BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION SOFTWARE ENGINEERING

DISSERTATION THESIS

Distributed Examination System Suitable for Distance Education

Supervisor
Lect. Dr. Grebla Horea

Author
Macarie Cristian

2021

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INGINERIE SOFTWARE

LUCRARE DE DISERTAŢIE

Sistem distribuit de examinare adecvat pentru educația la distanță

Conducător ştiinţific
Lect. Dr. Grebla Horea

Absolvent
Macarie Cristian

2021

Abstract

The subject of this thesis is the problem of online examination, which has proven to be of high interest nowadays due to the current world crises, and the problem of choosing an appropriate software architecture for building such an application.

Therefore, the main goals are to provide an analysis and a comparison of different software architectures used for designing distributed systems both at the system level and at the individual microservice level while also considering the software architecture evolution. Moreover, the paper presents besides the theoretical foundations required for designing a distributed examination system a proposed model built with respect to the Microservices architecture.

The proposed model aims to ease the job of both teachers and students which have adopted distance education regarding examination by supporting the creation and management of exams and questions while also providing real time insights and statistics about exam answers.

Additionally, the model focuses on providing a system which should easily satisfy the technical needs of a modern application, thus promoting extensibility, scalability and maintainability by leveraging the advantages provided by the chosen software architecture and the modern technologies.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Contents

List of Figures

Introduction

In our days the educational system is moving towards distance education faster than ever before. This comes as a consequence of the new Coronavirus Disease (COVID-19) outbreak [16]. Moreover, this fast transition towards the new normal of online education highly increases the need of online educational platforms and online examination applications. However, due to the advanced technologies and the highly available communication networks which are available today, the process of building such a platform only remains a matter of choosing the frameworks and the software architecture which best suits this use case.

The purpose of this thesis is to present a way of building a minimum viable product of an online examination platform which should leverage the advantages provided by the modern software architectures and frameworks.

The application is aimed towards making life easier for both students and teachers which implement distance education. The thesis focuses on providing a system which should easily satisfy the technical needs of a modern application, thus promoting extensibility, scalability and maintainability via the nature of the chosen microservices software architecture.

In more details, the main functionalities of the practical application regarding to a teacher are the creation of reusable multiple choice, single choice or free text questions, the creation of time boxed exams composed by choosing a set from those questions and the real time monitoring of the exam answers. The main functionalities regarding to students are the ability to surf and resolve the available exams.

Thesis Outline

The **first chapter** presents some theoretical information regarding software system architectures suitable for distributed systems. The chapter is split into five sections as follows:

The first section aims to present the evolution of distributed software architecture towards the Microservices architecture. It starts with a brief description and then follows with some concrete architectural patterns like Service Oriented architecture and Microservices architecture.

The second and third sections presents some of the challenges one might face when dealing with the Microservices architecture and a short comparison of software components and libraries.

The forth section briefly describes the microservices communication patterns and then presents in more details the API-Gateway pattern. The pattern with respect to which the practical application's architecture complies.

The fifth section dives further and presents an architectural pattern which should be considered at the level of an individual microservice together with its advantages and flaws.

The **second chapter** starts with the description of the problem treated in the thesis along with the proposed model for solving it. Moreover, there is also described the functionality of the practical application and the architectural decisions considered when building it. Furthermore, the chapter presents some use cases of the application which are described in details and backed up by architecture, use case, sequence and databases diagrams. The last section presents the main technologies used in the implementation process.

The **final chapter** presents the conclusions alongside with the current state of the solution for the described problem, a short auto-evaluation of the advantages and limitations of the proposed solution and the description of the ideas for future improvements to it.

Chapter 1

Theoretical Background

1.1 Distributed systems towards Microservices

1.1.1 Brief Introduction

Nowadays most of the software systems are distributed and the reason behind this is the availability of the communication networks which have greatly evolved. This also had a great impact on the way those applications are developed as well as on their infrastructure which has to cope with this new distributed environment [2]. Moreover the need of being able to deploy applications to a large number of people online, as well as to do this in real time, highly encouraged the development of distributed applications.

At the beginning, the network layer which allows the applications to communicate between themselves was not taken into consideration when designing such applications due to the poor network capabilities, but this has changed with the development of more powerful networks. This influenced the popularity growth of the client-server model which considers multiple clients who demand services from a server. In this model, the base communication mechanism was the Remote Procedure Call also known as (RPC). Furthermore, due to the lack of end-user resources, most if not all computing was designed to be done on the server side.

Later, with the growth of more available and faster network connections, as well as with the development of lightweight communication protocols such as Application Programming Interfaces (APIs), the client-server model became even more used as its biggest drawback namely the bandwidth consumption was no longer significant due to those improvements.

Another model proposed to fulfill the problem of integrating distributed applications, a model which highly facilitates the reusability software quality attribute, was the Service Oriented Architecture (SOA). The architecture aims to integrate different services across the web while relying on simple type of RPC, namely the Simple Object Access Protocol (SOAP). Furtherly, there were proposed other paradigms in which all the communication between the services was meant to be done only through APIs calls due to the fact that they are lightweight, scaled more easily, more reliable and use the resources in a more efficient way. One of the main ability provided by such a system is agility, thus explaining the high interest of designing applications which comply with the Microservices architecture [1].

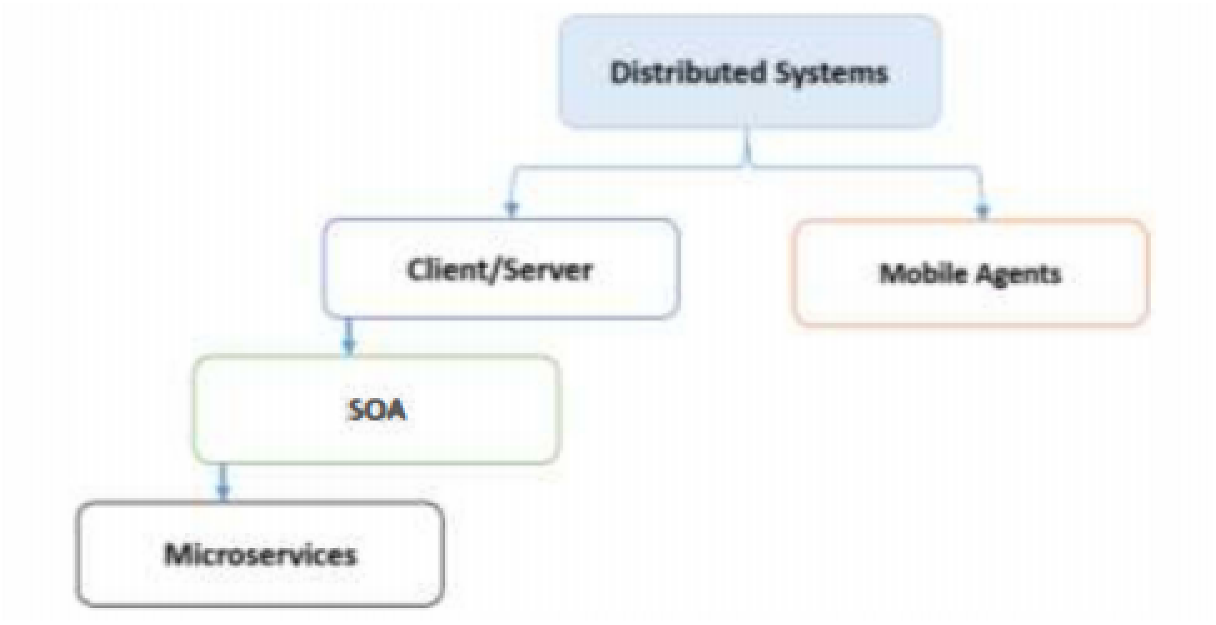


Figure 1.1: Evolution distributed systems [1]

In the following there will be roughly presented the evolution form basic software systems architecture such as Client-Server to Service Oriented Architecture and then to Microservice architecture.

1.1.2 Client-Server Paradigm

The Client-Server model is a highly used paradigm in the development distributed software systems. It usually describes the relation between two software programs, a client and a server which can be located on the same machine or split on different ones. Usually the client is the one requesting a resource, also known as the requester, while the server is the one providing the requested service or resource. Most often, this request-response communication mechanism is facilitated by RPCs or Representational State Transfer (REST). Those mechanisms came with a great impact on the model itself and helped it to evolve from a two tier architecture to multiple layer architectures [1]. The Client-Server architecture varies and depends on the number of layers it is built upon.

The two tier architecture in such a system usually consists of only the database server and a client. This client application, also known in this architecture as thick client, is responsible of connecting via the network to the database, do the computations required for the business logic and also present the result back to the end-user. This kind of design is to be considered in cases in which the client has to access the database directly without involving any other intermediary middleware. The Figure 1.2 clearly depicts this architecture.

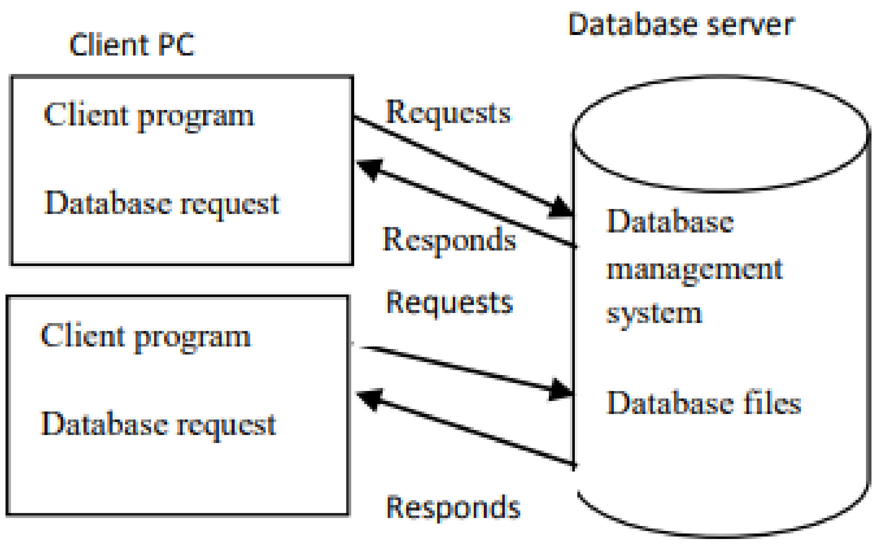


Figure 1.2: Two tier architecture [3]

The three tier architecture enhances the components required in a two layer by adding an application server which serves as a middleware between the databases server and the client.

The client becomes much lighter as it only contains the presentation logic while the business logic and the communication with the database is moved to the application server (the middleware) which runs on a separate machine. Moreover, the nth tier architecture only implies the existence of multiple application servers [3].

One of the biggest concerns of this model at its beginnings was the limited bandwidth. There were many models which tried to solve this issue, mainly by reducing the frequency and quantity of calls targeting the server. One of the models proposed the use of a so called 'eBUCS' protocol which basically suggest having a virtual database on the client, thus reducing the amount of calls made to the actual database server. This problem was later solved by the emergence of cloud based architectures which provided support for network resources.

Furtherly, despite the fact that this model can prove to be flexible and scalable it also comes with some flaws. The scalability is not that trivial as one needs a whole new server machine up and running in order to scale the system, thus the time required increases as well as the cost which is also influenced by the cost of hardware. Moreover, the maintenance is also quite difficult to be done as it is hard to split the duty to multiple people. Another drawback is the fact that this model is not able to integrate autonomous services. This is not the case in other architectures such as SOA [1].

1.1.3 Service Oriented Architecture (SOA)

The Service Oriented Architecture is by definition a design in which various services cooperate in order to provide some end functionality. Moreover, the services are usually different software programs which communicate to each other via network calls.

The SOA design is meant to be a solution for the problems raised by the large monolithic applications. The overall aim of the design is to make software easier to maintain and change as one could effortlessly change the underlying implementation of a service if this doesn't imply altering its interface. This results in software which is highly reusable since two or more applications could rely on the same service without having to know the implementation details encapsulated in it [23].

This architecture is considered to be the most successful evolution of the prior Client-Server model. A typical SOA structure can be seen in the Figure 1.3.

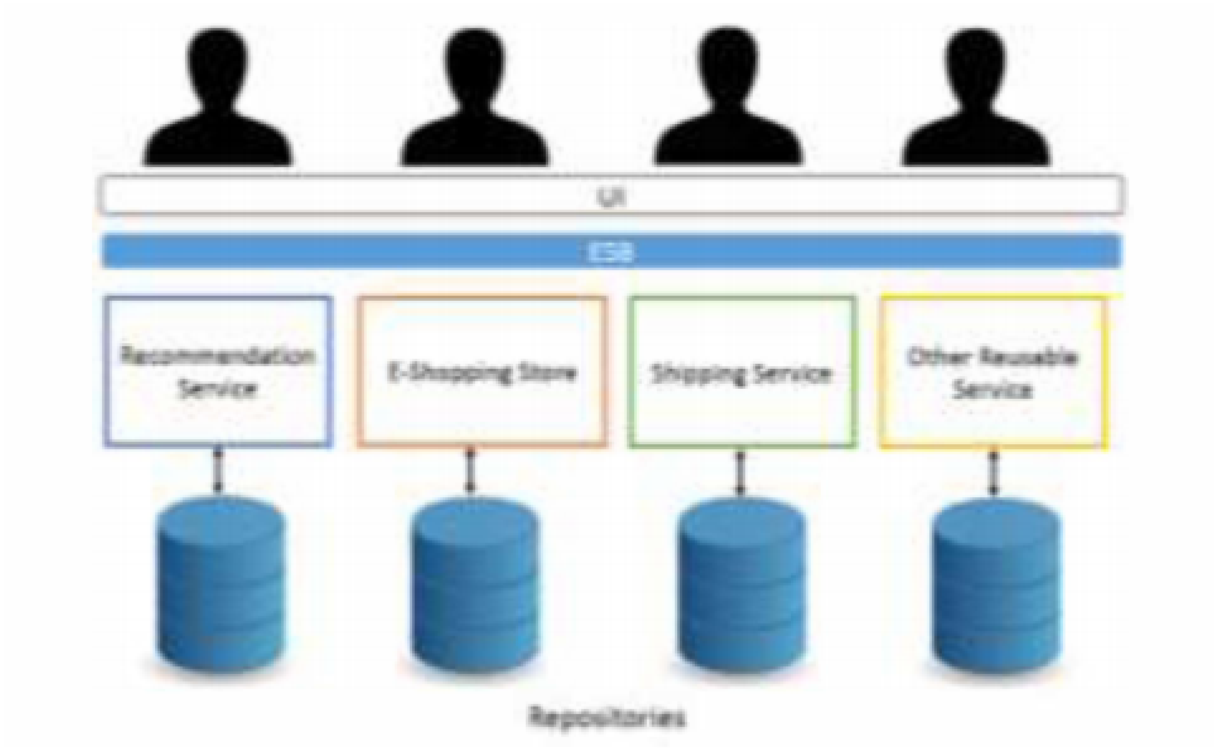


Figure 1.3: SOA basic structure [1]

In the presented figure one can straight forward see the basic elements which form the design, namely a set of services which are connected through an Enterprise Service Bus (ESB) and on top of which there is a user interface. Usually the Service Bus also comes with a mechanism which facilitate the registration of widely located services. When looking at the use cases, a trivial one would be something like this: a client would request a service through the user interface, the Service Bus would handle the request and translate it to the message type which suites the targeted service.

