

Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering



Master's Thesis

**Web application development using .NET and
microservices architecture**

Vladyslav Odynets

© 2023 CULS Prague

DIPLOMA THESIS ASSIGNMENT

Vladyslav Odynets

Informatics

Thesis title

Web application development using .NET and microservices architecture

Objectives of thesis

Research the best practices of .NET web application development using a microservices architecture. Build a web application that will allow students and teachers to share their works, and exchange useful materials. Use all the practices and methods of developing applications using a microservices architecture.

Methodology

The first step in creating the diploma thesis would be the analysis of existing articles, books, and other materials, that describe developing server-side applications using a microservices architecture. There is a need to find out suitable conditions and situations when this architectural approach would be considered more efficient for the business/company and would ensure the long-term maintainability of the application.

Moreover, another part of the qualitative research would be interviews with the software developers and software architects from different companies, which would be conducted in order to obtain the latest information and real-world examples of using the microservices approach in the building of server-side applications.

The second step is to create an application itself, using the best practices, that were identified in the previous step. This would be done to verify the theoretical background of the current work and provide the usage case for selected best practices and developmental patterns.

The proposed extent of the thesis

60-80 pages

Keywords

.net, microservices, web application, backend

Recommended information sources

Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional; 1st edition (August 20, 2003). ISBN-13: 978-0321125217.

Horsdal, Christian. Microservices in .NET Core: with examples in Nancy 1st Edition. Manning; 1st edition (February 3, 2017). ISBN-13: 978-1617293375

Whitesell, Sean. Pro Microservices in .NET 6: With Examples Using ASP.NET Core 6, MassTransit, and Kubernetes. Apress; 1st ed. edition (January 2, 2022). ISBN-13:978-1484278321

Expected date of thesis defence

2022/23 SS – FEM

The Diploma Thesis Supervisor

Ing. Jiří Brožek, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 4. 11. 2022

Ing. Martin Pelikán, Ph.D.

Head of department

Electronic approval: 28. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Dean

Prague on 28. 03. 2023

Declaration

I declare that I have worked on my bachelor thesis titled "Web application development using .NET and microservices architecture" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the master's thesis, I declare that the thesis does not break any copyrights.

In Prague on March 31st _____ Vladyslav Odynets

Acknowledgment

I want to thank my supervisor, Ing. Jiří Brožek, Ph.D., for giving me the opportunity to complete this work under his guidance. His advice and encouragement were invaluable. I'm also grateful to the Czech University of Life Sciences Prague for providing me with the resources and support I needed to complete this project.

Web application development using .NET and microservices architecture

Abstract

The increasing complexity and scale of modern software applications demand architectural approaches that can accommodate rapid development, scalability, and maintainability. One such approach is the microservices architecture, which has emerged as a popular solution for building distributed systems. This thesis investigates the microservices architecture, its best practices, and the application of this approach using the .NET framework and Angular.

The theoretical part of the thesis covers the fundamentals of microservices, its advantages and challenges, and the best practices for building applications using this architecture.

The practical part of the thesis demonstrates the development of a university-related application that employs microservices architecture. This real-world example provides insights into the process of transitioning from a monolithic application to a microservices-based solution.

Thesis showcases the potential of microservices architecture in creating scalable, maintainable, and flexible applications.

Keywords: .NET, microservices, web application, backend

Vývoj webových aplikací s využitím architektury

.NET a mikroslužeb

Abstrakt

Rostoucí složitost a rozsah moderních softwarových aplikací vyžadují architektonické přístupy, které umožňují rychlý vývoj, škálovatelnost a udržovatelnost. Jedním z takových přístupů je architektura mikroslužeb, která se stala oblíbeným řešením pro budování distribuovaných systémů. Tato práce se zabývá architekturou mikroslužeb, jejími osvědčenými postupy a aplikací tohoto přístupu s využitím frameworku .NET a jazyka Angular.

Teoretická část práce se zabývá základy mikroslužeb, jejich výhodami a výzvami a osvědčenými postupy pro budování aplikací s využitím této architektury.

Praktická část práce demonstruje vývoj univerzitní aplikace využívající architekturu mikroslužeb. Tento reálný příklad poskytuje vhled do procesu přechodu z monolitické aplikace na řešení založené na mikroslužbách

Práce ukazuje potenciál architektury mikroslužeb při vytváření škálovatelných, udržovatelných a flexibilních aplikací.

Klíčová slova: .NET, mikroslužby, webová aplikace, backend

Table of contents:

1	Introduction	12
2	Objectives And Methodology.....	14
2.1	Objectives.....	14
2.2	Methodology	14
3	Literature Overview.....	15
3.1	An Overview of Microservices Architecture	15
3.1.1	Historical Context	15
3.1.2	Key Characteristics of Microservices	15
3.1.3	Principles of Microservices Architecture.....	16
3.2	Used technologies	17
3.2.1	The .NET Framework: ASP.NET Core 6 and 7.....	17
3.2.2	Front-end Web Framework: Angular.....	19
3.2.3	Database: PostgreSQL	20
3.2.4	Database: MongoDB.....	21
3.2.5	API Gateway: Ocelot	22
3.2.6	API Documentation: Swagger.....	22
3.2.7	Image Processing: Magick.NET	23
3.2.8	Integrated Development Environment: Visual Studio	23
3.2.9	Integrated Development Environment: VS Code	24
3.2.10	Entity Framework.....	25
3.2.11	C# Language	25
3.2.12	TypeScript Language	26

