Towards a reference dataset of microservice-based applications

Antonio Brogi, Andrea Canciani, Davide Neri,
Luca Rinaldi, and Jacopo Soldani

Department of Computer Science, University of Pisa, Italy

Abstract. The microservice-based architectural style is rising fast in enterprise IT. Tools and solutions for supporting microservices-based applications are proliferating. It is however often difficult to qualitatively/quantitatively assess and compare such tools and solutions, also because of the lack of reference datasets of microservice-based applications. The objective of this paper is precisely to set the ground of a first reference dataset of microservice-based applications.

1 Introduction

The microservice-based architectural style proposes to develop *"a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API"* [6]. Each microservice implements a precise business capability, and it can be deployed and scaled independently from all other microservices forming an application [9,18].

Microservices are already pervading enterprise IT [3]. Various companies are already delivering their business services with microservice-based applications, with Amazon and Netflix being the most prominent examples. As a consequence, researchers and practitioners are rushing to provide an adequate support to such companies. This resulted in a rapid proliferation of heterogeneous tools and solutions (e.g., Docker Compose [4], Apache Mesos [16], Kubernetes [17]), which aim at offering an enhanced support for developing, deploying, and/or managing microservice-based applications [3].

It is however often difficult to assess and compare existing solutions, as well as to show that a newly proposed solution is actually enhancing the support already provided by its competitors (e.g., by supporting additional functionalities, or by running the same business with better performances). This is also due to the lack of reference datasets of microservice-based applications. Such datasets would indeed permit developing a set of repeatable experiments for evaluating a solution (e.g., an orchestrator of microservice-based applications), or for comparing it with other existing solutions [11].

The main objective of this paper is to set the ground of μSET, a first reference dataset of microservice-based applications. The μSET dataset will permit assessing and evaluating a solution offering support for microservice-based applications. It will indeed provide a set of microservice-based applications, each permitting to evaluate a solution qualitatively (viz., by checking whether it supports

one or more desired functionalities) and/or quantitatively (viz., by measuring its performances). We also present a first set of microservice-based applications that are already included in μSET.

Beside checking whether a solution supports desired functionalities with desired performances, the applications contained in μSET will be exploitable by practitioners and researchers in two other ways. On the one hand, μSET dataset will permit performing a systematic comparison of existing solutions. The latter will then ease the choice of the most appropriate solution for developing, or managing a microservice-based application. On the other hand, μSET dataset will permit developing a set of repeatable experiments, to show that a newly proposed solution actually enhances the support provided by its competitors.

The rest of the paper is organised as follows. Sect. 2 presents the design of μSET, as well as the microservice-based applications it already contains. Sect. 3 presents an example of evaluation based on the applications currently available in μSET. Sects. 4 and 5 discuss related work and provide some concluding remarks.

2 The μSET dataset

We hereby present the design of the μSET dataset of microservice-based applications. The main requirements driving the design and population of μSET are the following:

- All applications in μSET will be publicly available.
- μSET will contain applications easy to understand and manage, as this will simplify setting up repeatable experiments based upon them.
- All applications in μSET will permit evaluating a solution qualitatively (by checking whether it supports a desired functionality) or quantitatively (by measuring the performances of a solution).

To satisfy the first requirement, we implemented μSET as a public GitHub repository[1]. We also started populating the dataset with a first set of microservice-based applications that, much in the spirit of unit testing [13], we crafted "ad-hoc" to test the support of a specific functionality.

More precisely, we selected five functionalities that are crucial for supporting microservice-based applications. Namely, we selected the functionalities to support *communication* between microservices [6], *fault resilience* [10], *horizontal scalability* [6], *replaceability* of a microservice [9], and *extensibility* of a microservice-based application [9].

We then defined five categories of microservice-based applications (viz., *communication support*, *fault resilience*, *horizontal scalability*, *replaceability*, and *extensibility*), one for each of the above mentioned functionalities. Each category will contain microservice-based applications aimed at "unit testing" whether/how a solution supports the corresponding functionality. For instance, an application

[1] `https://github.com/di-unipi-socc/microset`.

in the *horizontal scalability* category can be used both to (qualitatively) check whether a solution supports horizontal scalability and to (quantitatively) measure the performances of such solution when scaling its microservices (e.g., time and resources needed to scale in/out a microservice).