SOA is often used in enterprise applications which are rather huge. Those include applications like e-banking systems, e-shopping platforms or even social media platforms. This kind of applications usually should provide the capacity to support large amounts of continuous requests.

When looking at the flaws of this design, there was observed that the monolith representation of the services built within this architecture are not satisfying business expectations. This is related to the fact that the requirements for software development are moving towards building as much functionality as possible while also considering means of reducing the time and the resources used. One of the big drawbacks appears when trying to scale such a system as although there may be multiple services in such a monolithic architecture, they all have to be deployed at once. Hence, that in order to scale such a system once could create multiple instances of the system but the issue is that all of them would be identical. In this case, even if only one service needs to be scaled, a whole new instance will be created, thus wasting resources. This architecture would be more suitable for large applications which have large codebases, but obviously those kind of applications come with their common problems such as the increased difficulty for understanding or modifying the code base. Another straight forward problem of such systems is that one would have to redeploy the whole system even though only some small changes were made [1].

1.1.4 Microservices Architecture

Nowadays the software systems are changing at a greater peace than ever before. Often, requirements of such systems implies that the user experience should be more and more interactive and engaging for the end user. Moreover, the application functionality is expected to be enriched and modified often during the lifetime of a product in order to satisfy this dynamic user experience. When considering the points stated before, one can see that there is the need of an architecture which facilitates frequent updates and enrichment of software systems. Furthermore, this also suggests that older monolithic architectures are no longer suitable in this scenario. As a solution for those modern needs the proposed architecture is the Microservices Architecture [1].

This architectural approach is built around the idea of developing a software system as a group of small services which are independent of each other. Each of them would reside in a separate process while the communication between them would be done through lightweight mechanisms, in most cases by relying on Hyper Text Transfer Protocol (HTTP). Another important characteristic of such services is that they could be deployed and scaled independently. Moreover, when compared to monolithic architecture from a scalability point of view, the Microservices Architecture can scale in multiple dimensions while the other only in one [4].

The Figure 1.4 presents a way one could design the previously presented system, which was represented using SOA, with respect to the Microservices Architecture.

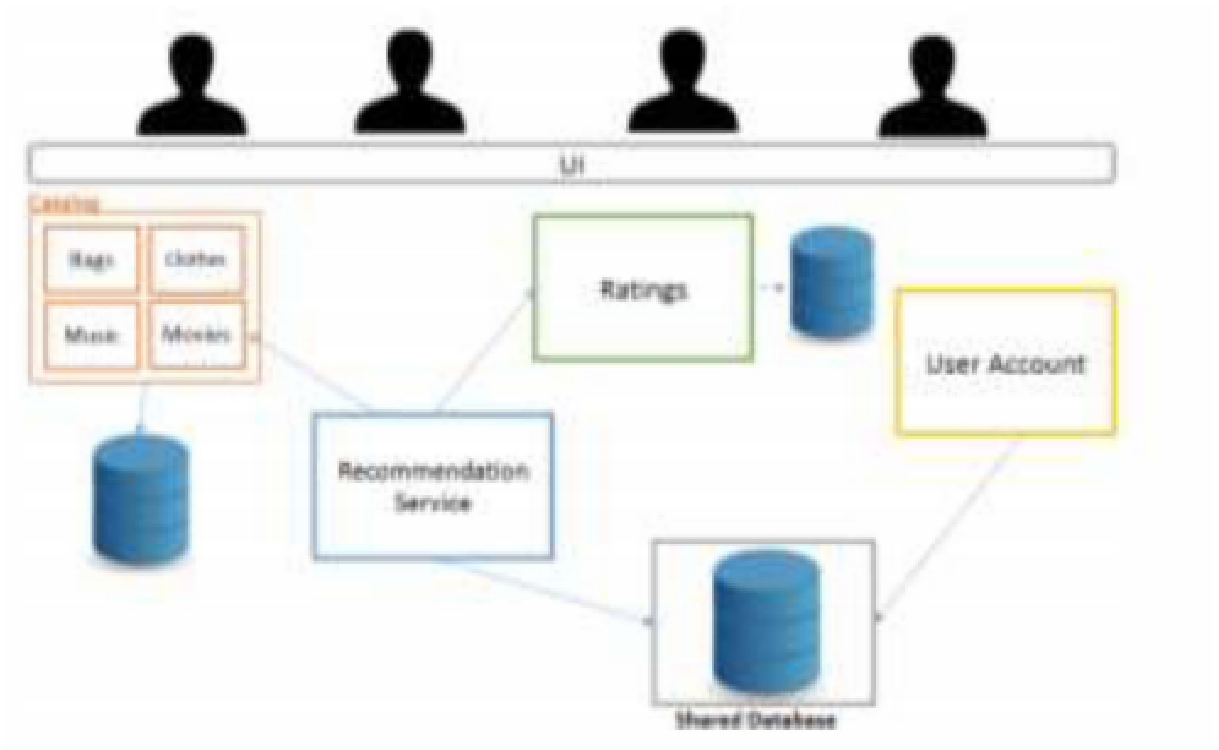


Figure 1.4: Microservices approach of the SOA system [1]

The figure depicts an online shopping system composed form four services namely the catalog, the users, the recommendations and the ratings service, designed as independent services. Any of those could be broken into even smaller independent services, thus enhancing the reusability of the software. As for an example in this manner, the recommendations service might need to interact only with movies service. When looking at the database, it can also be split into smaller independent databases, thus distributing the load which otherwise would be targeting only one database instance.

When compared to SOA, the Microservices Architecture is considered to be a way of building SOA or that SOA is just a condensed view of the microservices within a monolith. One of the differences between the two architectures is that in the Microservices Architecture the service bus is no longer present. The latter aims on keeping only dumb pipes for inter-services communication while also moving the control to the endpoints. By doing this the decoupling between the services is increased. Moreover, the services are also broken into smaller components [1].

1.2 Challenges of Microservices Architecture

Likewise any other architectural approach the Microservices Architecture also comes with its own flaws. Some of the most common drawbacks of the approach are the added complexity by the need of building a distributed system, the increased difficulty of testing such systems, the need of implementing and managing means of inter-service communication and also last but not least, a way of managing the distributed transactions. Furthermore, with this approach the deployment process will increase in complexity and will also require better coordination between the development teams.

One important step when building such a system is deciding on what is the best way of defining how the system break down into micro-services should happen.

The first straight forward way of doing that is by splitting the services with respect to the use case. Other approaches suggest the splitting of systems by verbs [5], an example service in this system would then be called Backup or Login, or the splitting by nouns.

One interesting way of splitting applications built with monolithic architecture is by using the scaling cube [6] as depicted in the Figure 1.5.

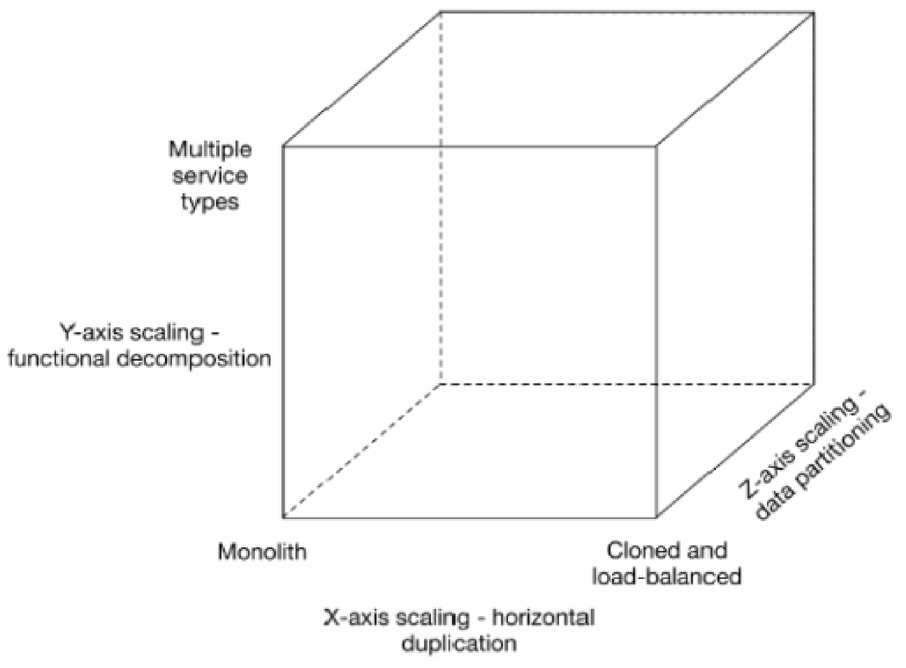

Multiple service types
Y-axis scaling - functional decomposition
Z-axis scaling - data partitioning
Monolith
Cloned and load-balanced
X-axis scaling - horizontal duplication

Figure 1.5: Scaling cube [4]

In this approach the horizontal scaling, which implies the creation of multiple similar server instances, all residing behind a load balancer, is represented on the X-axis. The Z-axis represents the splitting of thing which are similar. This means that even if each of the duplicated servers have and run the same code, they are only responsible for a subpart of the system's data, thus partitioning the data. Moreover, if one is to consider the case of fragmented databases, there is the need of an additional proxy component which is smart enough to forward the requests to the right server whose database contains the requested data. Moving on to another example for the Z-axis scaling, one could think of a system which offer services to both paying users and non-paying users. In this case the code which runs on the servers is the same but the server capacity is influenced by the amount paid. Together those two types of scaling aim to expand the capacity and the availability of the system.

The remaining Y-axis scaling comes as a rescue for the drawbacks generated by the other two means of scaling, namely the increased complexity of both the application itself as well as the increased difficulty of the development of such an infrastructure. The Y-axis scaling represents the decomposition of the system's functionality, in other words, splitting the things which does not make sense together. By following this scaling approach, one would break a monolith system into multiple services, each of them being responsible of only a piece of the overall system's functionality.

Independently of how one would choose to do the partitioning of the system, there is one principle that should govern over each service and it is the Single Responsibility Principle. This means that any service must have only a small number of responsibility in order for it to be more reusable [4].

1.3 Microservices, Components or Libraries?

Usually a software component is defined as piece of software which can be interchanged with another or can be enhanced independently. Libraries are just software components which are linked and used within an application. Services are components residing in different processes. The main difference between services and libraries is related to the way in which the communication is handled. When looking at the inter-services communication one can agree that it is mainly relying on RPCs, while the way of communicating with a library is through function calls which happen in memory. Hence, the services are closer to components than to libraries.

Furthermore, the deployment process within the microservices concept, which requires that each service should be deployed independently of the others, is another reason for the previous statement.

The other way around, when considering the implications of changing a library within a system, the obvious one is that the whole system requires a redeployment. Sadly, this kind of scenario could also happen in the case of services if for example the interface of one microservice is changed. In this case the changing of a service is no longer a trivial step as it also has implications to the other services which were relying on the previous interface. This highlights a key concept which should be considered when building microservices and that is to proper design the contracts of the services in order to embrace interface changes with ease. Without this proper design the system will eventually become hard if not impossible to maintain.

Additionally, when considering the microservices approach, the main strength is the physical isolation as this highly facilitates the scaling process [4].

In order to complete, here are some basic things which are usually required when building microservices [4]:

- Synchronous means of communication following the request-response pattern.
- Asynchronous means of communication through event patterns via various kind of underlying transports.
- A simple message format for serialization, examples mention JavaScript Object Notation (JSON), Extensible Markup Language (XML) and others.

1.4 Microservices Communication Patterns

1.4.1 Brief Description

Whenever the Microservices architecture is considered, everyone will agree that the network part of the overall system plays a key role, likewise in any other distributed system. Therefore in the following there will be presented some common communication approaches.

The most straight forward approach one could think of for performing inter-services communication is via direct calls as it can be seen in the Figure 1.6 [4].

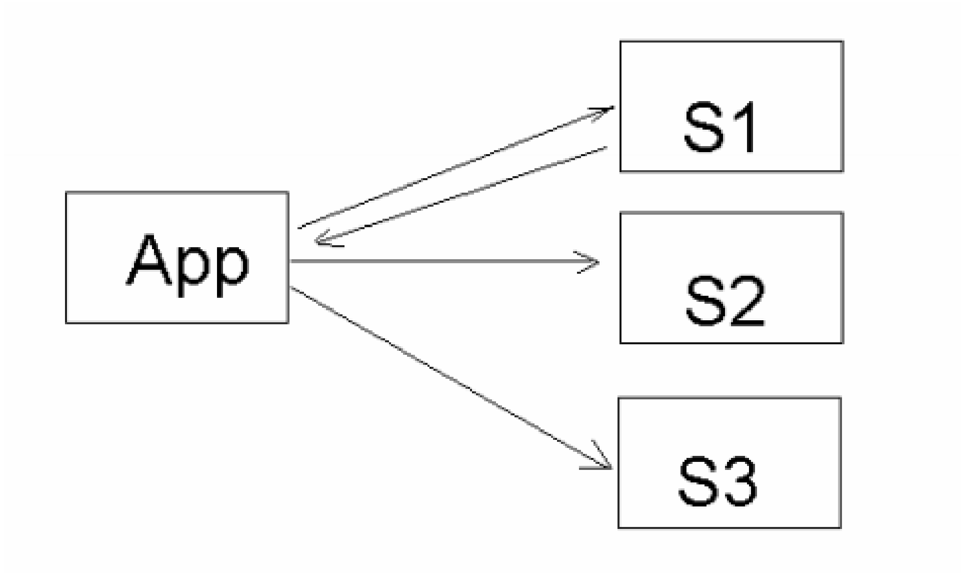


Figure 1.6: Direct calls [4]

There is no secret that this approach provides the most flexibility in such a system. As an example, one could think of an isolated web server which is able to call multiple backend services in order to render content for the end-user. However, this approach can also cause problems in terms of latency if the remote service calls have an increased delay. Thus, this leads to the next communication pattern, one which aims to solve this problem and decrease the number of required remote calls.

This approach implies the use of an additional component within the system, one which should serve as a common point for all the traffic targeting backend services. This extra component is usually referred to as gateway [4]. The Figure 1.7 depicts a basic system structured by following this service composition pattern. More about the API-Gateway pattern will be discussed in the following subsection.

