Flask Dashboard: A Lightweight Analytics Platform for Visualizing Evolving Python Web Service Utilization and Performance

NAMES ORDER TBA

Johann Bernoulli Institute for 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

"There is no getting around it: you are building a distributed system" argues a paper from half a decade ago [1]. Indeed even the simplest student project web application ends up implemented as two-tier architectures with a front-end implemented with web technologies and a service backend usually a REST API.

Many contemporary programming languages are offering libraries, modules, or frameworks that facilitate the development of such architectures. Python is an example of such a language. Python is currently one of the most popular programming languages for service implementation on the back-end side of web applications[1]. At the time of writing this paper[2] Python is the 4th most popular programming language according to the Tiobe Index[3].

Flask[4] is a very popular Python web framework[5]. It provides simplicity and flexibility by implementing a bare-minimum web server, and thus advertises as a micro-framework. The Flask tutorial shows how setting up a simple Flask "Hello World" web-service requires no more than 5 lines of Python code [2].

To the best of our knowledge, 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 services:

1) Use a commercial monitoring setup that usually requires setting up a different server that treats the subject API as a black-box (see Sec. VIII).
2) Implement their own analytics toolchain potentially requiring multiple person-months.
3) Live without analytics insight into their services[6].

For projects which are done on a budget (e.g. research projects, startups) the first and the second options are often not available due to time and financial constraints. Furthermore, while adopting 3rd-party analytics solutions like e.g. Google Analytics might be an option, a critical insight into the evolution of the exposed services of the web application, see for example [3], is missing due to the fact such solutions have no notion of versioning/integration with the development life cycle.

To avoid such projects ending up in the third situation, in this paper we present a low-effort, lightweight service monitoring API for Flask-based Python web services that is easy to integrate and provides multiple perspectives on the performance and utilization of the subject API.

VA: Mircea: consider removing the subsection header completely to save space

Structure of the Paper

ML: to update! The remainder of this paper is structured as follows: In Sections IV, V, VI we dedicate a separate section to three types of analysis that Flask Dashboard supports and present a the way the visualizations support these types of analysis.

II. Case Study

Zeeguu[7] is a platform and an ecosystem of applications for accelerating vocabulary acquisition in a foreign language [4]. The architecture of the ecosystem has at its core an API implemented with Flask and Python and a series of satellite

[1]Searching for projects written in Python on GitHub returns more than 500K open source projects as results

[2]End of June 2017

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

[4]http://flask.pocoo.org/

[5]More than 25K projects on GitHub (5% of all Python projects) are implemented with Flask (cf. a GitHub search for "language:Python Flask")

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

[7]https://www.zeeguu.unibe.ch/

applications that together offer three main intertwined features for the learner:

1) Reader applications that provide effortless translations for those texts which are too difficult for the readers.
2) Interactive exercises personally generated based on the preferences and past likes of the learner.
3) Article recommendations which are at the appropriate level of difficulty for the reader. The difficulty is estimated based on past exercise and reading activity.

The core API provides correspondingly three types of functionality: contextual translations, article recommendations, and personalized exercise suggestions. The core API of system is a research project, which sustains at the moment of writing this article the reading and practice of about two hundred active beta-tester accounts.

In the remainder of this paper, we will use the Zeeguu API as a case study. All the figures in this paper are captured from the actual deployment of Flask Dashboard in the context of the Zeeguu API [8].

III. The Flask Dashboard

The Flask Dashboard as well as the web application that is being monitored in the case study is written in Python using Flask. This makes binding to the web services of the application relatively easy, as well as adding additional routes to the service for interacting with the Flask Dashboard.

To start using our Python library for service visualization, and assuming Flask is already installed, one needs to install the Python package[9] and simply add two lines of code to their Flask web service:

```
import dashboard
...
# flask_app is the Flask app object
dashboard.bind(flask_app)
...
```

After binding to the service, the Flask Dashboard becomes available at the `/dashboard` route of the Flask application. A custom route can also be defined by the programmer in a configuration file.

During binding, the Flask Dashboard will search for all endpoints defined in the target application. These will be presented to the user in the tool web interface, where the user can select the ones that should be monitored, see Fig. 1.

In order to monitor an endpoint, the Flask Dashboard creates 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. Todo: Data collected by the wrappers are persisted in...

There are two main categories of visual perspectives that are available using Flask Dashboard:

1) *Overviews* that present information and measurements about all the endpoints of interest, and

[8]Within the Flask Dashboard the figures are interactive offering basic data exploration capabilities: filter, zoom, and details on demand[5]

[9]Section VII-D shows how to install the package

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 Zeeguu app are shown such that a selection can be made for monitoring them

2) *Detailed information* about the measurements pertaining to a specific endpoint.

