AN IMPLEMENTATION OF MICROSERVICES ARCHITECTURE USING HETEROGENOUS TOOLS AND LIBRARIES

Aleksandar Milanović, Irfan Fetahović\*

State University of Novi Pazar, Vuka Karadžića 9, Novi Pazar, Serbia

\* ifetahovic@np.ac.rs

Microservices architecture is a new paradigm for creating modern software systems. It structures an application as a collection of loosely coupled, fine-grained services which communicate through lightweight protocols. Due to its advantages it became widely adopted and used by many leading companies in the software industry. In this paper we discuss microservices architecture and present Netflix Clone application which was developed using this architectural style. The observed benefits of using microservices are: increased modularity, easier understanding, development, and testing. This approach facilitates development parallelization by allowing small, autonomous teams or individuals to independently develop, deploy and expand services.

**Keywords**

Microservices, software architecture

# Introduction

In traditional monolithic applications, all functions are encapsulated in one single application. This approach can have multiple advantages, such as rapid development and testing, and easy deployment, but only when the whole application is relatively small. However, modern software systems are becoming increasingly complex and this architectural style leads to weaknesses such as poor reliability and limited scalability [[1](#LDe19)]. In recent years, microservises have become widely adopted paradigm in developing software systems and many companies, including Amazon, Twitter, PayPal and others began to implement microservices on the cloud or carried out migration from legacy monolithic to microservice architecture [[2](#HKn18)].

Modern software systems require high concurrency, high availability, high scalability, high cohesion, and low coupling. Microservices are an architectural style in developing modern software applications. The style is based on decomposing a monolithic application into multiple, loosely coupled microservices, which can be independently developed, extended, deployed and tested. Microservices have following characteristics: independent development, independent deployment, independent release, high concurrency, high availability, high scalability, high cohesion, and low coupling [[3](#ASi16)]. They communicate and exchange data through lightweight HTTP protocol and mechanism like REST API or message bus.

Differences between monolithic and microservices architecture can be summarized as follows [[4](#GLi20)]:

Monolithic architecture: big component size; one single point of failure; holistic creation and deployment; shared database; uses the same programming language and framework; unable to scale on demand

Monolithic architecture: small, fine-grained multiple components; no single point of failure; each service is built and deployed independently; private database; heterogeneous in terms of programming languages and frameworks; scale on demand

Microservices architecture divides service side applications into multiple loosely coupled services oriented to different and precise tasks and responsibilities. In order to achieve this, each service executes in different containers and each container has its own private database that cannot be accessed directly by other containers [[5](#AFu18)]. Container technology became a lightweight alternative to hypervisor-based virtualization for application that do not require extreme security or strict isolation. They provide near bare-metal performance while requiring fewer computer resources to deploy, run, and manage compared to hypervisors. Adding or removing functionality from the existing image is easy, which makes them more flexible and customizable [[6](#DNJ18)]. Docker has emerged as the most popular container technology and we used it in our example [[7](#DJa16)].

Microservices architecture has two major challenges: performance issues due to network operations and data consistency due to existence of private databases. The microservices architecture requires continuous interaction between the services and procedure through network infrastructure which can cause additional latency and performance degradation due to limited network bandwidth or network congestion.

In microservices architecture, data is stored in private databases of different microservices, so data consistency is a challenging issue that needs to be addressed. Clone-based method tries to solve this problem by creating a clone of each private database. Each update in the private database will be followed by a broadcast to other cloned databases in order to maintain data consistency. The disadvantages of this approach are: more space to store data and increased network communication which can cause extra delay and speed loss [[8](#MEK19)].

# Implementation

In order to demonstrate microservices architectural style we have developed Netflix Clone application. The architecture of the application is depicted in Figure 1. Application frontend, i.e. user interface, is written in the JavaScript programming language, while backend is a combination of couple of technologies, for which three programming languages were used - JavaScript, Java and Python.

