way. Since the tool is made to also monitor Python-based Flask applications, extending such an application with the monitoring dashboard will be made easier.

2.2 Service evolution

During the development lifetime of a piece of software, there will be updates that should make the software better in some way (Papazoglou, Andrikopoulos, and Benbernou, 2011). These updates come in the form of new versions of the software. For example, if a bug is found, oftentimes the developers will try to fix this bug. After they are confident their changes to the software got rid of the bug, they have a newer version of their software. There are version control systems that provide a way for the developers to track these versions of their evolving piece of software.

2.2.1 Git

One example of a version control system is Git⁷. It is free, fast, open source and probably by far the most popular choice. It is used for tracking changes in files as well as allowing easy collaboration by more people on the same project or even on the same file. Git was created in 2005 for the development of the Linux kernel (Chacon and Straub, 2014).

GitHub⁸ is a web service that facilitates the hosting of repositories that are using version control (using Git). A repository is used for storing the source code of a software project, where collaborators can update the source code by creating 'commits' and pushing them to the repository. This way, evolution of a project is tracked by retaining the complete history of the repository and its commits.

Using Git is fairly easy. After installing Git, a repository can be initialized by creating a new project folder or by moving to an already existing one. Inside this folder, in the terminal, the command `git init` will initialize the repository. What this does is create a hidden folder named `.git`, in which the magic of Git happens. The repository is now set-up and ready to be used.

Initially, the repository has only one 'branch', named `Master`. This is by default the main branch of a repository. A branch represents an independent line of

⁷<https://git-scm.com/>

⁸<https://github.com/>

development within your project repository. Files on one branch can be changed without affecting the ones on other branches. Branches can be used to create different features of the software under development. After the feature is implemented, the branch containing this feature can then be merged back into the `Master` branch to add it to the main line of development.

A command that is very useful is `git status`, which will give some information about the current branch, as well as a list of added, modified or deleted files. Running this command after creating a new file results in Git saying that the newly created file is untracked. To track the changes made to this file, it can be added to Git by issuing the command `git add <filename>`. To add everything to Git, the command `git add .` can be used.

Now that new, updated or removed files are added to Git, they can be ‘committed’ to the repository. This will create a new ‘commit’, a snapshot of the files, and add it to the version control system. A commit has a branch it is applied on, the user that applied it, a time stamp, a commit message that describes the commit, and the list of changed files of course. These commits can be seen as versions of the software project. Git will automatically hash these commits based on the contents and the commit message, resulting in a unique character string that represents the commit. This hash could be used to denote the version of the software, since it describes a snapshot of it during the development phase.

Updates from collaborators can be downloaded to the local repository by issuing `git pull`. Updates in the local repository can be uploaded to enable collaborators to download them by issuing `git push`. Using these commands, commits will be downloaded and uploaded, respectively. Differences between the last commit and newly downloaded updates can be viewed by issuing `git diff`.

The commands described here are only a small subset of the ones available through Git. This makes it a great free and open source version control system to be used for software projects under development.

2.2.2 TravisCI

For software projects that have version control, there is the possibility of automatic continuous integration testing. The idea of continuous integration is that during the development of a piece of software, it would be tested for integration a few times a day for example. This way, developers gain more confidence in the newer versions of their software, in the sense that these updated versions do not break the functionality that was previously there. A form of testing the continuous integration would be running unit tests (see Chapter 2.4 for further explanation about unit tests).

TravisCI⁹ is an example of a continuous integration service that builds and tests software projects hosted on GitHub for free. To configure Travis to do this for your repository, a file named `.travis.yml` should be added to the root of the repository. This file specifies the different settings that the developer would like Travis to use, and what to build. An example of such a file would be something like:

```
services:
  - mysql

before_install:
  - mysql -e "create database IF NOT EXISTS test;" -uroot
  - pip install coveralls
```

⁹<https://travis-ci.org/>

```
language: python
python:
  - "3.6"

install: "python setup.py develop"

script: "./run_tests.sh"
```

This file specifies the services that Travis should use in order to test the project, the commands that it should run before the installation of the project, as well as the programming language and version of that language. It needs to specify how to install the project, and scripts to run the tests the project could contain.

When continuous integration testing is enabled on Travis for the repository you added the `.travis.yml` file to, whenever a new commit is made, Travis will automatically run the build and tests for this new version of the project. It will notify the developer whenever a build failed, so that he knows he has broken the build with his last commit. This way, the developer can fix this and create an updated version of the project again, or he could choose to revert back to some earlier version.

2.3 Measuring service performance

There are two environments in which one could evaluate the performance of a web service as the service evolves during its development phase (Ellison, 2015). First off, we have a kind of static evaluation in the development environment. This includes unit testing, which will be explained in the next section. Secondly, we have a dynamic evaluation environment, namely the production environment. Since the web is dynamic, the only way to obtain statistical information about a service under development is by actually deploying it and then collecting the information.

2.3.1 Development environment

The development environment is also called *sandbox*. This is where unit testing is performed by the developer. End-users have no access to this environment. Since this is the case, tests can be run as many times as the developer desires, and the service can be interrupted or down as long as he likes.

2.3.2 Production environment

The production environment is sometimes also called the *live* environment, due to the fact that end-users have access to this environment. They will be able to directly use the service that has been deployed in this environment. Usually, only major versions of the application will be deployed here, since deployment often requires a service interruption.

2.4 Unit testing

Unit Testing is a level of testing where the components of a piece of software are tested individually (Ellison, 2015). Its purpose is to validate that each component of the software performs like it is supposed to. It is a kind of white-box testing,

where the tester knows the internal implementation of the component being tested. During the evolution of a piece of software under development, if one unit test fails suddenly where it succeeded before, the developer knows that the newest version of the software has a bug that prevents the software from working the way it is supposed to.

2.4.1 Creating tests

Creating a unit test is relatively easy, as the internal implementation of the thing you are testing is known to you. There are libraries for writing unit tests for every major programming language. In the case of Python, the library called `unittest` would be a good example of such a unit testing framework. Setting up unit tests here is quite straightforward (PythonSoftwareFoundation, 2010):

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        with self.assertRaises(TypeError):
            s.split(2)
```

Naturally, we need to import the `unittest` library. Then, we can define a unit test class, which contains the actual tests the developer wants to run. Such a class will be a subclass of `unittest.TestCase`. Each test will be defined as a separate function inside this subclass. Following the naming convention, the names of the test functions will start with the word 'test'. Inside those tests, one has access to a set of tools one might need. In the example above, only a small subset of the available tools are demonstrated. This shows that using only a few of the tools will suffice to meet the needs of most testers.

In the example, we have three tests. The first will test whether or not the conversion to upper case of some string results in some other string. This is done by using the equality assertion. The second will test whether or not some string consists of upper case letters. This is done by either using the assertion of something being false, or the assertion of something being true. The third test will test the equality of two given values again, as well as testing if some function call with certain arguments will throw an error of some type.

These basic tools will grant the developer/tester to test almost everything they would want to. After creation of these tests, they have to be actually run, of course.

2.4.2 Running tests

To run a unit test case like the one used as an example in the previous section, two simple lines of code have to be appended to it:

```
if __name__ == '__main__':
    unittest.main()
```

This will run the unit tests in the class defined in this python script. When running unit tests from another script or from the interpreter for example, two different lines of code should be appended to the test case:

```
suite = unittest.TestLoader().\
    loadTestsFromTestCase(TestStringMethods)
unittest.TextTestRunner(verbosity=2).run(suite)
```

These will also give the tester a finer level of control over the tests to be run and the output verbosity of the tests.

Las but not least, a developer can use the test discovery functionality. This means that on the command line, when the working directory is your project directory, a command can be run to execute the unit tests. This command will search for all unit tests within your project directory and then execute them:

```
$ python -m unittest discover
```

When the names of the tests follow the naming convention, the test discovery functionality will be able to find them automatically. After the execution of the tests, the command line will output some information about the tests that have been run. When the `-v` flag is appended to the command, the output will be verbose.

The output of running the example will be something like:

```
test_isupper (test.TestStringMethods) ... ok
test_split (test.TestStringMethods) ... ok
test_upper (test.TestStringMethods) ... ok
```

```
-----
Ran 3 tests in 0.001s
```

```
OK
```

It shows that each of the three test cases have been run successfully. This means that all assertions were evaluated and the results were as expected. Now suppose we make one of the assertions fail, like for example changing the assertion in `test_upper`:

```
self.assertEqual('foo'.upper(), 'FOOD')
```

The output will now be something like:

```
test_isupper (test.TestStringMethods) ... ok
test_split (test.TestStringMethods) ... ok
test_upper (test.TestStringMethods) ... FAIL
```

```
=====
FAIL: test_upper (test.TestStringMethods)
```

```
-----
Traceback (most recent call last):
  File "test.py", line 6, in test_upper
    self.assertEqual('foo'.upper(), 'FOOD')
AssertionError: 'FOO' != 'FOOD'
- FOO
+ FOOD
?    +
-----
```