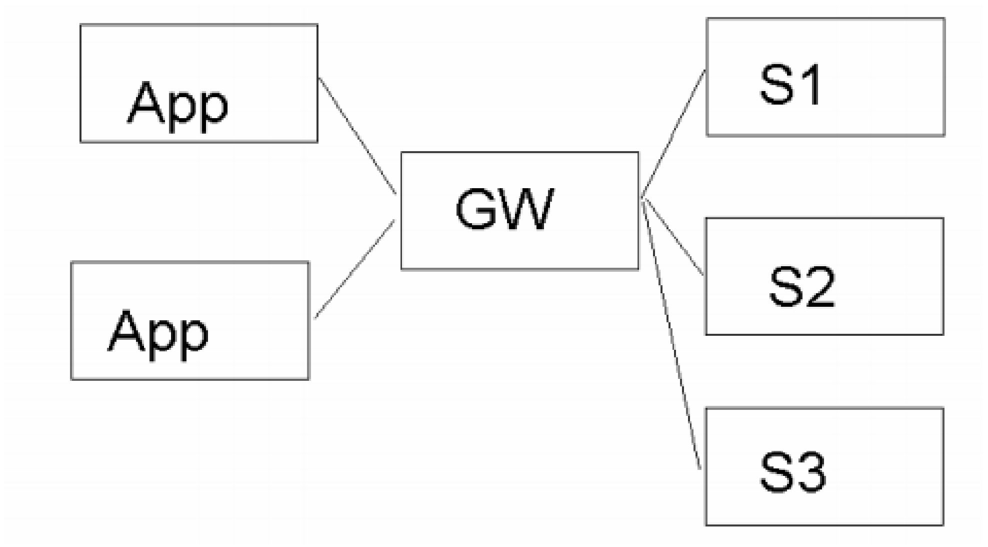


Figure 1.7: Microservices Gateway [4]

Another highly used communication pattern in the Microservices architecture is one which heavily relies on some message bus. This pattern is most convenient to be used in applications in which the inter-services communication is rather asynchronous. This means that the requests can be sent while the response can be received at a later time, so no need of immediate responses. The Figure 1.8 presents a basic representation of such a system.

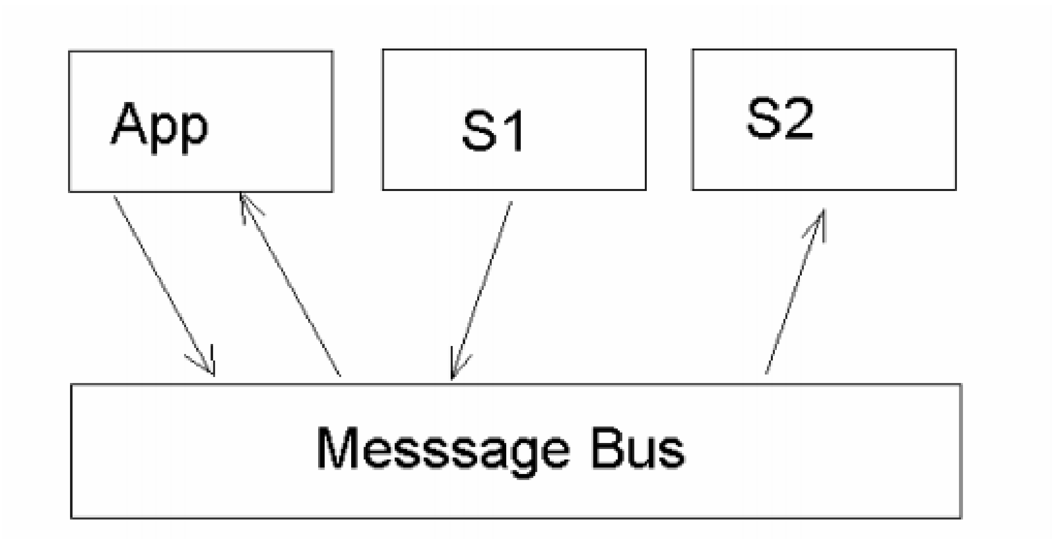


Figure 1.8: Message Bus pattern[4]

The reasons for using this communication pattern which relies on publish-subscribe model are quite obvious and mainly refer to extensibility, scalability and the way in which it facilitates the process of deployment. Therefore, the process of adding new components to the system is simplified, as it does not imply any changes to the already existing components. Deployment of new functionality is also independent of the deployment of the other components and the service bus itself can use load balancing, thus improving the scalability of the system [4].

1.4.2 The API-Gateway Pattern

The concept behind this service composition pattern is built upon the idea that microservices should expose their functionality, so it can be used within other services, through an Application Programming Interface (API). Moreover, this idea of aggregating a group of services in order to be used by an end-user application implies the need of a new backend mechanism to do this job, namely the API-Gateway.

The API-Gateway component represent the entrance point in such a system. It is usually responsible of forwarding the requests coming from the end-user application to the desired microservice and of aggregating the results when the business logic requires it. Furthermore, it can also provide dedicated APIs for each and every client, thus providing an easier integration. Other common tasks handled by an API-Gateway could be the implementation of authentication and authorization mechanisms, the provisioning of different kinds of applications insights, the provisioning of means for communication between applications which use dissimilar protocols for communication or the implementation of strategies for limiting the network traffic for protection against basic brute force attacks. In addition, when looking from a scalability point of view, the gateway component can also leverage its knowledge about the underneath services location and act as a load balancer.

The main objective of this approach is to facilitate the system interactions and enhance the efficiency by decreasing the necessary number of requests coming from end-user applications. In addition, the gateway component can also facilitate the communication between the system and other platforms by exposing tailored APIs for each of them and also by acting as a façade and hiding the underneath microservice interfaces. As a result of this, both the client applications and the microservices can be developed independently by relying on the low coupling between the two. This means that splitting or merging any of the underneath microservices does only imply changes at the gateway level.

In the Figure 1.9 the gateway acts as the central point for the traffic between a couple of client applications and the microservices system. Because of the gateway component's design, which exposes tailored APIs for each of the two client platforms, the logic from within the two client applications is reduced by not having to deal with redundant information as it would have otherwise [8].

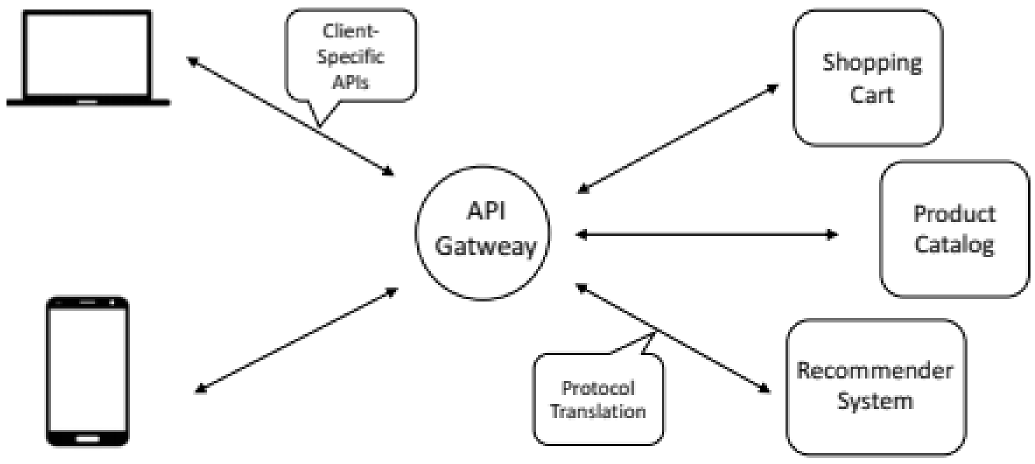


Figure 1.9: The API-Gateway pattern [8]

Furtherly there will be presented some of the strengths for the presented approach:

- Developing new functionality within the system is easier with this approach and this is mainly due to the fact that the API-Gateway pattern encourages the provisioning of customized APIs [10].
- The approach facilitates the process of changing or enhancing the system, as a response for business change requests, as it provides low coupling, thus any microservice can be modified without interfering with the rest of the system [10].
- The gateway component assures backward compatibility for its clients which rely on the provided custom tailored APIs [13].

Yet there were also noticed some flaws for the approach:

- If the gateway component is not designed properly, it can act as a bottleneck for the entire system as all the communication traffic flows through it [15].

- The creation of custom tailored API interfaces within the gateway component can make it more complex and harder to maintain [10].
- The gateway maintainability can be also hampered in cases in which different types of client applications rely on the same API, since those may require custom changes in case of gateway API level changes [10].
- The API-Gateway approach is no longer a good choice from a scalability point of view when the system is composed of a large number of microservices. In those cases, routing all the communication traffic through one gateway is no longer efficient [14].

1.5 Individual Microservice Architecture

Whenever implementing a service in a Microservices architecture there are certain guidelines that must be followed in order to fully leverage the advantages and the benefits of following this architecture. The next subsections aim to briefly present some key concepts regarding the domain architecture of microservices followed by the presentation of a more granular perspective of the Microservices architecture by discussing a proposed architecture for building the individual service in such a system, namely the hexagonal architecture.

1.5.1 Key Points of Microservices Domain Architecture

In a Microservices architecture, the guidelines regarding the functionality of the domain are provided by the domain architecture. Usually when building a system which follows the Microservices architecture it is recommended to let the design logic of individual microservices to be determined independently rather than forcing it to follow a chosen pattern. This decision diminishes the dependency between different development teams working on different microservices. However, there should also be a collection of lightweight rules and good practices which one must follow when building a microservice so that key points like maintainability, adaptability or ease of replacement are considered.

Furtherly, there is some guidance on how to determine possible flaws of the domain design for a microservice.

One first thing that should be considered when choosing the domain architecture for a microservice is the cohesion within the service. Sometimes the global domain architecture within the system has a great influence to one of an individual service. However, there is a base rule which should govern over a suite of microservices and it states that the system must be built in such a way as to decrease the coupling between different microservices while assuring the high cohesion within a microservice. Thus, the idea of single responsibility also apply to the domain architecture of a microservice which should assure that it only cares about one piece of the domain. When noticing low cohesion within a microservice, breaking it into multiple microservices must be considered. By doing this, the previously presented attributes of a system, namely the maintainability, adaptability and the ease of replacement will be enhanced.

Another point worth considering when designing domain architecture of a microservice is encapsulation. This means that the internal representation of the data should not be available directly to any other external service. The recommended approach, which makes it easier to change and therefore to develop the internal domain structures of a microservice, is to only allow access via interfaces. By doing this, the internal domain changes will no longer be reflected to other services, thus the only thing remaining to be understood by the other services are the external interfaces. Moreover, this also have a positive impact by enhancing the low coupling between the microservices within the overall system.

Last but not least another other thing that should be taken into account but which is rather hard to implement in a distributed system are the transactions. A transaction represents a set of operations which either must be all executed successfully or otherwise none of them should be executed, thus preserving the data consistency. Although the implementation of transactions which extend over multiple services is rather hard, it can be achieved to some extent by making use of the message based communication between the services. Furthermore, in order to avoid the involvement of multiple services in a transaction, the domain architecture should assure that each operation exposed through an external interface runs within only one transaction [24].

1.5.2 Hexagonal Architecture (Ports and Adapters)

The Hexagonal architecture mainly concentrates on the business logic and functionality of the application. The name of hexagonal comes from the multiple interfaces which compose a system built by following this architectural pattern and which are depicted as edges of the hexagon. In the Figure 1.10, such interfaces are the one which exposes the functionality for the users and the one exposing functionality for the admin.

Those interfaces could be used through different kind of adapters, for example one might access the user functionality via a HTTP adapter. One other example are the tests which access the functionality through a dedicated adapter. Moreover, there is another adapter which provides the means of accessing functionality through REST, thus enabling the inter-services communication.

In this architectural pattern, the external interfaces have beside the role of receiving and handling requests from external services also the role of communicating to other services or external systems. As an example, the communication with the database is facilitated by a dedicated database adapter, the communication with some other external microservice is done through a dedicated REST adapter and the same applies to the test.

Furthermore, the Hexagonal architecture is also known by the name of ports and adaptors. This is due to the fact that each side of the system like the ones presented in the figure: the user, the admin, the events or the data are considered to be ports. Above them, the adapters provide implementation for the ports by the means of lightweight communication mechanisms such as REST.

Often, the graphical representation of this pattern depicts the incoming ports on the left side of the hexagon while the outgoing ports are represented on the right side. The incoming ports are responsible of processing requests and exposing the microservice's functionality to other external services or systems while the outgoing ports are intended to reach out to other systems and fetch data [24].

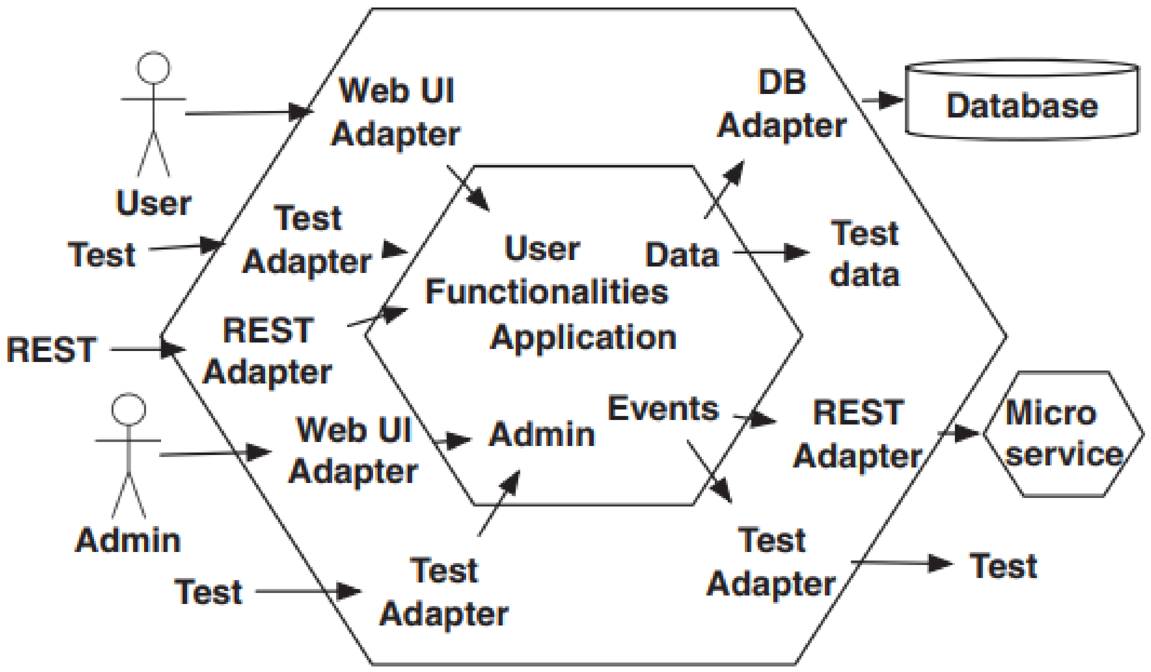


Figure 1.10: The Hexagonal Architecture pattern [24]

The base concept of the Hexagonal architecture is to encapsulate the core business logic in a sort of a kernel while only providing means of communication to external systems via dedicated adapters. It comes as an approach for extending the old layered architecture. In the traditional layered architecture there would be a presentation layer responsible for the user interface, a persistence layer which should facilitate the communication with the database and the business logic would reside somewhere in between the two, usually in a dedicated layer.

When looking at the hexagonal architecture one can observe that communication is handled through the adaptors which access the core logic via ports. In comparison with the layered architecture, in the hexagonal architecture there are much more ports exposed than just a persistence and a presentation layer. Moreover, the ports are meant to be independent of the specific communication protocols and implementation details of the adapters.

