Visualizing Evolving Service Performance in Python

NAMES ORDER TBA

Johann Bernoulli Institute of Mathematics and Computer Science
University of Groningen
Groningen, the Netherlands
Email: {v.andrikopoulos,m.f.lungu}@rug.nl, {t.klooster.1,p.p.vogel}@student.rug.nl

***Abstract*—The abstract goes here.**

I. Introduction

Every system is a distributed system nowadays [1].

Python is currently one of the most popular programming languages. At the time of writing this paper[1] Python is the 4th most popular programming language cf. the Tiobe Index[2].

Todo: Flask summary goes here

A search on GitHub with the keyword “language:Python” returns more than 500K open source projects written in the language. If we restrict the search by adding the keyword “Flask” we obtain a listing of 25K projects, that is, 5% of all the Python projects. Flask – advertised as a *micro-framework* – is a lightweight alternative to web site and service development.

However, there is no dedicated solution for monitoring the performance of Flask web-applications. Thus, every one of those Flask projects faces one of the following options when confronted with the need of gathering insight into the runtime behavior of their implemented service:

1) Use a heavyweight professional API monitoring setup they require setting up a different server ML: Thijs, Patrick: can we find a few examples of professional but overkill tools? ideally they require setting up a bunch of servers, and writing configs in XML!
.
2) Implement their own analytics tool
3) Live wihout analytics insight into their services [3]

Todo: For the first point in the list, we can also argue that analytics solutions like Google Analytics can be used, but they have no notion of versioning/integration with the development lifecycle. Feel free to cite [2] for service evolution purposes

For projects which are done on a budget (e.g. research projects) the first and the second options are often not available due to time and financial constraints.

To avoid these projects ending up in the third situation, in this paper we present a low-effort, lightweight service monitoring API for Flask and Python web-services.

[1]June 2017

[2]TIOBE programming community index is a measure of popularity of programming languages, created and maintained by the TIOBE Company based in Eindhoven, the Netherlands

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

To start using our Python library for service visualization, one needs one line of code to connect their Flask application object with the dashboard and an additional one to import the library:

Listing 1. Configuring the Dashboard is straightforward

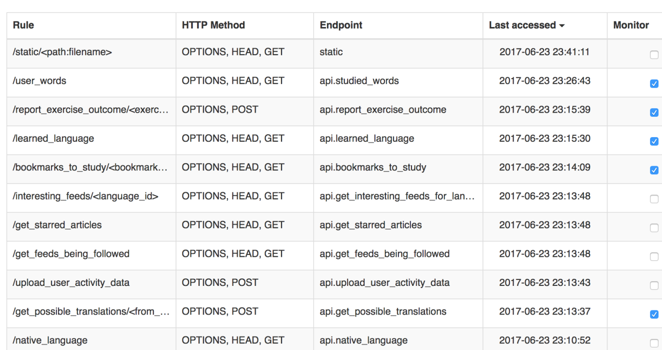
```
import dashboard
dashboard.bind(app=flask_app)
```

After binding to the service, Dashboard will search for all endpoints defined in it. These will be presented to the user, which can select the ones that should be monitored, see figure 1. In order to monitor an endpoint, Dashboard will create a function wrapper for the API function that corresponds to the endpoint. This way, the wrapper will be executed whenever that API call is made. The wrapper contains the code that takes care of monitoring an endpoint.

By default, Dashboard is available at the `/dashboard` route of the Flask app. A custom route can also be defined by simply adding one extra line of code:

Listing 2. Configuring the Dashboard with a custom route for it to be accessed on is straightforward

```
dashboard.config.link = 'custom-link'
```