```
Ran 3 tests in 0.001s
```

```
FAILED (failures=1)
```

From this, we can see that only two of the three tests have passed, while the one that we altered, did not. Since one of the tests failed, the whole unit test run gets marked as being failed. Now that we have a failing test, the library gives additional information about why that test failed. It gives information about the file in which the failing test resides, the line number of the code that causes the test to fail, and the values that have been tested for the assertion. In this case they were 'FOO' and 'FOOD', which obviously are not equal. The difference between them is also shown.

This process of writing tests and running them will help the developer to gain more confidence in the current version of the software, when the tests all pass. Another advantage of unit testing is that this way, bugs can be found early in the development cycle, making them relatively less costly to deal with.

A software development process that makes use of unit tests the most would be Test-driven development (TDD). This process has a very short development cycle that consists of turning requirements into test cases first, after which the actual software is improved to pass these newly added tests. This way, all components of the software are tested so that they will behave exactly the way they are supposed to. This also inspires confidence in the software under development.

2.5 Related work: Software performance visualization

Techniques to visualize the performance of software are being used for quite a while now. An example of a tool that does this specifically for web services that use the Simple Object Access Protocol (SOAP) is Web Services Navigator (Pauw et al., 2005). It visualizes characteristics and components of service-oriented architectures. Information about web service requests and responses is tracked and stored. Then, the information is retrieved and visualized in order for the developer to gain more insight into the performance of the service under development.

Within the research area of software visualization, a few related techniques are summarized in the following sections. They consist of visualizing software parallelism and visualizing the evolution of a software project. This makes them relate to the research that is done here, since the former is a kind of software performance monitoring, while the latter is a form of software evolution visualization. These aspects are both part of this research as well.

2.5.1 Visualizing parallelism

The complexity of modern software rises continuously. Optimizing the performance of large programs becomes therefore more and more difficult. Performance analysis and visualization are therefore very important in the software development process. One way of performance analysis is to study execution traces. This means recording the history of process events and inter-process messages in a parallel application. Visualizations of these recordings will give the developer insight into the performance of such an application. (Isaacs et al., 2014)

A case study in the paper written by Isaacs et al. (2014) investigates an 8-ary merge tree calculation. The parallel execution traces of the original implementation of this calculation is visualized for 1024 and 16384 processes. The visualization shows that in both cases there is a lot of white space, indicating processes are waiting. Of course, the waiting time of processes in a parallel program should be minimized. This possibly makes the program as a whole execute faster and take up less resources. So, after this discovery, the developers improved the implementation of the calculation. A new visualization, using the same amount of processes, shows greater parallelism and less white space. Using this visualization technique, this case study shows that performance of software is improved by letting the developers see what aspects of their software could be performing sub-optimally.

To optimize the performance of software, one approach developers could use is parallelism. Tasks that a piece of software would normally run sequentially, might be able to run in a parallel manner just as well. Oftentimes this leads to faster run times of the same executed task. Therefore, this approach is one that could be used in certain cases to improve software performance drastically.

Unfortunately, identifying possible pieces of code that could be run in parallel has proven to be more difficult as the complexity of the software increases. There are automatic methods of finding these pieces of code, but they can only identify parallelism within simple loops or at the instruction level of program execution.

In a paper written by Wilhelm et al. (2016), a visualization framework for identifying parallelism in a piece of software is presented. This framework consists of three views for parallelism detection. It is part of *Parceive*, a tool that traces C, C# and C++ programs. Its goal is to assist in detecting parallelism opportunities at various granularity levels. It utilizes static binary analysis and dynamic instrumentation to collect the trace data. This is related to the research presented in this thesis, since it consists of performing static and dynamic performance analysis on a web service, saving the obtained data, and visualizing it in the most useful way.

2.5.2 Visualizing project evolution

Since the research in this thesis is about visualizing the performance of software as it evolves over time, some kind of version control system like GitHub would be appropriate to use. It has built-in graphs that show information about the repositories, but Feist et al. (2016) state that those fall short on showing the effective contributions made by each collaborator. Therefore, they present *TypeV*: a tool for visualizing Java source code repositories. This tool does not use information like additions and deletions in lines of code, but rather extracts detailed type information by using differences between abstract syntax trees (ASTs) of committed code. This way, additions and deletions in terms of declarations and type invocations are used and can

therefore be visualized this way too. The data can now be visualized by grouping by kinds of additions and deletions, which gives more insight in the software evolution.

Visualizing the evolution of a software project can also be done by using a tool called *SoftwareNaut*, presented by Lungu, Lanza, and Nierstrasz, 2014. The tool they present can be used for architecture recovery, which is needed for systems whose initial architectures have been eroded. When dealing with a large system, this cannot be done manually, without the assistance of tools for the recovery process. The tool supports this process by visualizing the software and by interactively exploring it. The tool consists of features like filtering and details on demand, as well as the support for multi-version software systems. The latter enables the evolution of the system to be visualized in terms of the analysis results of the tool. *SoftwareNaut* uses version information of a software system to be able to assist in the evolutionary software architecture recovery process. The tool presented in this thesis also uses version information of a software system, in this case to monitor the evolving performance of this system.

Chapter 3

Implementation

The main goal of this research is to develop a tool that enables its user to gain insight into the evolving performance of a web service during its development phase. This tool should perform analytics on the monitored service, while changing the existing service in the least amount possible. The usage of this tool should be made as simple as possible, so that the developer has to make the least amount of effort to monitor his service. In order to track the evolution of the system, the tool should allow integration with Git. To add the monitoring dashboard to the existing monitored service, the tool has to allow integration with Flask services. For the automatic unit testing, the tool should allow for integration with TravisCI.

The programming language Python was used in the implementation of this tool. This was decided since this language is very expressive and thus needs fewer lines of code than other languages might to achieve the same result. It also a language that scores high in the code readability aspect, which is also a good reason to work with the language. Python also has a vast collection of available libraries which are free to use, which makes the life of a developer easier still.

A library used in the implementation of this tool is Flask. Flask is a free-to-use library for Python that enables the developer to implement and deploy a web application with minimal effort, due to the way Flask can be set-up in a Python project. As mentioned before, with a few lines of code, a fully functional web application can be created and deployed. This saves the developer really a lot of effort, since using this library prevents the developer from reinventing the wheel.

3.1 The Dashboard

First off, a short description of how the dashboard was made is given. This includes the technology stack that was used, as well as some of the bigger design decisions that were made during the development of the tool.

3.1.1 Solution design

To measure execution times of a web service, in this case a Flask application, the tool should attach itself to this app. This way, when a request comes in, a timer could be started. Then when the app handled the request and sends back a response, the timer could be stopped. The measured time is the execution time the service needed to handle that specific request.

Data collection is needed to satisfy the requirement of performing analytics on a monitored service. To be able to collect data, the dashboard has to know when a request comes in on the monitored web service. To achieve this, wrapper functions are used. A wrapper function is a subroutine in a piece of software whose main purpose is to call another subroutine. This comes in handy, since when a request comes

in, a subroutine is called that handles this request and possibly render a HTML page as a response. If this subroutine could be wrapped by another, this other subroutine could then collect information about the incoming request and the request handling.

Python has a very nice design pattern that enables the achievement of this effect. This is called the `FunctionWrapper` pattern. What this does is demonstrated in the following example¹:

```
def trace_in(func, *args, **kwargs):
    print "Entering function", func.__name__

def trace_out(func, *args, **kwargs):
    print "Leaving function", func.__name__

@wrap(trace_in, trace_out)
def calc(x, y):
    return x + y
```

Calling `print calc(1,2)` would then result in the following output:

```
Entering function calc
Leaving function calc
3
```

This shows that by adding the `@wrap` annotation before a function definition, when that function gets called, the function wrappers are also executed. This is exactly what is needed for the data collection of the incoming requests. These data should then be persisted, which can be done by using a database. After persistence, the database lets the dashboard easily retrieve all of its past measurements for visualization purposes.

For the collection of data, the dashboard should support a few methods that take care of this:

The first is the collection of the last access times of all of the endpoints found in the web application that is being monitored. This means that the dashboard tracks every request the web service gets, and see for which endpoint the request is intended. It then updates the last access time of this endpoint to the time the request came in. The way this is done is by finding all of the functions that act as a route of the monitored web application. This means finding all of the functions that get executed when the corresponding requests to the web service are made. When these functions are found, the dashboard adds a wrapper function to each of them. This wrapper function then retrieves the current time and the name of the endpoint that it is wrapped around. Then, this information is stored in the database by updating the time stamp of the last access time of this specific endpoint.

The second method is the collection of execution time data. This is only done for the subset of the endpoints that the user of the dashboard selects. The dashboard adds another wrapper function to each of the selected endpoint functions. This wrapper function retrieves the current time before execution of the wrapped endpoint function, as well as the current time after the execution of the endpoint function. It can then calculate the difference between the two to get the time it took for the request to be handled by the corresponding endpoint function. This result

¹<https://wiki.python.org/moin/FunctionWrappers>

is then stored in the database by adding the execution time in combination with the executed endpoint to a table in the database.

The third method is the collection of unit test execution time data. Unit tests can be run automatically using a continuous integration tool like Travis, as mentioned in Section 2.2.2 on page 9. This enables the developer of the web application to let the unit tests be run automatically when a new commit is made on GitHub. This automatic run of the unit test suite could then post its results to the deployed dashboard, so that the user of the dashboard can inspect them. This way, the data collection for the unit testing can be implemented. Whenever Travis detects a new commit, it starts to build the service under development. Then, it searches for any and all unit tests that are present in the project, after which it runs them. Just before and just after a test is run, the current time is retrieved. The difference between these two times is then calculated as being the execution time of that specific run of that specific test. This result is then appended to a list of execution times. When all tests have been run and the execution times are all appended to this list, the list is then sent to the deployed dashboard. It then stores the results in its database, such that they are included in the visualizations on the dashboard. Section 3.2.6 on page 30 explains how to set-up the automatic unit test monitoring.

3.1.2 Data visualization

When the dashboard has collected some data, the visualizations become active. The types of graphs used in the dashboard are heat maps, bar charts, box plots, time series and dot plots.

The heat maps are used for visualizing the behavioral patterns of the users of the service. It shows times of the day where the service has higher loads and the times of the day where the service has lower loads. From this, usage patterns can be spotted. This could be used to adapt the service to increase performance during times of the day when there is a high load. Such a graph has the 24 hours of every day on the y-axis, while on the x-axis the previous 30 days are listed. So for every hour of every day, the corresponding cell is colored with a warmer color when the load is high, while the cell is a colder color when there is a low load at that time.

The bar charts give more insight into the distribution of the requests of the users of the monitored service over the different endpoints that are being monitored. For every day, there is a stacked bar, where each segment has a different color and represent the number of requests made for a certain endpoint that day. This gives insight into which endpoints receive the most requests each day and also which endpoints are becoming more or less popular.

The box plots are a type of graphical representation of groups of numerical data, by showing the quartiles of these data (see Figure 3.1). The quartiles of a data set are the three points that divide the data set into four equal groups. One of these points is the median, which is the 'middle' value of the data set. Box plots can also have lines extending from them, which are named 'whiskers'. They represent the variability outside the upper and lower quartiles. When present, outliers in a data set are represented by individual points. These box plots are used by the dashboard for depicting collected measurement data, in this case for certain groups of execution time data. They were chosen since they nicely show the spread of data in a set, which proves to be quite handy for quickly obtaining insight into the differences in execution times in this case.

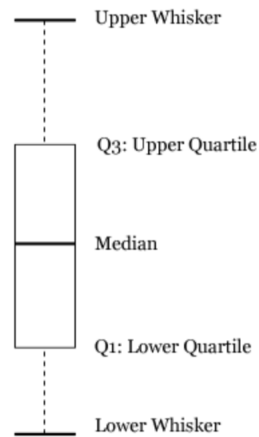


FIGURE 3.1: Example of a box plot with whiskers

The time series plots have on the y-axis the execution time in milliseconds or number of hits, while the x-axis contains the actual time series. This gives insight into the execution time evolution over time and the number of hits over time for a certain endpoint. This could show increases or decreases in endpoint popularity, as well as the days for which this specific endpoint has decreased performance and/or a high load.

The dot plots show the average execution time of a certain endpoint for a certain user. The user could be determined by the user name as well as the IP address the request was made from. These plots show differences in execution times over the evolution of the web service for a specific user, as well as differences in execution times for different users of the same version of the service. This could be used for spotting users that experience exceptionally high execution times, such that these users could be investigated further. Then the service could be updated to improve the experience for these users.

These are all the different types of graphs used by the dashboard to visualize the measured data. The graphs mentioned here are explained further with examples in Section 4.2 on page 33.

3.1.3 Technology stack

The programming language Python² was used in the implementation of this tool. A library for this language that was used is Flask³.

Another library that was used is SQLAlchemy⁴, which is a toolkit for using SQL in Python. Since the dashboard makes use of a database to store the measurements in, a library that provides a way of creating and interacting with such a database would save a lot of development effort. SQLAlchemy was used since this is a library that gives the developer many choices for the type of database he wants to use. Also, the library comes with an extensive amount of functionality that proves to be quite useful. The database type used by this tool is SQLite⁵, since this is a very simple solution that provides a database without being a client-server engine. This means

²<https://www.python.org/> (version 3.6.1)

³<http://flask.pocoo.org/> (version 0.12.2)

⁴<https://www.sqlalchemy.org/> (version 1.1.11)

⁵<https://www.sqlite.org/> (version 3.18.0)

that the dashboard can directly interact with the database file without setting up a database server, which makes it a simpler solution.

Plotly⁶ is the library that is being used by the tool for visualization of the obtained data. This library provides a way to easily create interactive graphs for on-line use, which is perfect for the implementation of the dashboard. This library saves a huge amount of implementation effort for the data visualization part, since it provides the developer with a large amount of different interactive graphs and charts that can be rendered and placed on a web page quite easily.

In addition to these main libraries that were used, a few others were used:

First off, the `configparser` library (in Python version 3.6.1) provides a way of easily parsing a configuration file. In our case this would be a file named `dashboard.cfg`, containing some custom settings of the dashboard.

Secondly, `psutil`⁷ was used for retrieving additional information about the current CPU usage and memory utilization. This comes in handy when some request to the web service being monitored gets flagged as being an outlier, in which case additional information is logged. This gives the dashboard user more insight into what might have caused that specific request to take longer to be handled than others of the same type.

