IEEE-SEM, Volume 8, Issue 7, July 2020
ISSN 2320-9151

Test automation in microservice architecture

Nikolay Raychev

Department of Computer Science, Varna University of Management, Varna, Bulgaria.
Email: nikolay.raychev@vumk.eu

ABSTRACT

This is a short review article on how we tested Filipo's monolithic application and what has changed with the transition to microservice architecture. Containers and microservices are becoming increasingly popular, providing additional capabilities. But like any other new technology, they put new challenges on developers. The proposed methodology is systematized as sequence of actions and flexible, therefore it is easy to understand and use. Optional performance of some activities, such as defining non-functional tests, allows the test process to focus on the specific needs of the user and performing step-by-step testing. The proposed methodology can be used by different specialists for different test purposes. Another advantage of the proposed methodology is that it uses a small set of test artifacts. When implementing full coverage tests based on the methodology, it is possible to achieve a high level of automation, where it is necessary to ensure only the tested business process.

Keywords : Test automation, microservice architecture

1. INTRODUCTION

What is the architecture of microservices?

Simplified architecture of microservices is a way to organize an application server. In essence, this is simply a service-oriented response to the emergence of practices such as DevOps. If SOA does not regulate the size of services and what exactly they should do, then there are some speculative limitations within the architecture of microservices. A microservice is a specific entity that contains a small functionality that manages and provides some data to external services.

Sometimes they give information that the microservice is 500 lines of code. But this is not mandatory; the point is that these services are small enough and form the same business processes as a monolithic back-end that works on many projects. Functionally - the same, the differences are in the organizational structure.

Where are the problems here? The number of services is growing rapidly and their connectivity is increasing. This complicates testing: checking the change of a microservice leads to the need to build a fairly thick layer of infrastructure from the many services with which the interacting microservice interacts. At some point in the development of the project, the number of connections and dependencies increases significantly and it becomes difficult to perform rapid and isolated testing. We start working slower. Development is slowing down. These and some other problems can be solved with test automation.

2. TRANSITION TO MICROSERVICES

For starters, consider the transition from a monolith to a microservice architecture. In the case of a monolith, in order to replace each piece in this system, we cannot roll it separately. You need to restore and completely update the back end. It is not always

rational and convenient. What happens when you switch to microservice architecture? We take the back-end and divide it into its constituent components, dividing them by functionality. We determine the interactions between them and get a new system with the same front end and the same databases. Microservices interact with each other and provide all the same business processes. For user and system testing, everything remained as before, the internal organization changed.

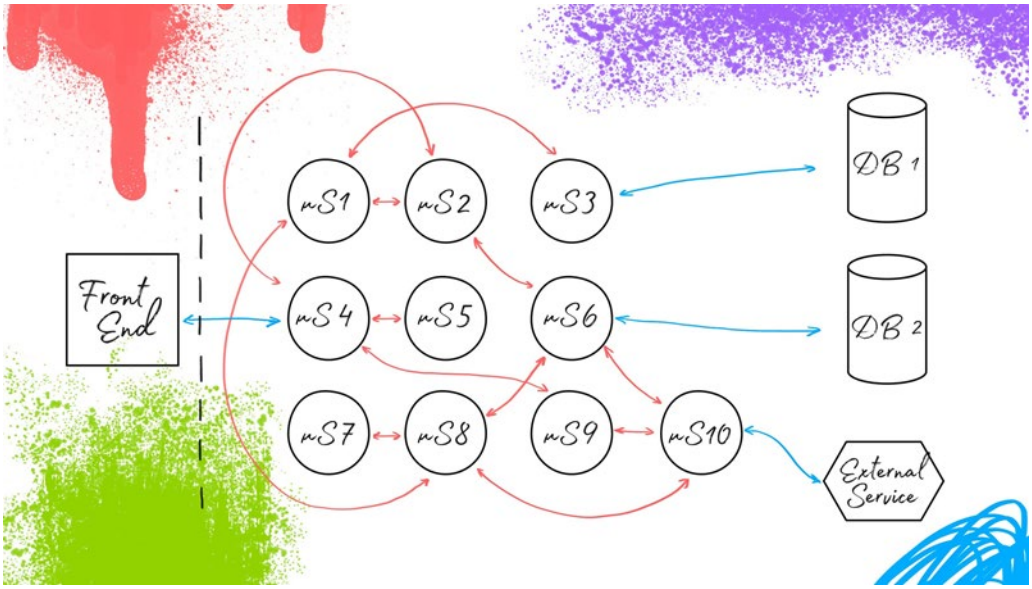


Figure 1 transition to microservices

What is he doing? You can make a change to the X microservice, which implements some of its functionality, and immediately deploy it to make it work. This all sounds very good. But, as always, there is a nuance: you need to check how microservices interact with each other.

a. Microservice architecture contracts

The interactions between the microservices are carried out through contracts. What does this mean for service-oriented architecture? A contract is a type of agreement that service developers create for external users to interact with each other. Service developers decide for themselves that you can ask them for an "X" and they will give you a "Y". The service can provide apples, tomatoes, plutonium to external users, sell children's clothes or TVs - it does not declare a clear focus on functionality. The internal content is regulated only at the level of common sense. Accordingly, this service dictates to external users what they will receive and how to access it.

If we increase the complexity of this task, when there will be no dozens of services (10-20-50), but hundreds (200, 400 ... 2000), then the traditional, "classic" contracts cease to work in a functional sense. And then it is necessary to modernize microservice architecture contracts. Developed a scheme of work for developers, in which they and the "end users" of the microservice exchange - this approach is called user contacts. Requests are now made by external users. Think of this type of conversation:

User 1: "I heard you were delivering apples. I need small and green. "User 2:" And I need huge red apples. "User N:" And I need you to bring three tons of apples. "

It turns out that consumers create contracts, specify requirements and direct them to the provider who fulfills them. What is the

plus? Since each service has a very limited number of users, getting three specifications from each and implementing them is much easier than trying to guess which "apples" should be provided in each case and what else might be needed. to add to these "fruits".

