Master's thesis

# Microservices and Monolithic Solutions in the Web Applications

Zielona Góra, August 2, 2020

Krystian Skibiński

# Introduction

**Master thesis progress:**

Done. Approved by the supervisor ✓

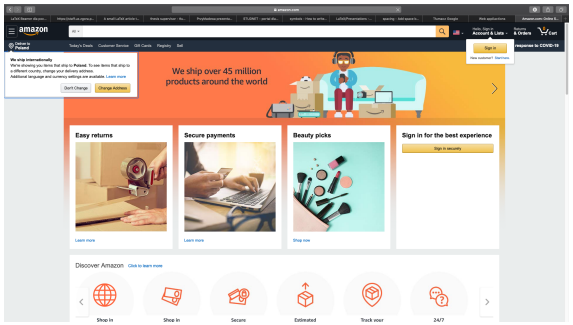
**Table of contents:**

# Abstract

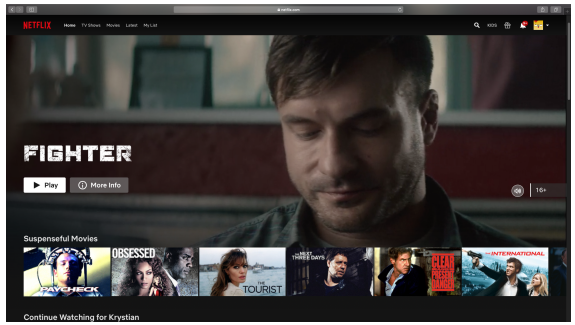
The thesis concerns on comparison of two popular approaches, microservices and monolithic, used in creating websites. Its purpose is to show the similarities and differences between these patterns, as well as their properties and to indicate cases for which the given architecture works best. For its needs, two websites were created using these approaches.

# Web Applications

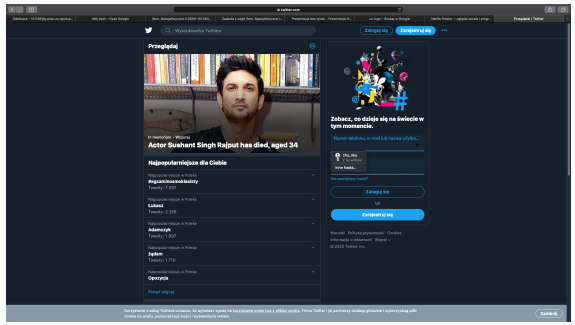
**Web applications** are computer programs, that use the browser as a *client*, which is used to provide and receive the data. Anther computer in that case called *server* is storing that provided by user data and if they are needed, share them.




(a) Amazon




(b) Netflix




(c) Twitter

Figure 1: Examples of the modern web applications.

# Three-layer application model

Basic web applications have a presentation layer written using *HTML*, as well as a program run on the server. Its task is service *databases*, i.e. management of permanent data that is related to software

- *HTML* - markup language used for documents displayed in browser.
- *Database* - an application that stores and manages solid data.

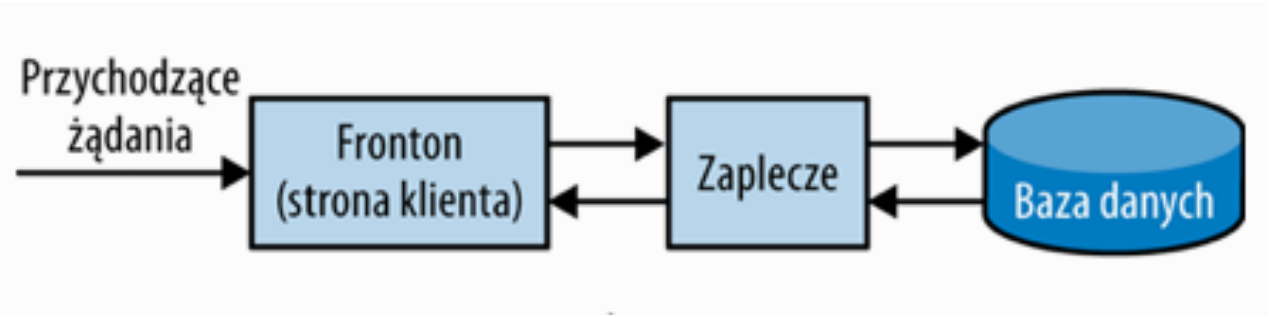



Figure 2: Three-layer application model scheme.