Sometimes the microservices built by following the hexagonal architecture do more than just to provide functionality to other systems through REST interfaces or other messaging interfaces. They can also be built to provide functionality directly to the user by the means of a web user interface. This represents a fundamental concept of the microservices, namely that they should do more than just to communicate to other services or systems, thus they might also facilitate user interactions via some kind of a user interface.

An advantage of the Hexagonal architecture is the fact that it facilitates the process of testing a microservice in isolation. This happens because the architecture provides the possibility of developing particular tests for each of the ports in which dedicated test adapters can be used. As microservices should be developed, tested and deployed as individual entities of the overall system, the possibility of easily testing each of those microserivices in isolation represents a big step forward.

Furthermore, there are more aspects that can be considered when building a microservice with respect to the Hexagonal architecture. One such aspect might be the development of the load balancing logic or of the resilience logic at the level of the adapter. Another aspect might be the breaking down of adapters and underlying logic into separate microservices. However, this also brings up on the plate all the complexity of a broader and sparser distributed system, but on the other hand the split would make it possible for multiple development team to work in parallel on the core logic and the adapter itself. As an example, one could imagine a team which implements a dedicated adapter which fits the needs regarding bandwidth consumption of their mobile client.

As an example for implementing a microservice with respect to the Hexagonal architecture, one could examine the order microservice depicted in the Figure 1.11.

The presented microservice provides three adapters, namely the web user interface adapter, the REST adapter and the test adapter, through which one could access the user functionality. The user interface adapter is meant to provide to the user direct access to the functionality, while the REST adapter serves as a mean of communication with other external microservices or systems. The two adapters represent just two alternative ways one could have access to the same functionality and are meant to be used by microservices which support user interface and REST integration.

The order events represents another interface of the microservice which is responsible of communicating to another microservice, named suggestively as delivery, whenever there is a new order to be delivered. However, the interface is also used on the other way around by the delivery microservice to communicate back to the order microserice whenever a delivery was successful. Therefore, the order events interface facilitates the bidirectional communication between the two services by not just providing the means of posting delivery data to the delivery microservice but also by providing the means to modify orders. Moreover, there is also a dedicated test adapter which aims to facilitate testing of the interface [24].

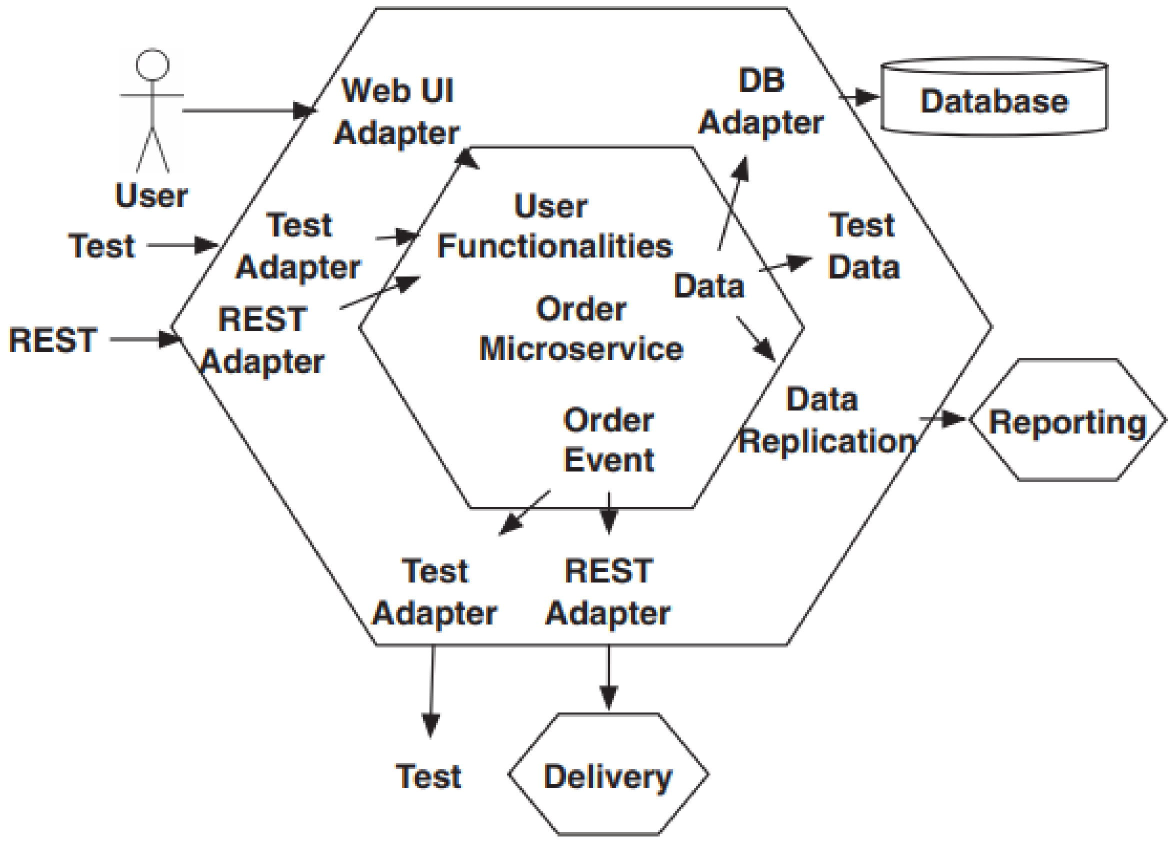


Figure 1.11: Hexagonal architecture example [24]

Another interface which is present in the system is the database interface. It is responsible of persisting the incoming orders into the database. It also comes together with its own dedicated database adapter and a test adapter which aims to ease testing. If looking from a layered architecture perspective, this interface would represents the persistence layer of the system.

The microservice also provides an interface for an extranal reporting microservice. Its responsibility is to send data regarding the orders in order for statistics and reports to be created, thus delegating the job of creating reports and statistics to a dedicated microservice.

The functionality of the example microservice is split with respect to parts of the domain into two interfaces, the users and the order event. This emphasize the base design of the architecture, namely the design based on the domain. Hence, the Hexagonal architecture facilitates the breaking down of functionality into clear-cut interfaces based on domain entities. This provides the possibility of splitting the interfaces and adapters into separate microservices whenever this brings an advantage [24].

Chapter 2

Application Description

2.1 Problem Statement

There is no doubt that nowadays the educational system is moving towards distance education or online education at a greater peace than ever before. This comes as a consequence of the new Coronavirus Disease (COVID-19) outbreak which highly contributed in speeding up the transition from the traditional way of teaching and learning, which required the physical presence of students and teachers, to a new way in which the internet and therefore the online educational platforms play a key role.

Many educators were surprised by this fast transition and so the teaching became quite challenging, especially for those living in countries in which the technology use is not yet widespread or where the educators were not previously required to have some basic digital knowledge [16].

Furthermore, besides the online courses and lessons there is yet another challenge to be tackled in this new environment, namely the online examination of the students. There are many systems out on the internet which were designed for online learning and testing. Often, such systems are design to handle the examination of quizzes composed from questions with simple answers which can be easily evaluated online.

With this in mind, I thought of a solution for the problem of online evaluation by means of a distributed web application which should facilitate the evaluation process. The platform aim to help students by providing easy access to their exams while also by making their interaction with the platform as intuitive as possible by presenting a minimalistic and user friendly interface.

Moreover, the platform also targets the educators and tries to make their job, which was already hampered by the fast changing environment, easier by providing them support in the process of formulating and gathering reusable exam questions and answers, in the process of building appropriate exams and also in the process of monitoring the exam results in real time.

After successfully logging in as a teacher, the application will reveal the teacher's main page by the means of a dashboard which comprises the main functionality in multiple sections.

One section of the teacher's dashboard provides the functionality of building different types of questions. The general structure of a question contains the main question or the requirement text, a short description which should instruct the students regarding the requirements and an arbitrarily chosen category which might be used later in order to facilitate the questions search. Regarding the answers of a questions, those can be added dynamically with respect to the type of a question, which currently can be one of single choice, multiple choice or free text.

The following section of the dashboard is the questions catalog. This sections provides to the teacher the possibility of freely surfing through questions by the means of an intuitive collapsible list which can also provide question details and present which is the correct answer or answers of a specific question.

The dashboard also provides a dedicated section which facilitates the exams building. The header structure of an exam contain a title, a description and the actual duration in minutes or hours, which can be easily selected through a really accessible slider component. The main part of the exam are obviously the questions or the tasks which compose it, thus this section also provides support for selecting the appropriate questions from the catalog while also providing quick links to their details. The questions which make it into an exam must be assigned a percentage weight which represents their value within the exam.

There is also a dedicated section for the exam catalog of a teacher which provides insights about the exams and the questions composing them.

Last but not least, the platform also provides, within the dashboard, a page with live statistics about the published exams. There the teacher can see in real time the submission of exam answers as well as answer statistics for each individual question of an exam.

Furthermore, the platform also provides the necessary means for students to take exams. By using their personal accounts, they are able to see their list of exams and take exams in the available time interval.

When looking from a more technical point of view, the platform aims to improve scalability and extensibility of the overall system by leveraging the strengths provided by the Microservices architecture. Moreover, as matters of orchestration and coordination, the system pursue the API-Gateway pattern [8] which implies an aggregation mechanism for composing multiple microservices by the means of a gateway. At a more granular level, each microservice is built by following the Hexagonal architecture [24] while regarding databases, in order to decrease the load of a single database, the system embraces the fragmented databases concept [24].

2.2 Implementation and Design

2.2.1 Application Architecture and Design

In Figure 2.1 there can be seen the system diagram which presents the client and the suit of microservices which compose the backend while also briefly depicting the protocols used for communication. Further, each microservice will be presented in more details.

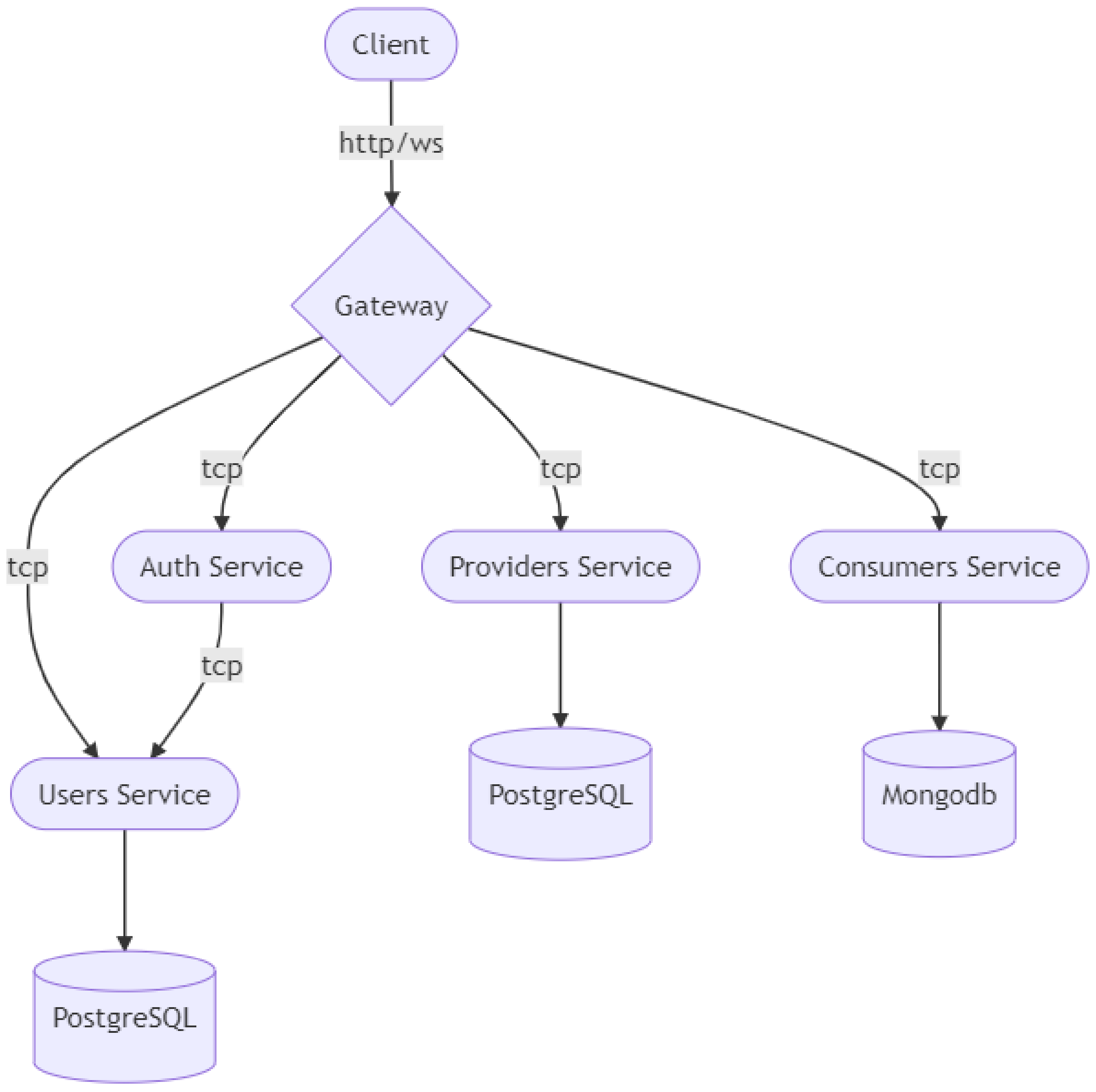


Figure 2.1: The system diagram

The **backend** of the application is built with respect to the Microservices architecture, more specifically by following the API-Gateway pattern. It is composed from five independent microservices, namely the Users Service, the Auth Service, the Providers Service, the Consumers Service and the Gateway. Likewise in an API-Gateway orchestration pattern, all the system traffic is forwarded through the Gateway which directly communicates to a client, which is running on a separate web server, via two dedicated ports and adapters, one which is responsible of handling the HTTP traffic while the other responsible of handling the web socket [18] traffic. Moreover, all the microservices are design with respect to the Hexagonal architecture while regarding the framework and the programming language used they are written in NestJs [7] which is a very well suited node [19] framework for building microservices.

The **Users Service** is one of the simplest microservices of the system and it is responsible of handling the user functionality as its name also suggests, thus it strictly maps one domain part. The microservice contains two interfaces and their dedicated adapters out of which one is an incoming port which manages the messages coming from the Auth Service while the other is an outgoing port which is responsible of communicating with a relational database. The service has a thin kernel which does some mappings and delegates the requests from one adapter to the other. Functionality wise, the microservice provide the possibility of fetching and persisting user accounts into the associated database. As matters of communication, the incoming port exposes a messaging interface which listen and respond to incoming messages via a request response pattern, thus this provides the possibility for other microservices to perform synchronous calls in order to access the user functionality.

The **Auth Service** is responsible of both the process of authenticating a user whenever he tries to login and the process of checking authorization of a user whenever the resource he tries to access is marked as protected or when the user tries to establish a web-socket connection. The microservice is composed from only one module which exposes two ports, one incoming port which provides the possibility of synchronously calling the login and the authorize methods in a request-response fashion and one outgoing port which handles the communication with the Users Service. Regarding the core functionality, it is encapsulated by the ports and handles the login logic, meaning the validation and verification of the received credentials which should match the ones saved in the Users Service's database, and the generation of a Json Web Token (JWT) [20]. Moreover, the core also handles the authorization logic by checking the validity of the provided JWT token and the required roles. Furthermore, any kind of exception raised within the Auth Service is wrapped into an RPC exception and shipped back to the caller.