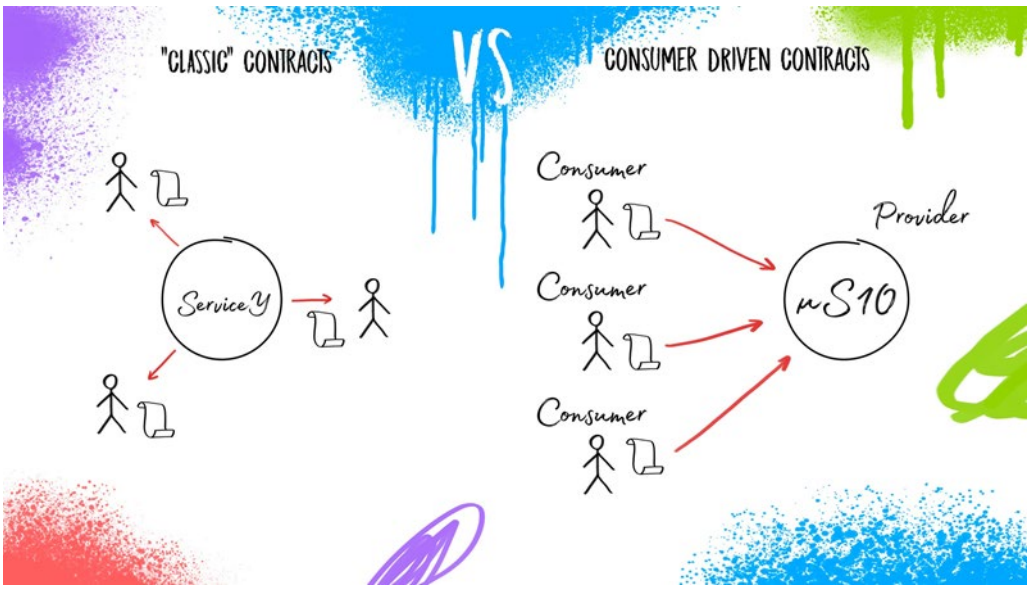


Figure 2 Microservice architecture contracts

For this approach, we, as a quality assurance department, must give a decent answer in the field of testing, so as not to slow down the process of rapid introduction of microservices.

b. CDC testing predicates (user-managed contracts)

What can't start working on CDC testing principles? The first. We will not be able to help if we see that our development process does not follow the process of working under the contract. The second thing is more technical: it's a post-acceptance (after PR, if you will) system that handles this communication flow between developers using contracts and signals our testing system to update, delete, and new ones appear. Accordingly, the relevant tasks are set in Jira so that the automators have time to "master" them. You can add everything else to this basic process - additional checks, gadget processing, but without control over the change of contracts for interaction between microservices will be difficult to live. Once we have completed these two points in some form, we can proceed with the implementation.

Implementation of automation for CDC

Our test automation system will be based on such a solution as PACT frameworks. What do you need to know about them? This is a protocol interaction with API: JSON over HTTP, there is nothing complicated about it. These solutions interact with our microservices and provide some additional functions for the organization of insulation and testing. What else can I say? I saw how this is implemented in one form or another in seven programming languages (Java, Javascript, Ruby, Python, Go, .NET, Swift). But if yours isn't on this list, don't worry: you can take a basic library and make your own bike, or write something similar to what's already implemented.

What else awaits us in almost 100% of cases? The first is a mockery of outside services. The problem is that it is difficult to main-

tain the relevance of how the external service behaves and to maintain this in all the little ones. How to solve this problem? Depends on you. It will probably be necessary to direct more resources to this or to limit the scope of the tests. And second: excerpts from databases. You will need to cut the combat databases into a light form so that they contain all the necessary data for testing in our system. Then you get a visually relevant result.

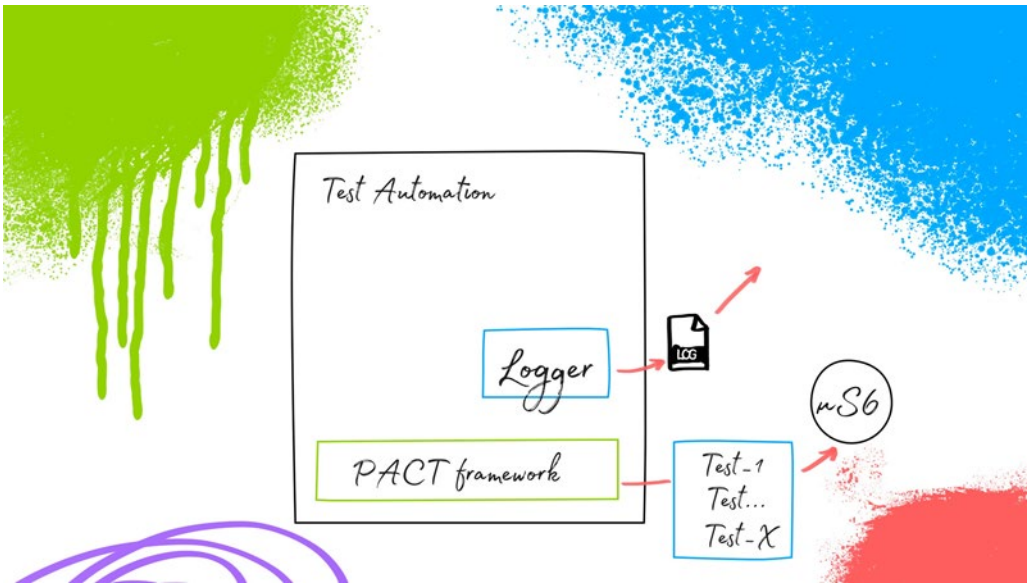


Figure 3 Implementation of automation for CDC

Let's move on. The first thing I immediately wanted to do in this project was informative logging. Representation of test results is always required. The simple truth: if the results are impossible to read and understand, no one will look at them. I must say that basic registration under the PACT is very weak.