# Monolithic applications

**Monolith** means solid stone block. This sentence perfectly reflects the nature of this type of architecture. The application was written based on a three-layer model, except that the frontend and backend have a common code base. They are placed in a one repository and run with one executable file.

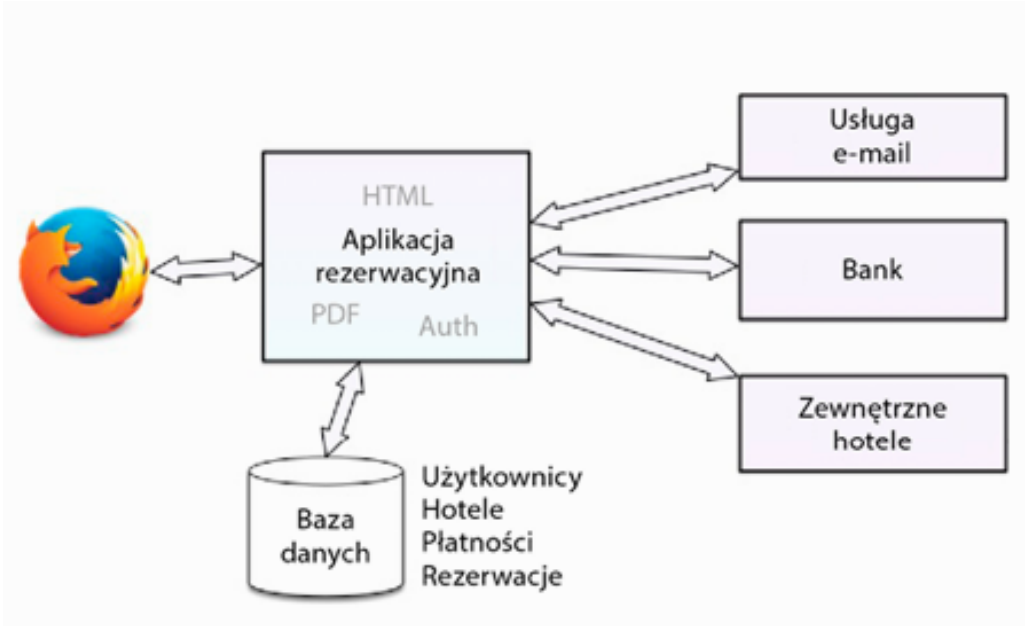



Figure 3: Scheme of the centralized service that book a hotel.

# Architecture of the monolithic applications

Logical layers are commonly used to creat monolithic applications, that is why many techniques have been created to allow them to be organized. The most popular of them is the division of application layers into those that will be responsible for data access (Data Access Layer) by their management (Business Logic Layer) and user interactions (User Interace).