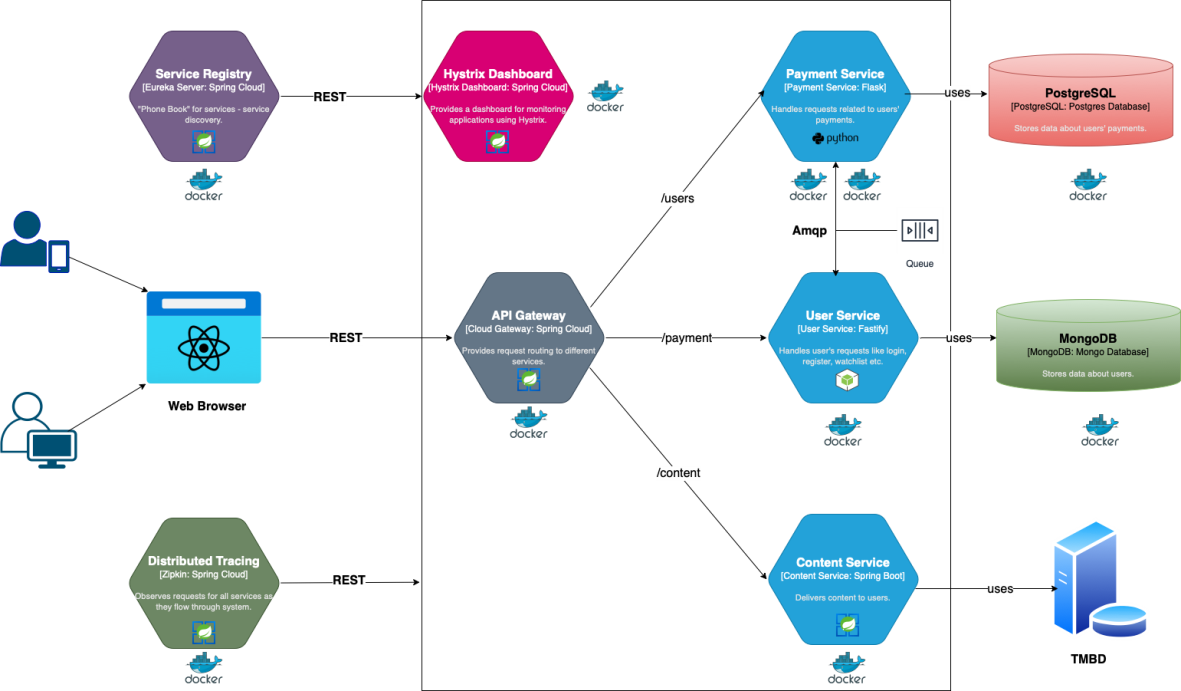


Figure - Application architecture

Many libraries and frameworks were used to create both parts of the application, which will be discussed below.

## Frontend

The user's interaction with the application takes place via a web browser. The user accesses the Single Page Application (SPA) developed using the Next.js framework. Here user can perform all actions like singing up, singing in, browsing content, liking/disliking TV shows and movies, adding them on the watch list, changing his payment plan and more. Next.js is a framework on top of a very popular JavaScript library called React.js. React is currently the most used frontend web framework for developing dynamic single page applications, and Next.js adds additional level of simplicity [[9](#Ada20)]. Some of the features that make Next.js a natural choice for this kind of app are: server-side rendering, easy client-side routing out of the box, automatic code splitting, ease of upgrading etc.

Figure 2 shows some examples of the application’s UI: Welcome page, profile selection, content browsing, user’s watch list and Manage profile and payment info.