In the remainder of the paper we present several of these perspectives.[10]

IV. Service Utilization

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

Figure 2 shows a first perspective on endpoint utilization that Flask Dashboard provides: a stacked bar chart of the number of hits to various endpoints grouped by day 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 foreign language reading the users are doing
2) **api.report_exercise_outcome** is an indicator of the amount of foreign vocabulary practice the users are doing

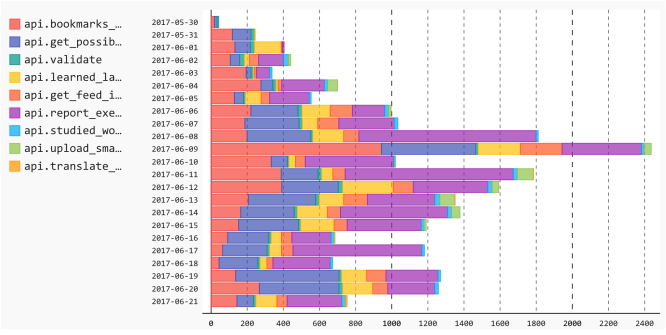

Fig. 2. The number of requests per endpoint per day view shows the overall utilization of the monitored application

Besides showing the overall utilization, this endpoint provides to the maintainer with information relevant for decisions regarding endpoint deprecation – one of the most elementary ways of *understanding the needs of the downstream*[6]. In

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

our case study, the maintainer realized that several endpoints which they thought were not being used, contrary to their expectations, were actually being used.[11]

Todo: Add the time series graph and discuss it before the heatmap? We can then sell the heatmap better ML: Not sure about which graph you refer to here V

A second type of utilization question that the Flask Dashboard can answer automatically regards *cyclic patterns of usage per hour of day*.

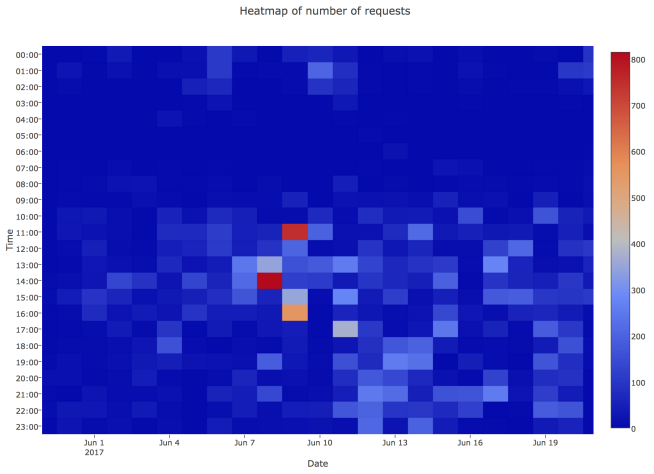

Fig. 3. Usage patterns become easy to spot in the requests per hour heatmap

Figure 3 shows the the API not being used during the early morning hours, and that most of the activity focused around working hours and some light activity during the evening. This is consistent with the fact that the current users are all in the central european timezone.

V. Endpoint Performance

The Flask Dashboard also collects information regarding endpoint performance. The view in Figure 4 summarizes the response times for various endpoints by using box-and-whisker plots.

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:

[11]Usage information can also be used to increase the confidence of the maintainer that a given endpoint is not used, although it can never be used a proof.

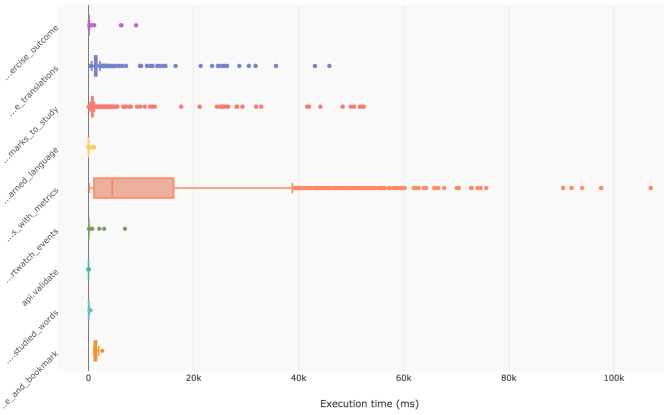

Fig. 4. The response time (in ms) per monitored endpoint view allows for identifying performance variability and balancing issues

```
dashboard.config.git = 'path/to/.git'
```