Thirdly, `colorhash`⁸ was used for hashing the name of a page of the monitored web application to a color in the form of a red-green-blue (RGB) color value. This assigns a color to every page, which is used in the visualization of the measurement data of such a page. Hashing is used to map data of one form (in this case a string of characters) into a different form (in this case an RGB color value). This way, the same color is used in the visualization of the collected data of some web application page.

Finally, `requests` (in Python version 3.6.1) is used for its functionality to do HTTP requests to some web address. The specific functionality needed from this library is for doing a POST-request to the `/dashboard/submit-test-results` page of the dashboard, containing the collected results of running the test suite of the monitored application. This is explained further in Section 3.2.5 on page 30.

The software applications used for the development of the tool, were the following: The integrated development environment (IDE) that was used for implementing the tool is PyCharm⁹. This free-to-use IDE provides a Python source code editor, build automation tools and a debugger, as well as code completion and intelligent on-the-fly error checking. This makes it a very powerful tool for Python software development. These great features make choosing an IDE very easy.

To test the web application and actually see what it looks like when it is deployed, two different web browsers were used. One is Google Chrome¹⁰, which is the browser that has the greatest market share (netmarketshare.com, 2017). The other is Mozilla Firefox¹¹, which has the third greatest market share (netmarketshare.com, 2017).

⁶<https://plot.ly/> (version 2.0.12)

⁷<https://pypi.python.org/pypi/psutil> (version 5.2.2)

⁸<https://pypi.python.org/pypi/colorhash> (version 1.0.2)

⁹<https://www.jetbrains.com/pycharm/> (version)

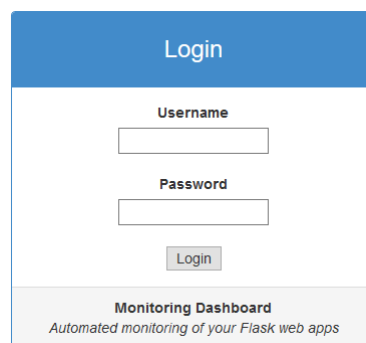
¹⁰<https://www.google.com/chrome> (version 59.0.3071)

¹¹<https://www.mozilla.org/en-US/firefox/> (version 54.0.1)

Another application that was used for working in collaboration on the implementation of the tool is Git¹². This provides a great way of version control and issue tracking, as mentioned in Section 2.2.1 on page 8.

3.1.4 Dashboard look

Having talked about the internals of the dashboard, now it is time for its appearance. This dashboard satisfies the requirement to enable its user to gain insight into the evolving performance of a web service during its development phase. When the dashboard is deployed and the user visits it, a login page is shown and the user has to login to be able to use the dashboard. This login page looks like Figure 3.2. The user should fill in the user name and the corresponding password, after which the login button should be pressed. The dashboard redirects the user to the measurements page by default, upon successful login.



The image shows a web form for logging into a 'Monitoring Dashboard'. The form is centered on a white background. At the top, there is a blue rectangular header with the word 'Login' in white text. Below this header, the form contains two text input fields. The first is labeled 'Username' and the second is labeled 'Password'. Both labels are in a small, bold, black font. Below the 'Password' field is a small, rectangular button with the text 'Login' in a light gray font. At the bottom of the form, there is a light gray footer area containing the text 'Monitoring Dashboard' in bold and 'Automated monitoring of your Flask web apps' in a smaller font below it.

FIGURE 3.2: Dashboard - Login page

The measurements page of the dashboard looks like Figure 3.3. On the top of the page, there is a bar that contains the name of the tool along with its short description. In the top-right corner of this bar, there is a drop-down menu where the user can go to the settings page or logout of the dashboard. On the left-hand side of the page, there is a navigation bar that enables the user to jump from one section of the dashboard to another. The three main sections are *Measurements*, *Rules* and *Testmonitor*. The page that is currently open is highlighted in this navigation bar. The remainder of the page contains the actual content. In the case of *Measurements*, the first page tab that is shown here would be the overview. This contains some information about the endpoints that are being monitored by the dashboard. The contents of this overview are explained further in Section 3.2.5 on page 28.

¹²<https://git-scm.com/> (version 2.10.1)

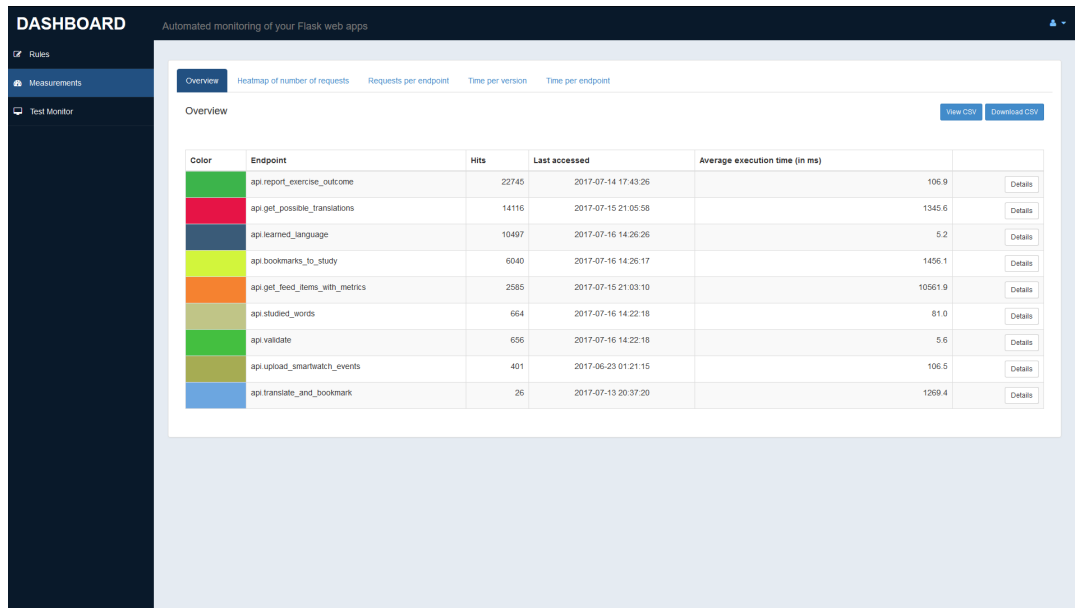


FIGURE 3.3: Dashboard - Measurements page

The first link in the navigation bar on the left-hand side of the page is the Rules page. This looks like Figure 3.4. This page contains a list of all of the rules of the service being monitored that the dashboard automatically found. The user can select any of these rules to enable the monitoring of this specific rule. When no rules are selected, no rules are monitored and so the dashboard does not collect request data. It collects the last access times for the endpoints, though.

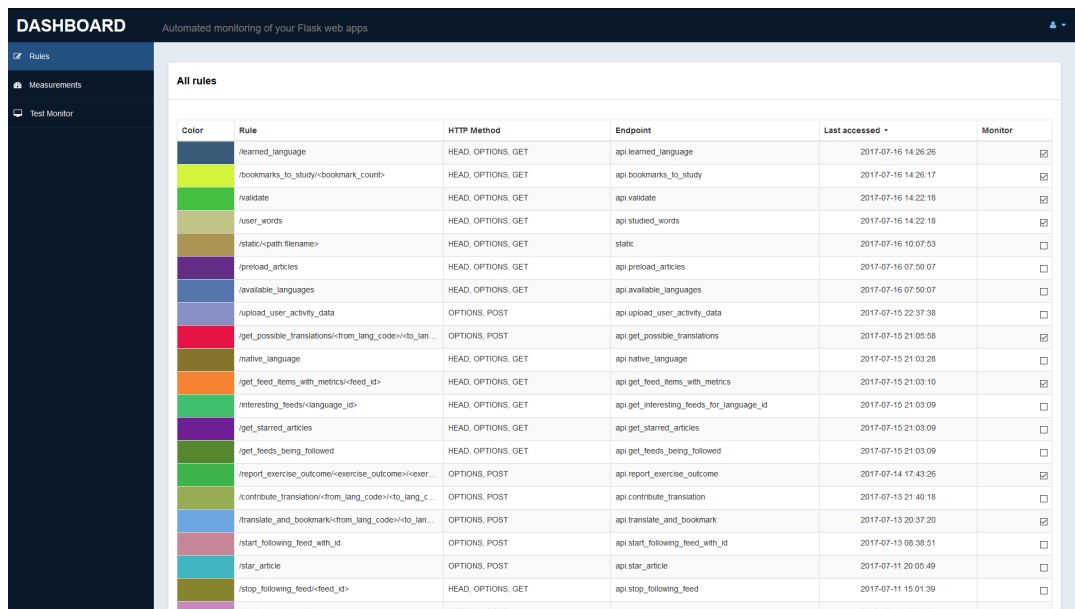
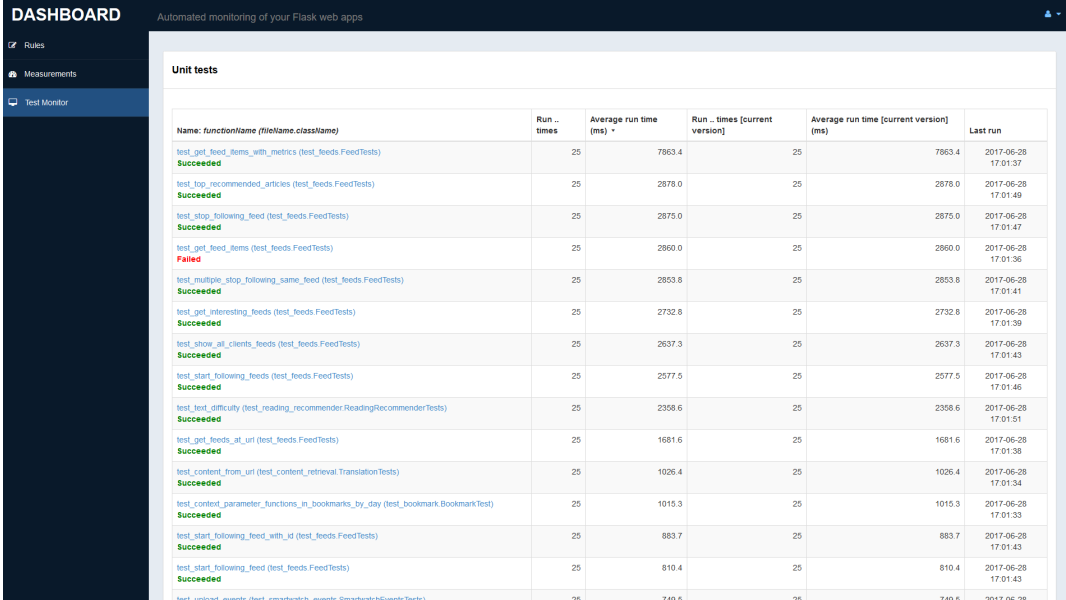


FIGURE 3.4: Dashboard - Rules page

The final link in the navigation bar is the one which leads to the Testmonitor page. This looks like Figure 3.5. This page contains all of the unit tests that the dashboard automatically found. When no tests have been run yet, the status would

be `Never` run in black. When the current status of some test is `Succeeded`, this is shown in green. When the current status of some test is `Failed`, this is shown in red. When the user clicks on a certain test, the user is redirected to the details page of that specific unit test. This is discussed further in Section 4.2, starting on page 33. Oftentimes, not all of the tests are relevant for the monitoring of web service evolution. Only the tests which are testing a single endpoint each should be used for the monitoring. This could be achieved by allowing the user to select a subset of the tests that have been found, just like the user can do with the endpoints. Only the selected tests could then be used for the evaluation of the service performance evolution.



DASHBOARD Automated monitoring of your Flask web apps

Unit tests

Name: function(name (filename.classname))	Run .. times	Average run time (ms) *	Run .. times [current version]	Average run time [current version] (ms)	Last run
test_get_feed_items_with_metrics (test_feeds FeedTests) Succeeded	25	7863.4	25	7863.4	2017-06-28 17:01:37
test_top_recommended_articles (test_feeds FeedTests) Succeeded	25	2878.0	25	2878.0	2017-06-28 17:01:49
test_stop_following_feed (test_feeds FeedTests) Succeeded	25	2875.0	25	2875.0	2017-06-28 17:01:47
test_get_feed_items (test_feeds FeedTests) Failed	25	2860.0	25	2860.0	2017-06-28 17:01:36
test_multiple_stop_following_same_feed (test_feeds FeedTests) Succeeded	25	2853.8	25	2853.8	2017-06-28 17:01:41
test_get_interesting_feeds (test_feeds FeedTests) Succeeded	25	2732.8	25	2732.8	2017-06-28 17:01:39
test_show_all_clients_feeds (test_feeds FeedTests) Succeeded	25	2637.3	25	2637.3	2017-06-28 17:01:43
test_start_following_feeds (test_feeds FeedTests) Succeeded	25	2577.5	25	2577.5	2017-06-28 17:01:46
test_test_difficulty (test_reading_recommender ReadingRecommenderTests) Succeeded	25	2358.6	25	2358.6	2017-06-28 17:01:51
test_get_feeds_at_url (test_feeds FeedTests) Succeeded	25	1681.6	25	1681.6	2017-06-28 17:01:38
test_content_from_url (test_content_retrieval TranslationTests) Succeeded	25	1026.4	25	1026.4	2017-06-28 17:01:34
test_content_parameter_functions_in_bookmarks_by_day (test_bookmark BookmarkTest) Succeeded	25	1015.3	25	1015.3	2017-06-28 17:01:33
test_start_following_feed_with_id (test_feeds FeedTests) Succeeded	25	883.7	25	883.7	2017-06-28 17:01:43
test_start_following_feed (test_feeds FeedTests) Succeeded	25	810.4	25	810.4	2017-06-28 17:01:43
test_upload_events (test_smartwatch_events SmartwatchEventsTests) Succeeded	25	749.5	25	749.5	2017-06-28

FIGURE 3.5: Dashboard - Testmonitor page

The link for the settings in the drop-down menu in the top-right corner of the page leads to a page showing the current dashboard settings, which looks like Figure 3.6. It shows the current link for visiting the dashboard, along with the current version of the monitored service. It shows the current user variable, the location of the database, the specified location of the unit tests of the project, and the login credentials of the user.

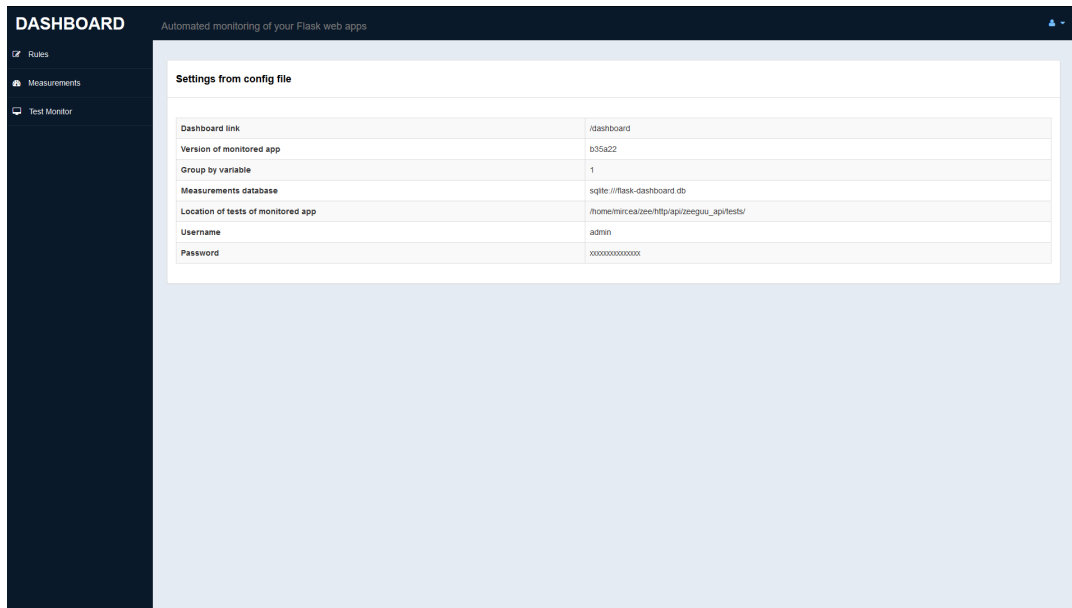


FIGURE 3.6: Dashboard - Settings page

Lastly, the design of the page containing the detailed results of a certain endpoint looks like Figure 3.7. It shows the version of the service where the endpoint first appeared in, along with the time stamp of when that version became active. This page has a tab pane where the user can go through the different visualizations for that specific endpoint. All of these visualizations are discussed in more depth and in the context of the case study discussed in Section 4.2, starting on page 33.

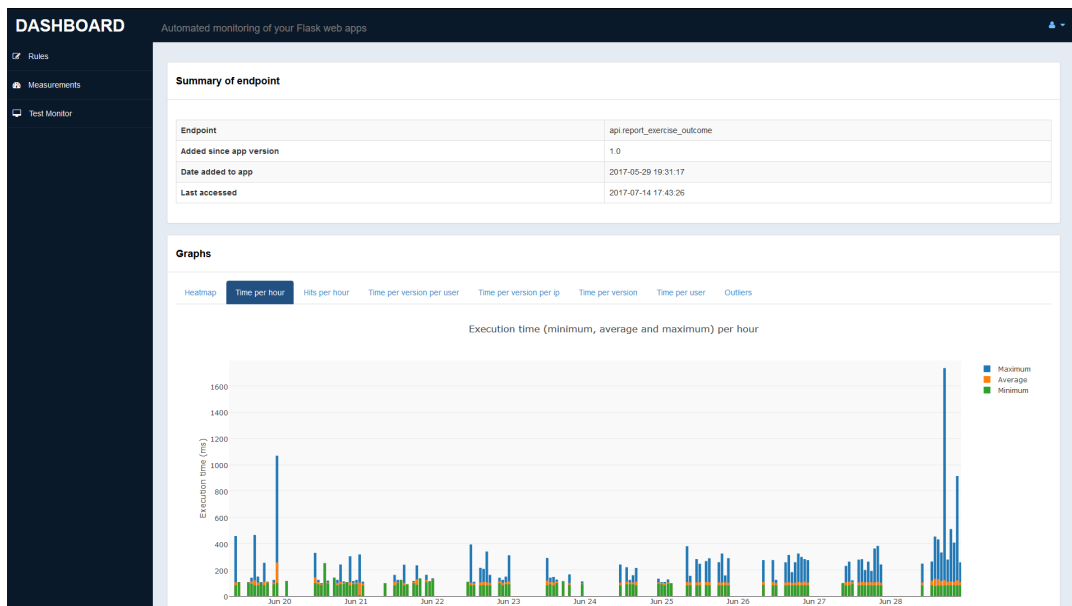


FIGURE 3.7: Dashboard - Endpoint page

3.2 Dashboard usage

3.2.1 The Flask app

A Flask application has at least a main Python script containing the code for setting up the app. To satisfy the requirement of integration with Flask services, the dashboard uses Flask as well. The often-used "Hello World" example was shown in Chapter 2 on page 7. To satisfy the requirements of changing the existing service in the least amount possible and that attaching the tool to a Flask app should be as simple as possible, the tool is implemented such that only two lines of code have to be added in the main Python script. In case of the example:

```
from flask import Flask
import dashboard

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

@app.route("/")
def hello():
    return "Hello World!"
```

The added lines are `'import dashboard'` to import the library, and `'dashboard.bind(app)'` to bind the tool to the Flask application.

3.2.2 Configuration options

The user of the dashboard can specify a number of configuration options. The dashboard can optionally be given a configuration file to read these options. This can be done by simply inserting one additional line of code into the main Python script. In case of our example, the code would become:

```
...
app = Flask(__name__)
dashboard.config.from_file('dashboard.cfg')
dashboard.bind(app)
...
```

To show all of the different configuration options that are available, an example of `dashboard.cfg` would be (where `<proj>` is the path to your web service project and `<serv>` is the URL to your live service):

```
[dashboard]
APP_VERSION=1.0
CUSTOM_LINK=dashboard
USERNAME=admin
PASSWORD=admin
GUEST_USERNAME=guest
GUEST_PASSWORD=guest_password
DATABASE=sqlite:///<proj>/dashboard.db
GIT=<proj>/git/
TEST_DIR=<proj>/tests/
N=5
SUBMIT_RESULTS_URL=<serv>/dashboard/submit-test-results
```

```
OUTLIER_DETECTION_CONSTANT=2.5
COLORS={'main':[0,97,255], 'static':[255,153,0]}
```

To explain this configuration file in more detail, each option is listed below:

- First off, using `APP_VERSION`, one can manually specify the version of the web service. This functionality can be used when there is no integration with Git. By default, this value is `1.0`.
- Then, with `CUSTOM_LINK`, the user can define the URL where the dashboard is available at. This can be necessary when the web service already has an endpoint named `dashboard`, which is the default link.
- A user can also set a custom user name and password for logging in to the dashboard as administrator, which by default is `admin` for both `USERNAME` and `PASSWORD`.
- Users that do not have administrative privileges can be given other login credentials. They can be specified by setting `GUEST_USERNAME` and `GUEST_PASSWORD`, which by default is `guest` and `guest_password`, respectively. Users that login using these credentials can only see the visualizations and the results of the monitoring, but cannot change anything in the dashboard.
- A custom `DATABASE` name and location can also be set. In this case, we use SQLite, so we prepend the location where we want the database to reside with `sqlite://`, and append the database name to this new string. By default, the value is `sqlite:///flask-dashboard.db`, which places the database relative to your project and name it `flask-dashboard`.
- A very important option is `GIT`, which specifies where the `.git` folder of your web service project resides. This is used by the dashboard in order to satisfy the integration with Git requirement and thus tracking the versions of the service automatically. By default, when this value is not specified, the dashboard uses the manual `APP_VERSION` for determining the version of the service. The dashboard uses the `.git` folder to automatically find the current `HEAD` of the repository, from which the most recent commit can be retrieved and used as the version of the service. When a new version of the service is deployed, the dashboard detects this and tag all new measurements with this new version.
- Also, a directory can be specified where the unit tests reside. By default, `TEST_DIR` has no value. When specified, the dashboard searches for unit tests within this directory. The pattern that the dashboard scans for is `*test*.py`, which means that it finds unit tests that are defined in Python files with `test` anywhere in the filename.
- Then, a number for `N` can be specified, which denotes the number of times each unit test should be executed when running the tests. By default, this is `1`, so each test runs only once. This functionality can be useful when users want more measurements of the same test, since they can vary due to load of the service, load of the CPU and memory usage for example.
- Also, a URL can be specified by setting `SUBMIT_RESULTS_URL`, where the unit test results should be submitted to. Since unit tests are oftentimes not run on the deployed service but instead in another (local) environment or for

example on Travis¹³, this link is necessary to be able to view the unit test results on the dashboard of the live service.

- A constant for outlier detection, `OUTLIER_DETECTION_CONSTANT`, can also be set. This constant is multiplied with the current average execution time of a certain endpoint. The resulting value is used as a threshold to determine if some request took longer than usual and if so, it gets flagged as an outlier and additional information about that specific request is collected.
- Lastly, the option to manually define a RGB color for a specific endpoint, user or version is available by setting `COLORS`. The value for this should be a Python dictionary¹⁴, which by default is an empty one. By default, semi-random colors are used. If a user wants to set the color of some endpoint named 'main' to blue, the key-value pair `'main': [0, 97, 255]` could for example be added to the dictionary.

3.2.3 User variable

For some endpoints, it might be that the performance depends on the user that made the request to that endpoint. Execution speed can depend on the way the endpoint is used, for example if one user has much more data than another. Endpoints that have to do something with these data obviously have higher execution time when there are more data. In order for the developer to gain more insight into why some user has a higher response time than another, functionality is added to the dashboard to achieve this.

The dashboard can be configured to associate a request with some user of the monitored service. The Flask architecture uses global request objects to store information about the requesting entity, which can be taken advantage of for linking a user to some request that is being monitored. This could be implemented as a for example turning a session ID into a user ID, by retrieving the session ID of the request first, after which the corresponding user ID is retrieved from the web service, given that session ID. A possible implementation for some specific Flask app could be something like:

```
def get_user_id():
    sid = int(flask.request.args['session'])
    session = User.find_for_session(sid)
    return user_id
```

Now that a function is defined for obtaining the user variable, it can be set in the configuration for the dashboard like so:

```
dashboard.config.get_group_by = get_user_id
```

This last line of code assigns the newly defined function to the dashboard, which in turn is able to group the monitored incoming requests by user. This gives the user the option to see a visualization where the measurements are grouped by user, to spot possible users which have a user experience that is worse than intended.

The complete code of our small example would now be:

¹³<https://travis-ci.org/>

¹⁴<https://docs.python.org/2/tutorial/datastructures.html#dictionaries>

```
from flask import Flask
import dashboard

app = Flask(__name__)

def get_user_id():
    sid = int(flask.request.args['session'])
    session = User.find_for_session(sid)
    return user_id

dashboard.config.from_file('dashboard.cfg')
dashboard.config.get_group_by = get_user_id
dashboard.bind(app)

@app.route("/")
def hello():
    return "Hello World!"
```

3.2.4 Binding

When binding to an app, routes for the dashboard are added to the existing Flask app. These routes are bindings of functions to their corresponding URLs. This way, when the URL for such a function is requested, the function gets executed. These additional routes make up the interactive dashboard.

The dashboard searches for all endpoints present in the Flask app. This basically means that it goes over all valid URLs of the service, and see what their corresponding functions are. The dashboard stores these in its database, along with whether or not that specific endpoint should be monitored. This way, the user can make a selection of endpoints that he is interested in. This can be done on one of the pages of the dashboard, which lists all endpoints that have been found and gives the user the option to check any endpoint for monitoring.

The last access time is tracked for every endpoint, the ones that are being monitored as well as the ones that are not. This way, the user already has some kind of notion about which endpoints are being used more frequently than others. The user can also decide based on this which endpoints should definitely be monitored and which ones have lower priority.

The dashboard also searches for unit tests within the project. When found, they are also saved in the database and listed on a different page of the dashboard. This way, users can already see the tests that can be run and monitored by the dashboard.

3.2.5 Dashboard routes

There are quite a number of routes that are added to an existing Flask app, when a dashboard is bound to it. By default, the base URL is that of the existing Flask app, with `/dashboard` appended to it. The main routes that are added are:

<code>/dashboard/login</code>	<code>-> log in to the dashboard</code>
<code>/dashboard/logout</code>	<code>-> log out of the dashboard</code>
<code>/dashboard/settings</code>	<code>-> show the current settings</code>
<code>/dashboard/rules</code>	<code>-> list all found endpoints</code>

<code>/dashboard/measurements</code>	-> show overall measurements
<code>/dashboard/result/<e></code>	-> show endpoint measurements
<code>/dashboard/testmonitor</code>	-> list all found unit tests
<code>/dashboard/testmonitor/<t></code>	-> show test measurements
<code>/dashboard/submit-test-results</code>	-> submit test measurements
<code>/dashboard/export-data</code>	-> export measurement data
<code>/dashboard/download-csv</code>	-> download measurement data

For most of the endpoints listed above, it is quite clear what they do. For the ones that are a bit more complex, an additional description is provided below.

- The `/dashboard/rules` page contains a table with all of the endpoints that the dashboard has found when scanning the monitored Flask application. For every entry of this table, the URL is given, along with the name of the endpoint. Every endpoint gets its own color, such that the visualizations of measurements of these endpoints are colored accordingly. With every endpoint, the allowed HTTP methods are also shown. For every endpoint that was already visited since the deployment of the dashboard, a last access time is also given. Last but not least, every endpoint has a box that can be checked when the user wants that endpoint to be monitored.
- The `/dashboard/measurements` page contains a tabbed view that consists of one tab with a table of a summary of the monitored endpoints, while the other tabs contain plots of the overall measurements of the service. The plots that are available here are named *Heatmap of number of requests*, *Requests per endpoint*, *Time per version*, and *Time per endpoint*.

The summary table has entries for every monitored endpoint. Its name, color and last access time are shown here again, but now also the total number of hits a specific endpoint has gotten is shown. For every endpoint, the average execution time is also in the table. Every entry has a link that redirects to the measurements and their visualizations of that specific endpoint. This page is explained next.

- The `/dashboard/result/<e>` page gives the user more detailed information about a specific endpoint. This page contains a number of plots that show all kinds of visualizations of the measured data for that endpoint. The plots that are available here are named *Heatmap*, *Time per hour*, *Hits per hour*, *Time per version per user*, *Time per version per ip*, *Time per version*, and *Time per user*.

Information about the endpoint is also shown, like the version of the service it first occurred in, along with the date it was added to the service. There is also a tab for gaining more insight into outliers, which are requests that took far longer than the average execution time for this specific endpoint. A constant can be specified by the user that is multiplied with the current average execution time. When a request takes longer to handle than this time, it gets flagged as an outlier and additional information is collected like request variables, CPU and memory usage, and the stack trace.

- The `/dashboard/testmonitor` page consists of a table with all of the unit tests that the dashboard has found. For every test, the current status is shown. This would be `Never run`, `Succeeded` or `Failed`. For each of the tests, the

number of times this test has been run overall is shown, as well as the number of times on the current app version. An overall average execution time along with the average execution time for the current app version is given as well. The final column specifies the time stamp of the time the test has been run last. In addition to this table, there is also a graph that shows box plots of the execution times per run of the test suite.

- The `/dashboard/testmonitor/<t>` page can be visited by clicking any of the unit tests in the table on the previously mentioned page. It shows a box plot graph of the execution times for every run of the test suite again, but this time only for the selected test. This enables the user to gain more insight into the performance of a specific test.
- The `/dashboard/submit-test-results` page cannot be visited by the user of the dashboard, but instead it is there so that an automatic run of the test suite is able to post the test results to the deployed dashboard. By doing so, this enables the user of the dashboard to see the visualized test result data on the deployed dashboard.

3.2.6 Travis integration

Now that the dashboard is deployed and working properly for the web service that it is configured to monitor, it is time to configure Travis to automatically run the unit tests of the project and post the test results to the deployment of the dashboard. This satisfies the requirement of allowing for integration with TravisCI. This way, the unit test results are available to the live dashboard to create visualizations of for the user to inspect. In order to achieve this, the project repository should be hosted on GitHub, such that the developer can link the GitHub account to Travis. This way, automatic continuous integration testing and automatic unit testing can be turned on for the repository. For every new commit that is made to the repository, Travis detects it and build the project according to a build specification that should be made in a file called `.travis.yml`. This file should reside in the root of the project repository. An example of such a file is given in Section 2.2.2 on page 9.

A few steps have to be taken in order to get Travis working with the dashboard and the monitored service. In the future, these steps could be made easier by facilitating this configuration differently, but for now, the following steps have to be taken:

1. The first is copying the file called `collect_performance.py` to the directory where the `.travis.yml` file resides. The `collect_performance.py` is downloaded along with the dashboard. It is a python script that can be run by Travis in order to find all unit tests of the project, run them and post the results to the link to the deployed dashboard.
2. Now, the configuration file for the dashboard (`dashboard.cfg` for example) should be updated to include at least three additional values, `TEST_DIR`, `SUBMIT_RESULTS_URL` and `N`. The first specifies where the unit tests of the project reside, the second where Travis should upload the test results to, and the third specifies the number of times Travis should run each unit test, as discussed in Section 3.2.2.
3. Then, a dependency to the dashboard should be added to the `setup.py` file of the web service. This is so that Travis tests the continuous integration of

the web service while it includes the dashboard plug-in. This can be done by simply adding `flask_monitoring_dashboard` to the `install_requires` list argument of the `setuptools.setup()` function call.

4. Lastly, in the `.travis.yml` file, two commands should be added to the script section:

```
script:
- export DASHBOARD_CONFIG=dashboard.cfg
- python ./collect_performance.py
```

The first command specifies an environment variable that contains the configuration file to be used. The second command runs the python script that handles the automatic unit testing and the sending of the results to the dashboard. The rest of the `.travis.yml` file could be used to specify the build Travis has to do to test the continuous integration or run some other type of testing.

Whenever a new commit is made to the repository, this is detected by Travis and the unit tests are run the specified amount of times. The results are added to a list, after which the list is sent to the deployed dashboard. For every test, the result consists of the name of the test, its execution time for that run, the time stamp when that run took place, whether or not the test passed, and which iteration of the tests the result belongs to.

Chapter 4

Evaluation



FIGURE 4.1: The Zeeguu platform

4.1 Case study

The case study of this research is the deployment of the dashboard on a live web service that has quite a few users. This web service is called Zeeguu¹ (see figure 4.1). It is a platform for accelerating vocabulary learning in a foreign language (Lungu, 2016). At the moment of writing this thesis, the platform has about 200 active beta-testers.

The core of the system consists of an application programming interface (API) implemented with Python and using Flask. Together with a number of connected applications, it offers three features for the language learner:

1. **Contextual translations** - Providing effortless translations for the texts that prove to be too difficult for the reader.
2. **Personalized exercises** - Exercises that are generated based on the preferences of the learner. (Avagyan, 2017)
3. **Article recommendations** - Recommendations for articles which are deemed to be on an appropriate difficulty level for the reader. (Brand, 2017)

The core of this system provides the functionality to make these three features possible. The API of the system will be used in this research as a case study. Three

¹<https://www.zeeguu.unibe.ch/>

major versions of the dashboard were deployed on this API. The first version contained the functionality for monitoring the selected endpoints, as well as some basic graphs visualizing the execution times of these endpoints. The second version contained an improvement on the visualization aspect, consisting of additional and neater graphs. This version also came with a web site design update, making the dashboard look more professional. The third version consisted of the functionality for flagging outliers in the requests, as well as the functionality for integration with Travis for unit test monitoring. All of the obtained results will come from the actual deployment of the dashboard on the live Zeeguu API.

4.2 Results

After deploying the monitoring tool on the platform as described in the previous chapter, the maintainer of Zeeguu selected a few of the endpoints that he was interested in. These are the ones that the dashboard has been monitoring for five weeks now. On May 29, 2017, the dashboard was first deployed on the case study. On July 2, 2017, the case study ended, but only in the sense that results of the case study were taken from these five weeks. The maintainer of Zeeguu wanted the deployment of the tool to continue its monitoring, and for it to stay available for viewers which can then see the visualization part of the tool in action. The screen-shots of the dashboard and its visualizations were taken on July 16, 2017.

In the following sections, all of the results of the deployed dashboard are presented. The appendix on page A describes how to see the up-to-date visualizations of the deployed dashboard on this case study.

There are four main visual perspectives that the dashboard provides. One which presents usage information about all of the endpoints that were selected to be monitored, one which presents their execution times, one which presents user-specific performance experience information, and one which presents the performance unit tests in the form of their execution times.

4.2.1 Service utilization

A basic insight that a developer needs will be that of service utilization. The dashboard contains a few graphs that may help the developer in gaining this insight.

First off, there is a stacked bar chart that visualizes the number of hits to the selected endpoints, grouped by day. An example of this is shown in Figure 4.2. On the y-axis, the days are enumerated. The x-axis contains the number of requests. The different colors of the segments of the bars represent the different endpoints that are being monitored, as shown in the legend. This visualization shows that the service has 12.000 hits per day at its peak. Since the requests are grouped by endpoint, the activity distribution over the endpoints can be observed from this visualization as well. Also, the popularity of some endpoint can increase and decrease over a number of days, which will also show up in this graph.

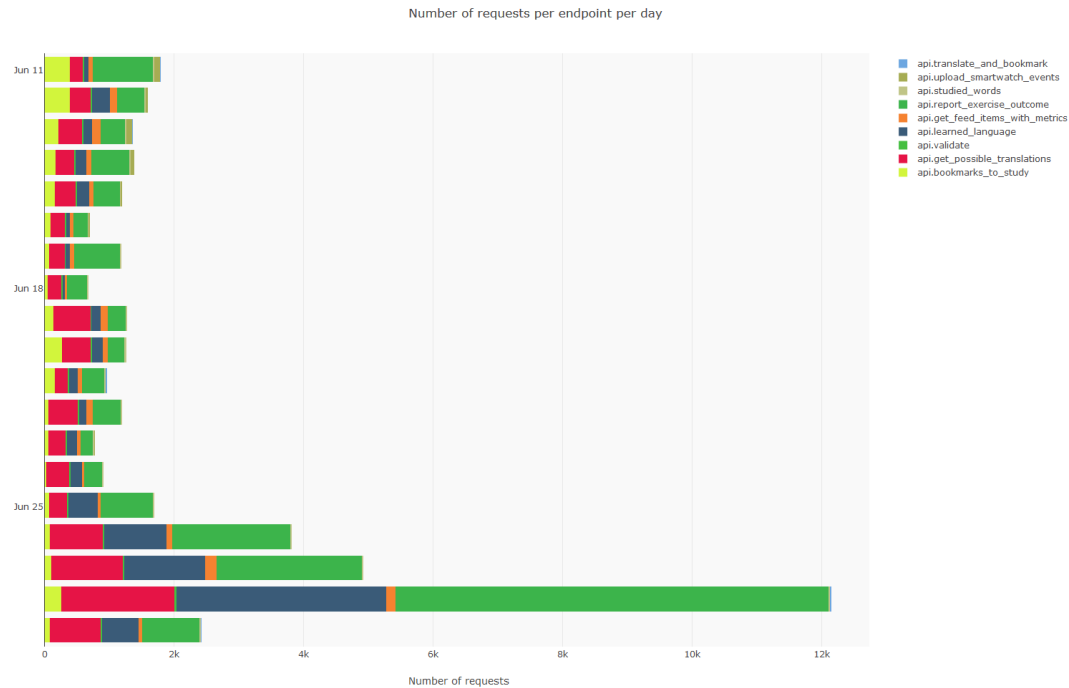


FIGURE 4.2: Measurements - Requests per endpoint

A related visualization is the heat map, which can show cyclic patterns of usage per hour of day. An example of this is shown in Figure 4.3. On the y-axis, the 24 hours of every day are enumerated. On the x-axis, the days are enumerated. This gives each day 24 cells that are colored with a color. The warmer the color, the higher the number of requests that hour. The colder the color, the lower the number of requests for that hour. The graph shows that the hour where there was a peak was at 9 PM on June 28, where the number of requests was about 2500. This also shows that the service is not used during the night, with most of the requests coming in on working hours and some activity in the evening. The graph shows that the spike of June 28 which was clearly visible in the previous visualization, was during the afternoon and the evening.

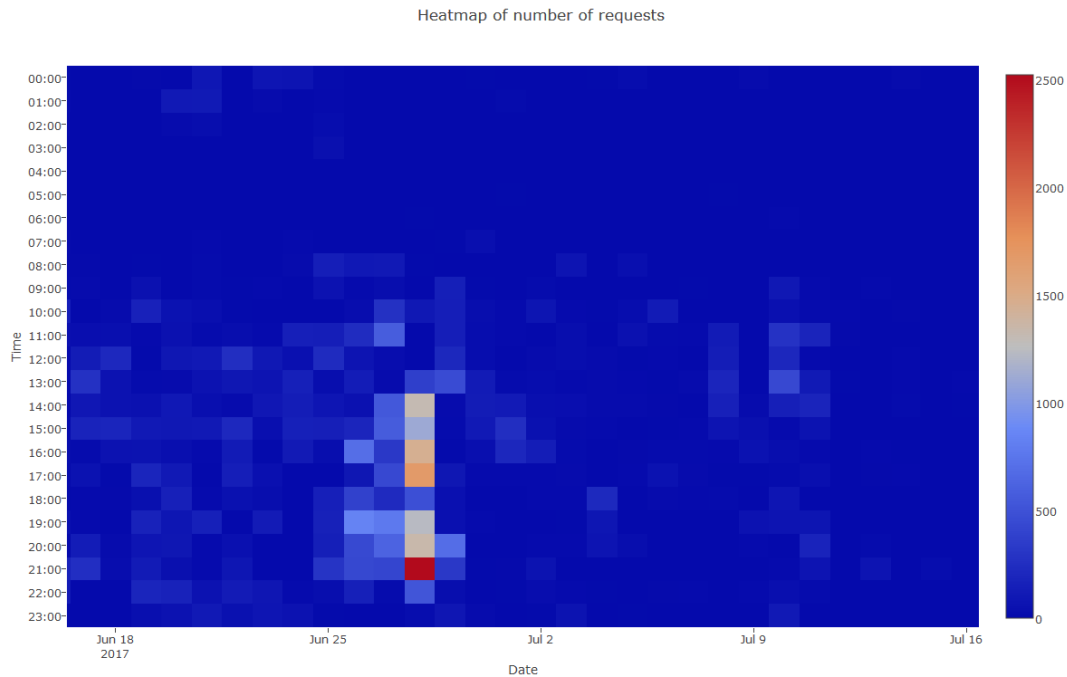


FIGURE 4.3: Measurements - Heatmap of number of requests

4.2.2 Endpoint performance

The dashboard also visualizes the performance of the individual endpoints. The dashboard contains a visualization that is made up of box plots, one for every monitored endpoint. An example of this is shown in Figure 4.4. On the y-axis, the endpoints that are being monitored are listed. The x-axis contains the execution time in milliseconds. The different colors represent the different endpoints again. This visualization summarizes the execution times of the endpoints and can help identify performance variability and balancing issues. This visualization shows that there are three endpoints that have a high variability in performance, and could therefore become a higher priority for the developer to be optimized.

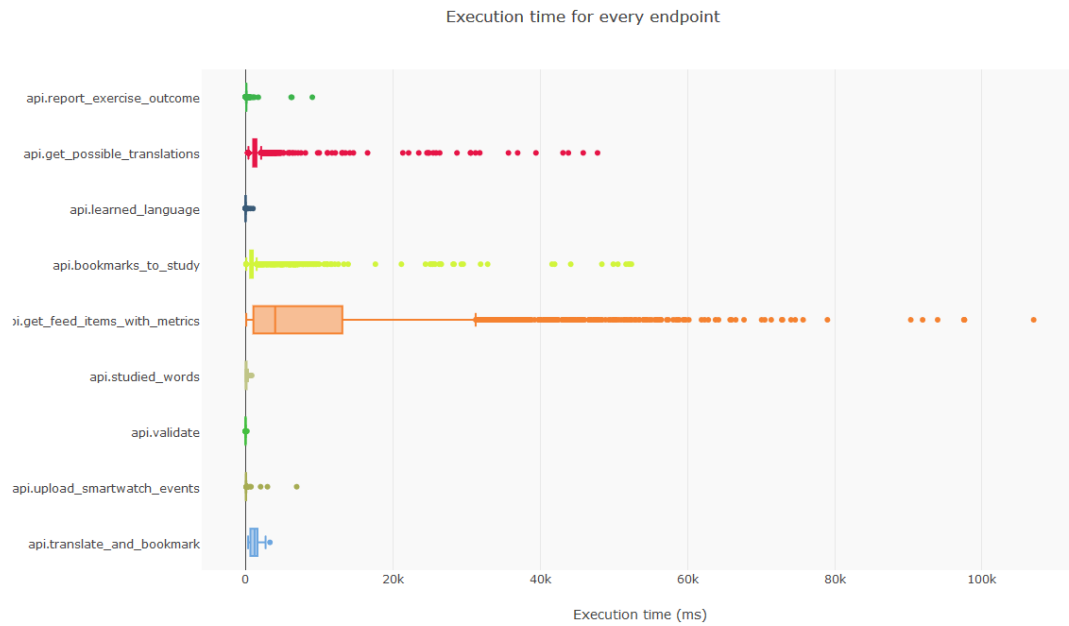


FIGURE 4.4: Measurements - Time per endpoint

Based on the previous visualization, the developer decided to try and improve the `get_possible_translations` endpoint. If we navigate to the results page of this endpoint, we can see the visualization in Figure 4.5. This is again a graph that contains box plots. On the y-axis, the version of the service along with its time stamp is listed. The x-axis contains the execution time in milliseconds again. This visualization shows that the performance gets better in the most recent versions of the service, since the median of the box plot moves to the left. In the most recent version, the median is about one second.

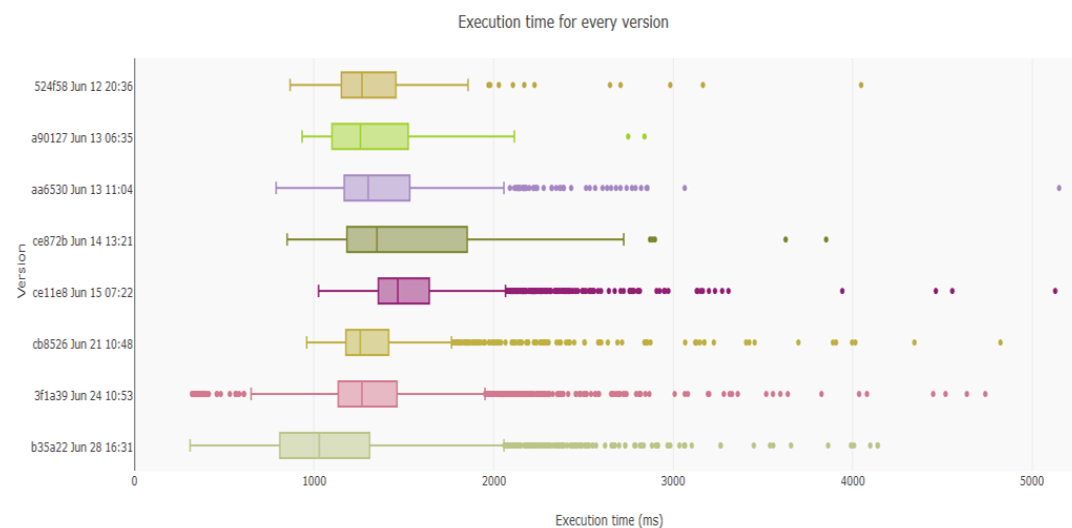


FIGURE 4.5: Endpoint - Time per version

4.2.3 User experience

Real-time computations can take longer for one user than it takes for another. This is due to the fact that such computations could become more complex and will take longer for users that have a higher load. For example, figure 4.4 shows the highest execution times and the highest variability for the endpoint named `get_feed_items_with_metrics`. This is due to the fact that users of Zeeguu can be subscribed to any number of article sources and for each of these, the system computes the personalized difficulty of each article for the user. A user that is subscribed to only one source will therefore experience faster execution times than one that is subscribed to thirty of them.

The dashboard provides a visualization that shows differences in experienced performance of multiple users. An example of this is shown in Figure 4.6. On the y-axis, a subset of the users of the service is listed. The x-axis contains the execution time in milliseconds again. This visualization clearly shows that the execution time for some user could be at least twice as long as that of another.

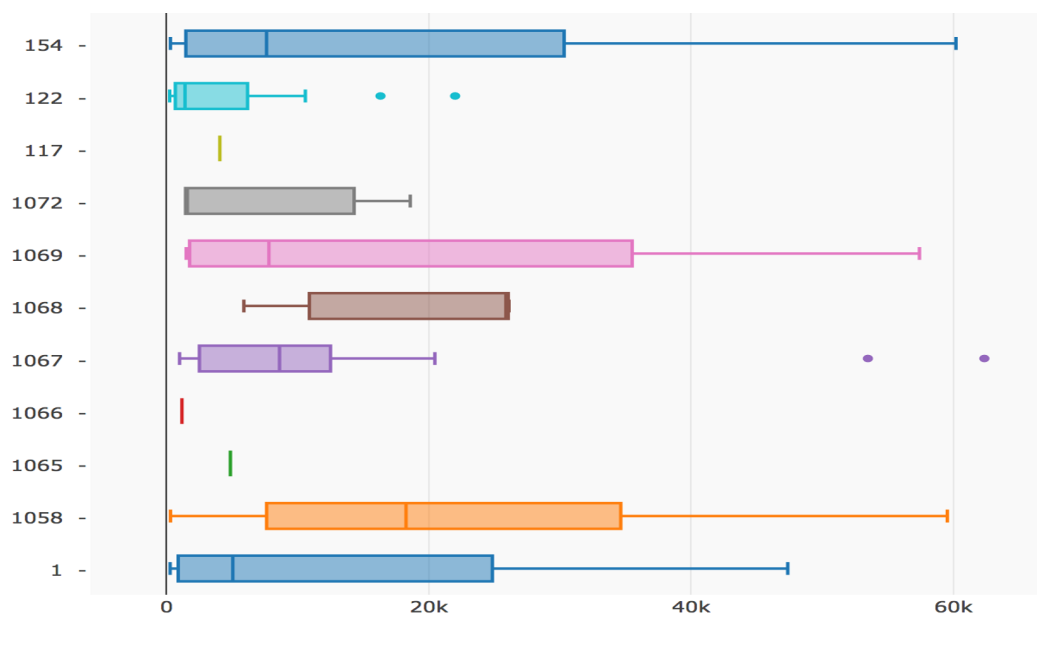


FIGURE 4.6: Endpoint - Time per user

The previous visualization presents all of the execution times of the requests of a certain endpoint for a certain user, but these requests are aggregated over the different versions of the service. The dashboard also has a visualization that also shows these different versions of the service. An example of this is shown in Figure 4.7. On the y-axis, a subset of users of the service is listed again. On the x-axis, the versions along with their time stamps are listed. For a certain combination of version and user, there could be a bubble. Such a bubble represents the average execution time experienced by that user for that version. The bigger the bubble, the longer the execution time. The visualization shows no clear trend. This could be due to the fact that the workload for one user is different than that of another, and the workload of a user could also change over time, becoming greater or smaller.

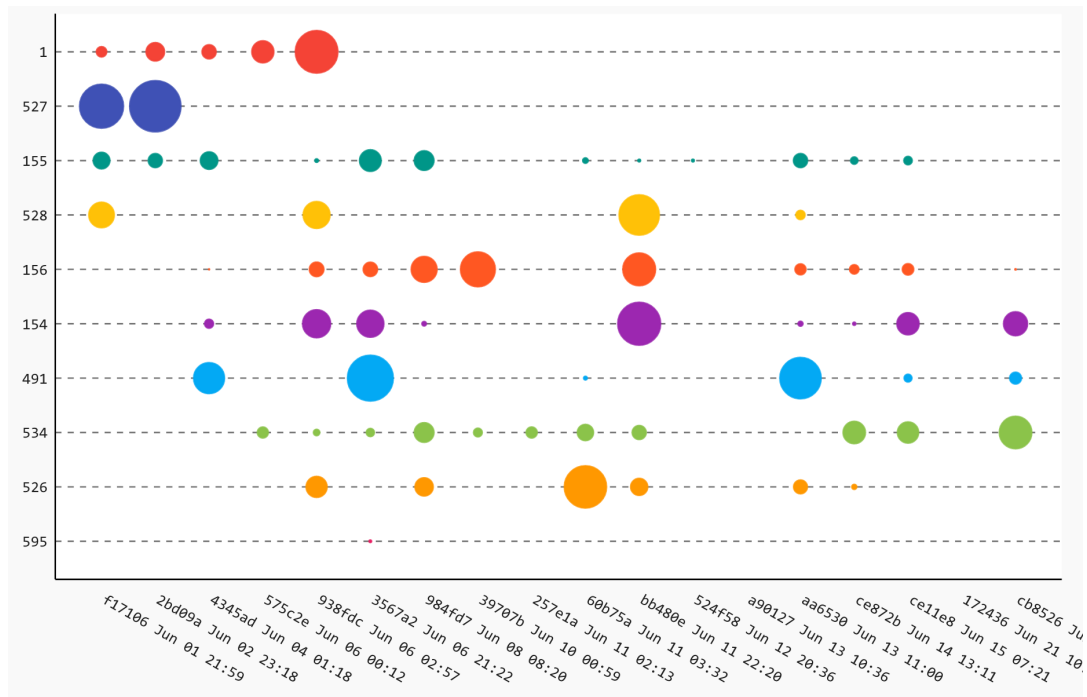


FIGURE 4.7: Endpoint - Time per version per user

4.2.4 Unit test performance