The **Providers Service** contains the base logic for the exams and questions. It is divided into three modules, one shared module which is responsible of establishing the connection to the other systems, namely the Gateway microservice and the database, another which is responsible of the questions logic and the third which is responsible of the exams logic. As depicted in the Figure 2.2 the microservice is composed of five ports out of which two are incoming ports while three of them are outgoing ports. The incoming ports are the Exams and Questions which are able to handle via messages communication both the request response pattern for the synchronous communication and the event based pattern for asynchronous communication.

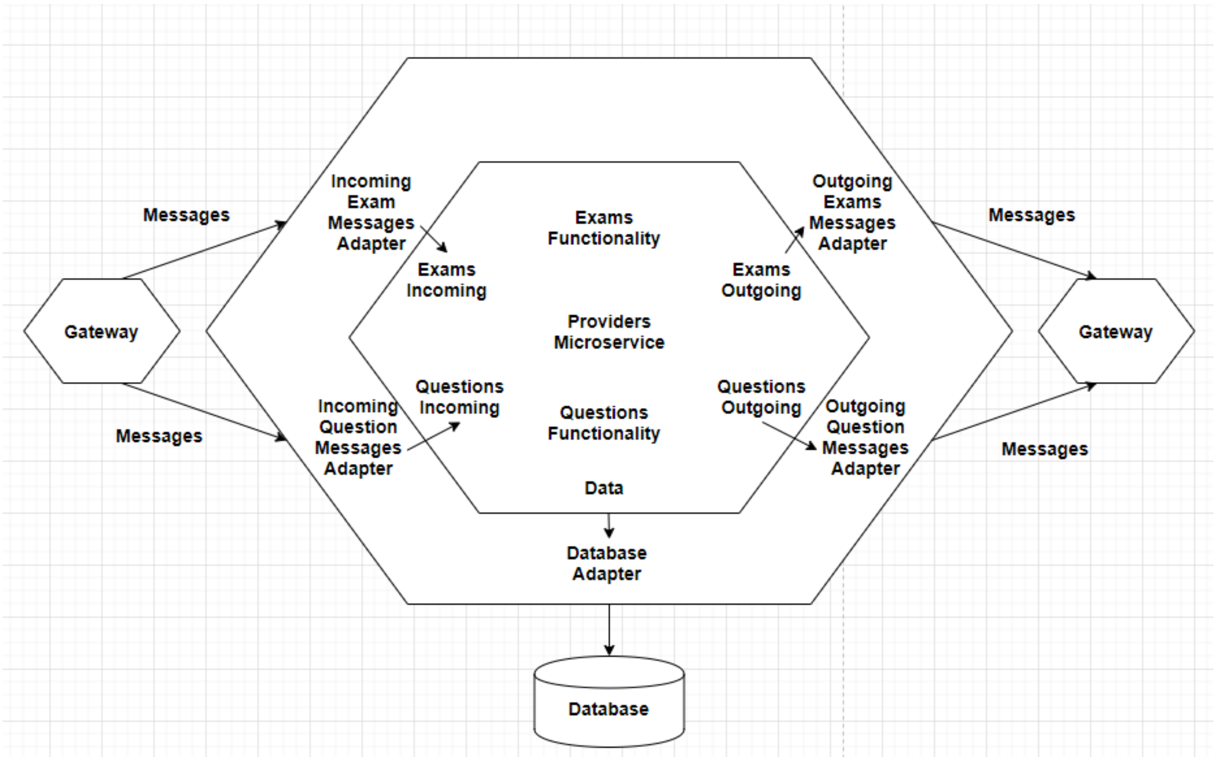


Figure 2.2: The Providers microservice

The synchronous communication is mainly used for actions which requires an immediate response, such as a fetch of the questions or exams for a user while the asynchronous communication is used for triggering the process of an exam creation or a question creation, which are actions that does not require an instant response with the newly created entities. The response for the asynchronous processes will then be later sent asynchronously via an outgoing port to the Gateway service which will then forward it to the client via a web-socket channel.

Regarding the outgoing ports, one of them handles the communication to the database for persisting data while the other two handles the asynchronous communication between the Providers Service and the Gateway.

The **Consumers Service** contains the logic regarding those who consume the provided exams, so it provides the functionality for the exam answers. The microservice is composed of three ports out of which one is responsible of handling both asynchronous events for persisting and processing the submission of an exam answer and synchronous events for fetching data about those answers. Another port is responsible of facilitating the communication with the database while the last one is responsible for emitting asynchronous events targeting the Gateway for providing real time data for the exam answers insights.

Last but not least, the **Gateway microservice** acts as a common point for all the traffic flowing from the clients towards different microservices and vice versa. As it can be seen in the Figure 2.3 it is also built with respect to the Ports and Adapters architecture.

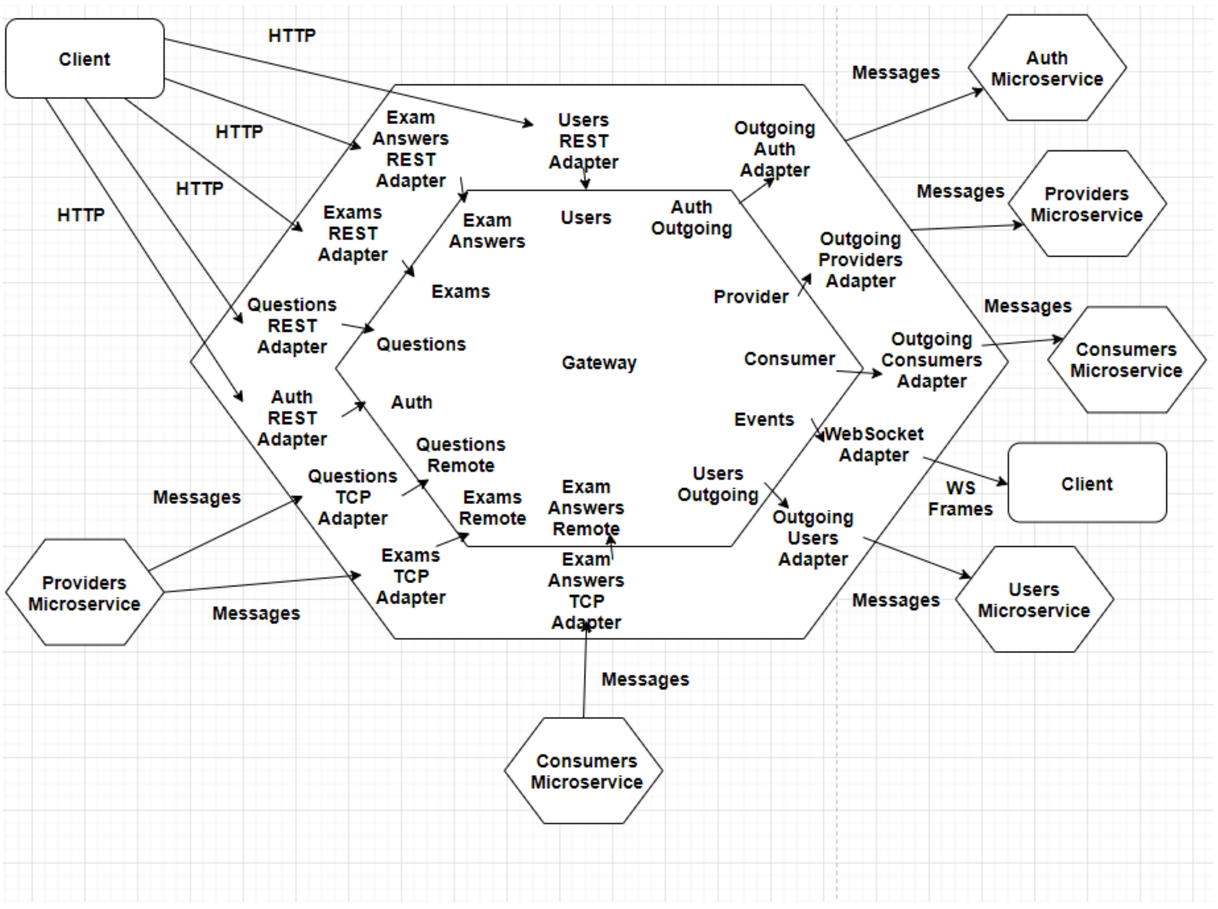


Figure 2.3: The System's Gateway

The Gateway is responsible of orchestrating the communication throughout the system. It exposes multiple incoming and outgoing interfaces with dedicated adapters for handling both the requests coming from the client and the messages coming from the other microservices from within the system. In more details, there are five incoming ports responsible of handling the HTTP requests coming from the client. There are the Exams and Questions ports which provide access to the respective functionality and have the job of forwarding the requests through the core and then delegate to the required microservice via dedicated outgoing ports. Another such example is the Exam Answers port which handles a similar flow but whose target in the end is the Consumer outgoing port and thus the Consumers Microservice. The same applies also for the Users incoming port. The Auth incoming port is responsible of handling the login flow by delegating the request to the Auth Microservice.

There are two more incoming ports, the Questions Remote and the Exams Remote, which listen for asynchronous messages coming from the Providers Microservice. Their job is to catch the messages and forward them though the core and then furtherly through the Events outgoing port in order to be sent via a web socket communication channel to the client, thus providing real time updates. The same applies for the Exam Answers Remote, the only thing different in this case is that the source for the messages is the Consumers Microservices. Those messages assures the real time updates of the client graphics and exam answers insights.

Regarding the outgoing ports, they are each handling the communication with one of the microserivces within the system. As for an example, the Provider port communicates via the Outgoing Provider Adapter to the Providers microservice both in an asynchronous and a synchronous manner. The same applies for the Auth Outgoing, the Consumer and the Users Outgoing ports.

From the authentication and authorization point of view, the Gateway delegates this responsibility to the Auth microservice. However, the Gateway is still responsible of setting up HTTP only cookies [21] whenever the authentication process is successful. The mechanism of HTTP only cookies was chosen in order to not provide the possibility of the generated JWT token to be read on the client side. This decision made the token not accessible through JavaScript code but also introduced two new problems, namely the client can no longer decode the token and thus the user details are no longer available on the client and the system is now more vulnerable to CSRF [22] attacks. However, because those problems are common, there are quite simple fixes for them, thus the problem of having user information on the client side can be easily fixed by providing a dedicated protected endpoint which provides account information based on the currently authenticated user. The solution for removing the threat of a CSRF attack was to also provide a generated CSRF token in the header for each modifying HTTP request (PUT, PATCH, POST, DELETE).

Regarding the authorization for the establishment of the web socket connection, the gateway, which also encapsulate a web socket server, also delegates this responsibility to the Auth microservice. The authorization process takes place in this case at the moment of the handshake when the process of protocol switching is happening.

The **client application** for the system is built as an Angular [9] application which is composed from four modules, as depicted in the Figure 2.4, and runs on a dedicated web server. It provides lazy loading for its modules and is built with respect to the Flow architecture (Redux [11]), facilitated by the Angular counterpart NgRx [25].

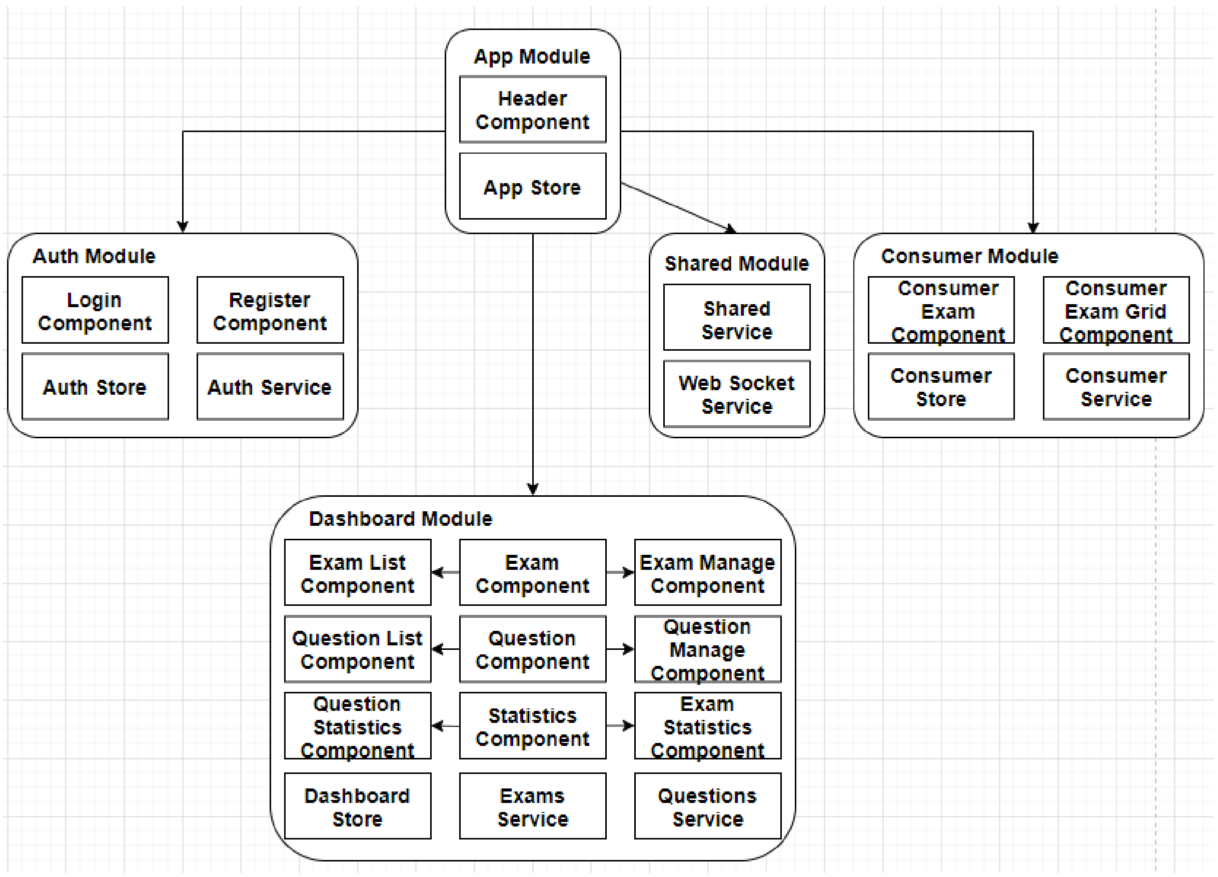


Figure 2.4: The Structure of the Client Application

One of the modules is the Auth module which handles all the logic regarding the login and register functionality. Another module is the Dashboard module which is responsible of all the logic regarding the teacher related functionality, namely adding and surfing through questions, creating and publishing exams and displaying real time insights about the exams submissions through some graphics. There is also the module responsible for the student functionality, the consumer module, which provides means for finding and taking exams. Furthermore, the forth module of the client application is a shared module which mainly handles the communication with the web socket while also exposing the common functionality and dependencies for all the other modules.

