A Low-Effort Analytics Platform for Visualizing Evolving Flask-Based Python Web Services

Patrick Vogel, Thijs Klooster, Vasilios Andrikopoulos, Mircea Lungu

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

***Abstract*—Tens of thousands of web applications are written in Flask, a Python-based web framework. Despite a rich ecosystem of extensions, there is none that supports the developer in gaining insight into the evolving performance of their service. In this paper, we introduce Flask Dashboard, a library that addresses this problem. We present the ease with which the library can be integrated in an already existing web application, discuss some of the visualization perspectives that the library provides and point to some future challenges for similar libraries.**

I. Introduction

There is no getting around it: you are building a distributed system argues a paper from almost half a decade ago [1]. Indeed, since then, even the simplest second-year student project is a web application implemented as two-tier architectures with a front-end implemented in Javascript and HTML 5 and a service backend, usually a REST API.

Python is one of the most popular programming language choices for implementing the back-end 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]).

Within the Python world, Flask[4] is a very popular 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].

Despite their popularity, to the best of our knowledge, there is no simple solution for monitoring the evolving performance of Flask web applications. Thus, every one of the developers of these 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 tool which treats the subject API as a black-box (e.g. Pingdom, Runscope).
2) Implement their own ad-hoc analytics solution, having to reinvent basic visualization and interaction strategies.
3) Live without analytics insight into their services.

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, even adopting 3rd-party analytics solutions, 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 and no integration with the development life cycle.

To avoid projects ending up in the third situation, that of living without analytics, in this paper we present Flask Dashboard — a low-effort service monitoring library for Flask-based Python web services that is easy to integrate and provides multiple perspectives on performance and utilization.

As a case study, on which we will illustrate our solution, we are going to use an open source API which was in the third situation presented above for more than one year:

II. Case Study

Zeeguu[6] 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 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.

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

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 [7].

III. The Flask Dashboard

In this paper we are introducing Flask Dashboard, a drop-in Python library that allows developers to monitor their Flask-based Python web applications with minimal effort.

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[8] 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.

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

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. Data collected by the wrappers are persisted in a local database.

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

In the remainder of the paper we present several of these perspectives.[9]

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[10]. Figure 2 in particular shows that at its peak the API has about 12.000 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, and
2) `api.report_exercise_outcome` is an indicator of the amount of foreign vocabulary practice the users are doing.

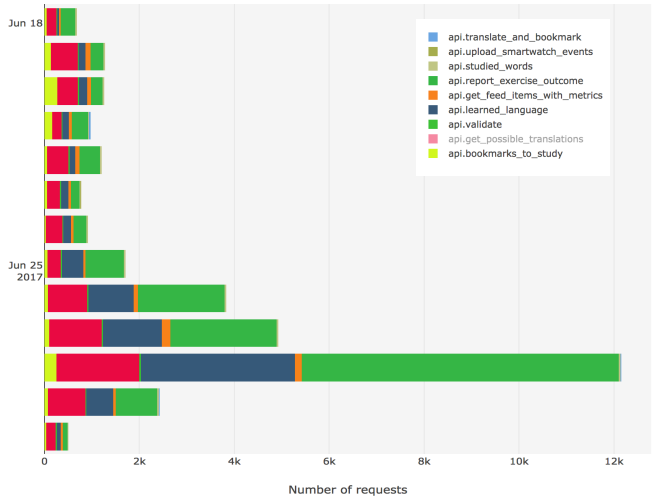

Fig. 2. The number of requests per endpoint per day view shows the overall utilization of the monitored application

Besides showing the overall utilization, this endpoint provides 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 our case study, the maintainer realized that several endpoints which they thought were not being used, contrary to their expectations, were actually very much in use[11].

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

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

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

[10] Endpoint colors are the same in different views

[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.

A second type of utilization question that the Flask Dashboard can answer automatically regards *cyclic patterns of usage per hour of day* by means of a heatmap, as in Fig. 3.

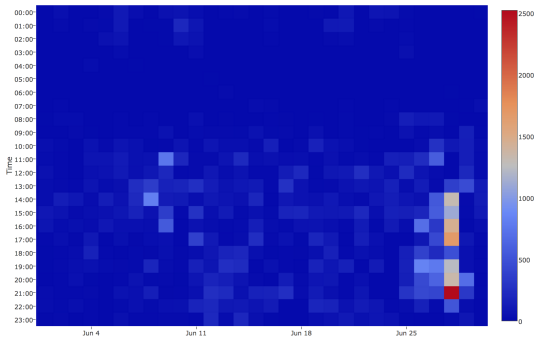

Fig. 3. Usage patterns become easy to spot in the requests per hour heatmap

Figure 3 shows the API not being used during the early morning hours, with 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. Also, the figure shows that the spike in utilization that was visible also in the previous graph happended in on afternoon/evening.