Rule	HTTP Method	Endpoint	Last accessed	Monitor
/static/<path:filename>	OPTIONS, HEAD, GET	static	2017-06-23 23:41:11	☐
/user_words	OPTIONS, HEAD, GET	api.studied_words	2017-06-23 23:26:43	☑
/report_exercise_outcome/<exerc...	OPTIONS, POST	api.report_exercise_outcome	2017-06-23 23:15:39	☑
/learned_language	OPTIONS, HEAD, GET	api.learned_language	2017-06-23 23:15:30	☑
/bookmarks_to_study/<bookmark...	OPTIONS, HEAD, GET	api.bookmarks_to_study	2017-06-23 23:14:09	☑
/interesting_feeds/<language_id>	OPTIONS, HEAD, GET	api.get_interesting_feeds_for_lan...	2017-06-23 23:13:48	☐
/get_starred_articles	OPTIONS, HEAD, GET	api.get_starred_articles	2017-06-23 23:13:48	☐
/get_feeds_being_followed	OPTIONS, HEAD, GET	api.get_feeds_being_followed	2017-06-23 23:13:48	☐
/upload_user_activity_data	OPTIONS, POST	api.upload_user_activity_data	2017-06-23 23:13:43	☐
/get_possible_translations/<from_...	OPTIONS, POST	api.get_possible_translations	2017-06-23 23:13:37	☑
/native_language	OPTIONS, HEAD, GET	api.native_language	2017-06-23 23:10:52	☐

Fig. 1. All of the endpoints of the Flask app are shown such that a selection can be made for monitoring them

II. Case Study

Zeeguu case study description to be used as running example throughout the rest of the paper [3]

Architecture: series of web and mobile applications built aroud a core web service implemented in Python and Flask which provides:

- contextual translations
- reading recommendations
- exercises

We have this system for helping learners read texts that they like, and enable them to practice with exericses generated on their past readings.

III. The Dashboard

The Dashboard as well as the web service that is being monitored in the case study is written in Python using Flask. This makes binding to the web service relatively easy, as well as adding additional routes to the service for interacting with the Dashboard. There are three main endpoints that are available using this tool:

1) one which shows a list of the endpoints of the service that can be monitored
2) one which contains an overview of the measurements of the selected endpoints
3) one which contains more detailed information about the measurements of a specific endpoint

As mentioned before, the first endpoint enables the user of the tool to specify which endpoints should be monitored. The second endpoint consists of two parts, one of them being a table that shows for every monitored endpoint the number of hits it has gotten, the time it was last accessed and its average execution time. The second part is a view with four graphs which show:

- A heatmap of the total number of requests to the monitored endpoints
- A stacked bar chart that shows the total number of requests to the monitored endpoints per endpoint per day
- A boxplot graph showing the average execution time per version of the web service
- A boxplot graph showing the average execution time for every monitored endpoint

The third main endpoint that the Dashboard has consists of three parts. One of them is a table that shows additional information about a specific monitored endpoint, like the version of the web service in which the endpoint was added to it along with the date. The second part is again a view, this time with seven graphs which show information specific to one monitored endpoint:

- A heatmap of the number of requests
- A time series of the minumum, maximum and average execution time
- A time series of the number of hits per hour
- A chart that shows the average execution time per user per version of the service
- A chart that shows the average execution time per IP Address per version of the service
- A boxplot graph showing the execution time per version
- A boxplot graph showing the execution time per user

The third part contains information about outliers in the measurements if they exist for that endpoint. There is information here about the time when the outlier was recorded, its execution time, its request url, request values, request headers, request environment, the cpu usage and memory usage at the time of recording the outlier and last but not least the stacktrace at that time.

In the remainder of the article:

- all the views are screenshots from the actual tool; in the tool they are interactive with the user being able to zoom in, pan, etc.

IV. Overall Endpoint Utilization

The most fundamental insight that a service maintainer needs regards service utilization.

