ASSIGNMENT OF BACHELOR'S THESIS

Title:	Advertising server using microservice architecture
Student:	Serhii Holovko
Supervisor:	Ing. Filip Glazar
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2021/22

Instructions

The goal of this work is to implement the prototype of a web application which will represent the advertising server. Completed server-side part of the application will be the prototype of the real cars-advertising platform like Cars.cz etc. The main functionality of the student's platform will be searching, comparing cars and creating user's own advertisements. Student will implement the backend part in Java, specifically in the Spring Boot. Implementation will aim on analyzing and designing the backend architecture using microservices. The backend documentation will be created for later frontend implementation.

1. Analyze existing solutions of similar implementations.
2. Based on conducted analysis design the backend part of the application.
3. Perform realization of this part of software.
4. Set up CI and provide the implementation of appropriate tests.
5. Estimate results of testing and usability of the application.
6. Design necessary modifications of the application for real use.

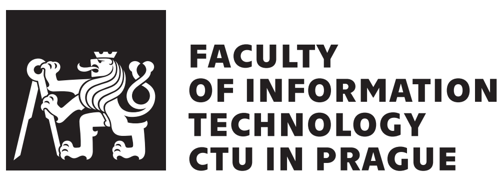
References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 25, 2020



Bachelor's thesis

Advertising server using microservice architecture

Serhii Holovko

Department of Web and Software Engineering
Supervisor: Ing. Filip Glazar

June 4, 2020

Acknowledgements

I would like to thank my supervisor Ing. Filip Glazar for his time and help and my family which has been supporting me all my life.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 4, 2020 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Serhii Holovko. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Holovko, Serhii. *Advertising server using microservice architecture*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Tato práce popisuje hlavní zásady při návrhu a implementaci softwaru v rámci architektury mikroslužeb. Po teoretické části následuje demonstrace skutečného procesu vývoje, což čtenáři umožňuje pochopit základy a vývoj distribuované architektury. Dále jsou diskutovány výhody a nevýhody dané architektury.

Klíčová slova mikroslužby, škálovatelnost, backend, webová aplikace, systémový navrh, průběžná integrace

Abstract

This thesis describes the microservice architecture essentials. The theoretical part is followed by the real application development process demonstration which makes the understanding of the distributed architecture deeper and enables the creation of a similar microservice application quite effortless. Moreover, the disadvantages of such architecture are discussed as well as advantages and the summarized opinion is demonstrated at the end of the thesis.

Keywords microservices, scalability, backend, web application, system design, continuous integration

Contents

Introduction **1**

1 Services collaboration **3**
1.1 HTTP . . . 3
1.2 Network communication . . . 3
1.3 Service discovery . . . 4

2 Fault tolerance and resilience **7**
2.1 Timeouts . . . 7
2.2 Retries . . . 7
2.3 Circuit Breaker . . . 8
2.4 Deadlines . . . 8

3 Testing **11**
3.1 Monolithic testing . . . 11
3.2 Microservices testing . . . 13

4 Advantages of the microservice architecture **15**
4.1 Distributed work . . . 15
4.2 Scalability . . . 15
4.3 Deployment . . . 16
4.4 Fault tolerance and resilience . . . 16

5 Disadvantages of the microservice architecture **17**
5.1 Complexity . . . 17
5.2 Culture . . . 17
5.3 Security . . . 18
5.4 Communication . . . 18

6 Practical part **19**

6.1 Design 19
6.2 Technologies 22
6.3 Data layer 27
6.4 Implementation 27

Conclusion **37**

A Acronyms **39**

B Contents of enclosed CD **41**

Bibliography **43**

List of Figures

1.1 HTTP communication 4
1.2 Client-side service discovery pattern 5
1.3 Server-side service discovery pattern 5

2.1 Retries pattern 8
2.2 Microservices chain communication diagram 9
2.3 Timeouts in microservices communication 9

6.1 Cars.cz technology stack 20
6.2 Microservices' data sources 21
6.3 Autos backend rough model 22
6.4 Eureka console 28
6.5 Catalog relational model 29
6.6 Rating relational model 30
6.7 JWT structure 33
6.8 Jenkins dashboard 34
6.9 JMeter success result table 35
6.10 JMeter fail result table 36

List of Tables

6.1 Catalog endpoints table 29
6.2 Rating endpoints table 30
6.3 Creation endpoints table 31
6.4 Similarity endpoints table 31
6.5 Gateway endpoints table 32

List of Listings

6.1 Eureka main class 24
6.2 Eureka application.properties 25
6.3 Eureka client application.properties 25
6.4 Eureka client pom.xml 25

Introduction

Nowadays creating an information system consists in building a web interface also known as a web application. Almost every system uses a 3-tier architecture that is divided into 3 big components (client-side, server-side, and data-side) each developed by a huge team. A lot of tools and additional software products are trying to make developers' work easier and applications more efficient, nevertheless in some cases, it does not have such a huge impact on the development process.

The implementation of a massive system requires teams of several, dozens or even hundreds of developers. This is a challenging area for the collaboration of a sizable amount of people. Moreover, applications are expanding due to the shifting of final projects' requirements and the system's extension. That is the case when developers have to rather understand the previous technical solutions and improve it or to extend the existing software by a separate "module".

Furthermore, an application can be used by thousands of people and that is why it has to be scalable. Modern solutions offer patterns for painless horizontal scalability, which stands for "running" software on several physical machines. Such a conception offers the ability to increase application efficiency by adding extra servers but not increasing the performance of the individual machine.

The thesis is logically divided into 2 big parts. The first one covers the theoretical aspect of the microservice architecture describing at the beginning problems of microservices communication and general concepts for achieving services wiring. Then resilience issues and solutions are demonstrated. After that, one of the most essential themes is explained in – testing patterns that is followed by discussing the advantages and disadvantages of the microservice architecture. The second part of the thesis describes the process of creating real advertising backend software using approaches and concepts from the theoretical section.

The primary aim of this thesis is to analyze the difference between the

classic monolithic approach and the conception of decomposition software, specifically the backend part of the application. In addition, the backend separation process which is achieved by implementing the microservice architecture will be discussed in the thesis. As an example of this approach, the realization of the backend part of the car selling application will be presented. The advantages and disadvantages will be discussed as well as mistakes in designing such a distributed system. Finally, performance testing results will be demonstrated.

Chapter 1

Services collaboration

Individual services are in fact just separate backend servers that can communicate with each other. That is why when talking about microservices, one of the first problems for discussion is a communication of different system modules (services). In the regular monolithic application components' binding is implemented by direct calling module's functionality (working with Java object, JavaScript function, etc.). However, microservices are frequently not running on the same virtual machine and that is the reason why their communication is performed through a network.

1.1 HTTP

HTTP stands for Hypertext Transfer Protocol. Nowadays it is a very popular practice in communication between browsers and servers. The main conception lies in that data are usually stored on a server or accessible through it. Most servers support HTTP protocol which makes it possible to communicate with them and retrieve some information. The process of getting information is achieved by making a HTTP request to a server and receiving a HTTP response see Figure 1.1. The most typical case of using the HTTP protocol in regular life is loading a web page from a browser [1].

1.2 Network communication

As previously mentioned, microservices communication is realized through a network. Most of the existing solutions use the regular HTTP protocol for services collaboration, which can be actually easily applied with the help of existing frameworks. The first problem of communication using the HTTP protocol is retrieving target URLs. This issue can be solved in hardcoding URLs and requesting necessary services by their location. However, this approach has a lot of disadvantages.

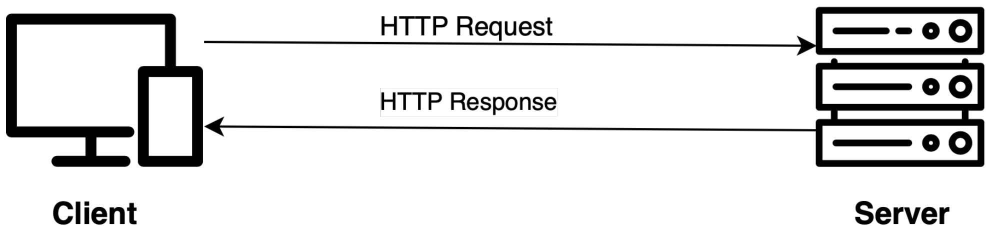


Figure 1.1: HTTP communication

One of them is the demand for changing the target URLs when some services modify their locations or ports. In that case, every microservice which uses the actualized microservice has to change target endpoints. That can lead to a huge amount of refactoring work.

Another disadvantage is the impossibility of deploying microservices with hardcoded target URLs to a cloud environment. That is caused because of the dynamic locations of applications in a cloud. So, it becomes impossible to request a microservice in such format:
GET: `https://myapplication/user:8080`.

Furthermore, concerning load balancing hardcoding services' endpoints makes it much harder for any microservice to use other services, because it is not longed clear to which URL (endpoint) send a request as a load balancer creates several services with different locations [2].