We then developed one application for each category, each designed to be as simple as possible (but not simpler). We hereafter present such applications, by first explaining their rationale, by describing the applications themselves, and by finally showing an example of how they permit checking whether an orchestrator of microservice-based applications supports the corresponding functionalities.

2.1 Communication support

Rationale. Microservices need to communicate each other via lightweight mechanisms, often being HTTP resource APIs. Any solution aiming to support microservice-based applications should hence provide functionalities to support communication between microservices.

Description. The `communication-support` application is composed by two microservices, viz., *backend* and *frontend* (Fig. 1). The *backend* microservice offers

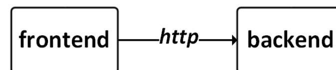

Fig. 1. Microservices in the `communication-support` application.

an HTTP API. The latter exposes just one operation, requiring no input parameters, and returning a JSON object containing a randomly generated number.

The microservice *frontend* offers an HTTP endpoint, through which it serves automatically generated HTML pages. Whenever a client connects to *frontend*, the latter invokes the API of *backend*. If *backend* answers to *frontend*, *frontend* returns an HTTP 200 response containing a HTML page rendering the information obtained from *backend*. Otherwise, *frontend* returns an HTTP 500 error.

Example of test. An orchestrator of microservice-based applications supports communication if it can deploy and interconnect *frontend* and *backend*, so that one can afterwards connect to *frontend*, and *frontend* returns an HTTP 200 response. If *frontend* instead returns an HTTP 500 error, this means that it is not able to communicate with *backend*.

2.2 Fault resilience

Rationale. Microservice-based applications should be designed by taking into account that their microservices may fail. Any solution supporting microservices-based applications should hence manage failures by allowing to restart failed microservices.

Description. `fault-resilience` is composed by a single microservice (*app*), which can be used to check whether a solution is capable to manage a microservice failure (Fig. 2). *app* is designed to fail after a given period of time (10 seconds, by default).

Fig. 2. Failure behaviour of the microservice *app* in the `fault-resilience` application.

Example of test. An orchestrator of microservice-based applications supports failures if it can automatically restart *app* whenever it fails. This can be tested by checking whether *app* is running and responding right after the failure period is expired.

2.3 Horizontal scalability

Rationale. Microservices are independently deployable and scalable by definition. Being able to horizontally scale a microservice is fundamental to improve the performances of a microservice-based application.

Description. `horizontal-scalability` is a microservice-based application designed to check whether a solution can deal with the horizontal scaling of a microservice. It is composed by two microservices, *consumer* and *producer*, communicating via HTTP.

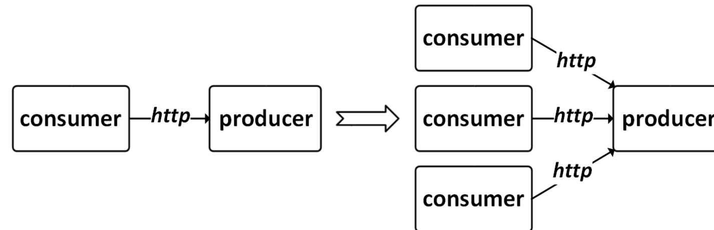

Fig. 3. An example of how to scale the number of microservices in the `horizontal-scalability` application.

The *producer* is a web service generating a finite stream of random numbers, and offering an endpoint to consume the numbers in the stream. The *consumer* is a script iteratively invoking the endpoint offered by the *producer*. Each time a *consumer* invokes such endpoint, the *producer* returns the next number available in the stream, until all numbers have been consumed.

The *producer* also records the time interval needed to consume all numbers and the amount of different *consumer*s that have required at least a number.

Fig. 3 shows an example of how to scale the microservices in `horizontal-scalability`. Initially, only one *consumer* and one *producer* are running. The number of *consumer*s is then increased to three, to reduce the time needed to consume the stream.

Example of test. An orchestrator of microservice-based applications supports horizontal scalability if it permits changing the amount of running *consumer*s. This can be tested, for instance, by scaling the *consumer*s as in Fig 3, and by checking that the *producer* returns 3 as the amount of different *consumer*s that consumed at least a number[2].

2.4 Replaceability

Rationale. Replaceability is the ability to replace a microservice with another microservice offering the same functionality. Solutions that aim to support microservice-based applications should hence permit replacing a microservice independently of the others.

Description. `replaceability` is a microservice-based application that permits checking whether a solution supports the replaceability of a microservice with another. It is composed by two microservices, viz., *frontend* and *backend* (Fig. 4).