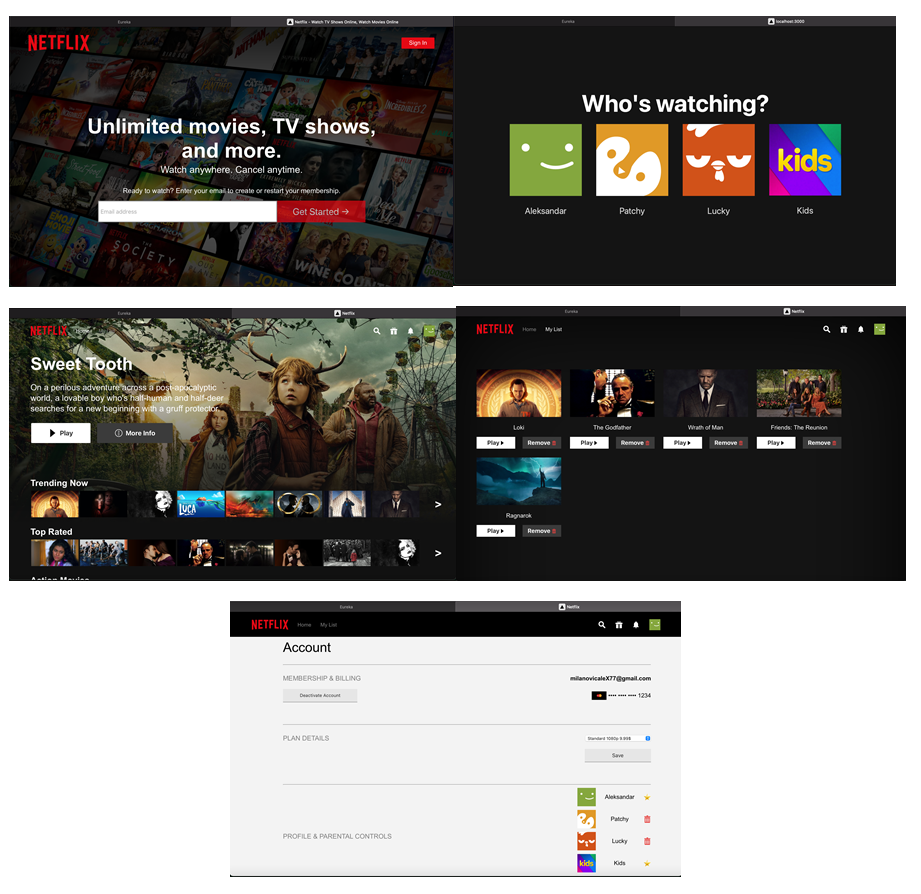


Figure - User interface

## Backend

Backend consists of total of seven microservices, and those are: User Service, Payment Service, Content Service, API Gateway, Service Registry, Hystrix Dashboard and Distributed Tracing. The whole backend architecture lives on Spring Cloud ecosystem, which provides a vast amount of tools to develop, monitor and control the behaviour of microservice applications. Besides Spring Cloud, there are more backend frameworks used for some services, and we will describe each of them below. Frontend and backend communicate via REST API.

**User Service**

User Service is responsible for all things regarding users and their personal information. This service handles request for authentication, authorization, managing profiles, liking/disliking content, adding content to the watch list, but it also communicates with Payment Service when user wants to change their payment plan. Two most interesting features this service provides are: authentication (via email and password, with email confirmation), JWT (Json Web Token) authorization and internal communication with Payment Service.

When user signs up for the first time, or whenever signs in again, frontend receives a token called JSON Web Token, which contains encrypted information about that user, so he can later access different things on the platform. This token is checked by backend at every request in User and Payment Services, and it controls whether user can access certain data in application or not.

The other interesting part is internal communication with Payment Service. So, as we know, payment services are often very slow with transactions, as everything has to be checked and cleared, so implementation here is based on message queues. Payment Service is responsible for getting and deleting information about user’s current plan, but User Service has a role when user signs up for the first time. When user makes a request to change his current subscription (plan), User Service will call Payment Service via AMQP protocol, using the CloudAmqp (RabbitMq solution in cloud). It will put that message in a queue with using appropriate topic, and Payment service will consume it and respond. So as we can see, User Service acts like a Producer here, while Payment Service is a Consumer. The important takeaway here is that this communication is asynchronous, so there will be no blocking in the system. User Service is implemented with Fastify Node.js framework and uses NoSQL MongoDB databases.

**Payment Service**

As mentioned above, this microservice handles all requests related to payment plan – subscription. It provides an API for retrieving, updating, and deleting payment information about user. It also receives messages from User Services when subscription is created. It’s implemented using Python, with Flask micro framework, and uses PostgresSQL behind the scenes. Reason for this is that this service is quite simple, so Flask really came as a natural choice here.

**Content Service**

This service is the last one that is related to customer. It provides an API for browsing content. It has different routes that return tv shows and movies based on the choses category from the request. This service is developed with Spring Boot Java framework, and it communicates with external TMDB database which stores all data about TV shows and movies. TMDB provides a free public API accessible with API key.

**Spring Cloud Services – API Gateway, Eureka Service Registry, Zipkin Distributed Tracing, Hystrix Dashboard**

These four remaining services are part of Spring Cloud infrastructure that has a big role in adding a layer of abstraction to the services, handling failures and monitoring the system [[10](#Pio18)]. As the names suggest, these services are all written in Spring Boot.

API Gateway serves as an entry point of the entire backend system, where communication between frontend and backend services starts. It routes user’s request based on the route, as it is shown on the architecture diagram. We have three routes - “/users”, which sends request to the User Service, “/payment” sends it to the Payment Service, and “/content” to the Content Service. It also restricts from which origin requests can be sent, it this case, only our Frontend application can do it. This is managed by configuration of CORS (Cross Origin Resource Sharing). There are many more features that Spring Cloud Gateway includes, like Circuit Break integration, Path Rewriting, Rate Limiting etc.