Regarding the Redux store, it is designed with respect to the modules of the client application. This means that there is a dedicated state and reducer for each of the three main modules, namely the Dashboard module, the Consumer module and the Auth module. In order to handle asynchronous calls the NgRx provides the concept of effects or side effects which via the help of RxJs [26] operators can behave like reducer cases with the difference that they are not synchronous. Moreover, those effects can be well integrated in the normal Redux flow because they can also dispatch one or more actions further to be processed by other effects or reducers. The client application also implements such an effect for each of the main modules.

Furthermore, the client application routes are protected via authentication and authorization guards which only allow access if the set of defined preconditions are met. This means that for example the routes which should be available only for teachers are both restricted on the frontend level and the backend level, providing an enhanced security and better user experience.

2.2.2 Use Cases

The following Figure 2.5 presents the main use cases of the application comprising both the functionality for the teacher and the student.

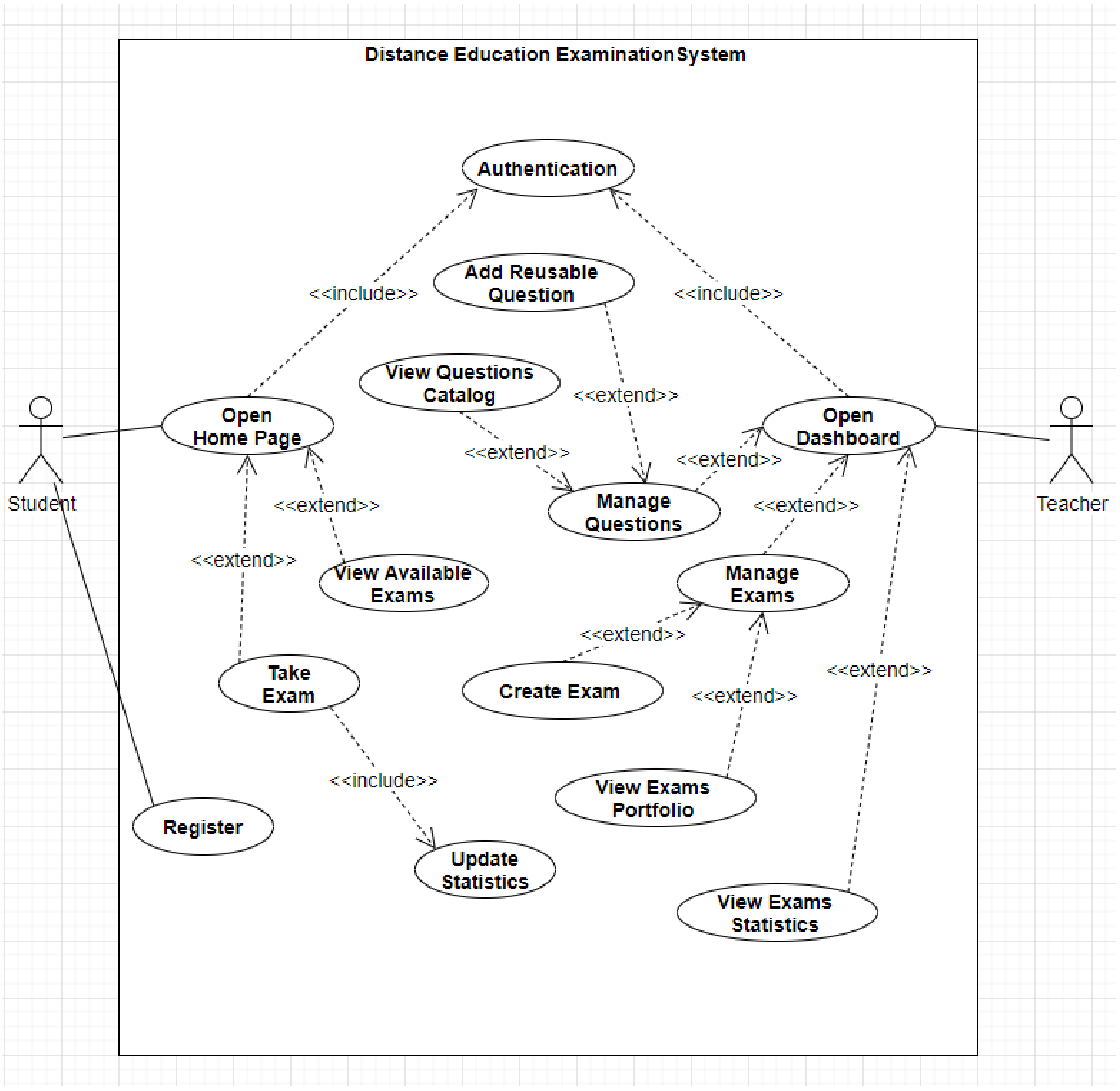


Figure 2.5: Use case diagram for main functionality of the system

Furtherly, some of the depicted use cases will be presented in more details, thus a short description, the pre-conditions and post-conditions required and the main flow will be described for each presented use case.

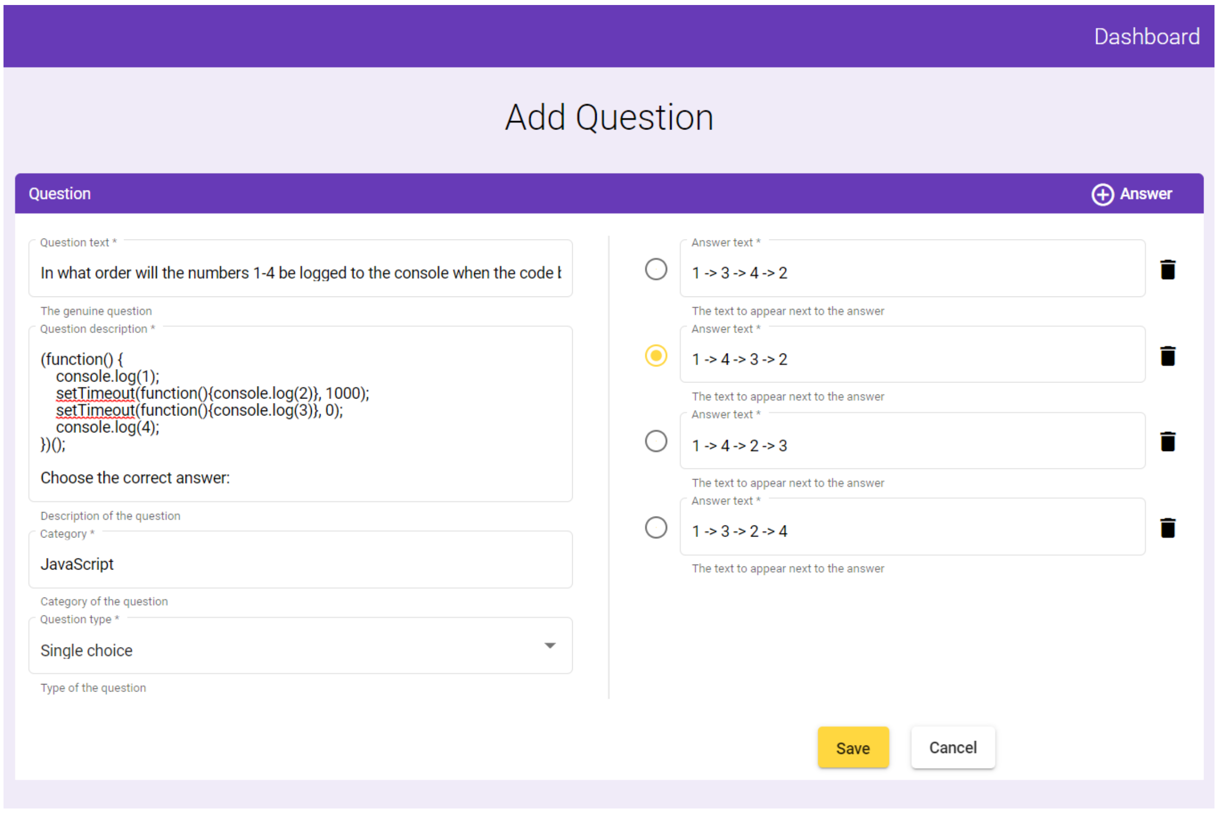
Use Case: Add Reusable Question

Actor: Teacher

Short Description: A teacher wants to add a new single choice question to his questions catalog in order to be later able to use it for creating an exam.

Precondition: To be logged in the application with a valid teacher account and have a token which is not expired.

Postcondition: The teacher can see the newly added question in his questions catalog. Moreover, the question is now available to be selected when creating a new exam.



Question text *
In what order will the numbers 1-4 be logged to the console when the code b
The genuine question

Question description *

```
(function() {
   console.log(1);
   setTimeout(function(){console.log(2)}, 1000);
   setTimeout(function(){console.log(3)}, 0);
   console.log(4);
})();
```

Choose the correct answer:

Description of the question

Category *
JavaScript
Category of the question

Question type *
Single choice
Type of the question

Answer text *
1 -> 3 -> 4 -> 2
The text to appear next to the answer

Answer text *
[✓ 1 -> 4 -> 3 -> 2]
The text to appear next to the answer

Answer text *
1 -> 4 -> 2 -> 3
The text to appear next to the answer

Answer text *
1 -> 3 -> 2 -> 4
The text to appear next to the answer

Figure 2.6: Adding a reusable question

Main Path:

1. Login into his teacher account.
2. Open the Add Question section from the dashboard's side menu.
3. Fill in the required fields: the question text, the question description and the question category.

4. Select the type of the question which can be one of single choice, multiple choice or free text.

5. Add on the right column an arbitrary number of answers and selecting the right one in case of a single choice question or the right ones in case of a multiple choice question.

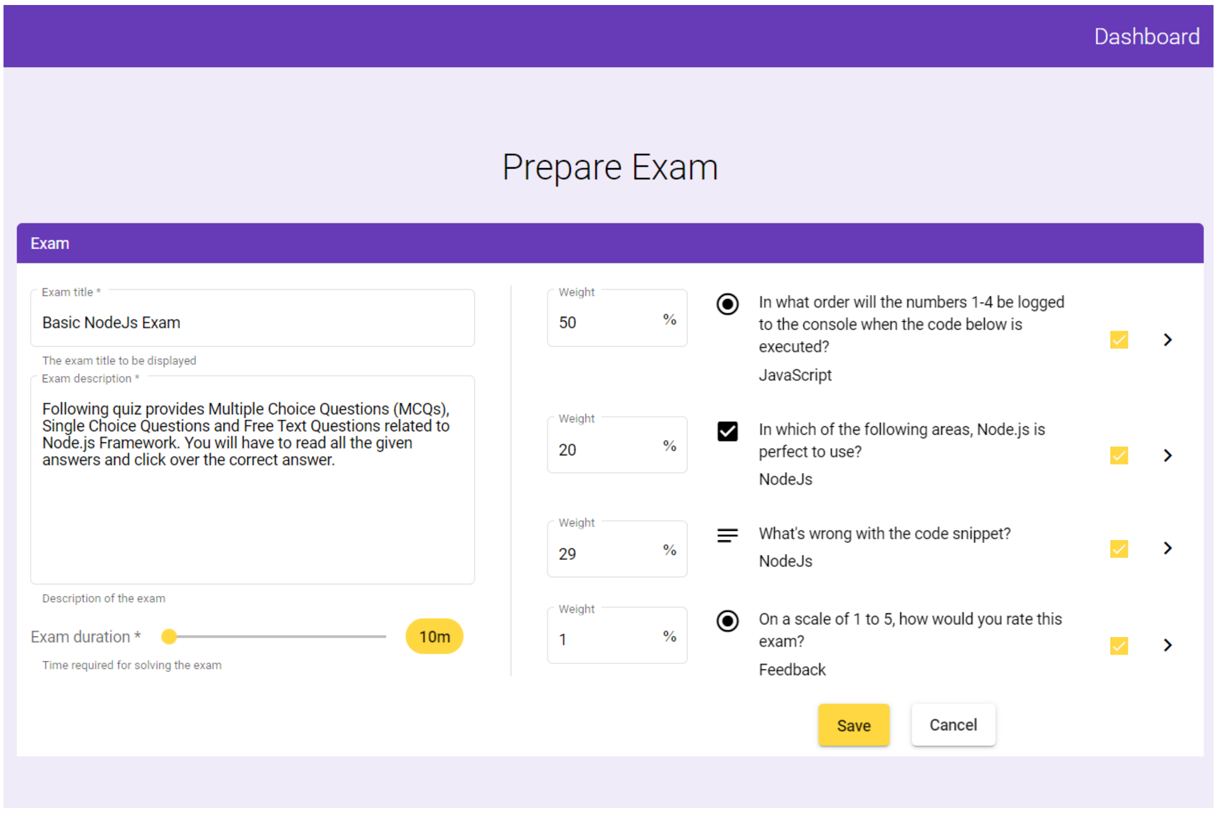
6. Press the Save button and save the question.

7. The user has now triggered an asynchronous process for adding a new question. When the process is completed the user will be notified through a message which directly links to the newly created question.

Use Case: Create Exam

Actor: Teacher

Short Description: A teacher wants to compose and add a new exam to his exams portfolio in order to be later able to test the knowledge of his students for his NodeJs lecture.