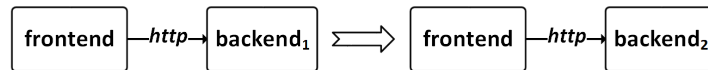

Fig. 4. Two consecutive configurations of `replaceability` application.

The *backend* microservice offers an HTTP API endpoint that returns a randomly generated number each time is called. It is available in two different implementations, namely $backend_1$ and $backend_2$. $backend_1$ returns only even random numbers, while $backend_2$ returns only odd numbers.

The *frontend* microservice serves HTML pages. Whenever a client connects to *frontend*, the latter invokes the API of the backend. If the backend answers to *frontend*, then *frontend* returns an HTTP 200 response containing an HTML page rendering the information obtained from backend. Otherwise, *frontend* returns an HTTP 500 error.

Example of test. An orchestrator of microservice-based applications supports replaceability if, after deploying *frontend* with $backend_1$, it permits replacing $backend_1$ with $backend_2$ without restarting or reconfiguring the *frontend*. Moreover, when the $backend_1$ is deployed, the *frontend* must return even numbers, while with $backend_2$ the *frontend* must return odd numbers.

[2] The length of the stream can be configured so that all *consumer*s can consume at least one number.

2.5 Extensibility

Rationale. Extensibility is the ability to add and integrate a new microservice in a microservice-based application independently of its other microservices. Solutions offering support for microservice-based applications should hence permit adding a new microservice in a existing application (without requiring to reconfigure the other microservices in such application).

Description. `extensibility` is a microservice-based application composed by three microservices, viz., $frontend_1$, $frontend_2$, and *backend* (Fig. 5). The *backend* microservice offers an HTTP API, which exposes one operation returning a JSON object containing a randomly generated number.

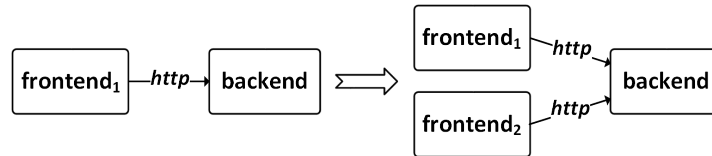

Fig. 5. Two consecutive configurations of the `extensibility` application.

$frontend_1$ and $frontend_2$ invoke the API of *backend* to get a randomly generated number, but they implement two different operations. $frontend_1$ returns the largest prime factor of the number received from the *backend*. $frontend_2$ checks whether the number received from the *backend* is prime or not. Both $frontend_1$ and $frontend_2$ render the response in a HTML page.

Example of test. An orchestrator of microservice-based applications supports extensibility if, after deploying $frontend_1$ and *backend*, it is possible to add $frontend_2$ without touching the other components. Afterwards, both $frontend_1$ and $frontend_2$ must return an HTTP 200 response (when invoked), as this means that they can communicate with *backend*.

3 μSET at work

In order to test whether a solution (e.g., an orchestrator of microservice-based applications) supports a functionality, it is first required to select an application from μSET that test such functionality and then build a test on top of it. The test should check whether the solution permits describing the application, running it and finally verifying that application behaves as expected. This testing approach can be iterated over all the functionalities which are considered by the μSET dataset consider, in order to determine the functionalities provided by a solution.

We now show how we used the above approach to check whether Docker Compose[3] supports all functionalities discussed in Sect. 2 (viz., *communication support*, *fault resilience*, *horizontal scalability*, *replaceability*, and *extensibility*)

We specified each microservice-based application of μSET in Docker Compose by means of a compose file (which combines the containers packaging the microservices forming an application). We then developed a set of test scripts, each executing the compose file of an application and checking whether the corresponding functionality is supported[4]. Below we provide some additional details on the test we developed, and we discuss the results we obtained.

Communication support. We created a compose file that specifies the two containers packaging *frontend* and *backend*, and the connection link from *frontend* to *backend*. We created a test script that runs such compose file (to build, start, and interconnect *frontend* and *backend*). It then performs a HTTP request to *frontend*, and it checks whether the latter returns a HTTP 200 response.

Docker Compose passed the test we created. This is because it deploys first the container of *backend* and then the container of *frontend*, and since it creates a bridge network allowing such containers to communicate.