Eureka is a Service Registry service that holds the information about all microservices. Every microservice will register into the Eureka server and Eureka server knows all the services running on each port and IP address. Eureka Server is also known as Discovery Server. This means that each of our services registers itself with Eureka when they start up. Service Discovery mechanism helps API Gateway discover services it needs to route request accordantly.

Eureka UI is shown on Figure 3, where you can see all registered applications (services).

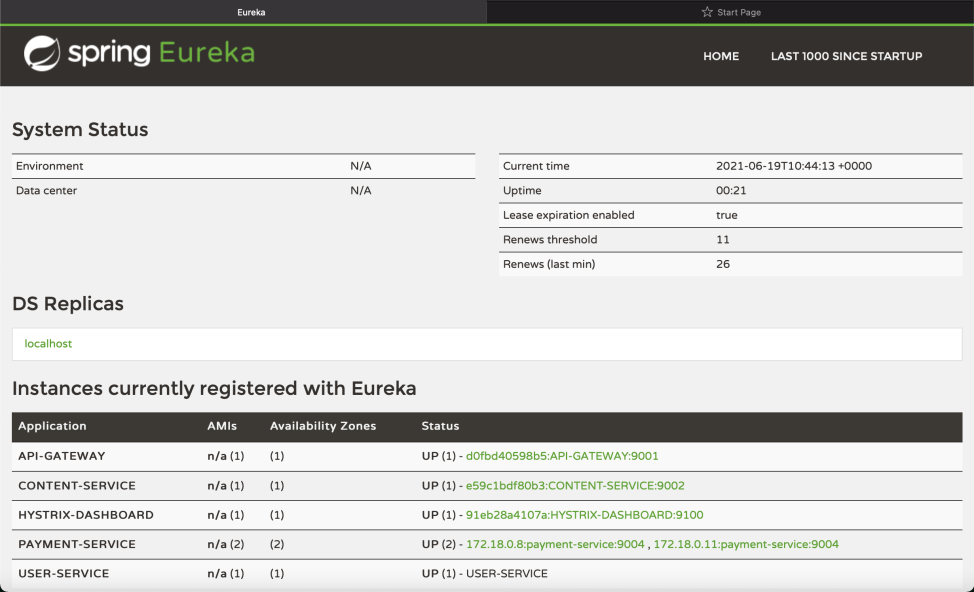


Figure 3 - Eureka UI

**Zipkin - Distributed Tracing**

Zipkin helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data. It identifies the failed microservices or the services having performance issues when there are many services call within a request. It is very useful when we need to track the request passing through the multiple microservices. It is also used for measuring the performance of the microservices. There is an integrated UI where you can jump in log file directly if you have Trace ID, or make different queries based on attributes like service, tags and duration. Figure 4 shows an example of tracing for some requests from the application. You can see which service handled the request, when, how long it processed it, and you can see more details as well.

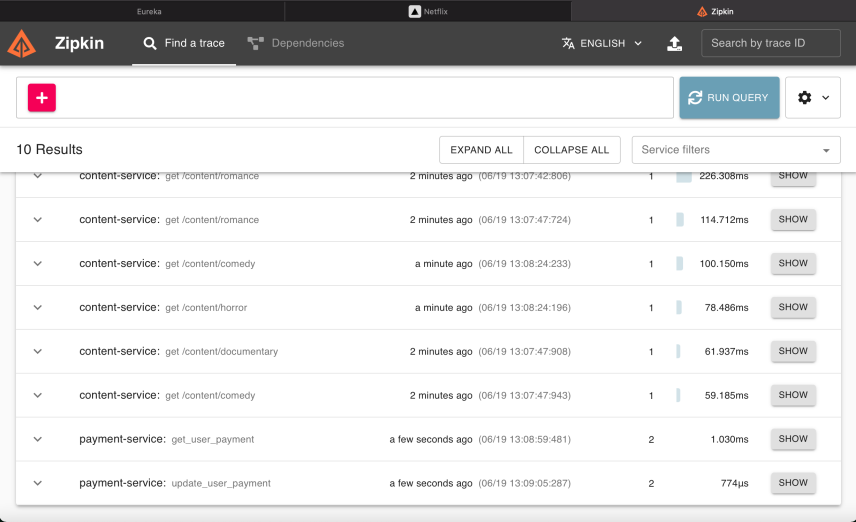


Figure 4 - Zipkin example

**Hystrix Dashboard**

The Hystrix Dashboard, depicted in Figure 5, allows you to monitor Hystrix metrics in real time.