In addition, one of the biggest disadvantages in hardcoding microservices' URLs is the environment changes which lead to changing all target endpoints in the whole application [2]. For example, when developing a distributed system on the local machine, target endpoints are presented in such form: `https://localhost:8080/...` However, after deploying on a server all services' locations will be changed and the complete application will need a lot of refactoring.

1.3 Service discovery

The process of retrieving the services locations is called service discovery [3]. The main idea consists in using the discovery server pattern, what in fact is just an additional layer between microservices which is implemented by an individual server.

1.3.1 Discovery server

In the beginning, all services, which are required to be accessible for other services, need to be registered in the discovery server. As a result, when

requesting any registered service, the first step is to retrieve the location of the necessary service from the discovery server, and then call the server by its URL. Furthermore, the discovery server very often supports load balancing functionality. So, a microservice backend can become extremely efficient only due to the use of this tool.

Moreover, registered microservices often has the ability to store other microservices locations in the cache. As a result, if the discovery server goes down, microservices can continue communicating with each other for some time using cached endpoints.

1.3.2 Discovery server concepts

By the way, discovery servers can be implemented in different ways. The first concept is called “client side service discovery”. The main idea is that client at first asks a discovery server, retrieves the target endpoint, and sends the request to the particular URL see Figure 1.2.

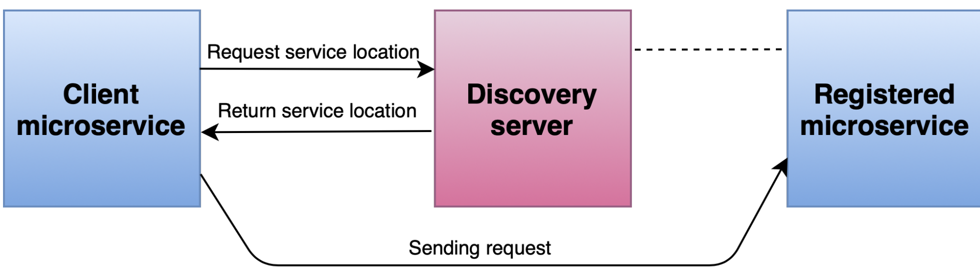


Figure 1.2: Client-side service discovery pattern

The second approach lies in the concept of a proxy server. The client service sends the request to the discovery server which then calls the necessary microservice and returns a response to the client see Figure 1.3.

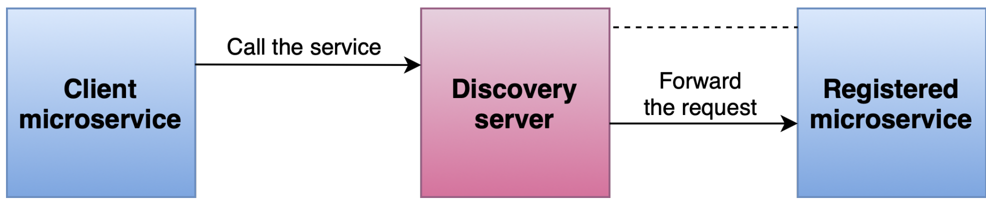


Figure 1.3: Server-side service discovery pattern

Chapter 2

Fault tolerance and resilience

The microservice architecture enables the system to behave flexibly due to the behavior of individual services. Microservice may go down, be overloaded, a data source can fall down or the network connection may be poor. A problem with an individual component should not lead to a complete application collapse as in the monolithic approach. Moreover, each request processing consumes one thread, that is why waiting for retrieving a microservice's response can lead to the client microservice overloading. Nevertheless, fault tolerance behavior is very often ignored by many companies as it requires a lot of additional work.

2.1 Timeouts

The main idea of this pattern focuses on that client does not have to wait for a service's response non-specified period of time. Instead of redundantly waiting for any microservice to return a result, it is considered a much better approach to close the connection and return an exception or any fake value that can be cached response, mock value, or even another microservice's response [4].

2.2 Retries

This technique lies in the principle of retrying a request if a microservice returns an error. It becomes necessary to set a number of attempts for retrying services' calls. However, this practice may lead to a redundant doubling of requests. Assuming that all microservices has a defined number of retries (3). If any service only returns errors, the component (C) which needs "error service" will call it 3 times and if the component C was called by a component B, B will receive an error from C and repeat the request to C 3 times, so the "error service" will be called $3 * 3$ times ultimately. Moreover, if the B service

is called by another component A, A in case of negative responses from the B service will make 3 requests to B. After every A request, B will call C 3 times, and as a result on every B request C will call the "error service" 3 times what leads to the exponential number of meaningless requests see Figure 2.1.

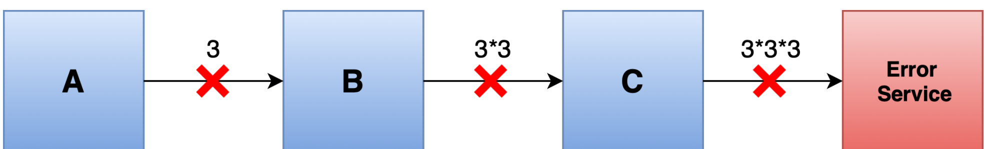


Figure 2.1: Retries pattern

That is the reason why a service doesn't have to retry a request a defined number of times, it needs rather check if the duplication of the same request makes sense. For example, if a server returns a 404 response, repeating a request is quite redundant. However, sometimes retrying a request can lead to a sending it by a load balancer to a more active server [4] or the response status code can be in 5xx format what can be sometimes resolved by repeating a call to service.

2.3 Circuit Breaker

Another pattern ensures that a service is not redundantly called if it is overloaded by requests or fallen down completely. The main conception consists in that last N requests are considered to be analyzed in some way to determine if the "error service" should be ignored for some time to recover from previous requests [5, Chapter Circuit Breaker].

This conception allows developers to set the logic for finishing sending requests to the "error service". Firstly a developer has to set the number of requests which are constantly analyzed (N). Then the amount of possible failed requests is defined. As a result, responses are continuously inspected if more then X requests failed during the last N calls.

2.4 Deadlines

This approach resolves the problem which can occur while the microservice A request a B service with 200 ms timeout see Figure 2.2, however, B may execute some business logic which takes 200 ms and then call the C service what is redundant, because the initial microservice A is no longer interested in the result, because of the defined timeout. This problem can cause redundant resource consumption. However, this issue can be solved by passing some metadata with services' calls. As a result, microservices can be able to

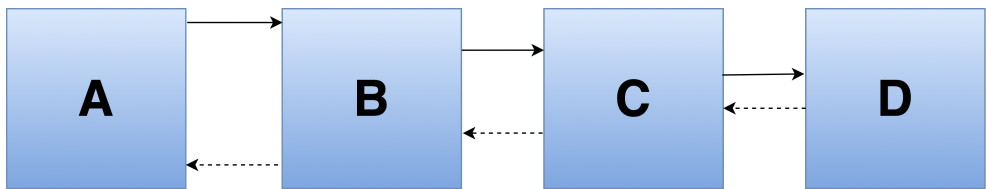


Figure 2.2: Microservices chain communication diagram

determine if it is still necessary to proceed with logic execution or a client is no longer interested in its response [5, Chapter Deadlines/distributed timeouts].

2.4.1 Timestamp

The timestamp approach focuses on establishing a deadline, after which service does not have to proceed with any logic execution, because the client service is not interested in the response. For example, if a service A calls a service B and B calls a service C see Figure 2.2, the first consuming service which makes a call (or frontend client) has to set a deadline for service B due to the timeout: $deadline = currentDatetime + Timeout$. And the deadline is sent as metadata to service B with a request. Then, the service B set the deadline for a C service. However, a problem is that microservices are frequently running on different servers which may have different time zones. As a result, a service may immediately stop executing its logic because it can improperly consider that the deadline is already passed. Furthermore, a microservice may continue running thinking that a deadline is not passed yet, what can be wrong if a server has the earlier current time. [5, Chapter Timestamp].

2.4.2 Timeout

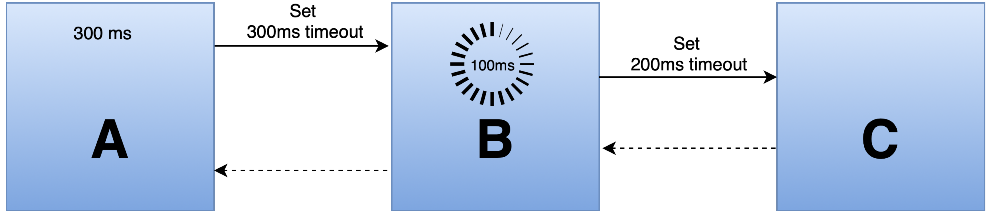


Figure 2.3: Timeouts in microservices communication

Another approach lies in the concept of setting timeouts for requested services. For example, A service has a timeout 300 ms for waiting for a re-

sponse from B, so when B will have to call a C service after 100 ms of executing its logic, the timeout for the C service will be the C timeout minus the time that the B has already consumed: $timeoutForC = timeoutForB - consumedTimeByB$. So $timeoutForC = 300\,ms - 100\,ms = 200\,ms$ see 2.3. But this conception has a disadvantage. Establishing a timeout for C may not be a guarantee that the C service will finish its work in the determined period after it is called, because we do not consider when the C service will actually begin processing a request, so the real deadline of the C service may be a little bit longer than the B service need it to be (because of the network latency, waiting for starting processing a request in the queue due to the server's overloading, etc.).