Fault resilience. We created a compose file that defines a container packaging the *app* microservice, and we indicated that such container must be restarted whenever it exits with an error (by setting the option `restart` of the container of *app* to `on-failure`). We created a test script that runs the above mentioned compose file, and which checks whether the container of *app* is restarted by Docker Compose after its first failure (by invoking *app* and verifying that it returns an HTTP 200 response).

Docker Compose passed the above explained test. This is thanks to option `restart: on-failure` of the container of *app*, which instructs Docker Compose to automatically restart such container whenever it fails.

Horizontal scalability. We created a compose file that defines the two containers packaging *producer* and *consumer*. We also created a test script that exploits such compose file to first run one *producer* and one *consumer*, and which then scales the number of *consumer*s up to three (by using the `docker-compose scale` command). The script also verifies that three different *consumer*s actually interacted with the *producer*.

Docker Compose passed also this test. It indeed natively supports the horizontal scalability of the containers forming an application, which can be scaled out/in by executing the `docker-compose scale` command.

Replaceability. We created a compose file that defines the two containers packaging *frontend* and *backend*, by also indicating that the actual implementation

[3] Docker Compose is an engine which permits deploying and managing multi-container Docker applications. Docker Compose permits describing the components of an application by using a `compose file` (specification file written in YAML). It is possible to find more information at [14].

[4] All compose files and test scripts are available in the GitHub repository of μSET.

of *backend* is $backend_1$. We then created a copy of such compose file, and we modified such copy by substituting the actual implementation of *backend* (from $backend_1$ to $backend_2$). Finally, we created a test script that first executes the initial compose file (hence running *frontend* and $backend_1$), and which checks that the actual implementation of *backend* can be changed from $backend_1$ to $backend_2$ by running the modified compose file.

Docker Compose passed also the above test. It can indeed detect the changes that occurred in the compose file describing a running application. Such changes are then processed by re-building and re-deploying only the containers they affect (without touching the other containers in the application).

Extensibility. We created a compose file describing the containers packaging $frontend_1$ and *backend*. We then created a copy of such compose file, and we modified such copy by adding the container packaging $frontend_2$. We created a test script that first executes the initial compose file (hence deploying $frontend_1$ and *backend*), and it then executes the modified compose file (to add $frontend_2$). Afterwards, the scripts checks whether $frontend_2$ is actually added to the application (without touching the other containers).

Docker Compose passed the test we developed. It can indeed detect that new containers are added to the compose file describing a running application, which are processed by building and deploying only such containers.

Summary. We developed five tests, from which Docker Compose turned out to support all functionalities covered by the applications currently in μSET.

4 Related work

The need for reference datasets allowing to set up repeatable experiments is widely recognised in computer science [8,12]. For instance, SPEC (*Standard Performance Evaluation Corporation*) is working since 1988 to produce, establish, maintain, and endorse a standardised set of performance benchmarks for computer systems [15]. Concrete examples are the reference applications contained in SPEC CPU2006 and SPEC Cloud_IaaS 2016. The former are designed to provide performance measurements that can be used to compare compute-intensive workloads on different computer systems. The latter instead measure the performances IaaS platforms, by stressing provisioning and runtime aspects of a cloud using I/O and CPU intensive workloads.

Other examples are MiBench [7] and PARSEC [1] which provide reference applications for evaluating embedded systems and shared-memory systems, respectively. DataGov [19] instead offers hundreds of thousands of reference datasets for evaluating approaches for information retrieval/data mining.

A reference dataset allowing to evaluate solutions for supporting microservice-based applications is however, to the best of our knowledge, missing. There exist some demo applications (such as the Sock Shop [20], or those available on eventuate.io [5]), which could be used to compare different solutions based on their performances. Such applications are however designed to demonstrate specific

solutions, and this makes them unsuitable to evaluate the actual performances of different and heterogeneous solutions.

The dataset we propose (viz., μSET) can hence provide a first reference for evaluating and comparing solutions offering support for microservice-based applications. μSET is indeed designed to permit evaluating different and heterogeneous solutions, both qualitatively (by allowing to check whether a solution supports a certain functionality) and quantitatively (by providing a reference to measure the performances of a solution).

Finally, there exists orthogonal approaches (e.g., [2]) that permit testing microservice-based applications. μSET, instead, permits testing solutions supporting the analysis/deployment/management of microservice-based applications.