It main purpose is to help reduce the time it takes to detect and recover from operational events. The duration of most manufacturing incidents (rarely due to Hystrix) becomes far shorter, with reduced impact, due to real-time insights into the behaviour of the system provided by the Hystrix dashboard.

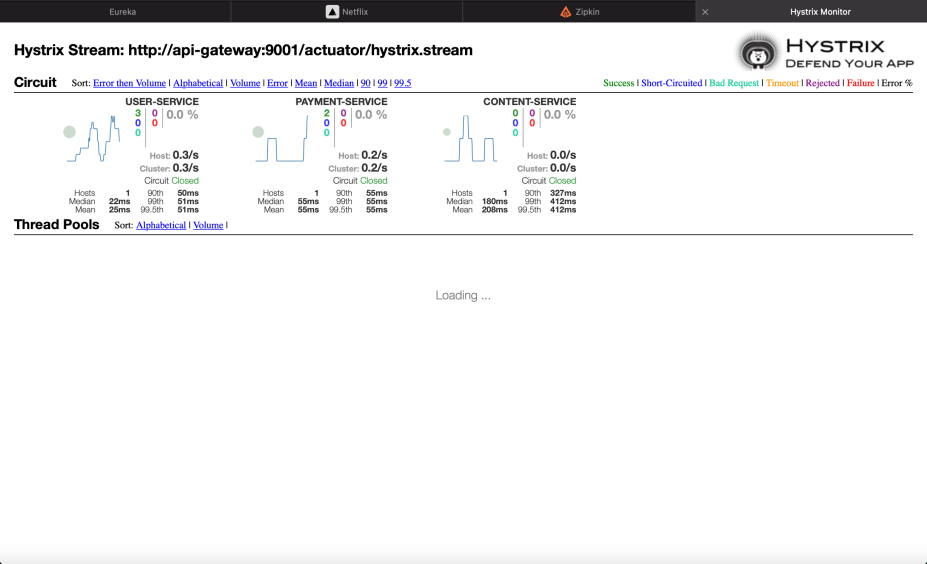


Figure 5 - Hystrix Dashboard

# Conclusion

As businesses struggle to meet the demand to automate data services to achieve efficiencies in business and to better handle the increasing complexity and communication requirements of a growing number of applications, wearables, and things like IoT into the overall picture, microservices will provide the flexibility, scalability and cost-efficiency needed to future-proof their IT infrastructure. This type of architecture really shines in separating business logic into small components that communicate with each other, but that is probably where things start to complicate too. Choosing the right tools such as protocols, frameworks and languages that work well with each other is not an ordinary task, but if it is done the right way, this type of architecture proves to be a very grateful one when writing applications.

# References

x

|  |  |
| --- | --- |
| [1] | L. De Lauretis, "From Monolithic Architecture to Microservices Architecture," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Berlin, Germany, 2019, pp. 93-96. |
| [2] | H. Knoche and W. Hasselbring, "Using Microservices for Legacy Software Modernization," *IEEE Software*, vol. 35, no. 3, pp. 44-49, May/June 2018. |
| [3] | A. Sill, "The Design and Architecture of Microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76-80, september 2016. |
| [4] | B. Huang, Z. Liang, M. Qin, H. Zhou and Z. Li G. Liu, "Microservices: architecture, container, and challenges," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Macau, China, 2020, pp. 629-635. |
| [5] | C. Fidge, O. Zimmermann, W. Kelly and A. Barros A. Furda, "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency," *IEEE Software*, vol. 35, no. 3, pp. 63-72, May/June 2018. |
| [6] | S. Garg, P. P. Jayaraman, R. Buyya, Z. Li and R. Ranjan D. N. Jha, "A Holistic Evaluation of Docker Containers for Interfering Microservices," in *2018 IEEE International Conference on Services Computing (SCC)*, San Francisco, CA, USA, 2018, pp. 33-40. |
| [7] | D. V. Nguyen and R. Smart D. Jaramillo, "Leveraging microservices architecture by using Docker technology," in *SoutheastCon 2016*, Norfolk, VA, USA, 2016, pp. 1-5. |
| [8] | M. E. Kholy and A. E. Fatatry, "Framework for Interaction Between Databases and Microservice Architecture," *IT Professional*, vol. 21, no. 5, pp. 57-63, September/October 2019. |
| [9] | Adam Boduch, *Next.js in Action*. New York, USA: Manning Publications, 2020. |
| [10] | Piotr Minkowski, *Mastering Spring Cloud: Build self-healing, microservices-based, distributed systems using Spring Cloud*. Birmingham, United Kingdom: Packt Publishing, 2018. |

x