Chapter 3

Testing

In this chapter, the process of inspecting the application functionality correctness will be demonstrated. Before diving into how microservices should be tested, common backend testing patterns will be discussed. After that, distributed inspection techniques and issues will be explained.

3.1 Monolithic testing

The monolithic testing process enables to realize basic software inspection techniques that are also used in testing microservices applications. This section is divided into explaining unit, integration, system, and acceptance test processes which are analyzed in corresponding subsections.

3.1.1 Unit testing

Testing of the smallest software piece independently from the rest of the application is called unit testing. It can be performed by checking only a supposed output of a function [6]. However, unit tests do not provide a guarantee that the whole application works as expected, due to the possibility of problems existence in the collaboration of different units of a system. For example, objects interaction can be set improperly, despite individual functionalities may work correctly and be unit tested, but their wrong cooperation can cause the whole application collapse. That is what integration testing is responsible for.

3.1.2 Integration testing

Assuming that separate system pieces are unit tested integration tests can be performed. This approach consists in testing how correct separate modules interfaces and data flows are configured [7, Chapter Example of Integration Test Case]. Integration tests actually check if the result of several modules/services collaborative work is the same as expected. Moreover, such way of testing can

be performed in different ways [7, Chapter Approaches, Strategies, Methodologies of Integration Testing]:

- Big Bang Approach
- Incremental Approach

3.1.2.1 Big Bang Approach

This approach stands for testing all the application modules wired together, which have its advantages and disadvantages. Big Bang strategy is a very convenient method of integrated testing for small application, because of its simplicity. However, it may become extremely hard to detect a bug in a colossal system this way. Moreover, all pieces of software should be implemented to perform a testing process, which leads to redundant waiting of QA specialists [7, Chapter Big Bang Approach]. Furthermore, some critical situations may not be tested because of trying to simulate all cases and that is why some scenarios of secondary modules' functionality can be missed [8].

Hence Big Bang testing approach is a very convenient way of testing smaller systems, where the only aim of the testing process is checking if the whole application works or not. But is not considered to be the best method for looking for a bug or a broken service.

3.1.2.2 Incremental Approach

Instead of testing the whole application (using the Big Bang approach), it may become much more convenient and efficient to wire two or several modules together and to add necessary modules one by one [7, Chapter Incremental Approach]. However, this way of testing can be divided into 3 categories:

- Top Down integration
- Bottom Up integration
- Hybrid/ Sandwich integration

The difference between the Top Down and the Bottom Up testing approach consists in that the Top Down method starts testing the main module adding less significant modules and the Bottom Up design does the same but in the opposite way. The Bottom Up approach starts with testing lower (more specific) modules wiring them with higher (main) ones.

The Hybrid method of integration incremental testing is the mix of the Top Down and Bottom Up testing approaches. Integration testing of bottom modules is performed at the same time as higher modules testing. The testing process is heading to the "middle" module layer to complete the final test of the "bundled" system.

3.1.3 System Testing

System testing is a process of inspection of the wired system with its external dependencies [9], which is actually a process of inspection of the whole application functionality. This way of testing is usually executed after unit and integration tests and simulate the real application flow. System testing is frequently performed using demo data and production environment.

It may become a little bit confusing how does the system testing differs from the Big Bang integration testing. However, the difference is very clear, Big Bang integration testing is the process of testing if wired modules work together as expected, but not meeting business requirements. On the other hand, the system testing tests specifications, if the application is implemented due to the final prototype and found bugs are not considered as product failure.

3.1.4 Acceptance testing

Acceptance testing is the final stage of inspecting an application that determines if a system fulfills client requirements. It is considered to be the last testing process before deployment. This testing phase inspects mostly the application use-cases to provide the completed product. It is frequently performed by a customer or some users who test the application. Actually, it is the process of inspection if the application can be used by real users.

3.2 Microservices testing

The microservices testing differs a little bit from the monolith application testing, whereas a lot of concepts and tools stay the same. Services' entire logic is inspected using unit testing and mocking responses from other services. The integration testing focuses on reviewing how do microservices communicate with each other and verifying real outputs, which has no difference with the integration testing of a monolith system. However, end-to-end microservice application testing can be extremely painful because of a system's fragmentation that differs from the monolithic approach in which the application can be easily deployed to the localhost and be effortlessly tested.

3.2.1 Contract testing

Contract testing can be extremely useful when testing microservices. This approach lies in inspecting if the service (provider) which is assumed to return some data, returns them in the consumer expecting format [10, Chapter Contract Tests]. Such a process is considered to be a black-box approach. A tester doesn't need to know how a service is implemented, it is required only to verify if the microservice's response fulfills the consumer-provider contract.

3.2.2 End-to-end strategies

As previously mentioned, the end-to-end testing process is a serious issue in the microservice world. In this section these problems will be discussed and several testing strategies will be demonstrated.

3.2.2.1 The full stack in-a-box strategy

It becomes almost impossible to execute the system testing in the old school way like testing on the local machine. If an application consists of 4–5 services, it can be quite simple to run all these services together, but managing the right branch of every repository (considering that every microservice is versioned in the separate repository), pulling the latest version of the branch and constantly starting services may lead to boilerplate work every time before beginning a test process [11]. Moreover, starting a system of dozens microservices on a local PC will likely make this machine useless for testing an application and potential scalability.

3.2.2.2 Personal deployment strategy

The personal deployment strategy focuses on creating a cloud environment for each tester where all services are deployed. Very frequently all QA specialists have individual Amazon Web Services accounts what makes scalability of the testing process much higher, because of the ability to run the system in the cloud right away.

3.2.2.3 The shared testing instances strategy

The shared testing strategy is considered to be a mix of the full stack in-a-box strategy and the AWS strategy (personal deployment strategy). Its concept is based on the idea that not all the application microservices have to run on the local machine, some services can be deployed to a remote server that can reduce the local PC load.

Chapter 4

Advantages of the microservice architecture

One of the initial steps in developing the information system consists in deciding if the application will be developed monolithic or distributed. This decision is one of the most important due to the complexity of converting the system from one architecture to another. Therefore, in this section advantages of the microservice architecture will be discussed.

4.1 Distributed work

The main advantage of using microservice architecture is the work distribution. Developing separate service becomes much more comfortable for programmers, they can easily realize their microservice in the programming language they want and not being aware of the technologies their colleagues use.

Moreover, when services are separated it becomes easier to maintain them because the team developing the separated microservice doesn't have to understand the whole logic of the application. When fixing some bugs, developers focus on the small bunch of logic.

4.2 Scalability

Concerning the colossal system which has been developing for several years, it becomes very hard to extend such application because it may be written in Cobol or the logic is so complex that it becomes unbelievably difficult to perform the extension. However, microservices let the developer just add a separate backend which will enhance the existing application. Such approach makes working with a legacy system much more comfortable, efficient and cheaper, because it is not any longer necessary to seek specialists in concrete

technology, furthermore a team can use any modern framework that improves a system development.

So when implementing a massive application that will probably have to be expanded in the future, it is a good choice to use a distributed architecture to have a possibility for painless scalability.

4.3 Deployment

First of all, the deployment term has to be explained. When developing an application the final stage assumes that the completed version will be published to any place from where it could be accessed. In the case of web applications, the software is usually launched on a server that is responsible for communicating with clients. A deployment is a group of actions whose goal is to make an application accessible from some environment [12].

Regarding the deployment, the microservice architecture enables the increase of the delivery speed. In case of small changes, recompiling and redeployment of the small service is much more efficient and less time consuming than manipulating with a large monolithic application [13], because the individual team can perform some repair and deploy their microservice independently from other teams. That is why the delivery process becomes more agile [14].

4.4 Fault tolerance and resilience

It is considered to be a serious advantage that microservice architecture can be fault-tolerant and resilient. Unlike a monolithic approach where one fault in the logic can cause the collapse of the whole application, well designed distributed system can tolerate if one microservice goes down in such a way that a client does not have to find out that fault happened [15]. The possibility to avoid a single point of failure makes a system resilient. The main idea is that when making a remote call from one microservice to another, the client service can react to the failure of the second service and either retrieve the cached response or mock the necessary value.