Figure 2 shows a first perspective on endpoint utilization that Dashboard provides: a stacked bar chart of the number of hits to various endpoints grouped by day [4] shows that at its peak the API has about 2500 hits per day. The way users interact with the platform can also be inferred since the endpoints are indicators of different activity types, e.g.:

1) **api.get_possible_translations** is an indicator of the amount of reading the users are doing
2) **api.report_exercise_outcome** is an indicator of the amount of vocabulary practice the users are doing

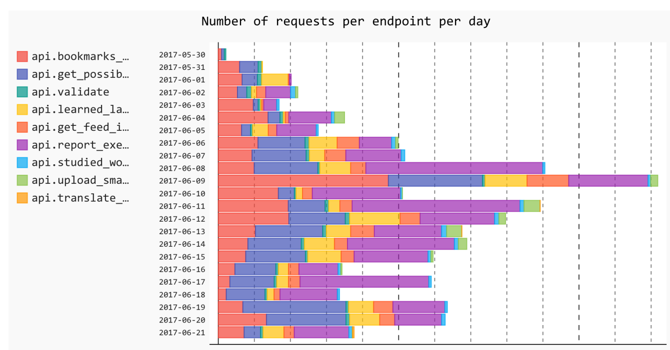


Fig. 2. The number of requests per endpoint per day view shows the overall utilization of the monitored application

This visualization also provides feedback to the maintainer when deciding aboiut endpoint deprecation, the most elementary way of *understanding the needs of the downstream* [4]. In our case study, the maintainer decided to not remove several endpoints once they saw that, contrary to their expectations, they were being used.[5]

A second type of *utilization* question that an API maintainer can answer by using the Dashboard regards cyclic patterns of usage during various times of day.

[4]We recommend obtaining a color version of this paper for better readability

[5]Usage information can also be used to increase the confidence of the maintainer that a given endpoint is not used, although it is not a proof.

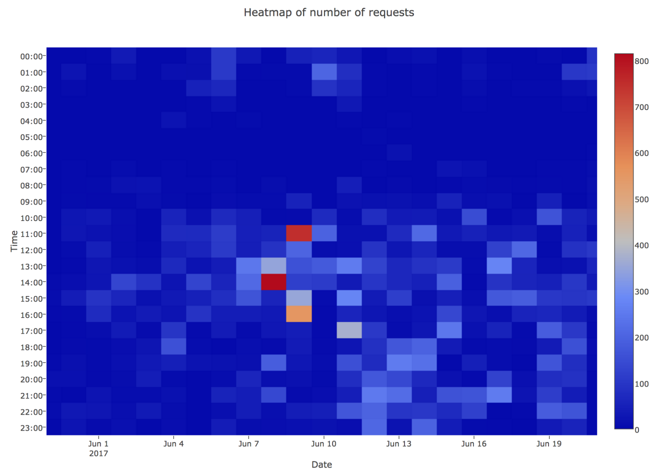


Fig. 3. Usage patterns become easy to spot in the requests per hour heatmap

Figure 3 shows the users of the Zeeguu API not practicing languages at night, but otherwise hitting the API around the clock with several hundred hits per hour.

V. Visualizing Service Performance

The Dashboard also collects information regarding endpoint performance. The view in Figure 4 summarizes the response times for various endpoints by using boxplots.

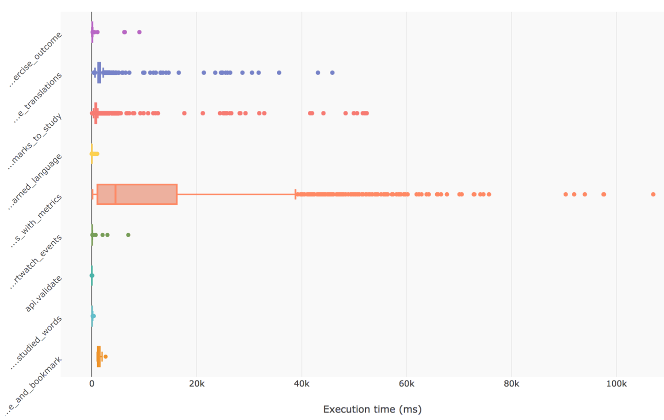


Fig. 4. The response time (in ms) per monitored endpoint view allows for identifying performance variability and balancing issues

After investigating this view it became clear to the maintainer that three of the endpoints had very large variation in performance. One of the three was most critical was optimized first: the **api.get_possible_translations** is part of a live interaction and it having such variable performance was a usability problem for the users of the reader applications.