I prefer to put it in a separate module, wrap it and wrap what we need there. First, divide the errors according to their sources. Next - everything you can implement, such as solutions such as Allure (a solution from Yandex to increase the ease of analyzing test results). You can do anything, but the need for information diaries must be taken into account.

The next, probably "captain's" moment is Config Reader. But as part of testing the architecture of microservices, this can be a bit complicated. We have two sources for Config Reader. The first is the PACT files and the second is the State.

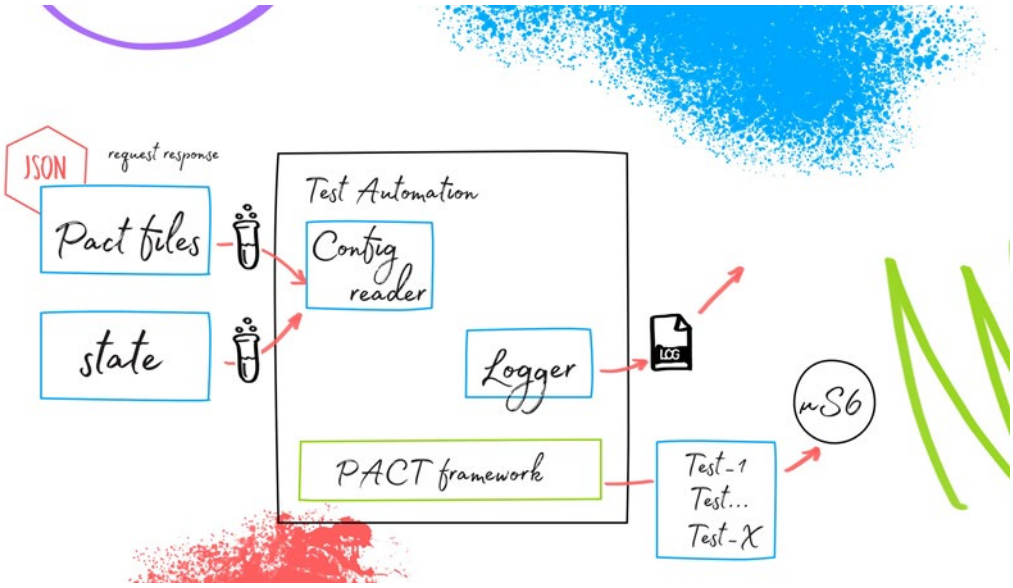


Figure 4 Config Reader

What is a .pact file? This is not some kind of encrypted binary file, but a simple JSON file that has a specific structure. In it, the consumer, the supplier is distributed (ie who and with whom interacts within the framework of this contract and in what role). The following describes the interactions (this is done by the developers): I am a user and I want this service (provider) to give me "little green apples"; waiting for such and such a response code, status, title, body, etc. There is a description field - it's just a description, a reason to remind developers what the contract was about, what meaning was invested in it.

And the most interesting is the state supplier. What is? In fact, this is the state in which the tested micro-service must be at the time of calls to it for a specific iteration of testing, for a specific request. States can describe both SQL queries (or other mechanisms for bringing a service to some state) and the creation of some data in our sample database. States is a complex module that can contain all types of entities that bring our service to the desired correct state.

It is important to note that the Runner Suite appears here (see the diagram below). This is the organization that will be responsible for conducting and configuring tests in a form convenient for the programmer / tester. It may not be written, but I would still highlight this point, as it is difficult to predict which tests will have to be run at one time or another on the project. As a result, we obtain the following core for test automation of microservice architecture:

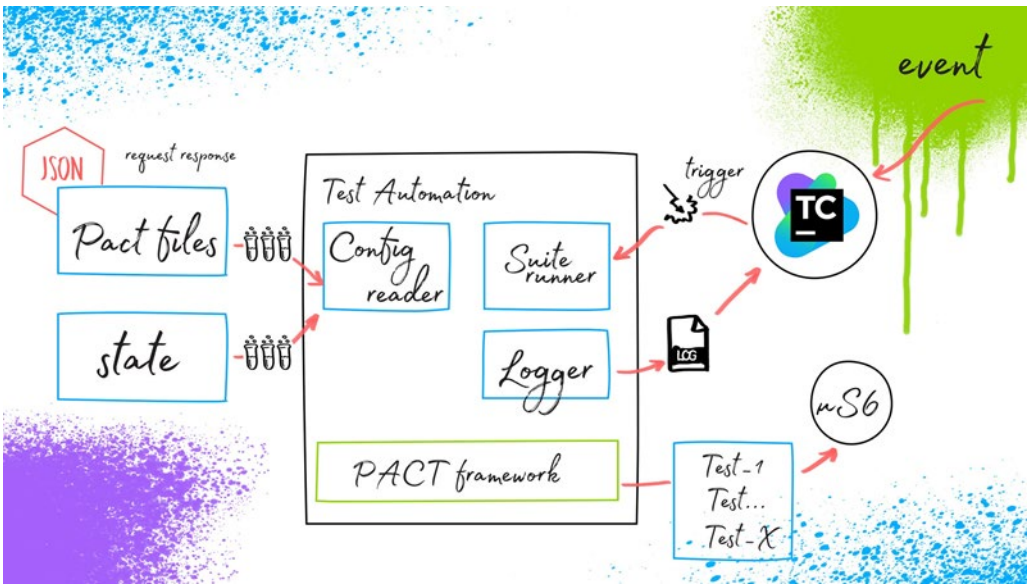


Figure 5 The implementation

The most important thing is the implementation. We must provide this scheme with the requirements for the presence of the described contracts to the developers. What do we get next?