Chapter 5

Disadvantages of the microservice architecture

Indeed, not every system needs to be implemented with microservice architecture [16]. This development choice may be redundantly very painful for programming teams during the project implementation. Moreover, since such a distributed way of developing is considered to be a very young approach, there are not a lot of best practices and it becomes quite hard to find strict microservices standards.

The aim of this chapter is to demonstrate the disadvantages of using the distributed software approach. Complexity issues as well as companies' culture, security, and microservices' communication problems will be shown in particular sections.

5.1 Complexity

Developing microservices is more complex than a monolithic application. Due to the possibility of using several stack of technology, a distributed system requires maintaining different frameworks, programming languages, database schemes and so on [16, Chapter Disadvantage #1].

In addition, refactoring a huge legacy system from the monolith to microservice architecture will take a huge effort [16, Chapter Disadvantage #1]. This process can take some time and thereafter costs a lot of money.

5.2 Culture

Each microservice is developed by a separate team. That is why developers in any team have to understand the full process of developing the application [16, Chapter Disadvantage #2: Microservices Require Cultural Changes]. Each team has to have enough experience in creating, deploying, and testing

microservice independently from other teams, that is why every team needs DevOps specialists to be able to create reliable microservice and comfortable workflow.

5.3 Security

Due to exposing communication to the network, security problems may become a nightmare. Moreover, one weak place in the service is a dangerous problem for the whole application, because of using the same logic in different services frequently [16, Chapter Disadvantage #4: Microservices Can Present Security Threats].

5.4 Communication

Communication of different application parts in the monolith architecture is implemented by using classes and functions from the same code bunch. Such an approach is a little bit faster compared to the microservice architecture, where calls to individual services are performed remotely. Communication inside such a distributed system has to be organized intelligently not to call redundantly the same functionality several times. However, the microservice approach is not considered to cause a huge delay because of the high actual network performance.

The microservices communication problem causes the requirement of using a collaboration framework. It leads to the demand for maintaining an additional piece of software what is another disadvantage.

Chapter 6

Practical part

In this chapter, the realization of the real advertising application will be shown. The main idea of this section is the demonstration of using basic tools, patterns, and frameworks for creating a complete distributed backend that can be later expanded by a frontend client.

The main idea of the application consists in creating advertisements for cars, retrieving them, looking for similar cars. The result of the realized system will be fully working REST API.

6.1 Design

The design of the whole application will include functional requirements, non-functional requirements, and the analysis of the cars.cz platform.

6.1.1 Functional requirements

Firstly, it is necessary to specify functional requirements to have a clear appreciation of the system's functionality.

- The system will be able to keep records of advertisements.
- The system will be able to assign a rate value to advertisements.
- The system will be able to recommend similar cars.
- The system will be able to return information about a seller.

6.1.2 Non-functional requirements

Then, non-functional requirements have to be specified to define the system's restrictions and facilities [17].

- Scalability: The system has to be able to be easily extended by a new functionality.
- High performance: The system has to be able to process 10000 users' requests per second or to be implemented to have such performance characteristics after some modifications.
- Network availability: The system has to be available through a network, especially using the HTTP protocol.

6.1.3 Known approach

Figure 6.1: Cars.cz technology stack

At the beginning of the thesis development, several analogous cars' selling information systems were supposed to be analyzed, however later the impossibility of discovering the backend architecture was realized. That is the reason why only one approach was investigated. As an example of a similar application the cars.cz system was analyzed. The first step was to inspect this platform with the `wappalyzer.com` web application which allows discovering the technology stack of the site. The result of the analysis demonstrates that cars.cz platform uses the Prototype JavaScript framework and that the backend is probably written in Java see Figure 6.1.

The Nginx is a server that is able to perform proxy and load balancing functionality [18]. Due to the fact that cars.cz uses the JavaScript framework, load balancing, and has a backend written in Java the assumption is that this web application is implemented using the distributed architecture.

As a result, a similar approach is used during the advertising server implementation. The backend will be separated into several microservices written in Java, the proxy server will be implemented which can perform a load balancing functionality in the future.

6.1.4 Model

Firstly, the rough and very abstract model of the application was designed see Figure 6.3. In this model, a service discovery problem and a database layer are not shown. The main purpose is to demonstrate how do microservices represent the complete backend of the application. Then, data sources associated with appropriate services are visualized see Figure 6.2.

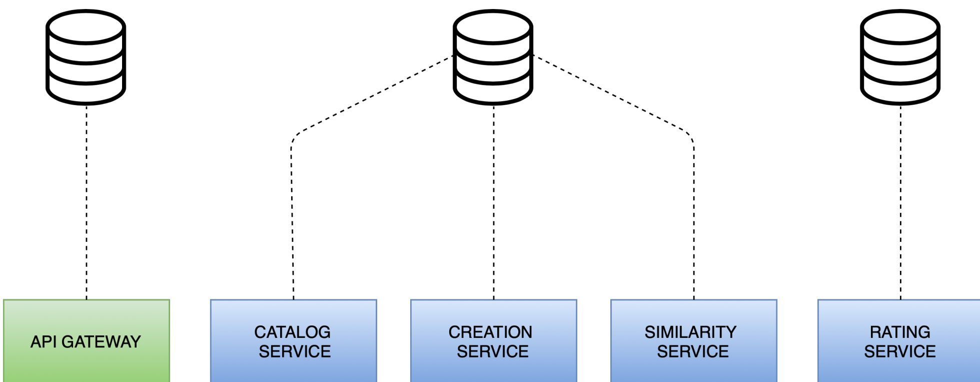


Figure 6.2: Microservices' data sources

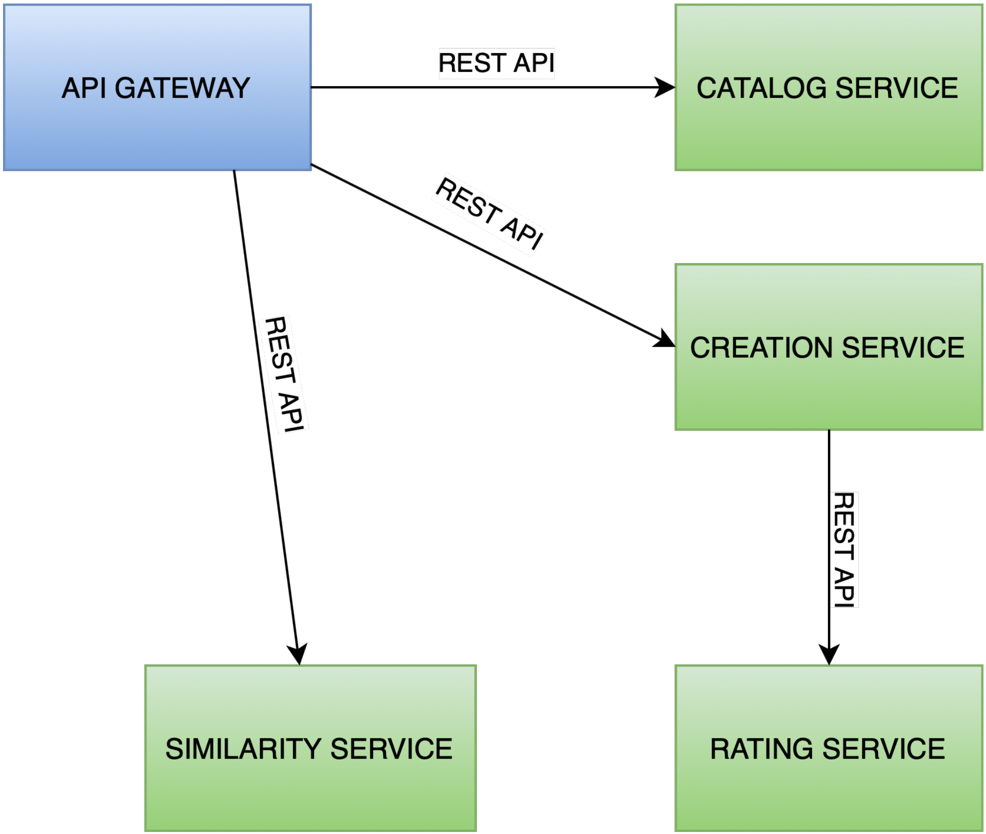


Figure 6.3: Autos backend rough model

6.2 Technologies

In this section used frameworks, tools and libraries will be shown and discussed. Furthermore, some basic concepts used in operating with various tools will be shown.

6.2.1 Programming language

As a programming language for implementing the backend part was chosen Java. The choice was affected by the popularity of this language and a big community. The advantage of such a famous programming language is the simplicity of looking for solutions for specific problems. However, Java is not considered as the easiest programming language to learn, neither the most painless way of implementing server-side software.

6.2.2 Web framework

As the main platform for implementing microservice architecture was chosen the Spring Boot framework. This framework is considered to be a very clear solution for implementing a REST API with Java. Moreover, a huge ecosystem has been developed around this platform. Many components of the Spring environment will be used and discussed as well.