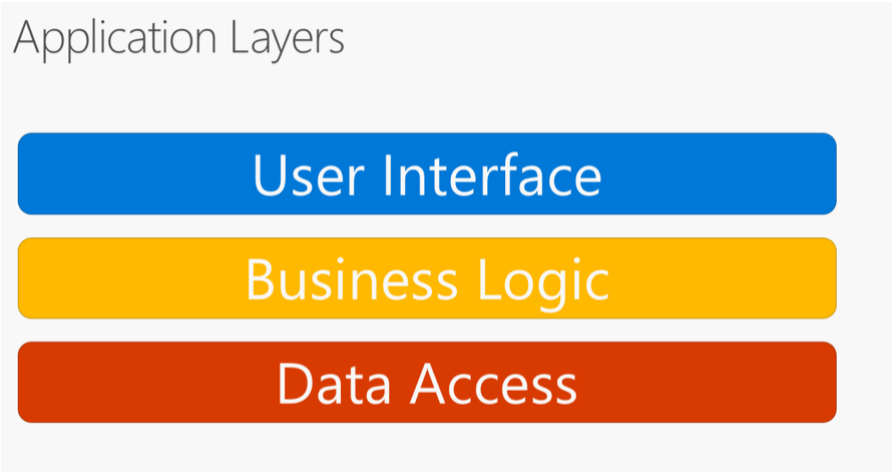



Figure 4: Three-layer application architecture.

# Architecture of the monolithic applications

Modern frameworks further fragment the individual layers, to provide access only for programmers with interfaces that they can use to create their own application. *Ruby on Rails* introduced *MVC* and *Django MVT*.

- *MVC, (Model-View-Controller)* - where models are the classes responsible for interacting with the framework's database mechanism. Controllers, which task is to receive data from the user, modify the model and refresh views. The views, on the other hand, are responsible for generating HTML templates and sending them to the application client (user's browser).
- *MVT (Model-View-Template)* - Instead of using *controllers*, that frameworks uses *templates*. They do not differ much from the *controllers*. Difference is that instead of classes generating HTML files, its creators have created a module responsible for processing such files with special tags. They provide additional logic to interact with the data sent through the view.

# What are microservices?

An opposition to the approach where one central application communicates with external services is to create several smaller programs working separately. Their code could be divided into several separately run processes that have different smaller tasks to accomplish.

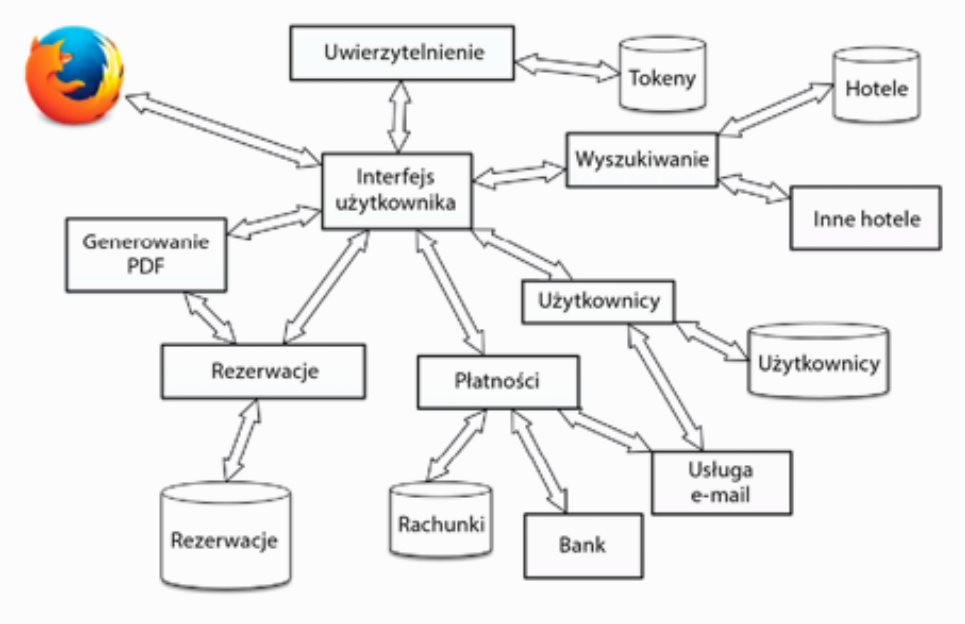



Figure 5: Monolithic application from figure 3 divided into microservices.