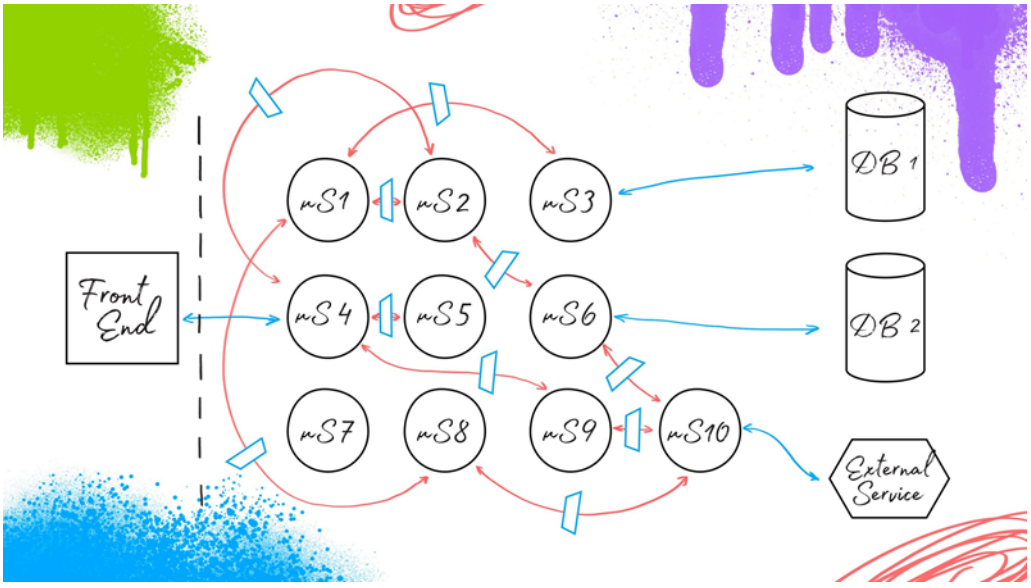


Figure 6 The implementation

We have information about each service: what it gives and what it wants from its "neighbors". Accordingly, using our test automation system using PACT files, we isolate our microservice to ensure the isolation of its testing, regardless of the external services with which it is integrated. And we provide states through layouts, stubs, database samples, or directly in some way to

change the service. We receive respectively isolated tests. Voila: we have the answer to the question of what to do with test automation during the transition from a monolithic to a microservice architecture.

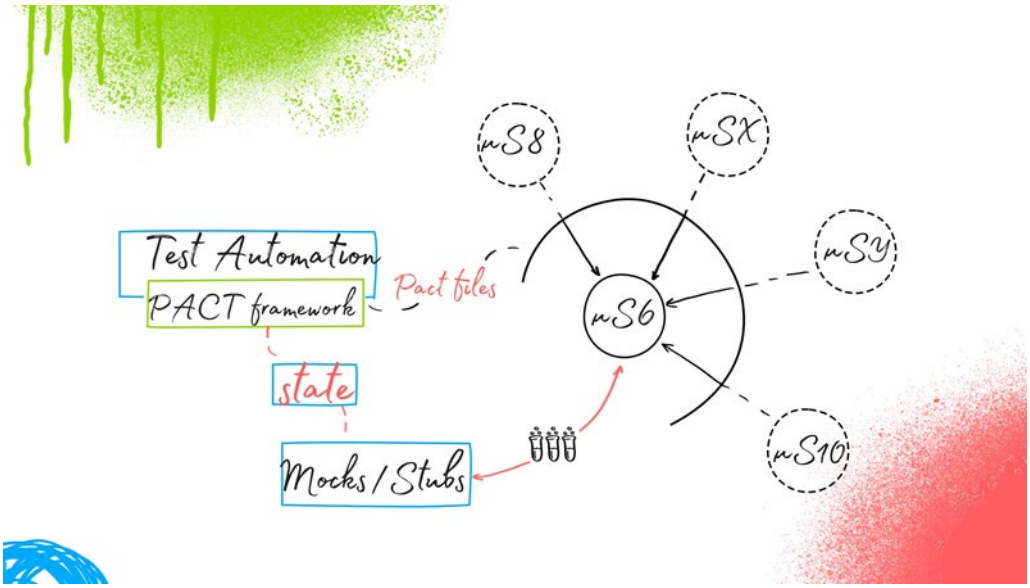


Figure 7 The implementation

What to consider when implementing

What should be taken into account when applying them? The first is the use of containerization and virtualization in the process of building / implementing a microservice. And our test automation system is screwed into a container in the same way. How you ensure the interaction between the microservice and the test automation system is no longer so important: make it as convenient as possible.

The second is updating the contract requirements files. There is such a problem that the requirements of the user to the supplier are starting to accumulate, and the developers are doing only what they urgently need now. Here you need to give developers a task at the level of test management or product management: for example, to work with this "queue" for an hour a week so that it does not grow. Of course, these tests are fast, can take a few seconds. But if their number is measured in thousands, it will complicate the updating of the tests. And on our part, through the system of the hook for engagement we will receive information and eliminate unnecessary self-tests. Basically the CDC is developed by the development documentation, so to speak.

Third: in this work it is necessary to introduce the process of updating states and data sets. What are data packets? During the development process, developers write some basic interaction scenarios. For example: "I need such a request and such an answer." And about everything else - in what framework it should exist, what values of parameters are possible and which are not, they forget. This needs to be checked. And here we must enter with our data-based approach to testing, applying this in our configurator: positive, limit and negative tests to ensure the sustainability of our microservice architecture in production. Without it, it is unlikely to work, every step right / left - and an error in interaction.

Accordingly, when new pact files appear, we hook tasks and fill in new / changed interactions with a large number of data sets in interactions, which is equal to a whole series of tests performed. We can populate and edit interactions manually or we can make generators to parameterize them and run tests.

As for the countries: if we take the automation of web testing, these are prerequisites, settings, FixTure; there is a problem with the microservices - there is no ready mechanism: you have to think about how you will interact with the change of the name of the United States in the PACT files in this module. The simplest of the mechanisms is the use of Aliases, which is essentially KDT (keyword-driven testing) in the test automation language. I am not ready to talk now about a more beautiful and elegant solution: I have not yet invented it.