Listing 1. Configuring the Flask Dashboard with the path to the .git folder enables the generation of evolutionary performance graphs

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

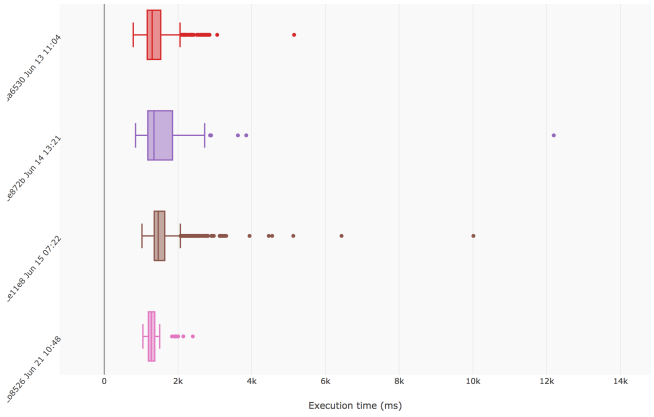

Fig. 5. Visualizing The Performance Evolution of the **api.get_possible_translations** endpoint

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 box plot moved to the left and there are fewer outliers.

The Flask Dashboard also collects **extra information 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 Flask Dashboard tracks for every endpoint a

```
# 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
```

Listing 2. Simply define a custom app-specific function for user retrieval and pass it to the Flask Dashboard to group information by user

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 Experience

For service endpoints which run computations in real time, the operator of a system must understand the endpoint performance on a per-user basis, especially for situations where the system response time is a function of some individual user load[12]

Flask 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 utilizes again the common expectations of Flask applications which offers a global `request` object which contains a `session` object which encapsulates information:

In the Zeeguu case study, one of the slowest endpoints, and one with the highest 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.

Figure 6 shows some of the results of calling the **api.get_article_difficulties** endpoint for various users. The figure shows that the response times for this endpoint can vary considerably for different users.

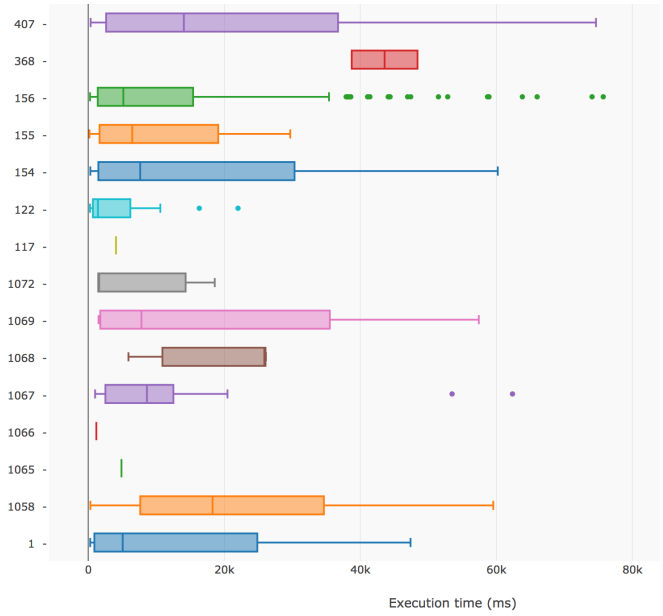

Fig. 6. The **api.get_article_difficulties** shows a very high variability across users

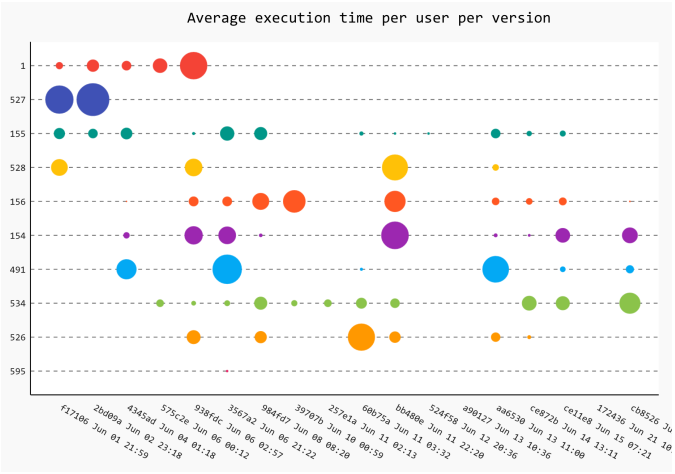

Fig. 7. Caption here

The limitation of the previous view is that it does not present the information also on a per version basis. To address this, a different visual perspective entitled `Evolving per-User Performance` can be defined. Figure 7 attempts to present the information that would be required in such an perspective:

For the situations in which the user information is not available, the Flask Dashboard tracks out of the box information about different IPs. In some cases this might be a sufficiently good approximation of the user diversity and identity.