5 Conclusions

Microservices are pervading enterprise IT, with various companies already delivering their business services with microservice-based applications. To provide an adequate support to these companies, researchers and practitioners are working day-by-day on enhancing the current support for developing, deploying and managing microservice-based applications. The result is an increasing number of heterogeneous solutions, offering similar functionalities in a different manner [3] (e.g., Docker Compose [4], Apache Mesos [16], Kubernetes [17]).

It is often difficult to choose the most appropriate solution fitting our needs, namely a solution offering the functionalities needed by our microservice-based applications with the desired performances. It is also difficult to give evidence that a newly proposed solution is actually enhancing the support provided by its competitors (e.g., by supporting additional functionalities, or by running the same business with better performances). This is because it is difficult to develop a set of repeatable experiments that can be used to evaluate a solution, and to compare its evaluation with respect to that of its competitors [11].

The μSET dataset proposed in this paper starts tackling this issue, by proposing an easy-to-use, reference dataset of microservice-based applications. Such dataset will indeed be exploitable to develop repeatable experiments evaluating solutions qualitatively and/or quantitatively. Its aim is indeed to provide microservice-based applications that permit checking whether a solution supports a desired functionality or measuring its performances (e.g., measuring the time and resources needed to add a new microservice to an already running application, to detect when a microservice fails, and to scale a microservice).

The population of μSET is currently ongoing. μSET already includes a first set of microservice-based applications, which are designed to “unit test” whether a solution provides some fundamental functionalities needed by microservice-based applications. It however requires to be extended to include other applications for checking functionality support (both for already selected functionalities and for functionalities yet to be selected), as well as more complex applications (crafted or taken from real deployments) that permit measuring the performances of a solution. We plan to continue extending μSET dataset as part of our immediate

future work, and we are willing to involve other practitioners and researchers in this extension process.

Finally, it is worth noting that μSET can be used not only to evaluate a single solution, but also to perform a systematic comparison of existing solutions. Solutions can indeed be evaluated by exploiting the microservice-based applications that will be contained in μSET, and this would provide enough information to compare them based on the functionalities and performances they provide. Such a comparison would then be very useful when looking for the most appropriate solutions for a microservice-based application. We hence plan to exploit μSET to systematically compare existing solutions as part of our future work.

References

1. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. pp. 72–81. PACT '08, ACM (2008)
2. de Camargo, A., Salvadori, I., Mello, R.d.S., Siqueira, F.: An architecture to automate performance tests on microservices. In: Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services. pp. 422–429. iiWAS '16, ACM (2016)
3. Di Francesco, P., Malavolta, I., Lago, P.: Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA). pp. 21–30. IEEE Computer Society (2017)
4. Docker, Inc.: Docker-compose. `https://docs.docker.com/compose/` (last accessed on June 16th, 2017)
5. Eventuate, Inc.: Eventuate example applications. `http://eventuate.io/exampleapps.html` (last accessed on June 16th, 2017)
6. Fowler, M., Lewis, J.: Microservices. ThoughtWorks, `https://martinfowler.com/articles/microservices.html` (last accessed on June 16th, 2017)
7. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop. pp. 3–14. WWC '01, IEEE Computer Society (2001)
8. Myers, G.J., Sandler, C.: The Art of Software Testing. John Wiley & Sons (2004)
9. Newman, S.: Building microservices. O'Reilly Media, Inc. (2015)
10. Nygard, M.: Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf (2007)
11. Pahl, C., Jamshidi, P.: Microservices: A systematic mapping study. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2. pp. 137–146. CLOSER 2016, SCITEPRESS (2016)
12. Roper, M.: Software Testing. McGraw-Hill, Inc. (1995)
13. Runeson, P.: A survey of unit testing practices. IEEE Software 23(4), 22–29 (2006)
14. Smith, R.: Docker Orchestration. Packt Publishing (2017)
15. Standard Performance Evaluation Corporation (SPEC): Benchmarks. `http://www.spec.org/benchmarks.html` (last accessed on June 16th, 2017)
16. The Apache Software Foundation: Mesos. `http://mesos.apache.org/` (last accessed on June 16th, 2017)

17. The Kubernetes Authors: Kubernetes. https://kubernetes.io/ (last accessed on June 16th, 2017)
18. Thönes, J.: Microservices. IEEE Software 32(1), 113–116 (2015)
19. U.S. Government: Data.Gov - The home of the U.S. Governments open data. https://www.data.gov (last accessed on June 16th, 2017)
20. Weaveworks, Inc.: Sock shop. https://microservices-demo.github.io (last accessed on June 16th, 2017)