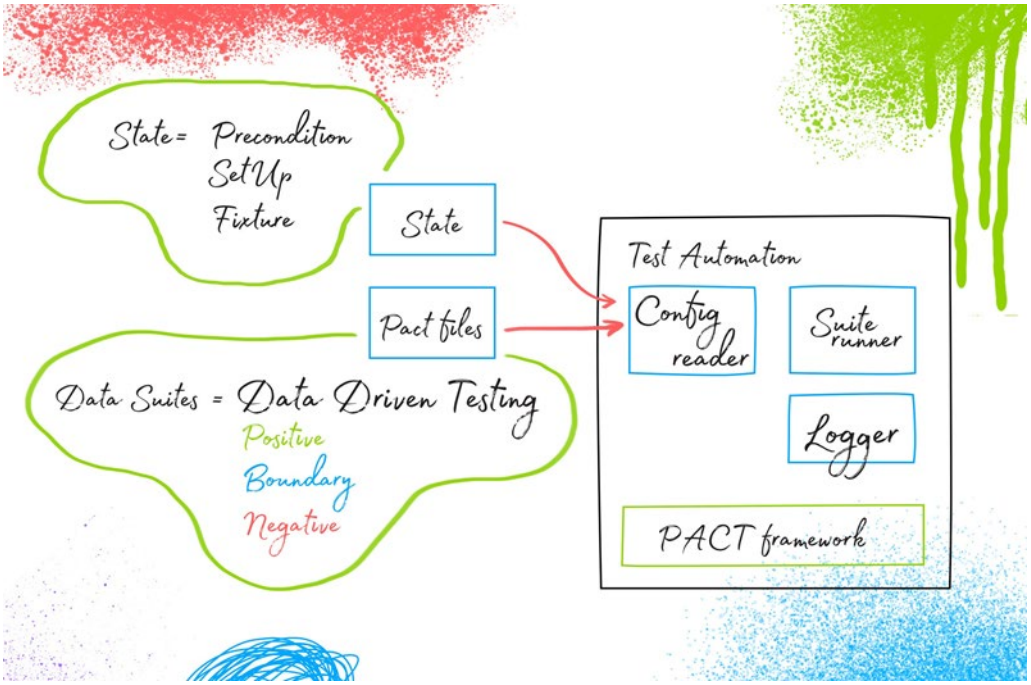


Figure 8 The implementation

3. TESTING IN THE AGE OF MICRO-SERVICES

Initially, the Filipo application was monolithic. Honestly, the monolith is still quite large, but there are more and more micro-services.

We have a user, there is a presentation layer, a business layer, a data layer and a database from which we receive them. When we had a large monolith, the object of testing was a backup application in Python, a frontend in Twig, and quite a bit in React.

The classic test pyramid for monolithic application looks like this:

1. Many test units.
2. Slightly less integration tests.
3. Even fewer end-to-end tests.

4. On top of that - an incomprehensible amount of manual tests.

I will tell you how we tried to get to such a pyramid and what we have from the infrastructure.

We had our own test framework in Python with PythonUnit under the hood. We also have our own system for generating test data. This is a resource manager that allows you to create any required resource for testing. For example, a user with money, with ads in a certain category of Filipo, a user with a certain status.

The reporting system is also self-written. In addition to it, we wrote our own jsonwire network. Grid is a system for orchestration with selenium nodes, it allows you to lift a selenium node on request.
We also have our own selenium mapper. We have written our own library that allows us to execute queries using the jsonwire protocol.

Our pipeline for CI looked like this: some CI event happens for example, it will be a boost to the repository. An artifact for testing was built in CI. A self-written parallel running test system analyzes the artifact and begins testing a bunch of different nodes.

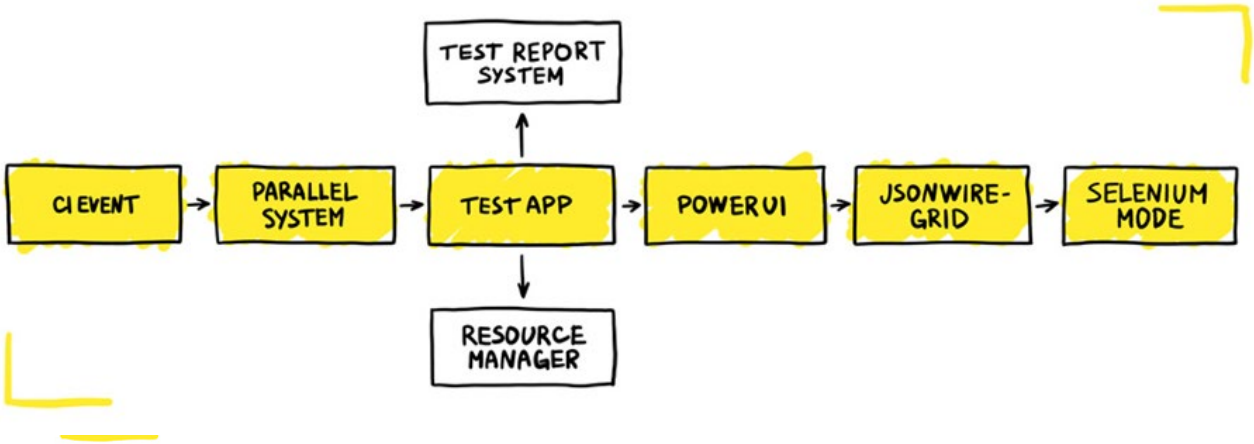


Figure 9 Pipeline for CI

As a test application, we used the Python implementation of Selenide, a full port with Java. But in the process we abandoned him because Selenide was difficult to maintain, he himself was no longer developing. We wrote our own, lighter, PowerUI, around which we built the architecture of test applications with custom matchrs, selectors, etc. This product has greatly simplified testing and reporting for us. Accordingly, the PowerUI then entered the selenium node via jsonwire-grid and performed the necessary actions.