6.2.3 Security

Almost all microservices are going to be hidden from public access. This will be ensured by the firewall tuning of individual services, nevertheless, this setup will not be realized in the scope of this thesis. The idea lies in the concept of communication through the proxy microservice which is called in this communication as the API Gateway service. This particular application server will collaborate with other services by forwarding all requests to concrete microservices and implement the registration and authentication functionality.

This part was realized by using the Spring Security framework. However, during the development, the complexity of this solution was realized and only basic functionality was used.

6.2.4 Object Relationship Mapping

As the application has to persist data in some way, using individual database rows as objects make the business logic implementation much easier. Nevertheless, the technique of manual mapping every retrieved data result to the Java object is considered as the archaic approach because of the existence of many frameworks solving this problem. For the development of this application was chosen Hibernate framework, which is the default JPA implementation bundled with the Spring framework.

Spring Data JPA: Spring environment provides a very convenient mechanism for working with ORM frameworks. Spring Data JPA uses Hibernate entirely and provides a feature for a convenient working with a data source called “Repository”. This mechanism allows to create an interface, which would inherit the `CrudRepository` interface and then to define the method in such way: Optional<Car> findByCarId(int carId); Then when autowiring such interface, Spring would provide the concrete bean with implemented methods which should be only defined in the interface.

6.2.5 Project management system

Maven management tool was chosen for the project administration. It is considered to be a default system for managing Java projects. It is very easy to use, nevertheless, Maven’s building process is not simply configured. Its

build-cycle is rather determined by Maven's convention than set up by a developer [19]. Maven build process consists of 3 life-cycles: default, clean, and site. Each has its own list of phases, which are responsible for specific build aspect.
Default: the default stage of the build is the main Maven group of phases. It consists of 23 phases and covers almost all necessary steps for building a Java program.
Clean: the clean life-cycle is responsible for removing all redundant files created by previous Maven build processes and has 3 phases.
Site: the site life-cycle has 4 phases whose goal is to create documentation for a Java software [20].

6.2.6 Service discovery

One of the advantages of using the Spring platform is the simplicity of adding all necessary functionality for operating with microservice architecture. One of the most significant microservice tools is the Eureka server, which was developed by Netflix and is widely used. This concrete technology was chosen because of its simplicity, convenience, and popularity.

Firstly, it is necessary to start a Spring Boot application and add the necessary Spring Cloud and Eureka dependencies. In the pom file basic Spring Boot and Spring Cloud dependencies are added, which makes the application ready to be launched.

Then the main Spring class should be annotated with `@EnableEurekaServer` and finally may look like in Listing 6.1.

```
1  @SpringBootApplication
2  @EnableEurekaServer
3  public class DiscoveryServerApplication {
4
5      public static void main(String[] args) {
6          SpringApplication
7              .run(DiscoveryServerApplication.class, args);
8      }
9  }
```

Listing 6.1: Eureka main class

Nevertheless, some additional properties can be specified in the `application.properties.yml` file. It is the configuration place for properties of specific Spring classes [21]. The `application.properties` file of the configured Eureka server is demonstrated in Listing 6.2.

Server port: Port of the discovery server is explicitly established due to the possibility of appearing the conflict of ports. For example, if the port of the Eureka server is not set, the default 8080 port would be used. As a result, when

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Listing 6.2: Eureka application.properties

starting another Spring application without the established port number, the exception will be thrown with the message: “Web server failed to start. Port 8080 was already in use”.

Another two properties tell the Eureka server not to be registered in any other Eureka server as a client. After this configuration process is finished the Eureka server can be launched and its the graphical interface is available at `http://localhost:8761` where all registered clients are displayed.

6.2.7 Eureka client

After setting up the Eureka discovery server, the next step is creating a microservice which will be registered in the Eureka server. A basic `application.properties` file has to contain 2 properties and is shown on the example 6.3.

```
server.port=8081
spring.application.name=cars-catalog-service
```

Listing 6.3: Eureka client application.properties

The server port number sets the port on which the application is running. The second property specifies the name of the service which is the identifier of the application which is used by the Eureka server. When a service is requested, it is called by its name and the discovery server determines the location and returns it, after that the request is sent to the particular Spring application.

Then, the Eureka client dependency has to be added to the pom file as shown in Listing 6.4. It is supposed that the Eureka client Maven project is inherited from the Spring Boot project.