[12]E.g. in GMail some users have two emails while other have twenty thousand and this induces different response times for different users

VII. Discussion

A. Integrating with Different Deployment Strategies in Order to Automatically Monitor System Evolution

The main goal of the Flask Dashboard design was to allow analytics to be collected and insight to be gleaned by making the smallest possible changes to the running API. To allow the collection of evolutionary information

This technique assumes that the web server code which is the target of the monitoring is deployed using `git` in the following way:

1) The deployment engineer pulls the latest version of the code from the integration server; this will result in a new commit being pointed at by the HEAD pointer than previously
2) The deployment engineer restarts the new version of the service. At this point, the Flask Dashboard detects that a new HEAD is present in the local code base and consequently starts

associating all the new data points with this new commit[13]

The advantage of this approach is the need for minimal configurability. The disadvantage is that it will consider the smallest of the commits[14] and the shortest lived commits[15], as a distinct way of grouping the data points.

Another possible extension point here is supporting other version control systems (e.g. Mercurial). However, this is a straightforward extension.

B. User-Awareness

User awareness as presented in Section VI is useful, but the scalability of the approach must be taken into consideration. In our case study we had about several hundreds users (of which about two hundred were active during the)

C. Utilization

There are other perspectives on service utilization that could be useful, that we did not list in this paper. For example, if the service is using OAuth, then together with every request, in the header of the request there is information about the application which is sending a request. But providing statistics about

D. Tool and Source Code Availability

Flask Dashboard is implemented for Python 3.6 and is available on the Python Package Index repository[16] from where it can be installed on any system that has Python installed by running `pip install flask_dashboard` from the command line.

The code of Flask Dashboard is published under a permissive MIT license and is available on GitHub.[17]

The images in this paper are screenshots of the interactive visualizations from the deployment of Flask Dashboard in the context of the Zeeguu core API. The actual deployment can be consulted online by the reviewers and readers of this article[18].

VIII. Related Work

There is a long tradition of using visualization for gaining insight into software performance. Tools like Jinsight [7] and Web Services Navigator [8] pioneered such an approach for Java and for Web Services that communicate with SOAP messages. Both have an "omniscient" view of the services / objects and their interactions. As opposed to them, in our work we present an analytics platform which focuses on monitoring a single Python web service from its own point of view.

Run-time monitoring of services [9]

An existing monitoring tool is Pingdom [19], which monitors the uptime of an existing web-service. This tool works by pinging the websites (up to 60 times) every minute automatically. Thus this creates a lot of overhead and is bound to be noisy since it will also be influenced by the speed of the network connection[20]

Todo: Runscope? Others?

[13]The Flask Dashboard detects the current version of the analyzed system the first time it is attached to the application object, and thus, assumes that the Flask application is restarted when a new version is deployed. This is in tune with the current version of Flask, but if the web server will support dynamic updates in the future, this might have to be taken into account

[14]Even one that modifies a comment

[15]A commit which was active only for a half an hour before a new version with a bug fix was deployed

[16]http://pypi.org/TODO

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

[18]https://zeeguu.unibe.ch/api/dashboard. Username: *guest*, password: *vissoft*

[19]https://www.pingdom.com/company/why-pingdom

[20]Another problem is that such a tool would

IX. Conclusion and Future Work

In this paper we have shown that it is possible to build a lightweight monitoring solution which provides plenty of feedback to a user with very little effort.

In the future we plan to investigate ... ML: TODO

- usability study with more systems: to improve the visualizations, discover other needs, wean out the less useful visualizations; we plan to announce it in the community
- supporting multiple deployments of the same applications across multiple nodes (e.g. for the situations where the application is deployed behind a load balancer)
- integration with testing as a complementary source of performance evolution

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] A. Ronacher, "Quickstart," http://flask.pocoo.org/docs/0.12/quickstart/#quickstart, 2010, [Online; accessed 25-June-2017].

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

[4] 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

[5] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *IEEE Visual Languages*, College Park, Maryland 20742, U.S.A., 1996, pp. 336–343.

[6] 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

[7] 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.

[8] W. D. Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. F. Morar, "Web services navigator: Visualizing the execution of web services," *IBM Systems Journal*, vol. 44, no. 4, pp. 821–845, 2005.

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