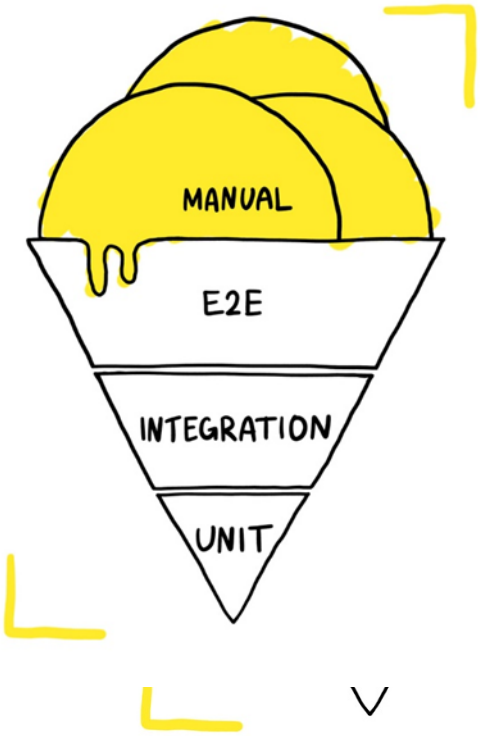
The testing application itself went to the resource manager to generate test data, and then sent the data to our Test Report System.

In general, we lived well in this paradigm. Initially, there were reports of a large monolithic Python application once a day, then their number rose to three and then to six. We had several thousand tests for E2E with a lot of coverage and they were quite light. On average, they ran for about a minute, with rare exceptions. A test that tested a huge piece of business logic could take two to three minutes. We had a minimum of manual regression and a minimum of manufacturing errors.

4. TESTING THE ARCHITECTURE OF MICROSERVICES

Over time, we began to move to microservice architecture. Its main advantages are scaling, speed of function delivery and fault tolerance.

With the monolith, the test pyramid failed. Instead, there was ice cream testing. What was the reason for this? The E2E tests, thanks to the developed infrastructure, were quite fast and did not cause much pain. So we focused on them. We could even ignore the single tests.



With the advent of microservice architecture, this approach ceases to work. Much of the business logic went into separate services and there were more and more of them. For 2020 we have about 2.5 thousand different repositories. In this case, when we conducted the E2E tests, some services, for example, may suddenly stop responding. If he dropped out, all the tests that went to this service and blocked for merging also started to fail. Accordingly, our time in the market simply fell because people could not merge due to falling tests. We were forced to sit and wait for the specific service to be received, to find out what was going on, to turn it around, or to deal with problems.

Then we introduced test karma. This is a very simple mechanism. It works on the basis of a self-written reporting system that has all the necessary historical data. Karma checks if the test fails and then continues to check if there is a similar error when running tests in other branches. The error is the trace hash. We take the full trace, hash it and save it. If we see that an error with such a hash appears on three more branches, we understand that the problem is not in the branch on which the tests are performed, but in the fact that it is of a general nature. If the error is general and not related to a specific branch, then we allow throttling that branch.

Yes, this way we mask the problems, but nevertheless this solution greatly simplifies the life of the developers. If the programmer missed some kind of error, we still have an implementation process. During the implementation, no karma works anymore, everything is manually approved by testers and developers, that is, we look directly at what happened and how it happened. If

we find problems, the redirect is blocked until the problems are resolved and an update is made.

The number of microservices is growing, but the number of testers is not very large. As a rule, we have one hand tester per unit. It is clear that it is quite difficult to fully cover the needs of all developers in the team. On average, these are a few front, a few back-end, more automation, etc.

To solve this problem, we started applying the Agile Testing methodology. The essence of this methodology is that we prevent mistakes, not look for them. We are discussing testing of product modifications. We immediately determine how we will test a characteristic: is it enough to cover it with a single test and if the unit test is sufficient, what type of test should it be. The discussion is conducted with the tester, who determines whether additional manual testing will be needed. As a rule, a single test is sufficient. Plus, the tester can suggest some other cases, and can also provide a checklist based on which we will write the necessary single or functional tests.

Our development comes from acceptance tests, that is, we always determine the tests that will be accepted during development. My colleague Alena from a neighboring unit has already told more about the transition to Agile Testing. In the article, she writes about the application of the methodology as an example for her team, but this applies to all Filipo.

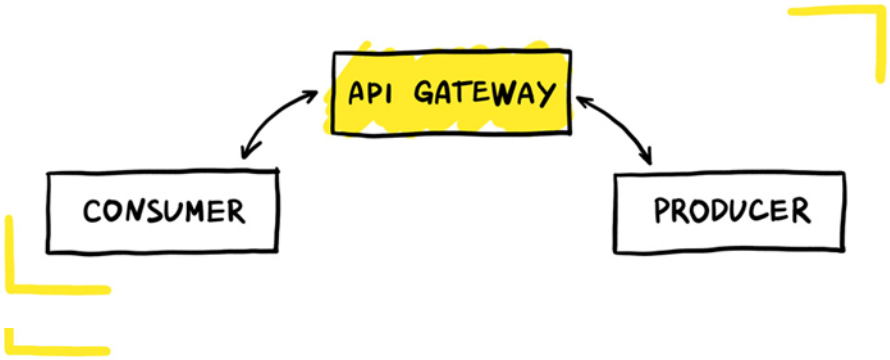
But Agile Testing is not possible without Shift-Left tests. Shift-left testing is a methodology in which we test each implementation and perform all necessary tests with each click. Deployment is impossible without this. But the tests should be light. The main essence of the approach is to find defects in the early stages so that the programmer can perform the necessary test at any time when writing the code. It also provides continuous testing in CI and the ability to automate everything.