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>
        spring-cloud-starter-netflix-eureka-client
    </artifactId>
</dependency>
```

Listing 6.4: Eureka client pom.xml

The main application class has to be annotated with the `@EnableEurekaClient` annotation. After that, a microservice will be registered in the discovery server as an Eureka client. Then the main class may be simply started and the microservice will be ready to process requests.

6.2.8 Unit and integration testing tools

For performing unit and integration testing the JUnit framework was used. This is considered to be the most popular framework for testing the Java code. [22] However, 2 JUnit widely used versions exist and the latest JUnit 5 framework was chosen because of its more capacious functionality which allows the test development to be more expandable in the future.

During the unit testing process, only limited logic blocks are inspected, that is why some related functionalities have to be mocked. As a result, additional dependency called Mockito was added to perform objects mocking.

6.2.9 Load testing tool

At the final stage of developing the application, performance tests were executed. This process was implemented by testing the load abilities of the backend. The main idea was to execute a lot of HTTP requests to test the abilities of the completed application what was achieved by using the JMeter tool. The results of this testing process will be shown and discussed in later sections.

6.2.10 Git

As the information system developed in the scope of this thesis is a software product the source code needs to be saved in storage and versioned in some way. That is why Git tool is discussed in this section.

Git is a distributed version control system, which means that almost all operations are performed on the local repository. As a result, it is not necessary to even have the Internet connection to manage the project. The state of the project is not controlled as states of individual files like in the Subversion, however individual snapshots are stored in the memory. Snapshots represent all source files in the time when the commit operation was performed [23, Chapter 1: Getting Started].

As the place for staging the Git repository the GitHub platform was used which allows to store and manage Git projects for free. The biggest advantage of keeping files on the remote server (in case of this thesis) is the ability to easily share the project with the tutor to continuously receive feedback.

6.3 Data layer

Due to the conception of loosely coupling, individual microservice is considered to operate with a separate data domain of a system. Therefore, each service has to have a personal data source that is responsible for the concrete data field [24]. Nevertheless, in the advertisement application, some microservices will persist data in shared data sources, because of the application's simplicity.

6.3.1 Database Management System

As the Database Management System (DBMS) for the application was chosen MySQL because of its simplicity, but any similar relational DBMS can be applied for similar application development purposes.

6.3.2 Database design tool

The process of the creation and extension of the data layer will be performed with the help of the MySQLWorkbench tool. This software will help to perform any database modifications as changing a model, manipulating with data, modification of the schema.

6.4 Implementation

The application implementation lies in realizing the backend part of the advertising server and testing it in a proper way, which will be demonstrated in this section. Nevertheless, some realization details are not shown in this section and are described in the documentation.

6.4.1 Microservices

In this section individual microservices implementation processes will be shown and discussed, and that their collaboration will be demonstrated as well. Each microservice is a registered Eureka client that allows to call them by retrieving their endpoints from the discovery server what was discussed in Section Service discovery.

6.4.1.1 Eureka service

The first microservice to be established is the service discovery server with the help of the Eureka library. Its launch is similar to launching a classic Spring Boot application. However it is necessary to add several dependencies, annotate the main class with `@EnableEurekaClient`, `@SpringBootApplication` and set necessary properties in the `application.properties` file what as was shown in section Service discovery. After the discovery server is launched its console is available at the port

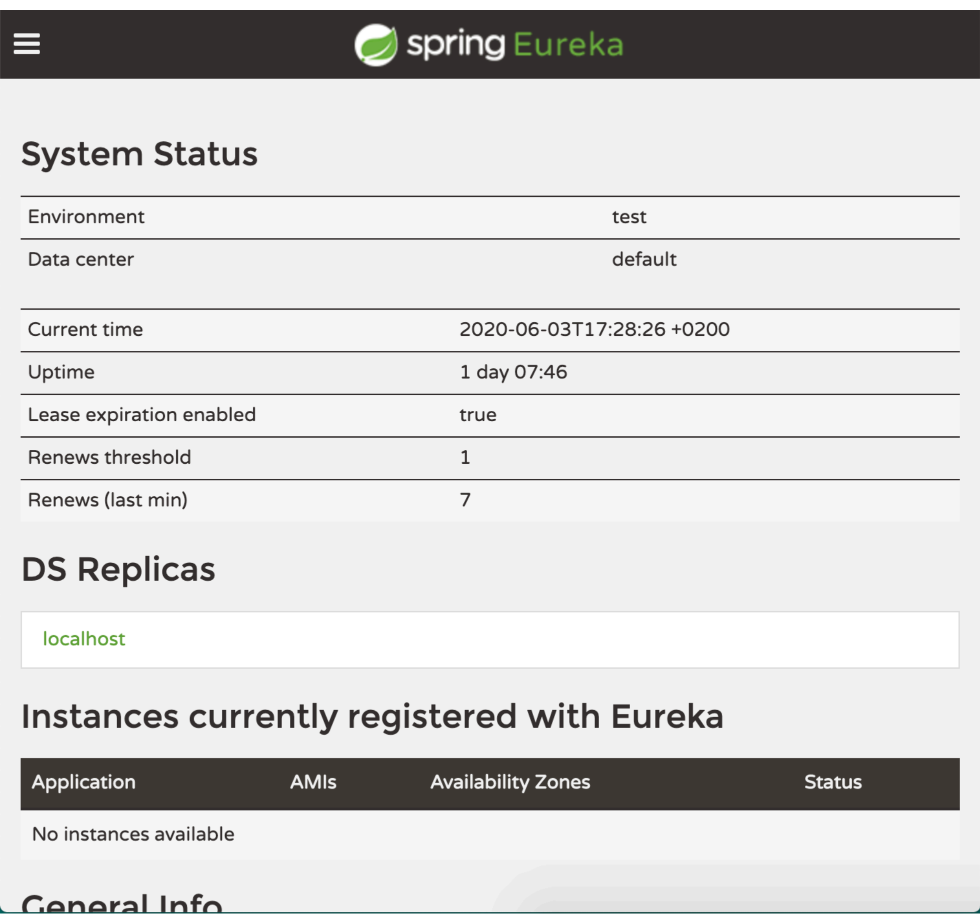


Figure 6.4: Eureka console

defined in the `application.properties` file. When opening the browser at `http://localhost:8761` the Eureka console is visible see Figure 6.4 Nevertheless, no services are started so far, that is why the application section contains the string "No instances available".

6.4.1.2 Catalog service

The main purpose of the Catalog microservice is to retrieve cars' advertisements. As the first step in developing this backend part of the application is defining corresponding REST API endpoints see Table 6.1.

The /adv endpoint returns all advertisements, the /adv/{id} sends back the detailed information about the advertisement. The /adv/{id}/author endpoint is responsible for retrieving data about the author of the concrete advertisement.

GET:	/adv
GET:	/adv/{id}
GET:	/adv/{id}/author

Table 6.1: Catalog endpoints table

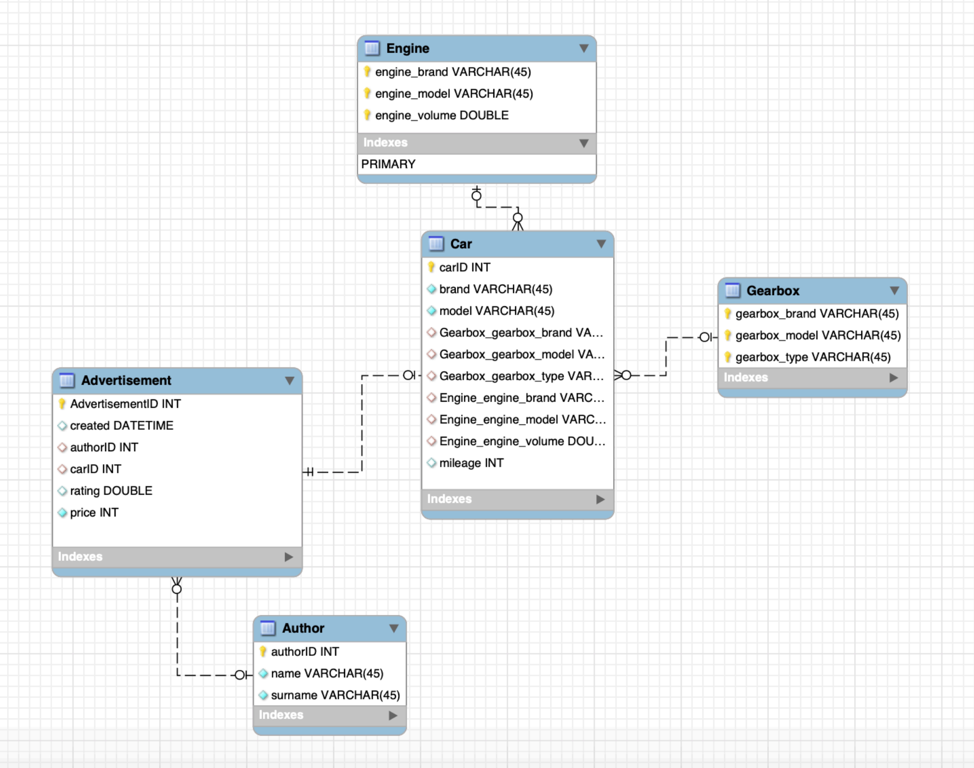


Figure 6.5: Catalog relational model

The start of creating the microservice is considered in establishing the data source. For this purpose, MySQL database management system was used. Using MySQLWorkbench tool the relational model was created see Figure 6.5. After that, the MySQL server is launched and the model is wired with it. The connection between the model and the data source makes possible the continuous synchronization between the model and the database.

After setting up the database, the Spring application is starting to be developed. First of all, necessary dependencies are added and data source properties are configured. Then, the Eureka client settings are set and the microservice becomes registered.

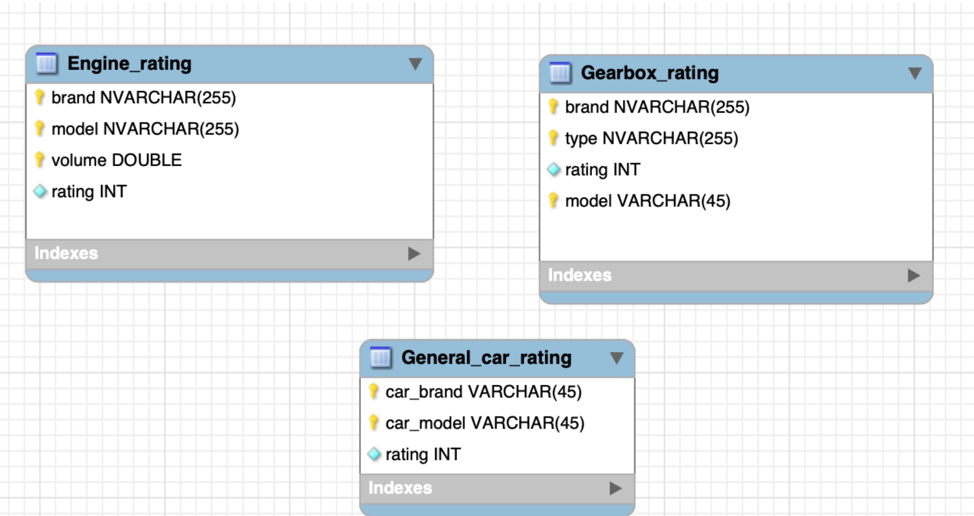


Figure 6.6: Rating relational model

6.4.1.3 Rating service

The Rating microservice is responsible for assigning the value from 0 to 10 to a particular car. A separate data source was established for the possibility to retrieve the information about the quality of particular gearboxes, engines, and specific cars. First of all, the relational model was created see Figure 6.6 and connected with the MySQL server. The data model describes the concept that each engine, gearbox, and car which is present is corresponding tables has to have the rating attribute which is retrieved by the Spring Rating application.

This Spring project has only one endpoint which receives the information about a car and returns a rating see Table 6.2.

POST:	/rating

Table 6.2: Rating endpoints table

The Spring controller obtains from the POST request such data: car brand, car model, engine brand, engine model, engine volume, gearbox brand, gearbox model, gearbox type. In case that any of three entities is not found the particular entity rating value is 0. For example, if a retrieved engine is not present in the database, then its value is 0. The final result of the calculation is described by the formula:

$$(generalCarRating + engineRating + gearboxRating)/3.0$$

Each rating value varies from 0 to 10 and that is why the final sum is divided by 3. The response contains the result of the rating calculation in the

response body.

6.4.1.4 Creation service

The Creation service is responsible for inserting a new advertisement to the database. Its data source is the same database as the Catalog microservice uses. The most interesting aspect of this service is the fact that for the insertion of a new advertisement it calls the Rating service to assign a rating value to the new advertisement. The communication between services is achieved through the Eureka server and the `RestTemplate` bean which is responsible for communicating with the discovery server and the Rating microservice.

The only endpoint in the Creation microservice is of type POST and takes the information about the advertisement, engine, gearbox, car and the author see Table 6.3.

POST:	/adv

Table 6.3: Creation endpoints table

6.4.1.5 Similarity

The Similarity service is the microservice which was designed to look for similar advertisements and as a result, the GET endpoint was created see Table 6.4. The data source of this Spring application is the same as the data source of Catalog and Creation service as the logic lies in retrieving necessary advertisements. Similarity criteria are the rating value and the price. The only necessary information for retrieving similar cars is the id of the car to which analogous autos is looked for.

GET:	/similar

Table 6.4: Similarity endpoints table

6.4.1.6 API Gateway

Another microservice solution used in the Advertising server is the conception of the API Gateway. This approach lies in requesting only the API Gateway microservice. Clients call only this proxy server which then retrieves individual microservice endpoints to which requests are forwarded. The framework which realizes such proxy-server behavior is called Zuul which was developed by Netflix and has many other features besides the proxy functionality [25].

The second problem which the API Gateway solves is the security issues. Due to the fact that all requests are going through the proxy server, only API

Gateway issues have to be resolved. Therefore, 2 endpoints are established for registering new users and authentication see Table 6.5.

POST:	/register
POST:	/authenticate

Table 6.5: Gateway endpoints table

Before demonstrating authentication, registering, and proxy functionality JWT tokens need to be discussed. The conception of JSON tokens lies in passing signed information at first from server to client and then from client to server with every request. Every JWT encoded string consists of 3 parts: header, payload, and signature, divided by a dot see Figure 6.7. The Header part describes the algorithm which was used for singing the token, the second part of the token is a payload and contains all necessary data which generally contains a username, user id but nothing private as passwords, cards' numbers, etc. Due to the fact that header and payload are just data encoded in base64, no confidential data have to be passed by a JWT token. Nevertheless, the server has to verify passed data in some way that is achieved by checking the 3-rd part of a token [26]. When creating a token, a signature is passed as the last part of a JWT. That is why in case of creating a token on the client-side by a hacker, the signature part cannot be faked because a hacker doesn't know the secret key to create a signature for the particular payload.

Register: the register endpoint is responsible for signing up new users. The controller receives username and password credentials in the POST request body, insert them to the database in case that the same username doesn't exist yet and returns a token.

Authenticate: the authenticate endpoint is related to checking in the database if the user credentials passed in the request body are valid and returning the JWT token.

One of Spring Security features is filtering. The API Gateway filters inspect every request coming through it, verify a JWT token, check if the username is present in the database (what can be considered as redundant but in this application, it is the additional security verification step) and forward the request to the particular microservice.

6.4.2 Continuous integration

First of all, continuous integration techniques need to be discussed. Continuous integration is the concept of having a 100% valid tested code that can be deployed at any time [27]. The high quality of source code is achieved by an immediate build and testing after pushing to the repository [28]. The big advantage is that the application is built not only once a day but continuously after each contribution.

Encoded

eyJhbGciOiJIUzI1NiIsInR5cCI6
IkpXVCJ9.eyJzdWIiOiIxMjM0NTY
3ODkwIiwibmFtZSI6IkpvaG4gRG9
lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36P
Ok6yJV_adQssw5c

Decoded

HEADER:

{
  "alg": "HS256",
  "typ": "JWT"
}

PAYLOAD:

{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
```