Exam title *
Basic NodeJs Exam
The exam title to be displayed

Exam description *
Following quiz provides Multiple Choice Questions (MCQs), Single Choice Questions and Free Text Questions related to Node.js Framework. You will have to read all the given answers and click over the correct answer.
Description of the exam

Exam duration * 10m
Time required for solving the exam

Weight 50 %	●	In what order will the numbers 1-4 be logged to the console when the code below is executed? JavaScript
Weight 20 %	✓	In which of the following areas, Node.js is perfect to use? NodeJs
Weight 29 %	≡	What's wrong with the code snippet? NodeJs
Weight 1 %	●	On a scale of 1 to 5, how would you rate this exam? Feedback

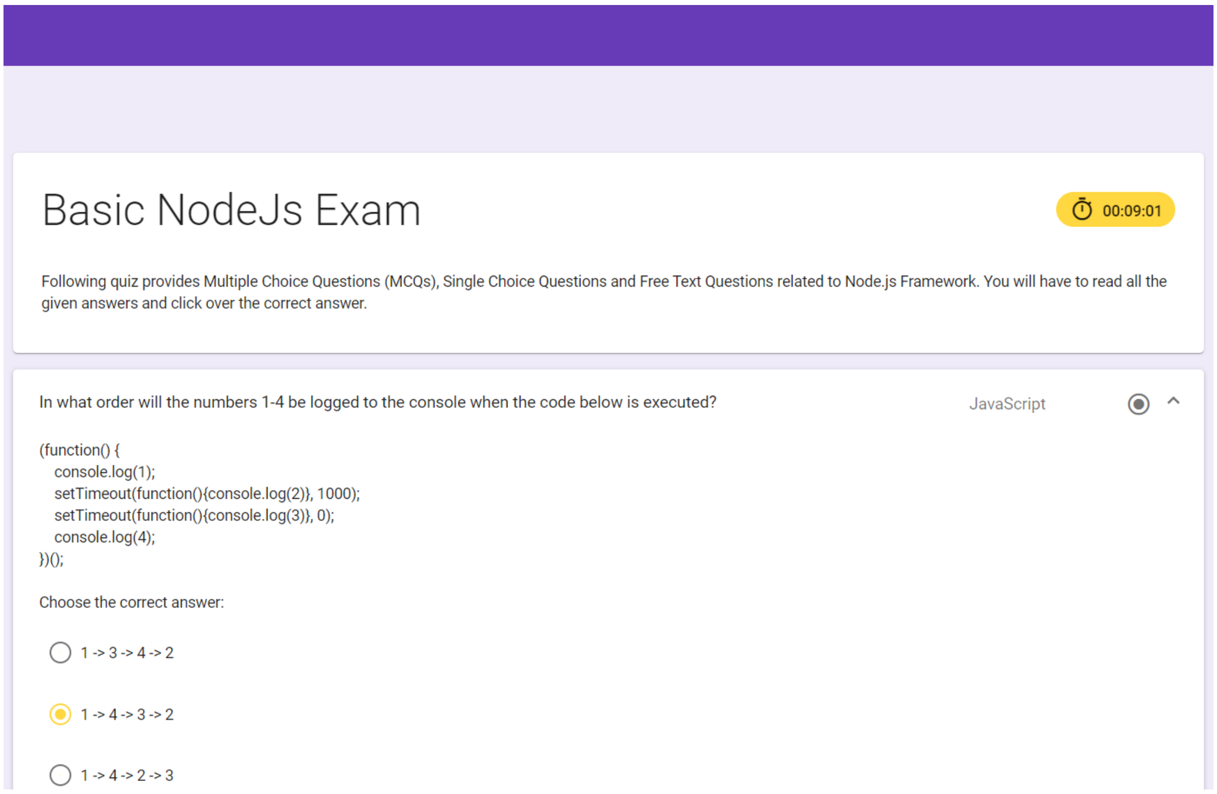
Figure 2.7: Creating an exam

Precondition: To be logged in the application with a valid teacher account and to have at least one available question in his questions catalog so it can be added to the exam.
Postcondition: The teacher can see the newly added exam in his exams portfolio together with details about the questions composing it while the students can see the new exam available on their home page.

Main Path:

1. Login into his teacher account.
2. Open the Prepare Exam section from the dashboard's side menu.
3. Fill in the required fields: the exam title, the exam description and the exam duration via the slider component.
4. On the right column select the questions which should be part of the exam, the available questions are the ones from the questions catalog.
5. Specify the percentage weight for each of the selected questions. The questions weights must sum up to one hundred.
6. Press the Save button and save the exam.
7. The user has now triggered an asynchronous process for adding a new exam. When the process is completed the user will be notified through a message which directly links to the newly created exam.

Use Case: Take Exam
Actor: Student
Short Description: A student want to take the exam for his NodeJs course in order to complete his exams session.
Precondition: To be logged in the application with a valid student account and have a token which is not expired. Moreover, the answer to the exam must be submitted within the available time provided.
Postcondition: The student is informed that his exam submission was successful while the teacher can perceive in real time the statistics regarding the exam answers.


Basic NodeJs Exam

Following quiz provides Multiple Choice Questions (MCQs), Single Choice Questions and Free Text Questions related to Node.js Framework. You will have to read all the given answers and click over the correct answer.

In what order will the numbers 1-4 be logged to the console when the code below is executed?

```
(function() {
   console.log(1);
   setTimeout(function(){console.log(2)}, 1000);
   setTimeout(function(){console.log(3)}, 0);
   console.log(4);
})();
```

Choose the correct answer:

- 1 -> 3 -> 4 -> 2
- [✓ 1 -> 4 -> 3 -> 2]
- 1 -> 4 -> 2 -> 3

Figure 2.8: Taking an exam

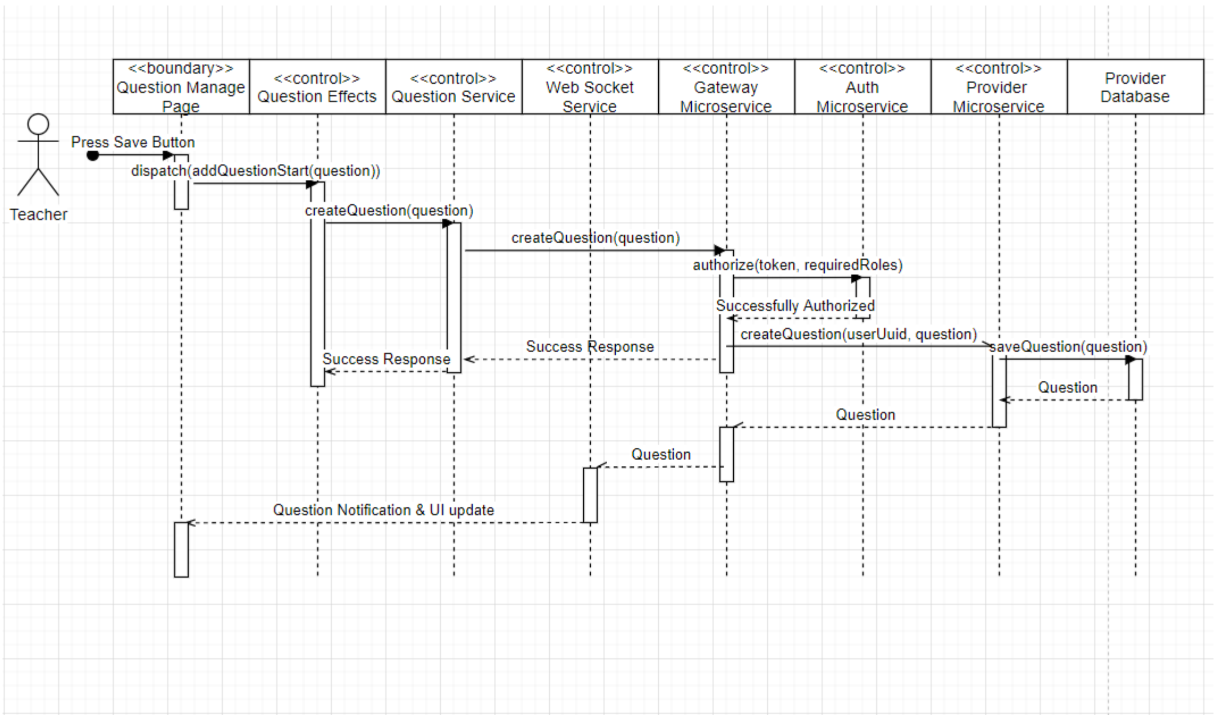
Main Path:

1. Login into his student account.
2. Choose the exam from the exams grid view.
3. Press the Start button and start taking the exam. This will fetch the exam questions and start the timer.
4. Answer to all the required questions by checking the correct result or by providing the requested text answers.
5. Press the Submit button and save his answers.
6. The user has now triggered an asynchronous process for saving his exam answers which also will provide real time updates for the exam's statistics. The student will be notified immediately after this process is started.

2.2.3 Sequence Diagrams

In the next pages there will be presented some sequence diagrams corresponding to the previously described use cases. However, the sequence diagram for the creation of an exam will be skipped since its flow is very similar to the add question use case.

Firstly, there will be presented the flow for the action of adding a new question by a teacher. The Figure 2.9 briefly depicts the sequence of events starting from the moment one triggers the action of creating a new question.



<<boundary>> Question Manage Page
<<control>> Question Effects
<<control>> Question Service
<<control>> Web Socket Service
<<control>> Gateway Microservice
<<control>> Auth Microservice
<<control>> Provider Microservice
Provider Database
Teacher
Press Save Button
dispatch(addQuestionStart(question))
createQuestion(question)
createQuestion(question)
authorize(token, requiredRoles)
Successfully Authorized
createQuestion(userUuid, question)
saveQuestion(question)
Question
Success Response
Success Response
Question
Question
Question Notification & UI update

Figure 2.9: Add Question sequence diagram

As presented in the diagram, the user starts by filling in the question's required fields and then presses the save button. This event is than handled in the Question Manage component which, after checking if the submitted form is valid, dispatches the addQuestionStart action with the new product's data. The action is then handled by an NgRx effect which calls the exposed create question endpoint of the Gateway via the Question Service. Before handling the actual logic for the HTTP request, the Gateway checks the authorization of the request via a synchronous call to the Auth Microservice. If the authorization is successful, the user details, which are present in the provided JWT token, will then be injected in the context of the request and the normal flow will be resumed.

Further, the Gateway triggers an asynchronous process by sending a message which targets the Provider Microservice and which also carries the user information and the new question's data. The Provider Microservice will then receive the message and after processing the question data it will both save it into a database via a dedicated outgoing port, and also will send an asynchronous message back to the Gateway.

When receiving the success message from the Provider Microservice, the Gateway will forward it through a web socket connection to all the active sessions of the user which initiated the process. The message is then received on the client application where a new NgRx action is dispatch, thus updating the store and therefore updating the displayed data through the NgRx selectors.

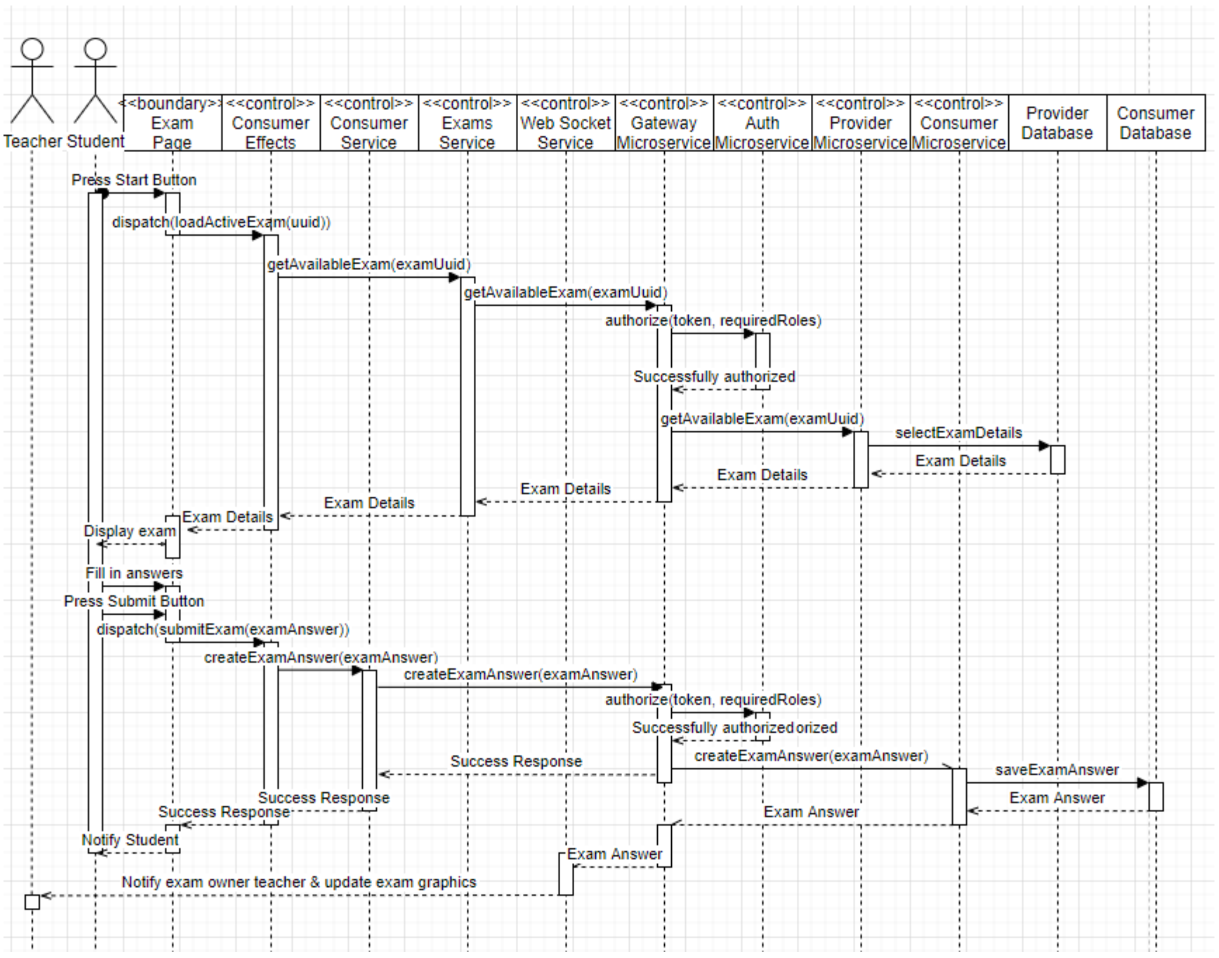

Teacher
Student
<<boundary>> Exam Page
<<control>> Consumer Effects
<<control>> Consumer Service
<<control>> Exams Service
<<control>> Web Socket Service
<<control>> Gateway Microservice
<<control>> Auth Microservice
<<control>> Provider Microservice
<<control>> Consumer Microservice
Provider Database
Consumer Database
Press Start Button
dispatch(loadActiveExam(uuid))
getAvailableExam(examUuid)
getAvailableExam(examUuid)
authorize(token, requiredRoles)
Successfully authorized
getAvailableExam(examUuid)
selectExamDetails
Exam Details
Exam Details
Exam Details
Exam Details
Exam Details
Display exam
Fill in answers
Press Submit Button
dispatch(submitExam(examAnswer))
createExamAnswer(examAnswer)
createExamAnswer(examAnswer)
authorize(token, requiredRoles)
Successfully authorized
createExamAnswer(examAnswer)
saveExamAnswer
Exam Answer
Success Response
Success Response
Success Response
Exam Answer
Notify Student
Exam Answer
Notify exam owner teacher & update exam graphics

Figure 2.10: Take Exam sequence diagram

The Figure 2.10 briefly presents the sequence of events for the Take Exam flow. At the beginning of the sequence the student is already on the page of an exam. As it can be seen from the diagram, the student starts by pressing the Start button. This action is being handled in the Consumer Exam Component which dispatches a loadActiveExam NgRx action together with the uuid of the requested exam. The action then triggers an asynchronous effect which makes an HTTP call to the Gateway microservice via the Exams Service.

In order to decide whether to continue the flow, the Gateway, checks the validity of the request by delegating the authorization responsibility to the Auth microservice via a synchronous call. If the received response is affirmative, the normal flow is executed furtherly and a synchronous request is made towards the Provider microservice. When receiving the request, the Provider microservice queries the details about the exam, including the questions and answer options, and then return the data to the Gateway. Next, the response is forwarded step by step until it reaches the NgRx effect which will then dispatch another action. The action will be furtherly handled by a reducer and the user interface will be updated through the selectors, thus the student is now able to view and solve the exam questions.

After filling in all the required answers, the student presses the Submit button which, in the same manner as in the previous flow will dispatch an action which will then be handled by an effect. Then an HTTP call will be made to the Gateway's createExamAnswer endpoint. The request will be authorized in the same way as the previous was and then an asynchronous process will be triggered for saving the exam answer while a successful response will be returned in order to notify the student that his answers were submitted successfully. The Consumer microservice will then save the exam answer into its associated not relational database and send another asynchronous message back to the Gateway. As in the flow of adding a question, the Gateway will receive the message and then forward it through a web socket connection to the Web Socket Service which will further update the NgRx store, thus the teacher who owns the exam will have the displayed charts updated via selectors.