# Microservices architecture

There is no single template to determine the structure of the entire system based on microservices. Unlike monolithic applications, small independent websites do not require an in-depth design of the class hierarchy or relationships between individual modules. In this case, the most important is to look at the entire system from the side of messages, not from the perspective of individual components. It is then easier to analyze the ways of interaction between them, find the right patterns and adapt their architecture well. This point of view makes clear that for microservices, the most important thing is to design a transport layer. The knowledge that particular service is actually a separate HTTP server is very important.

# Application project

**The goal:**

- Create two separate application, the first one based on monolithic approach. The second one on microservices.
- Implement database interactions for them.
- Make usage of all HTTP response types.
- Provide good communication layer for them.
- Keep a similar technology stack for these applications.

**Technology stack:**
*Python, Flask, HTML, PostgresSql, Vue.JS*

# Database schema

The database will have two main tables: users and tasks. They will be connected by the one-to-many relationships in many frameworks also called a foreign key. This means that one user will be able to have many tasks assigned to each other, while one task will be assigned to only one user.

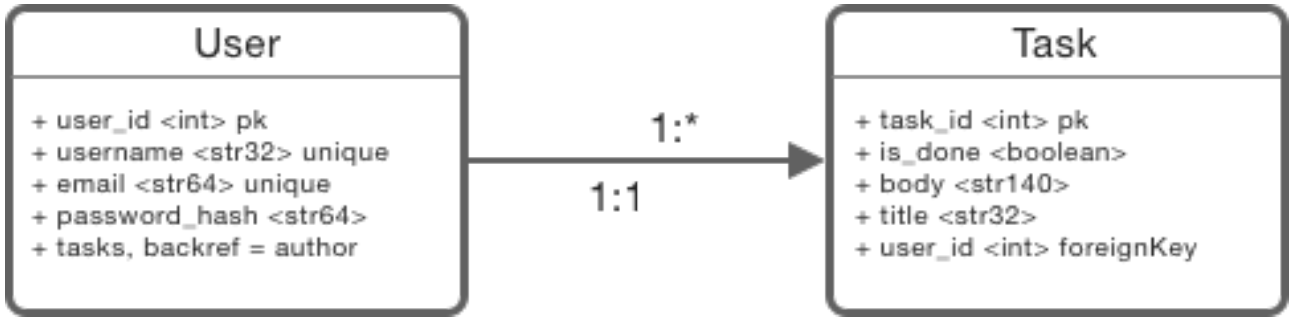



Figure 6: Application database schema.

# Architecures

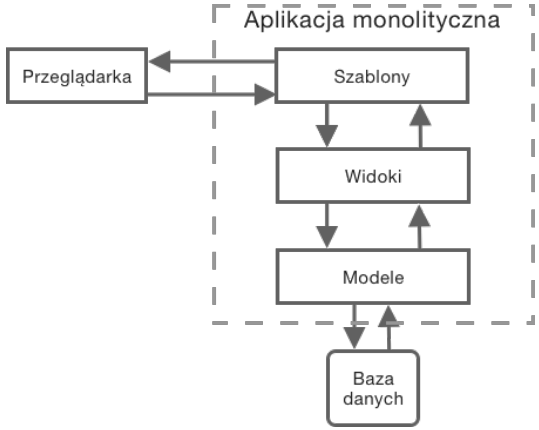



Figure 7: Monolithic application architecture schema.

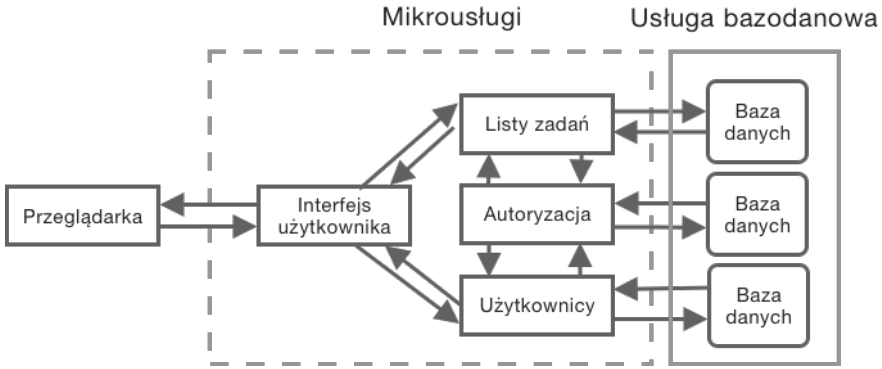



Figure 8: Microservices application architecture schema.