V. Endpoint Performance

The Flask Dashboard also collects information regarding endpoint performance. The view in Fig. 4 summarizes the response times for various endpoints by using a box-and-whiskers plot.

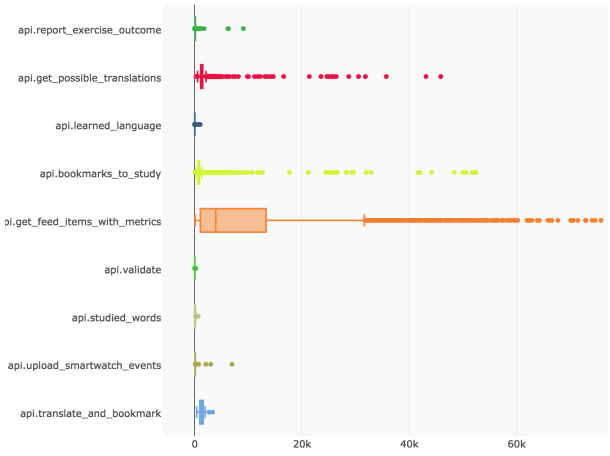

Fig. 4. The response time (in ms) per monitored endpoint view allows for identifying performance variability and balancing issues

From this view it became clear to the maintainer that four of the endpoints had very large variation in performance. The most critical for the application and consequently the one optimized first was the `api.get_possible_translations` endpoint which was part of an interactive loop in the reader applications that relied on the Zeeguu API.

However, with the current configuration of the tool, it would be impossible for the maintainer to see the improvements resulting from the optimization. One way to do this is to add an extra line of configuration to Flask Dashboard find the git[12] folder of the deployed service:

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

With this extra configuration, the Flask Dashboard can now automatically detect the current version of the project, and group measurements by version[13]. Fig. 5 is a zoomed-in version of such a view for `api.get_possible_translations` with versions increasing from top to bottom

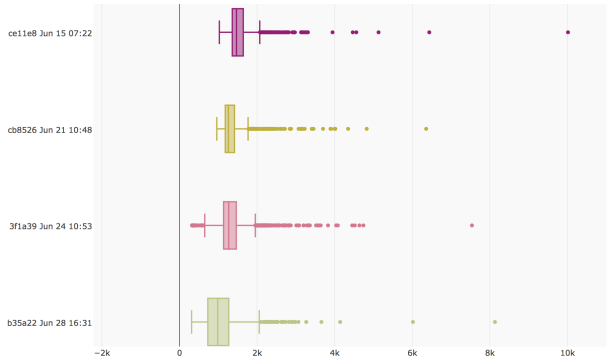

Fig. 5. Visualizing The Performance Evolution of the `api.get_possible_translations` endpoint

This view confirms that the performance of the translation endpoint improved in the recent versions: the median of the last three versions is constantly moving towards the left, and progresses from 1.4 seconds (in the top-most box plot in Fig. 5) to 0.8 in the latest version (bottom-most box plot).

The Flask Dashboard collects **extra information about such 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 without degrading overall performance, the Flask 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.

[12]https://git-scm.com/

[13]Alternatively, the maintainer can add version identifiers manually for the web application through a configuration file if the system does not use git.

VI. User Experience

For service endpoints which run computations in real time, the maintainer 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[14].

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

```
# 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
```

In the Zeeguu case study, one of the slowest endpoints, and one with the highest variability as shown in Fig. 4 is `api.get_feed_items_with_metrics`: 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_feed_items_with_metrics` endpoint for various users. The figure shows that the response times for this endpoint can vary considerably for different users.