2.2.4 Database Diagrams

As previously presented, the system embraces the concept of sparse databases meaning that each of the Users, Provider and Consumer microservices have its associated database. In the case of Users and Providers microservices there are two relational databases while regarding the Consumers microservice there is a NoSQL database associated. Furtherly there will be presented the diagram for each of those databases.

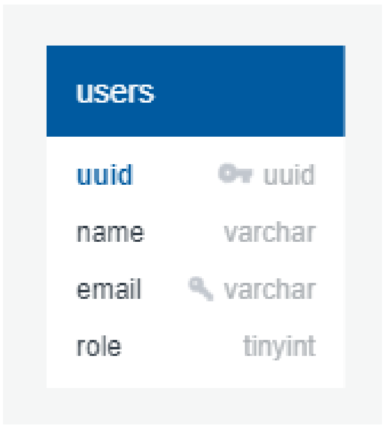
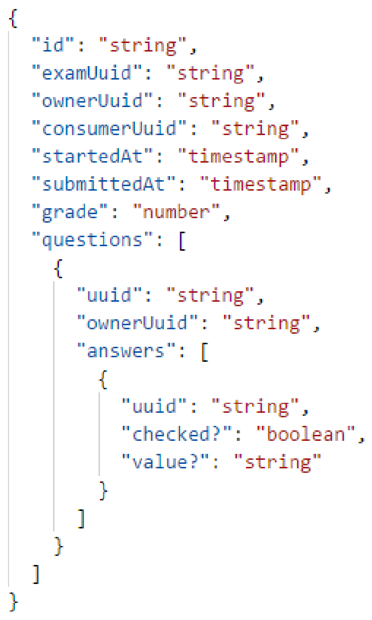


Figure 2.11: Users database diagram



```
{
  "id": "string",
  "examUuid": "string",
  "ownerUuid": "string",
  "consumerUuid": "string",
  "startedAt": "timestamp",
  "submittedAt": "timestamp",
  "grade": "number",
  "questions": [
    {
      "uuid": "string",
      "ownerUuid": "string",
      "answers": [
        {
          "uuid": "string",
          "checked?": "boolean",
          "value?": "string"
        }
      ]
    }
  ]
}
```

Figure 2.12: Exam Answers document structure of the Consumers Database

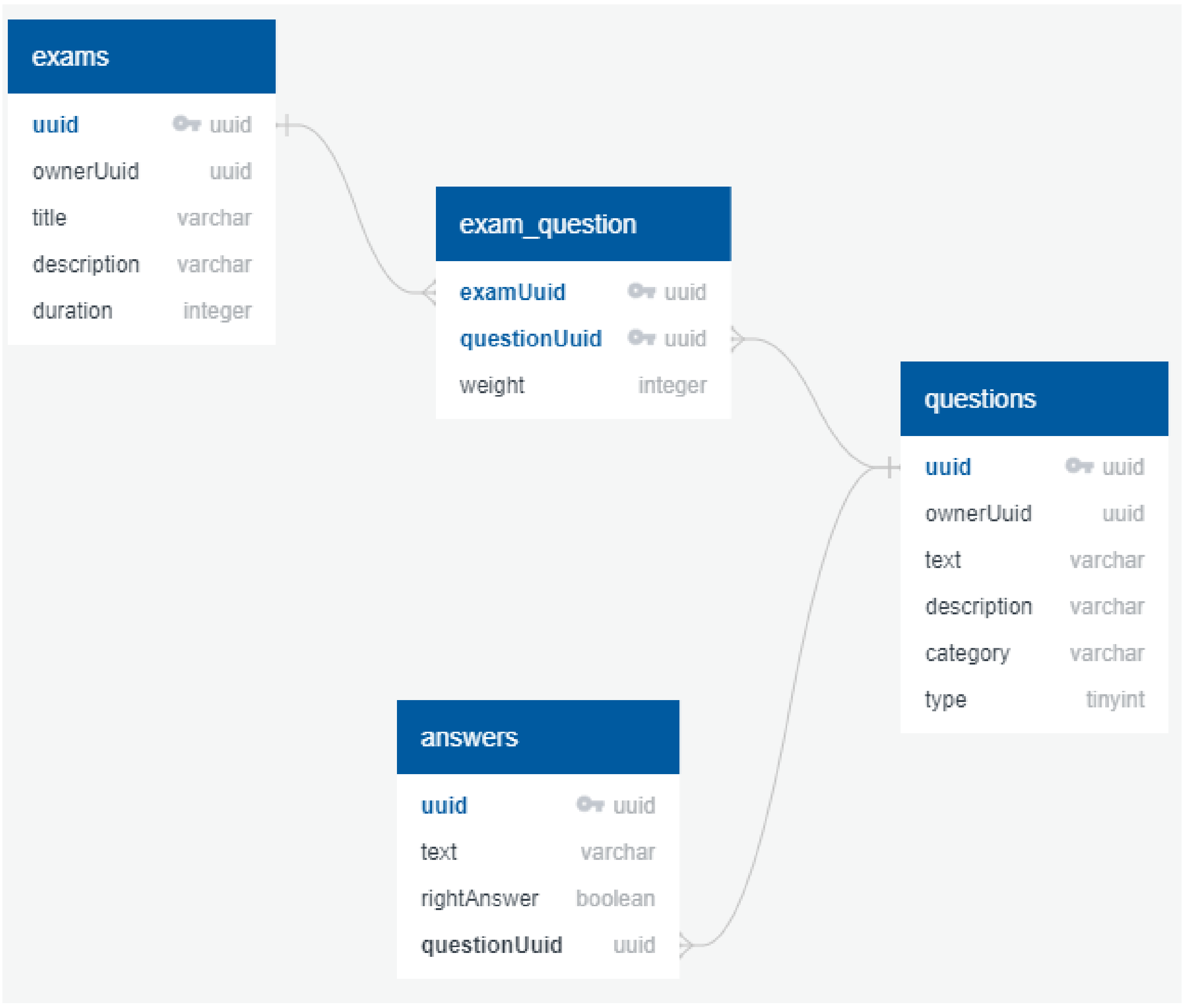


Figure 2.13: Providers database diagram

Both of the relational databases are built with PostgreSQL [27]. The Users database only hold the information about the user accounts including a role field used for implementing the simple role based authentication mechanism. The Providers database contain the information regarding the exams, the questions together with their answers and the associations between the exams and the questions which also contain the weight a certain question has for its associated exam. The Consumers database is built as a Mongodb [28] document database which only contains the exam answers collection used for the exam answers documents storage.

As the databases are split between multiple microservices the data association is only available via identifiers without direct relations. As an example, the association between an exam and its owner user is made through the ownerUuid field.

2.3 Technologies Used

For the development of the application a bundle of programming languages, technologies and frameworks were used and in the following some of them will be presented briefly.

The programming language chosen for writing both the backend microservices and the frontend client is **Typescript** [17] which is considered by many to be a super set for JavaScript which also provide data type support. It was built in order to facilitate the development of large scale applications. Moreover, when compared to JavaScript, Typescript helps in mitigating multiple problems causing run time errors related to mismatch data types, when using the latter by providing compile time errors instead. A great advantage of using typescript when referring to the client development is that it provides the possibility of using brand new features of the language even if the developed application will run in older browsers. This is due to the fact that Typescript is later compiled to a targeted version of vanilla JavaScript which can be run in older browsers.

The framework used for developing the backend microservices is **NestJs** [7]. It is well suitable for developing easily scalable and efficient Node.js backend applications. Moreover, it provides a mix of elements from Object Oriented Programming, Functional Programming and Functional Reactive Programming. The framework provides great support for building microservices and inter-services communication, websockets and Object Relational Mapping third party libraries, such as TypeORM [12], used for database interaction, which are backed up by an even greater documentation. Furthermore, the framework is heavily inspired from Angular meaning that it provides by default a module based architecture together with other highly useful concepts like the Inversion of Control in form of Dependency Injection, thus providing the possibility of a more reliable and readable code base.

The framework chosen for building the client application is **Angular** [9]. It is an open source framework and development platform written in Typescript which facilitates the development of sophisticated single-page applications. As an advantage of using Angular is the existence of concepts like Dependency Injection based on types, the possibility of using declarative templates with the possibility of Typescript code interpolation directly into HTML templates. Additionally, Angular also provides easy to implement constructs such as pipes for data formatting, guards for implementing routing restrictions or HTTP interceptors.

NgRx [25] is a framework used for developing reactive applications in Angular. It is a state management solution, relying on Rxjs library, which was highly inspired by the Redux library. Likewise in the Redux library, the NgRx framework provides fundamental constructs for state management such as actions, reducers for handling synchronous state mutations and effects for handling asynchronous code.

Chapter 3

Conclusions

Everyone can agree that nowadays distance education is going towards becoming more and more adopted due to external factors such as the new pandemic. Therefore, there is an increasing need for online platforms which support distance education and online evaluation. Moreover, due to the high volatility in regards of business requirements and the need for customized solutions which should satisfy the needs and preferences of different educational institutions and environments, the process of building an online evaluation system becomes not so trivial as one would expect. This means that such a solution should highly consider things like extensibility, scalability and maintainability and therefore the architectural design must be carefully chosen.

Furtherly, there will be briefly presented the relation of the current state of the system which made the subject of this thesis and other similar approaches by highlighting some similarities and differences. Moreover, there will be also presented some directions of improvement and further development.

When considering online examination platforms, one can easily notice that there is a wide range of applications available out on the web. One good candidate of such an application is Moodle [29]. It is a free learning management system that also facilitates the process of online evaluation through its quizzes features. In comparison with the application proposed in the thesis, Moodle is built with respect to a plugin architecture while the proposed one follows a microservices architecture.

The plugin architecture mainly implies the existence of a small core component on top of which multiple dependent plugins can be added in order to extend the functionality of the core, thus assuring the extensibility of the system. However, for a system which highly considers scalability, an application which relies on a suite of independent microservices is more appropriate as instead of having to clone the whole system, each microservice can be scaled independently.

Regarding the possibilities of further development of the current model, there is plenty of room for further improvements for the already existing features as well as for other new features. One straight forward extension would be the creation of a native mobile client application. This can be easily achieved due to the flexible architecture of the backend system, more specifically of the Hexagonal architecture with respect to which the Gateway microservice is built. Another highly needed extension would be the creation of user groups which would represent for example the students enrolled to a course. This will assure a much better management of exams and the creation of much complex statistics and insights regarding the exam answers.

Finally, this thesis aims to present a way for building an easily scalable, extensible and maintainable online evaluation system by leveraging the advantages of the microservices architecture and other modern technologies. In more details, it describes the road through software architectures which lead to microservices, a way of designing individual microservices by following the Hexagonal architecture and also presents the practical approaches and challenges for building the minimum viable product (MVP) of an online examination platform as the proof of concept.

Bibliography

[1] Salah, Tasneem, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. ”The evolution of distributed systems towards microservices architecture.” In 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), pp. 318-325. IEEE, 2016.

[2] Lamersdorf, Winfried. ”Paradigms of Distributed Software Systems: Services, Processes and Self-organization.” In International Conference on E-Business and Telecommunications, pp. 33-40. Springer, Berlin, Heidelberg, 2011.

[3] Oluwatosin, Haroon Shakirat. ”Client-server model.” IOSRJ Comput. Eng 16, no. 1 (2014): 2278-8727.

[4] Namiot, Dmitry, and Manfred Sneps-Sneppe. ”On micro-services architecture.” International Journal of Open Information Technologies 2, no. 9 (2014): 24-27.

[5] Hassan, M., Zhao, W., Yang, J. (2010, July). Provisioning web services from resource constrained mobile devices. In Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on (pp. 490-497).

[6] Abbott, Martin L., and Michael T. Fisher. The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise. Pearson Education, 2009

[7] NestJs official documentation: https://docs.nestjs.com

[8] Taibi, Davide, Valentina Lenarduzzi, and Claus Pahl. ”Architectural Patterns for Microservices: A Systematic Mapping Study.” In CLOSER, pp. 221-232. 2018.

[9] Angular official documentation: https://angular.io/docs

[10] Malavalli, Divyanand, and Sivakumar Sathappan. ”Scalable microservice based architecture for enabling DMTF profiles.” In 2015 11th International Conference on Network and Service Management (CNSM), pp. 428-432. IEEE, 2015.

[11] Redux official documentation: https://redux.js.org/introduction/getting-started

[12] TypeORM official documentation: https://typeorm.io

[13] Scarborough, Walter, Carrie Arnold, and Maytal Dahan. ”Case study: Microservice evolution and software lifecycle of the xsede user portal api.” In Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale, pp. 1-5. 2016.

[14] Jaramillo, David, Duy V. Nguyen, and Robert Smart. ”Leveraging microservices architecture by using Docker technology.” In SoutheastCon 2016, pp. 1-5. IEEE, 2016.

[15] Malavalli, Divyanand, and Sivakumar Sathappan. ”Scalable microservice based architecture for enabling DMTF profiles.” In 2015 11th International Conference on Network and Service Management (CNSM), pp. 428-432. IEEE, 2015.

[16] Özer, Mahmut, and H. Eren Suna. ”COVID-19 pandemic and education.” Reflections on the pandemic in the future of the worlds (2020): 157-178.

[17] Bierman, Gavin, Martín Abadi, and Mads Torgersen. ”Understanding typescript.” In European Conference on Object-Oriented Programming, pp. 257-281. Springer, Berlin, Heidelberg, 2014.

[18] Fette, Ian, and Alexey Melnikov. ”The websocket protocol.” (2011).

[19] Tilkov, Stefan, and Steve Vinoski. ”Node. js: Using JavaScript to build high-performance network programs.” IEEE Internet Computing 14, no. 6 (2010): 80-83.

[20] Shingala, Krishna. ”Json web token (jwt) based client authentication in message queuing telemetry transport (mqtt).” arXiv preprint arXiv:1903.02895 (2019).

[21] Barth, Adam. ”HTTP state management mechanism.” (2011).

[22] Barth, Adam, Collin Jackson, and John C. Mitchell. ”Robust defenses for cross-site request forgery.” In Proceedings of the 15th ACM conference on Computer and communications security, pp. 75-88. 2008.

[23] Newman, Sam. Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.", 2015.

[24] Wolff, Eberhard. Microservices: flexible software architecture. Addison-Wesley Professional, 2016.

[25] Farhi, Oren. Reactive Programming with Angular and ngrx. Apress:, 2017.

[26] Mansilla, Sergi. Reactive Programming with RxJS 5: Untangle Your Asynchronous JavaScript Code. Pragmatic Bookshelf, 2018.

[27] Momjian, Bruce. PostgreSQL: introduction and concepts. Vol. 192. New York: Addison-Wesley, 2001.

[28] Chodorow, Kristina. MongoDB: the definitive guide: powerful and scalable data storage. " O'Reilly Media, Inc.", 2013.

[29] Rice, William, and H. William. Moodle. Birmingham: Packt Publishing, 2006.