# Integration with hosting services

Containerisation of the application with the Docker platform ensures the system's operation regardless of the programmer's machine, but it can also be used to create a production environment and integration with the server infrastructure. Services such as Amazon Web Services have ready solutions for publishing applications inside containers.

**Link to docker documentation:** `https://docs.docker.com`

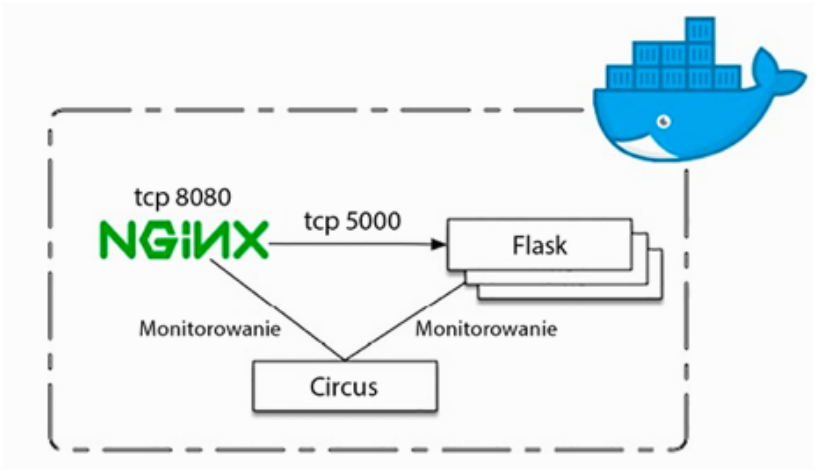



Figure 9: Example of the docker environment.