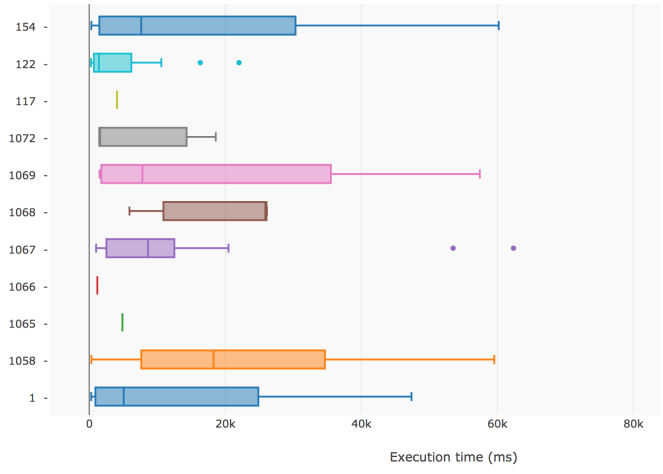

Fig. 6. The `api.get_feed_items_with_metrics` shows a very high variability across users

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

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.

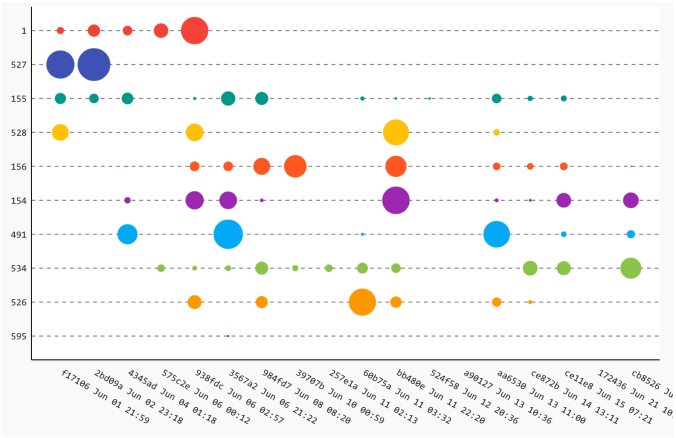

Fig. 7. This perspective shows that the evolution of response times for individual users (horizontal lines) across versions (the x-axis)

For the situations in which the user information is not available, the Flask Dashboard also 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.

In the interest of space limitations we are not showing this, and quite a few more visualizations provided by the Flask Dashboard. The interested reader is referred to Sec. VII-D for further information on how to install the tool and how to get access to the data of the case study.

VII. Discussion

A. Automatically Monitoring System Evolution

The main goal of the Flask Dashboard design was to allow analytics to be collected and insight to be gained by making the smallest possible changes to the running API. This technique assumes that the web application 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.
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[15].

[15]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

The advantage of this approach is the need for minimal configuration effort, as discussed in the presentation of the tool. The disadvantage is that it will consider on equal ground the smallest of commits, even one that modifies a comment, and the shortest lived of commits, e.g. a commit which was active only for a half an hour before a new version with a bug fix was deployed, with major and minor releases of the software. A mechanism to control which versions are important for monitoring purposes is therefore required to be added. A further 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 course of the study). However, different perspectives might be required for APIs with millions of users.

C. Other Possible Groupings

There are other groupings of service utilization and performance that could be important to the maintainer, that we did not explore 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. Grouping the information by application that sends the request could be important in such a context. In general, providing a mechanism that would allow very easy specification of groupings (either as code annotations, as normal code, or as configuration options) is an open problem that Flask Dashboard and any other similar library will have to face.

D. Tool and Source Code Availability

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[16].

Flask Dashboard is implemented for Python 3.6 and is available on the Python Package Index repository[17] from where it can be installed on any system that has Python installed by running `pip install flask_dashboard` from the command line.

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

From the perspective of service monitoring, our work falls within the server-side run-time monitoring of services [9]. While we don't implement the more advanced features of related monitoring solutions like QoS policies driving the monitoring, it presents nevertheless an easy to use approach support improving the performance of web applications.

IX. Conclusion and Future Work

In this paper we have shown that it is possible to build a low-effort monitoring solution which provides plenty of feedback to a user with very little effort. The user group that we are aiming for with this work is application developers using Flask and Python to build web applications with limited or no budget for implementing their own monitoring solutions. The emphasis is in allowing such users to gain insight into how the performance of the service evolves together with the application itself.

In the future, we plan to perform case studies with other sytstems, with the goal of discover other needs and to wean out the less useful visualizations in the Flask Dashboard. We plan to also extend the tool towards supporting multiple deployments of the same applications across multiple nodes (e.g. for the situations where the application is deployed together with a load balancer). Finally, we plan to integrate Flask Dashboard with unit testing as a complementary source of performance evolution.

[16]https://zeeguu.unibe.ch/api/dashboard; username: *guest*, password: *vissoft*

[17]https://pypi.python.org/pypi/flask-monitoring-dashboard/1.8

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

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.