To be able to see their improvements in action, the maintainer had to add an extra configuration information to be able to find the '.git' folder from where to retrieve the current version of the deployed application:

Listing 3. Configuring the Dashboard with the path to the .git folder enables the generation of evolutionary performance graphs

```
dashboard.config.git = 'path/to/git-root/of/app'
```

After redeploying the API, the dashboard can now automatically detect the current version of the project, and can group measurements by version. Dashboard can now generate the view in Figure 5 where the performance of the give endpoint is tallied by version.

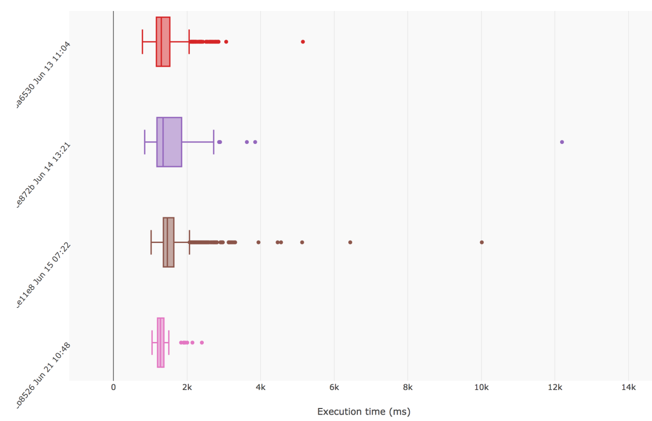


Fig. 5. Visualizing The Performance Evolution of the **api.get_possible_translations** endpoint

This way the maintainer could confirm that the performance of the translation endpoint improved: in the latest version (bottom-most box plot in Figure 5) the entire boxplot moved to the left and there are fewer outliers.

The Dashboard also collects **extra informaiton about outliers**: Python stack trace, CPU load, request parameters, etc. in order to allow the maintainer to investigate the causes of these exceptionally slow response times.

In order to address this, but not degrade the usual performance the Dashboard tracks for every endpoint a running average value. When it detects that a given request is an outlier with respect to this past average running value, it triggers the *outlier data collection routine* which stores all the previously listed extra information about the current execution environment.

VI. User Centered Visualization

For service endpoints which run computations in real time as they are called, there might be very different timings based on the different loads that are sent to the endpoint.

In our cases study, one of the slowest endpoints, and one with the higest variability is **api.get_article_difficulties**: it retrieves a list of recommended articles for a given user. However, since a user can be subscribed to anything from one to three dozen article sources, and since the computation of the difficulty is personalized and it is slow, the variability in time among users is likely to be very large.

Listing 4. Simply define a custom app-specific function for user retrieval and pass it to the Dashboard to group information by user

```
# app specific way of extracting the user
# from a flask request object
def get_user_id(request):
  sid = int(request.args['session'])
  session = User.find_for_session(sid)
  return user_id

# attaching the get_user_id function
dashboard.config.get_group_by = get_user_id
```

Dashboard provides a way of grouping information on a per user basis. However, to do this, the developer must specify the way in which a given API call can be associated with a given user. There are multiple ways, the simplest takes again advantage of the strengths of the Flask framework which offers a global request object which contains session information:

Sometimes, grouping the service calls per endpoint it is not sufficient. Figure 6 shows some of the results of calling the **api.get_article_difficulties** endpoint for various users.