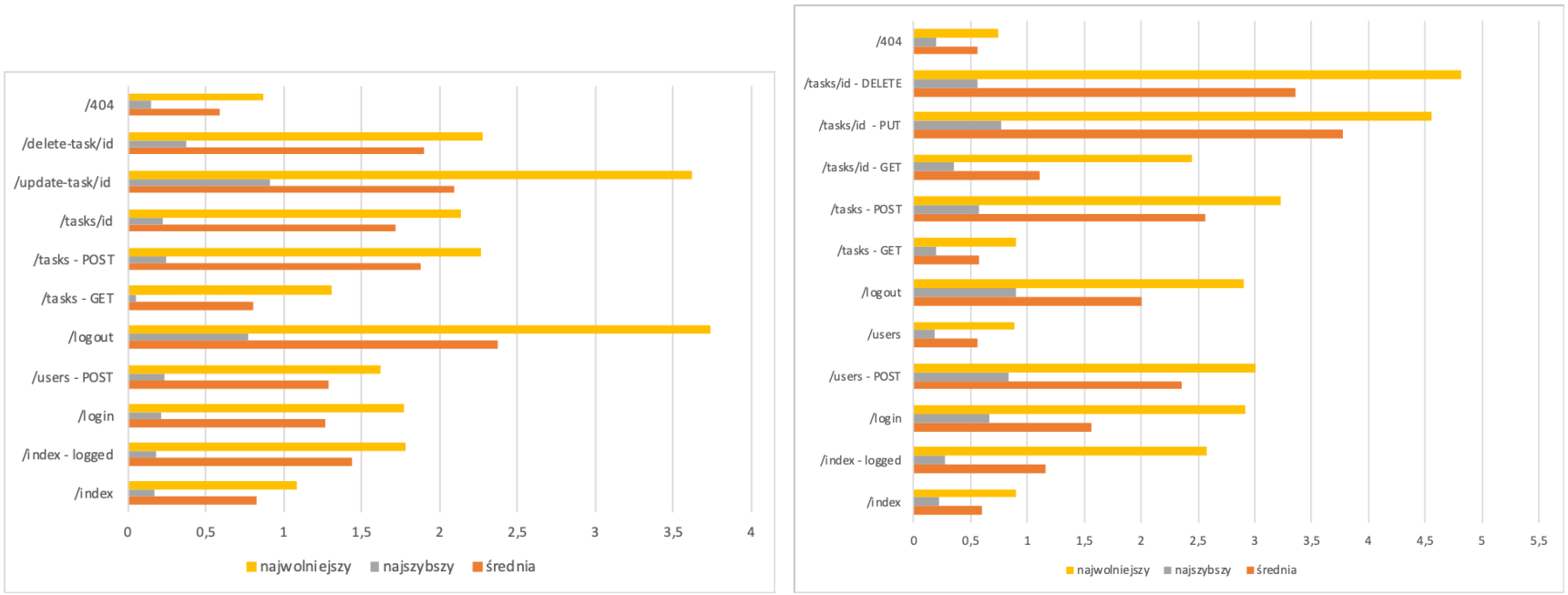
# Application tests

The studies in the thesis were about monolithic and microservices platforms performance, scalability and general opinion on writing applications and used technologies.

*Boom* - program that was used to measure application performance.
`https://github.com/tarekziade/boom`

**Test environment:** Machine with 2.7 GHz Intel Core i5 processor with RAM 8 GB 1867 MHz DDR3. *Boom* in the configuration of 100 users who sent a maximum of 1000 requests.