In addition to inspecting the performance of the live deployment of the web service based on dynamic requests by the users of the service, the dashboard also has visualizations for unit test performance during the evolution of the service.

On the `Testmonitor` page of the dashboard, there is a visualization for the overall execution times of the unit tests. An example of this is shown in Figure 4.8. On the y-axis, the number of the test suite (or the number of the Travis build) that the measurements belong to is listed along with the amount of measurements in that suite (between parentheses). The x-axis contains the execution times in milliseconds again.

This visualization shows that the most recent build resulted in the best overall execution times of the unit tests, since its median moved to the left compared to the builds that preceded it. When comparing Figure 4.8 to Figure 4.5, the trend of the unit test performance seems to match the trend of the live system performance. This means that unit test monitoring can indeed be applied to get an indication of the performance changes in the eventually live system. This visualization also shows one outlier that is extremely far away from the other data points in every build. This could be explained by the fact that one of the unit tests has an execution time that is very long, about 28 seconds.

In order to investigate this further, the user of the dashboard can go to the table on the dashboard above the visualization of Figure 4.8. This table contains all of the unit tests that have been run, along with their average execution times, number of times executed, and time of last execution. When sorting the table on execution time by clicking that column header, one unit test will stand out. This is the test named `test_get_feed_items_with_metrics`. When the user clicks on this test, a page containing the visualization for this specific test will be shown.

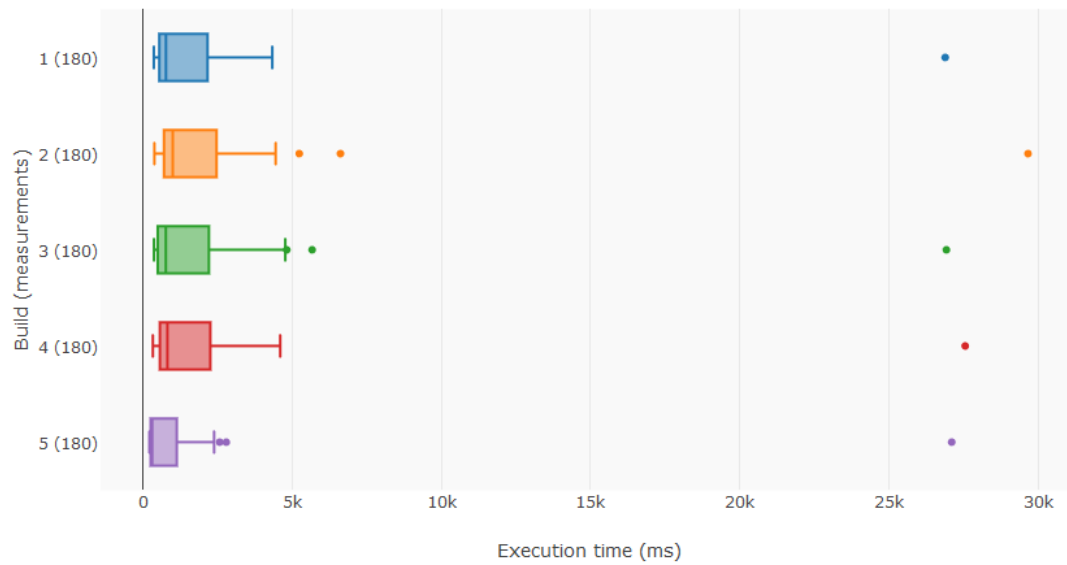


FIGURE 4.8: Testmonitor - Time per build

The visualization for a specific unit test could look something like Figure 4.9. As can be seen, the median does move to the left in the most recent build, which means the performance improves. Also, this unit test is indeed the one that produces the extreme outlier of 28 seconds. As can be seen, the unit test is being run five times each build. The outlier corresponds to the first run of the test, which will cache the result of some calculation making the successive calls have an execution time that is rather small compared to the original one.

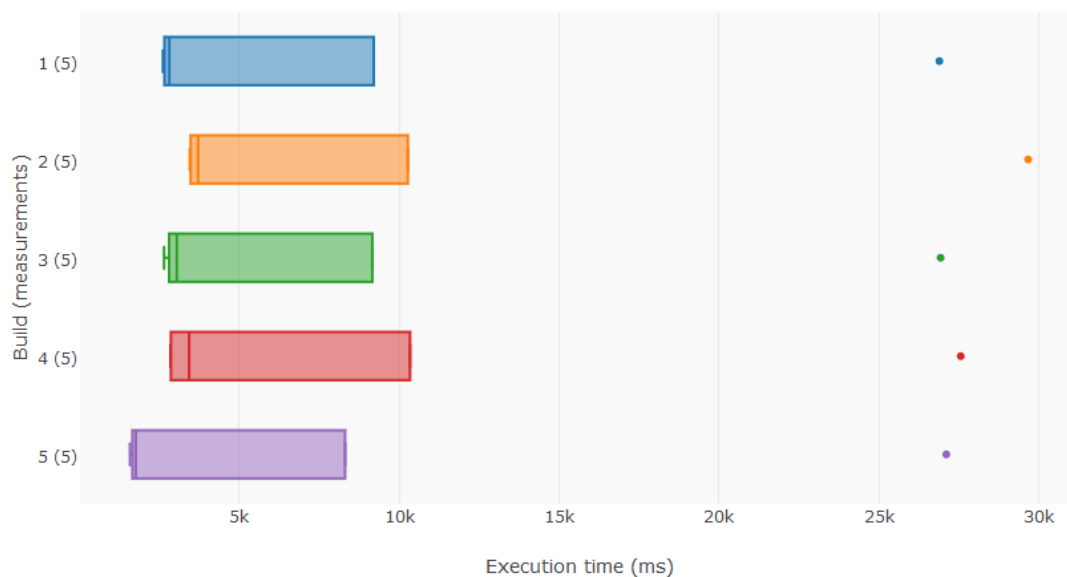


FIGURE 4.9: Testmonitor - Time per test

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Software monitoring can be done by observing the evolution of unit test performance as the software project evolves over time. It can also be done by observing the evolution of the live performance of a deployed software project as it evolves over time. Both of these approaches are implemented in the tool presented in this thesis. This shows that it is possible to create a monitoring solution that gives the developer of a web service insight into the utilization and performance of the service. It also shows that such a solution does not need to take a great amount of effort to deploy.

This research is aimed at web application projects that are under development and are using Python in combination with Flask. These projects may not have the time nor the money to either use a commercial tool or create their own. The dashboard provides a free-to-use way for these projects to easily achieve evolving performance monitoring. A similar architecture could be applied to other languages and frameworks in order to create a tool that provides evolving performance monitoring for those other languages and frameworks.

In the case study discussed in this work, the trend of the unit test performance seems to match the trend of the live system performance. This means that unit test monitoring can indeed be applied to get an indication of the performance changes in the eventually live system. By monitoring the unit test performance of a project, a developer can prematurely implement functionality or make certain functionality more high-performing, without having to deploy the project and test the live service first.

The research question of this thesis is:

"How to create a tool that allows the automatic performance monitoring of evolving unit tests for Flask services in a way that affects these services in the least amount possible?"

This research presents a way of achieving this. The tool allows automatic performance monitoring of Flask web services that are under development. It tracks the versions of these services, such that the evolution of the service is taken into account when visualizing the performance measurements on the dashboard of the tool. The tool is very simple to integrate in a pre-existing service, which will be changed minimally in order for the tool to work.

For the availability of the tool, see the appendix on page 43.

5.2 Future Work

5.2.1 Case studies

Future work for this research includes performing more case studies using other systems to deploy the dashboard on. The goal of this is to discover other needs of such a tool that are currently absent from the dashboard, as well as finding possible aspects of the dashboard that are less useful and could therefore be removed from the dashboard. This way, more essential and informative visualizations will have room to grow in the tool.

Another goal of doing more case studies is to see if the dashboard will scale with an increase in web service users. All of the visualizations should still behave as intended with a database that is considerably larger than the one being used in the case study within this research. The dashboard itself should also still perform quite well with a huge increase in data points it has to take into consideration and requests it has to monitor.

The purpose of these studies is to test and improve the aspects of usability, scalability, reliability, stability and security. The tool would not be very helpful if it did not have a high usability, or if it would not be very scalable to projects that have a massive user-base. Of course, any tool should be reliable and stable in order for it to be used professionally. Obviously, security is also a big thing that should not be forgotten, since the dashboard deals with user-specific information and when the security is compromised, the privacy of the users of the monitored service will be compromised as well. Security could be improved by encrypting the database, for example by hashing the entries. Communication with the dashboard could also be improved security-wise by only allowing HTTPS connections, which encrypt this communication.

5.2.2 Meta-dashboard

Another thing that could be viewed as possible future work would be the creation of a meta-dashboard. This would mean that one deployment of such a dashboard will enable its user to monitor multiple deployments of their web application. This way, results could be aggregated or shown for each separate project. This will give the user more options in the area of which deployments to monitor, since this will enable developers of applications with a load-balancer for example to also make use of the dashboard. Then the measurements of every node will be aggregated into one dashboard and they do not need an individual dashboard for every node.

A meta-dashboard could also be nice when developing multiple web applications, so that the user could login to this dashboard and then be presented with a list of applications that it is currently monitoring. The user could then select one of them to go to the visualizations for that specific application. This could be handy for software testers, who probably have to test multiple web applications at the same time.

5.2.3 Error tracking

The live tracking of requests which result in errors could be another possible extension of the research done here. The monitoring tool could track every incoming request to the monitored web service and see if the request gets handles properly. If this is not the case, and the service encounters an error while handling a request, the dashboard could notify the developer of this immediately. The dashboard could

then also collect the dynamic request values and the state of the service when the error occurred. This could help the developer to recreate the situation and maybe also improve the service so that the error will not be thrown again in the future.

5.2.4 Flask core

A final possibility of future work could be to try and make the monitoring tool available as part of the core of the Flask micro-framework. This would make it even easier for developers of Flask web services to deploy their service with built-in automatic monitoring of their evolving application. This would mean that the maintainer of Flask has to be contacted and asked for his opinion of the monitoring tool. When he is convinced of the usefulness of the tool and its quality, he might then decide it would be a nice addition to the currently available version of Flask.

Before this would even be considered, the tool in its current state should first endure additional testing and going through other case studies. The scalability, reliability and stability of the tool should be proven to be at a high enough level first. Several improvements will need to be made based on these tests and case studies, before the tool is ready to be shipped in the form of a default extension of Flask on its website.

Appendix A

Availability of Flask Dashboard

Most of the figures used in this thesis are screen-shots of the actual deployment of Flask Dashboard on the Zeeguu platform. If the tool is still deployed on this case study at the moment of reading this thesis, it should be available for viewing on-line at:

```
https://zeeguu.unibe.ch/api/dashboard
```

```
Username: guest
```

```
Password: vissoft
```

Flask Dashboard is available to download on the Python Package Index:

```
https://pypi.python.org/pypi/flask-monitoring-dashboard
```

It can be installed on any system that has Python installed by issuing the following command:

```
$ pip install flask_monitoring_dashboard
```

The source code of Flask Dashboard is published under a permissive MIT license on GitHub:

```
https://github.com/mircealungu/automatic-monitoring-dashboard
```

To install Flask Dashboard using the cloned GitHub repository, issue the following command:

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

Bibliography

- Avagyan, Martin (2017). *Building Blocks for Online Language Practice Platforms*. Bachelor Thesis, University of Groningen.
- Brand, Luc van der (2017). *Prometheus: Efficiency and Usability in a Personalized Multilingual Feed Manager*. Bachelor Thesis, University of Groningen.
- Chacon, Scott and Ben Straub (2014). *Pro Git*. <https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git>. [Online; accessed 13-July-2017].
- Ellison, Rich (2015). *How To Be A Tester*.
- Feist, M. D. et al. (2016). "Visualizing Project Evolution through Abstract Syntax Tree Analysis". In: *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 11–20. DOI: 10.1109/VISSOFT.2016.6.
- Fielding, Roy et al. (1999). "Hypertext transfer protocol-HTTP/1.1". In: *RFC 2616*.
- Haas, Hugo (2004). *Web Services Glossary, World Wide Web Consortium*. <https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice>. [Online; accessed 03-July-2017].
- Isaacs, K. E. et al. (2014). "Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time". In: *IEEE Transactions on Visualization and Computer Graphics* 20.12, pp. 2349–2358. ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2346456.
- Lungu, Mircea, Michele Lanza, and Oscar Nierstrasz (2014). "Evolutionary and Collaborative Software Architecture Recovery with Softwrenaut". In: *Science of Computer Programming* 79.0, pp. 204–223. DOI: 10.1016/j.scico.2012.04.007. URL: <http://scg.unibe.ch/archive/papers/Lung14a.pdf>.
- Lungu, Mircea et al. (2017). "A Low-Effort Analytics Platform for Visualizing Evolving Flask-Based Python Web Services". In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*.
- Lungu, Mircea F. (2016). "Bootstrapping an Ubiquitous Monitoring Ecosystem for Accelerating Vocabulary Acquisition". In: *Proceedings of the 10th European Conference on Software Architecture Workshops*. ECSAW 2016. Copenhagen, Denmark: ACM, 28:1–28:4. ISBN: 978-1-4503-4781-5. DOI: 10.1145/2993412.3003389. URL: <http://doi.acm.org/10.1145/2993412.3003389>.
- netmarketshare.com (2017). *Browser market share*. <https://www.netmarketshare.com/browser-market-share.aspx>. [Online; accessed 16-July-2017].
- Papazoglou, M. P., V. Andrikopoulos, and S. Benbernou (2011). "Managing Evolving Services". In: *IEEE Software* 28.3, pp. 49–55. ISSN: 0740-7459. DOI: 10.1109/MS.2011.26.
- Pauw, W. De et al. (2005). "Web Services Navigator: Visualizing the execution of Web Services". In: *IBM Systems Journal* 44.4, pp. 821–845. ISSN: 0018-8670. DOI: 10.1147/sj.444.0821.
- PythonSoftwareFoundation (2010). *Basic unittest example*. <https://docs.python.org/2/library/unittest.html#basic-example>. [Online; accessed 04-July-2017].

- Ronacher, Armin (2010). *Quickstart*. <http://flask.pocoo.org/docs/0.12/quickstart/#quickstart>. [Online; accessed 04-July-2017].
- Vogel, Patrick (2017). *Automatic Performance Monitoring of evolving live Web Services*. Bachelor Thesis, University of Groningen.
- Wilhelm, A. et al. (2016). "A Visualization Framework for Parallelization". In: *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 81–85. DOI: 10.1109/VISSOFT.2016.35.