During the Shift-Left tests, we break large, heavy tests into a bunch of small ones so that they start and start faster. We decomposed our huge E2E tests into component-interface tests, unit tests, integration tests, functional tests, that is, we started distributing E2E in the pyramid. Previously, the tests could take 20-30 minutes to run and required tambourine dancing. Now in each microservice the tests start in 5-10 minutes and the developer immediately knows whether he has broken something or not.

We also introduced a methodology for testing contracts. CDC acronym tests means "Consumer-Managed Contract Tests". When we click on the code change in the repository, a specially written broker gathers all the necessary information about what the CDC tests are written and additionally understands which service they refer to.

The CDC tests themselves are written by the service user. When the manufacturer rolls up, we conduct all written tests, ie we check whether the manufacturer does not violate the contract in any way.
Did the CDC tests help us? No, because there is a problem with the users themselves not to support their tests. As a result, the tests are unstable, so it turned out that our manufacturers can not deploy on time. I had to go and fix tests on the part of the users. Plus, all the tests were written differently, there could be a dozen different tests on one pen that tested the same thing. It is inconvenient and time consuming. That's why we gave up the idea of CDC tests.

We recently introduced PaaS. Our architectural team has developed a very convenient tool, thanks to which you can quickly deploy a service of boiler plates and immediately begin to develop it. At the same time, one should not think about bases, migrants and other infrastructural things. You can immediately start writing code and rolling migrations.



Now the service user and the service provider communicate with each other through Gateway Api. Api Gateway has contract validation based on short files. A short file is a very simple structured file that describes the interaction in the service. There is a short file from the manufacturer, it describes how we should communicate with this service. The user of copy-paste takes the pen he needs, the necessary structure, puts it all in his service and generates customers based on it.

Accordingly, we confirm the short files that everything is fine with us, that we do not disrupt any network interaction. Now such an assertion will even block the merger if the contracts are breached somewhere. We're just checking infrastructure contracts.

We also introduced such a thing as a service network. Service network - this is when a side car rises to the service code, which proxies all the necessary requests. The request does not go to the service itself, it is first received by the side car, which passes the necessary headers, checks, routes and transmits the request to the service. The service transmits the request back to the cart and so on in the chain.

We have implemented OpenTracing based on the side car This is a technology that allows you to fully track requests from the front to the final service and see what dumps were, how much was the request.

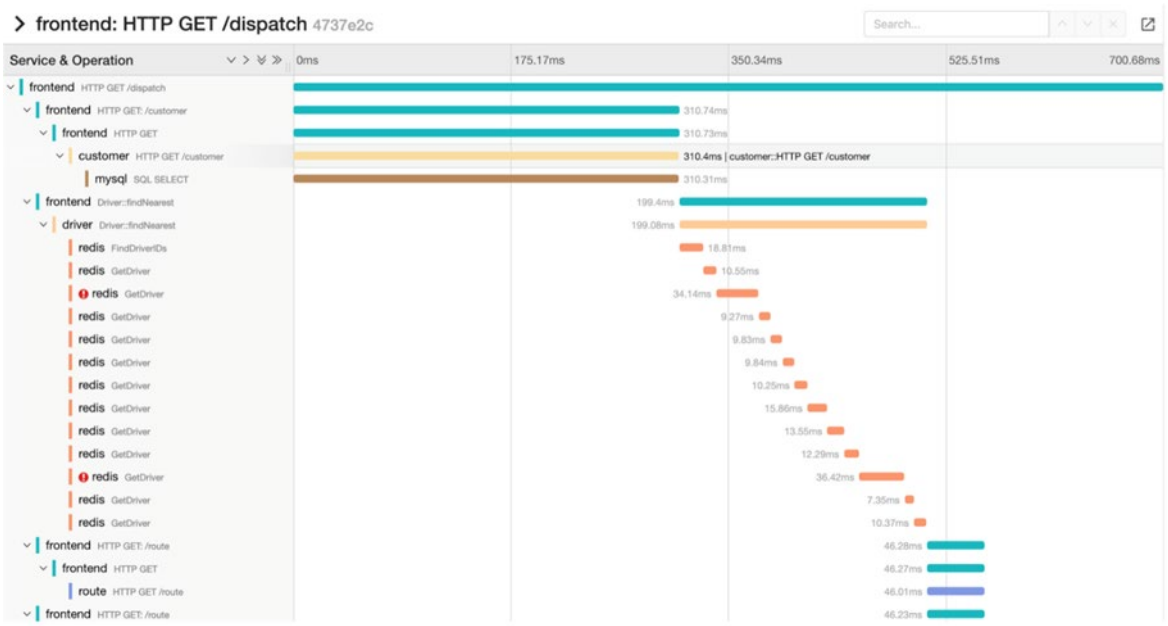


Figure 12 The interface of the Jaeger UI. The screen shows tracking of requests with execution time and route

We also did exquisite degradation tests with the help of the service network. The graceful degradation testing is when we turn off a service and check how the application will work for us. We are looking to see if the application throws an error or crashes completely. We do not want such a development, but as there are many microservices, the number of rejection points is also increasing. This type of testing allows you to check the behavior of the entire system when one of the services fails.

The screen shows a battle example from one of our tests during the passage. We turned off the service and checked if we would give the user a readable message.

Everything works thanks to the Netramesh service network. All you have to do is register the X-Route header, our side cart intercepts the request for the service and redirects it when necessary. In this particular case, we redirected it to nothing, as if the service had dropped out. We can tell him to return the 500th error or we can wait. Netramesh can do all this, the only problem is that you need to add the required title to all requests via the DevTools protocol.

In the dry residue

Now for testing in microservice architecture we use:

- Karma for E2E tests.
- Agile Testing methodology.
- PaaS with Api Gateway.
- Service network, thanks to which the tests for OpenTracing and Graceful Degradation work.

.