Figure 6.7: JWT structure

Jenkins: Jenkins is one of the most popular tools for CI. However, only the basic features of Jenkins are used in this thesis. This tool allows to create a separate server that can react to the changes made in the repository and then build, test, and return a result if integration manipulations were successful but for this purpose, a real server has to be established but not the local one. That is why the microservices integration process was completed by manual forcing Jenkins to build and run the "maven test" command.

Each item checks out the common Git repository which contains all microservices and runs the build and testing process of the particular microservice. The dashboard of the configured Jenkins build and test results of individual microservices is shown in Figure 6.8.

The basic Jenkins functionality is used to build and run tests processes of particular microservices. However, the real use of such CI tool should look a little differently. First of all, Jenkins has to be running on some server to be able to react to the changes pushed to the repository as Jenkins usually receives a notification from a Git hosting environment as GitHub. Another useful feature which can be applied in the future is called the Jenkins pipeline. Essentially it is a set of plugins used for executing commands for delivering the code from the repository environment to the final stage (QA environment, production environment) [29] which usually reacts to a "push" to a repository and execute predefined processes (build, test, etc.).

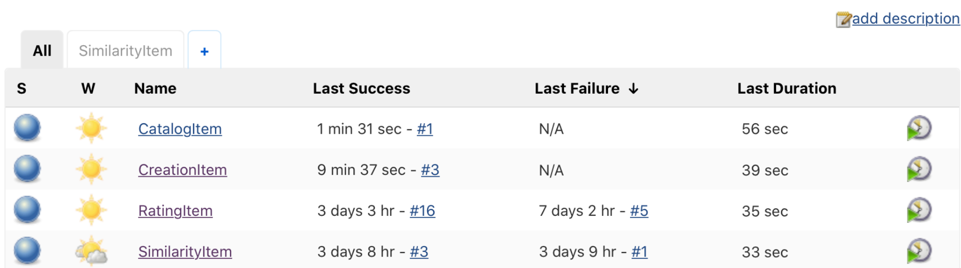


Figure 6.8: Jenkins dashboard

6.4.3 Performance testing

As part of the thesis, performance tests have been completed. The specification of this type of test is called stress testing which approach lies in the unreal loading the system [30]. In the case of the advertising server application, the performance test was ensured by sending an enormous number of requests through the API Gateway service to the catalog microservice.

The strategy for the stress performance testing is to detect the critical number of requests in which microservices are available to process and to define future modifications for the system scalability. The main parameters for executing such tests are the number of requests and the Ramp-Up Period which defines the period for sending all requests. For the first test, the number of calls to the microservice is 500 and the Ramp-Up Period is 3 seconds. The JMeter has the ability to execute such tests and to demonstrate results in a table see Figure 6.9. After inspecting the result of the testing, the successful outcome has been detected.

After such a result the more serious load process can be performed. The new number of requests is 10000 and the Ramp-Up Period stays the same (3 seconds). The result clearly illustrates that the capabilities of the service are not enough for processing such a high number of calls see Figure 6.10 However, almost all requests were completed and according to several tests the maximum number of calls for successful processing was determined as 4000.

Nevertheless, microservices can be duplicated as well as data sources can be modified for processing a higher amount of requests. The API Gateway can implement the load balancing logic as well as the discovery server. However such adjustments are not realized in the scope of this thesis but can be performed in the case that the system will be extended by a front-end client and the demand for scalability would be detected in the future.

#	Start Time	Thread Na...	Label	Sample Ti...	Status ↓
1	17:57:04.910	Thread Gr...	HTTP Req...	183	
2	17:57:05.013	Thread Gr...	HTTP Req...	170	
3	17:57:05.0...	Thread Gr...	HTTP Req...	160	
4	17:57:04.9...	Thread Gr...	HTTP Req...	198	
5	17:57:05.0...	Thread Gr...	HTTP Req...	182	
6	17:57:04.991	Thread Gr...	HTTP Req...	198	
7	17:57:05.031	Thread Gr...	HTTP Req...	160	
8	17:57:05.0...	Thread Gr...	HTTP Req...	147	
9	17:57:05.041	Thread Gr...	HTTP Req...	153	
10	17:57:05.0...	Thread Gr...	HTTP Req...	161	
11	17:57:05.0...	Thread Gr...	HTTP Req...	137	
12	17:57:04.9...	Thread Gr...	HTTP Req...	269	
13	17:57:05.0...	Thread Gr...	HTTP Req...	145	
14	17:57:04.9...	Thread Gr...	HTTP Req...	219	
15	17:57:05.0...	Thread Gr...	HTTP Req...	165	
16	17:57:04.9...	Thread Gr...	HTTP Req...	249	
17	17:57:04.8...	Thread Gr...	HTTP Req...	337	
18	17:57:04.919	Thread Gr...	HTTP Req...	283	
19	17:57:04.8...	Thread Gr...	HTTP Req...	343	
20	17:57:04.9...	Thread Gr...	HTTP Req...	277	
21	17:57:04.9...	Thread Gr...	HTTP Req...	296	

Figure 6.9: JMeter success result table

Sample #	Start Time	Thread Na...	Label	Sample Ti...	Status ↓
9582	18:10:13.695	Thread Gr...	HTTP Req...	1454	
9583	18:10:13.694	Thread Gr...	HTTP Req...	1455	
9584	18:10:13.573	Thread Gr...	HTTP Req...	1577	
9585	18:10:13.324	Thread Gr...	HTTP Req...	446	
9586	18:10:13.689	Thread Gr...	HTTP Req...	1462	
9587	18:10:13.681	Thread Gr...	HTTP Req...	1470	
9588	18:10:13.677	Thread Gr...	HTTP Req...	1475	
9589	18:10:13.674	Thread Gr...	HTTP Req...	1478	
9590	18:10:13.673	Thread Gr...	HTTP Req...	1479	
9591	18:10:13.592	Thread Gr...	HTTP Req...	1560	
9592	18:10:13.354	Thread Gr...	HTTP Req...	1798	
9593	18:10:13.594	Thread Gr...	HTTP Req...	1558	
9594	18:10:13.358	Thread Gr...	HTTP Req...	1795	
9595	18:10:13.591	Thread Gr...	HTTP Req...	1562	
9596	18:10:13.590	Thread Gr...	HTTP Req...	1563	
9597	18:10:13.587	Thread Gr...	HTTP Req...	1567	
9598	18:10:13.579	Thread Gr...	HTTP Req...	1575	
9599	18:10:13.578	Thread Gr...	HTTP Req...	1576	
9600	18:10:13.355	Thread Gr...	HTTP Req...	1799	
9601	18:10:13.576	Thread Gr...	HTTP Req...	1578	
9602	18:10:13.700	Thread Gr...	HTTP Req...	1448	
9603	18:10:13.357	Thread Gr...	HTTP Req...	1797	

Figure 6.10: JMeter fail result table

Conclusion

In the scope of this thesis, the complete research of the microservice architecture was performed. Many concepts and patterns of such an approach have been demonstrated and a wide determination of the idea of a distributed information system was presented. Moreover, the real application development process was realized and tools used during this process were shown as well.