# Endpoints performance

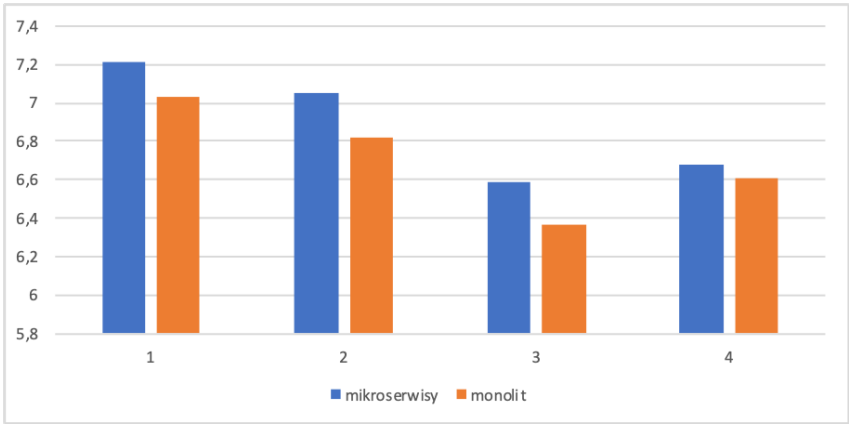



(a) Microservices

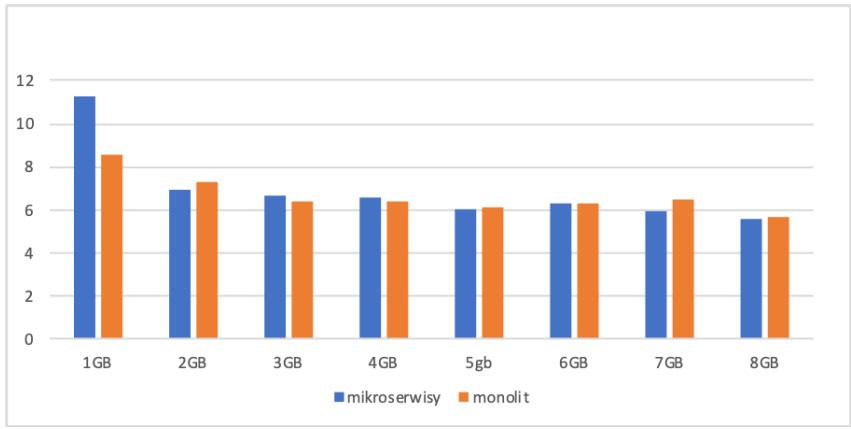
(b) Monolithic

Figure 10: Performance tests result response time(ms) foreach applications endpoint.

# Vertical scalability performance tests




(a) Request average time(ms) for amount of CPU cores (GHz).




(b) Request average time(ms) for amount of RAM(Gb).

Figure 11: Performance tests result for vertical scalability (RAM and CPU tests).

# Horizontal scalability performance tests

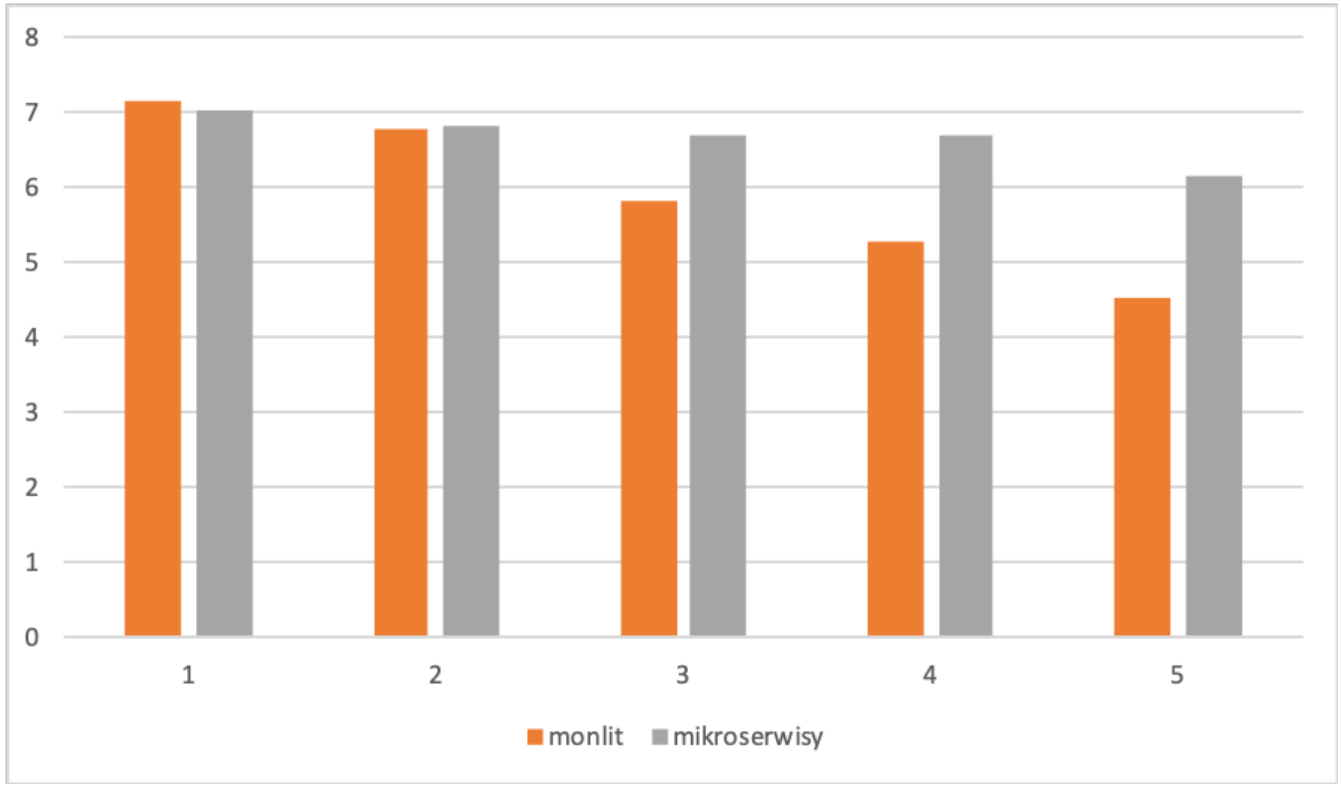



Figure 12: Request average time(ms) for amount of the containers for each service.