4 CONCLUSION

1. As for the hooks after completion, automation of monitoring, appearance and changes, the abolition of contracts between services - all this must be automated, it is part of the system. I would say that this is a piece of the kernel that is at the repository level.

2. A CDC model for developers is needed. This is the basis, without it nothing will work.

3. You will notice that the system is less prone to defects. In fact, all its parts are wrapped in insulation tests. We quickly locate defects, fix them quickly and have high test coverage. If defects occur at the system testing level, they will most likely be related to the front or some type of external service.

REFERENCES

[1] Aderaldo, Mendona, Pahl, Jamshidi, 2017. Benchmark requirements for microservices architecture research. Proc. 1st IEEE/ACM International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering, (ECASE@ICSE 2017), IEEE (2017), pp. 8-13

[2] Alshuqayran, Ali, Evans, 2016. A systematic mapping study in microservice architecture. Proc. IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA 2016) (2016), pp. 44-51

[3] Antonakakis, April, Bailey, Bernhard, Bursztein, Cochran, Durumeric, Halderman, Invernizzi, Kallitsis, et al., 2017. Understanding the Mirai botnet. USENIX Security Symposium (2017), pp. 1092-1110

[4] Avritzer, Ferme, Janes, Russo, Schulz, van Hoorn, 2018. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. Proceedings of the 12th European Conference on Software Architecture (ECSA) (2018), pp. 159-174

[5] Avritzer et al., 2020. Reproducibility Package for "Scalability assessment of microservice architecture deployment configurations: A Domain-based approach leveraging operational profiles and load tests" (2020)

[6] Avritzer, Menasché, Rufino, Russo, Janes, Ferme, van Hoorn, Schulz, 2019. PPTAM: production and performance testing based application monitoring. Companion of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE) (2019), pp. 39-40

[7] Avritzer, Tanikella, James, Cole, Weyuker, 2010. Monitoring for security intrusion using performance signatures. Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering (ICPE), ACM (2010), pp. 93-104

[8] Avritzer, Weyuker, 1995. The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Softw. Eng., 21 (9) (1995)

[9] Casalicchio, Perciballi, 2017. Auto-scaling of containers: the impact of relative and absolute metrics. Proc. FAS*W@SASO/ICCAC (2017), pp. 207-214

[10] Ferme, Pautasso, 2018. A declarative approach for performance tests execution in continuous software development environments. Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE) (2018), pp. 261-272

[11] Francesco, Malavolta, Lago, 2017. Research on architecting microservices: trends, focus, and potential for industrial adoption. Proc. 2017 IEEE International Conference on Software Architecture (ICSA 2017) (2017), pp. 21-30

[12] Heger, van Hoorn, Mann, Okanovic, 2017. Application performance management: state of the art and challenges for the future. Proceedings of the 2017 ACM/SPEC International Conference on Performance Engineering (ICPE) (2017), pp. 429-432

[13] Jiang, Hassan, 2015. A survey on load testing of large-scale software systems. IEEE Trans. Softw. Eng., 41 (11) (2015), pp. 1091-1118

[14] Kambourakis, Kolias, Stavrou, 2017. The Mirai botnet and the IoT zombie armies. Proceedings of the Military Communications Conference (MILCOM 2017), IEEE (2017), pp. 267-272

[15] Milenkoski, Vieira, Kounev, Avritzer, Payne, 2015. A. Milenkoski, M. Vieira, S. Kounev, A. Avritzer, B.D. Payne. Evaluating computer intrusion detection systems: a survey of common practices. ACM Comput. Surv. (CSUR), 48 (1) (2015), p. 12

[16] Newman, 2015. Building Microservices. (1st), O'Reilly Media, Inc. (2015)

[17] Pahl, Jamshidi, 2016. Microservices: a systematic mapping study. Proc. 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) (2016), pp. 137-146

[18] Schulz, Okanovic, van Hoorn, Ferme, Pautasso, 2019. Behavior-driven load testing using contextual knowledge—approach and experiences. Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE), ACM (2019)

[19] Taylor, Medvidovic, Dashofy, 2009. Software Architecture: Foundations, Theory and Practice. John Wiley & Sons, Inc. (2009)

[20] Ueda, Nakaike, Ohara, 2016. Workload characterization for microservices. Proc. IISWC (2016), pp. 1-10

[21] Vögele, van Hoorn, Schulz, Hasselbring, Krcmar, 2018. WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems. Softw. Syst. Model., 17 (2) (2018), pp. 443-477

[22] Weyuker, Avritzer, 2002. A metric for predicting the performance of an application under a growing workload. IBM Syst. J., 41 (1) (2002), pp. 45-54

[23] Miika K 2017 Transforming Monolithic Architecture Towards Microservice Architecture (Finland: Helsingin Yliopisto)

[24] Sourabh S 2016 Mastering Microservices with Java (United Kingdom: Packt Publishing Ltd)

[25] Balalaie A 2016 Microservices IEEE Software 33 42–52

[26] Bucchiarone A, Dragoni N, Dustdar S, Larsen S T and Mazzara M 2018 IEEE Software 35 50-55

[27] Su J M and Hsu F Y 2017 IIAI International Congress on Advanced Applied Informatics (IIAIAAI) 6
[28] Gan Y and Delimitrou C 2018 IEEE Computer Architecture Letters 17 155-158
[29] Fan C Y and Ma S P 2017 IEEE International Conference on AI & Mobile Services (AIMS) 109– 12
[30] Phil W 2019 Microservices Disadvantages & Advantages – Tiempo Development (USA: Tiempo Development)
[31] Li S, Zhang H, Jia Z, Li Z, Zhang C, Li J, Gao Q, Ge J and Shan Z 2019 Journal of Systems and Software 157
[32] Putra S D, Sutikno S, Rosmansyah Y and Asrowardi I 2014 Conference on ICT For Smart Society (ICISS)
[33] Asrowardi I, Putra S D, Subyantoro E and Daud N H M 2018 IT IJECE 8 4023