The thesis consists of 2 big components. The aim of the first one is to present different approaches in a theoretical manner. This part describes concepts of the microservice architecture in a quite fundamental way, however, some techniques were not explained because of their hugeness. One of them is the fallback strategy problem which is considered to be extremely intriguing and not even been solved in many companies that use the microservice approach.

The second part of the thesis is explaining the advertising server development process. The final results of testing demonstrated the ability of this project to be immediately used in the production in case that a frontend client would be provided. Even the performance modifications were explained what makes it possible to use the backend in the highly loaded system after providing some adjustments.

The thesis is considered to fulfill all requirements established at the beginning. Nevertheless, the final opinion regarding applying the microservice architecture in the real world is not so univocal. The approach chosen by a company or a group of developers about the application architecture has to be elected after performing some analysis and after realizing requirements and potential future evolutionary processes.

Appendix A

Acronyms

API Application Programming Interface

AWS Amazon Web Services

CI Continuous Integration

DBMS Database Management System

GUI Graphical user interface

HTTP Hypertext Transfer Protocol

JPA Java Persistence API

JSON JavaScript Object Notation

JWT JSON Web Token

PC Personal computer

QA Quality assurance

REST Representational State Transfer

URL Uniform Resource Locator

Appendix **B**

Contents of enclosed CD

- `src` .... the directory of source codes
  - `autos` .... implementation sources
  - `thesis` .... the directory of LaTeX source codes of the thesis
- `text` .... the thesis text directory
  - `thesis.pdf` .... the thesis text in PDF format
- `doc` .... the directory of documentation
  - `index.html` .... Documentation generated by Swagger

Bibliography

1. GOURLEY, David; BRIAN TOTTY, Marjorie Sayer; AGGARWAL, Anshu; REDDY, Sailu. HTTP: The Definitive Guide: Understanding Web Internals. In: Sebastopol: O'Reilly Media, 2002, pp. 3–4. ISBN 978-1-56592-509-0.
2. Service Discovery in a Microservices Architecture. *DZone* [online] [visited on 2020-05-17]. Available from: `https://dzone.com/articles/service-discovery-in-microservice-ecosystem`.
3. Service Discovery Definition. *Avi Networks* [online] [visited on 2020-05-17]. Available from: `https://avinetworks.com/glossary/service-discovery/`.
4. BEHARA, Samir. Making your Microservices Resilient and Fault Tolerant. *Samih Behara* [online] [visited on 2020-05-17]. Available from: `https://samirbehara.com/2018/08/06/making-your-microservices-resilient-and-fault-tolerant/`.
5. PERIKOV, Igor. 5 patterns to make your microservice fault-tolerant. *Medium* [online] [visited on 2020-05-17]. Available from: `https://itnext.io/5-patterns-to-make-your-microservice-fault-tolerant-f3a1c73547b3`.
6. Unit Testing. *Software Testing Fundamentals* [online] [visited on 2020-05-17]. Available from: `http://softwaretestingfundamentals.com/unit-testing/`.
7. Integration Testing: What is, Types, Top Down & Bottom Up Example. *Guru99* [online] [visited on 2020-05-17]. Available from: `https://www.guru99.com/integration-testing.html`.
8. Big Bang Integration Testing. *ProfessionalQA.com* [online] [visited on 2020-05-17]. Available from: `https://www.professionalqa.com/big-bang-integration-testing`.

9. What is System Testing? Types & Definition with Example. *Guru99* [online] [visited on 2020-05-17]. Available from: https://www.guru99.com/system-testing.html.
10. GHAHRAI, Amir. Testing Microservices – A Beginner's Guide. *DevQA.io* [online] [visited on 2020-05-17]. Available from: https://devqa.io/qa/testing-microservices-beginners-guide/.
11. LUMETTA, Jake. These are the most effective microservice testing strategies, according to the experts. *freeCodeCamp* [online] [visited on 2020-05-17]. Available from: https://www.freecodecamp.org/news/these-are-the-most-effective-microservice-testing-strategies-according-to-the-experts-6fb584f2edde/.
12. LEE, Roger. Software Engineering Research, Management and Applications: Studies in Computational Intelligence. In: Berlin: Springer, 2008, p. 30. ISBN 978-3-540-70561-1.
13. HAMRLA, Lukáš. *Zajištění škálovatelnosti webových aplikací s využitím architektury mikroslužeb*. Prague, 2018. Master's thesis. Czech Technical University in Prague.
14. KUPRENKO, Vitaly. 6 Key Benefits of Microservices Architecture. *Stackify* [online] [visited on 2020-05-17]. Available from: https://stackify.com/6-key-benefits-of-microservices-architecture/.
15. Making Your Microservices Resilient and Fault Tolerant. *DZone* [online] [visited on 2020-05-17]. Available from: https://dzone.com/articles/making-your-microservices-resilient-and-fault-tole-1.
16. WITTMER, Phil. The Top Microservices Disadvantages & Advantages. *Tiempo Development* [online] [visited on 2020-05-17]. Available from: https://www.tiempodev.com/blog/disadvantages-of-a-microservices-architecture/.
17. Non-functional Requirements: Examples, Types, How to Approach. *AltexSoft* [online] [visited on 2020-05-17]. Available from: https://www.altexsoft.com/blog/non-functional-requirements/.
18. What Is Nginx? A Basic Look at What It Is and How It Works. *Kinsta* [online] [visited on 2020-05-28]. Available from: https://kinsta.com/knowledgebase/what-is-nginx.
19. BUT, Colin. Ant vs Maven vs Gradle. *Medium* [online] [visited on 2020-05-17]. Available from: https://medium.com/@Colin_But/ant-vs-maven-vs-gradle-801fde21af80.
20. BAELDUNG. Maven Goals and Phases. *Baeldung* [online] [visited on 2020-05-24]. Available from: https://www.baeldung.com/maven-goals-phases.

21. Common Application properties. *Spring Boot Reference Documentation* [online] [visited on 2020-05-17]. Available from: `https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html`.
22. KOTHAGAL, Koushik. JUnit 5 Basics 1 - Introduction and agenda. In: *Youtube* [online]. 2019 [visited on 2020-05-21]. Available from: `https://www.youtube.com/watch?v=2E3WqYupx7c&list=PLqq-6Pq4lTTa4ad5JISViSb2FVG8Vwa4oc`. Free Java and JavaScript Courses and Tutorials.
23. CHACON, Scott; STRAUB, Ben. Pro Git: EVERYTHING YOU NEED TO KNOW ABOUT GIT. In: New York: apress, 2014, vol. 2, p. 31. ISBN 978-1-4842-0076-6.
24. SITAPARA, Jay. Data Management Patterns for Microservices Architecture. *DATAVERSITY Education* [online] [visited on 2020-05-17]. Available from: `https://www.dataversity.net/data-management-patterns-for-microservices-architecture/#`.
25. SALERNO, Rafael. Get to Know Netflix's Zuul. *DZone* [online] [visited on 2020-05-25]. Available from: `https://dzone.com/articles/spring-cloud-netflix-zuul-edge-serverapi-gatewayga`.
26. Introduction to JSON Web Tokens. *JWT* [online] [visited on 2020-05-25]. Available from: `https://jwt.io/introduction/`.
27. ROSSEL, Sander. Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment. In: Birmingham: Packt Publishing, 2017, chap. 1: Continuous Integration, Delivery, and Deployment Foundations. ISBN 978-1-78728-661-0.
28. CONTINUOUS INTEGRATION ESSENTIALS. *CodeShip* [online] [visited on 2020-05-26]. Available from: `https://codeship.com/continuous-integration-essentials`.
29. Pipeline. *Jenkins* [online] [visited on 2020-05-31]. Available from: `https://www.jenkins.io/doc/book/pipeline/`.
30. MATAM, Sai; JAIN, Jagdeep. Pro Apache JMeter: Web Application Performace Testing. In: New York: apress, 2017. ISBN 978-1-4842-2961-3.