# Conclusions

The conducted research indicates better performance of the monolithic website, but it should be noted that the created application is small, the HTML templates generated by it were small. Microservices, despite the fact that they send only JSON objects, still have to be serialized, which could burden the application without highlighting the advantages of this solution. In addition, the services would be launched inside containers, which, despite the fact that they can be used in production solutions, require more resources. Running more containers in a decentralized application could also affect their response times. Another factor that could affect the research was the hardware limitations of the physical machine. It was not a unit prepared exclusively for server solutions and it had a limited number of RAM, an older generation processor and a small disk space.

# Conclusions

Finally, one can come to the conclusion that the choice between monolithic architecture and microservices should be directed mainly due to the nature of the web application project. When it comes to created as part of a larger team that knows several technologies, it would be reasonable to divide the application so that each of them works on a separate service, then for larger services, it will be possible that the disadvantages of a solution based on microservices will be less significant. A monolithic application is still a good idea for smaller applications, where the team does not consist of a large number of people familiar with several technologies. Starting a project is then much simpler and the implementation of such an application is faster. In the event that it grows to a larger size, there is an option to rewrite it to a smaller application and change the architecture to that based on microservices.

# Figure source

- Figure 1: screens of the main pages: *Amazon* (`https://amazon.com`), *Netflix* (`https://netflix.com`), *Twitter* (`https://twitter.com`)
- Figure 2: source: Tarek Ziadé. Rozwijanie mikrousług w Pythonie. Budowa, testowanie, instalacja i skalowanie. Helion, 2018.
- Figure 3, 5: source: Susan J. Fowler. Mikrousługi. Wdrażanie i standaryzacja systemów w organizacji inżynierskiej. Helion, 2019.
- Figure 4: © Microsoft 2020. Dokumentacja Microsoft Build. https://docs.microsoft.com/pl-pl/.

*The rest of the figures is the author's own study.*

# References I


- Tarek Ziadé. Rozwijanie mikrousług w Pythonie. Budowa, testowanie, instalacja i skalowanie. Helion, 2018.
- Susan J. Fowler. Mikrousługi. Wdrażanie i standaryzacja systemów w organizacji inżynierskiej. Helion, 2019.
- Richard Rodger. Tao Mikrousług. Projektowanie i wdrażanie. Helion, 2019.
- Sam Newman. Budowanie mikrousług. Helion, 2019
- Daniel Nations. What exactly is a web application? Lifewire, 2019. https://www.lifewire.com/what-is-a-web-application-3486637.
- © Microsoft 2020. Dokumentacja Microsoft Build. https://docs.microsoft.com/pl-pl/.
- "Ruby on Rails" David Heinemeier Hansson. All rights reserved "Rails". Dokumentacja Ruby on Rails. https://guides.rubyonrails.org.

# References II


- © 2005-2020 Django Software Foundation and individual contributors. Dokumentacja Django w wersji 3.0. https://docs.djangoproject.com/en/3.0/.
- © Copyright 2010 Pallets. Dokumentacja Flaska w wersji 1.1.x. https://flask.palletsprojects.com/en/1.1.x/.
- Copyright © 2014-2020 Evan You. Dokumentacja Vue.JS w wersji 2.x. https://vuejs.org/v2/guide/.
- Copyright © 2013-2020 Docker Inc. Dokumentacja Dockera. https://docs.docker.com/.