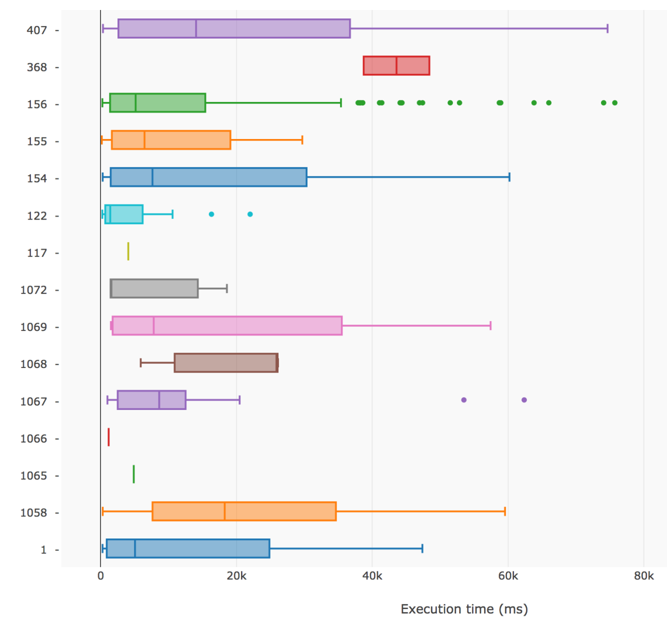


Fig. 6. The **api.get_article_difficulties** shows a very high variability across users

Different users might have different experiences: - a user has 10K emails one has 10 emails - a user is subscribed to 20 feeds one to 2 feeds

The system will have different processing times. It is important for the DevOps-er to be able to understand the difference in performance on a per user basis.

If we try to show also per version:

VII. Tool Availability

The code of Dashboard is available under an open-source permissive MIT license on Github [6].

[6]https://github.com/mircealungu/automatic-monitoring-dasboard

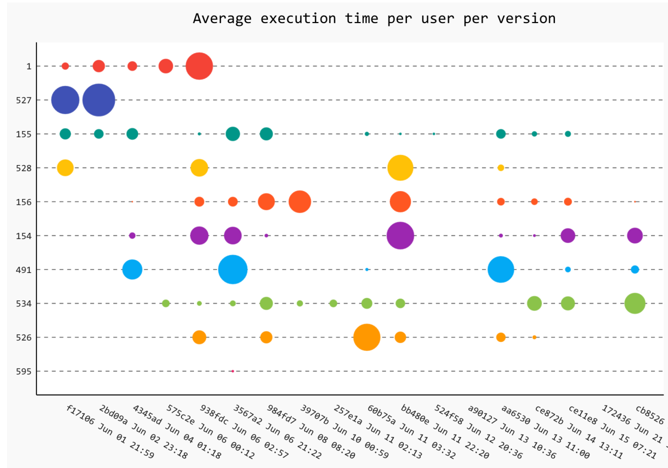


Fig. 7. Caption here

The images in this paper are screenshots of the actual deployment of the tool which can be found at https://zeeguu.unibe.ch/api/dashboard. For the readers of this paper to be able to see the tool in action, they can login with the username and password: guest, dashboardguest!. ML: Thijs, Patrick: we should add a guest/guest username password which is allowed to only visualize things, but not modify anything, and not export anything!

VIII. Related Work

Java Visualization [5]

Run-time monitoring of services [6]

IX. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] M. Cavage, "There is no getting around it: you are building a distributed system," *Communications of the ACM*, vol. 56, no. 6, pp. 63–70, 2013.

[2] M. Papazoglou, V. Andrikopoulos, and S. Benbernou, "Managing evolving services," *Software*, vol. 28, no. 3, pp. 49–55, 2011.

[3] M. F. Lungu, "Bootstrapping an ubiquitous monitoring ecosystem for accelerating vocabulary acquisition," in *Proccedings of the 10th European Conference on Software Architecture Workshops*, ser. ECSAW 2016. New York, NY, USA: ACM, 2016, pp. 28:1–28:4. [Online]. Available: http://doi.acm.org/10.1145/2993412.3003389

[4] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "A quantitative analysis of developer information needs in software ecosystems," in *Proceedings of the 2nd Workshop on Ecosystem Architectures (WEA'14)*, 2014, pp. 1–6. [Online]. Available: http://scg.unibe.ch/archive/papers/Haen14a-QuantitativeEcosystemInformationNeeds.pdf

[5] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the execution of java programs," in *Revised Lectures on Software Visualization, International Seminar*. London, UK: Springer-Verlag, 2002, pp. 151–162.

[6] C. Ghezzi and S. Guinea, "Run-time monitoring in service-oriented architectures," in *Test and analysis of web services*. Springer, 2007, pp. 237–264.