3.3	Best Practices for Building Microservices Architectures	26
3.3.1	Main practices.	26
3.3.2	Example project structure:	29
3.4	Transitioning from Monolith to Microservices	32
3.4.1	The Challenge of Starting with Microservices.....	32
3.4.2	Separating a Monolith into Modules.....	33
3.4.3	Data Management Strategies.....	33
3.4.4	Modular monolith architecture.....	34
4	Practical part	36
4.1	Application Functionality and Features	36
4.1.1	User Registration and Authentication	36
4.1.2	Role-Based Behavior	38
4.1.3	Content Creation: "Add" Page	39
4.1.4	Content Discovery: Select Page.....	42
4.1.5	View post page.....	46
4.1.6	Profile Page: Displaying User Information and Posts.....	47
4.2	The process of building the application.	48
4.3	Overview of the application: Client side.....	49
4.4	Overview of the Application - Server Side	52
4.4.1	Uni.WebApi	53
4.4.2	Gateway.WebApi.....	61
4.4.3	Post.WebApi	62
5	Results and discussions	66
6	Conclusions	68
7	References	69

8. List of figures..... 74

1 Introduction

The software development landscape has undergone significant transformations over the years, with modern applications demanding increased flexibility, scalability, and maintainability. Microservices architecture has emerged as a popular approach to address these challenges, offering a modular and decentralized solution for building robust and efficient applications. As more organizations adopt microservices, there is a growing need to explore best practices and practical applications of this architecture to maximize its benefits.

This thesis delves into the world of microservices architecture, focusing specifically on building applications using the .NET framework. The primary goal is to provide a comprehensive understanding of the key principles and practices of microservices architecture, and to demonstrate its practical implementation through the development of a real-world application.

The application in question will be built using a server-client architecture, with both server and web client as separate components. The server side will be implemented using the latest ASP.NET Core versions 6 and 7, with microservices as its foundation. The web client will be a single-page application (SPA) built using the latest Angular version.

In addition to the core technologies, the thesis will explore a range of supplementary tools and libraries used in the development process, such as Ocelot for API gateway functionality, Swagger for automatic documentation generation, and the MagickImage library for image manipulation. Furthermore, the application will utilize PostgreSQL and MongoDB as its databases, with a discussion on the benefits and use cases for each.

The thesis will also include a qualitative research component, examining the conditions and situations in which microservices architecture is the most beneficial for businesses and their long-term application maintainability. This will involve interviews with software developers from various companies to gather real-world examples and insights into the practical application of microservices.

By the end of this thesis, the reader will have gained a solid understanding of microservices architecture, its best practices, and its real-world applications using the .NET

framework. This knowledge will serve as a valuable resource for software developers and architects looking to adopt microservices architecture in their own projects and organizations.

2 Objectives And Methodology

2.1 Objectives

Research the best practices of .NET web application development using a microservices architecture. Build a web application that will allow students and teachers to share their works, and exchange useful materials. Use all the practices and methods of developing applications using a microservices architecture.

2.2 Methodology

The first step is the analysis of existing articles, books, and other materials, that describe developing server-side applications using a microservices architecture. There is a need to find out suitable conditions and situations when this architectural approach would be considered more efficient for the business/company and would ensure the long-term maintainability of the application.

Moreover, another part of the qualitative research would be interviews with the software developers and software architects from different companies, which would be conducted in order to obtain the latest information and real-world examples of using the microservices approach in the building of server-side applications.

The second step is to create an application itself, using the best practices, that were identified in the previous step. This would be done to verify the theoretical background of the current work and provide the usage case for selected best practices and developmental patterns.

3 Literature Overview

3.1 An Overview of Microservices Architecture

3.1.1 Historical Context

The concept of microservices can be traced back to the early 2000s, with roots in the service-oriented architecture (SOA) and domain-driven design (DDD) paradigms (Newman 2015). The term "microservices" was coined around 2011 during a workshop attended by software practitioners (Fowler and Lewis 2014). The growing need for scalable, modular, and maintainable applications led to the emergence of microservices architecture as a response to the limitations of traditional monolithic architectures (Dragoni et al. 2017).

Over the years microservices became more and more common in commercial development because of their key characteristics. One of the classic appliances of microservice architecture we can see in Twitter, where developers noticed that load on Twitter APIs which is takes care about tweets is much bigger then on other parts like registration and user settings APIs. So upsides of separating that part to allow it scale independently were huge.

3.1.2 Key Characteristics of Microservices

Microservices architecture is characterized by several defining traits, which contribute to its ability to accommodate modern application requirements. One of the main features of microservices is their modularity. They are organized as a collection of small, independent, and loosely coupled modules, each responsible for a specific functionality (Newman 2015). This modularity allows for greater flexibility and maintainability of the system.

Autonomy is another crucial characteristic of microservices. Each microservice can be developed, deployed, and scaled independently, reducing the dependencies and complexities associated with monolithic architectures (Richardson 2018). This autonomy helps organizations achieve increased agility and responsiveness in their software development processes.

In the context of microservices, the concept of bounded context from domain-driven design (DDD) plays a significant role in organizing the system (Evans 2003). It ensures that related functionalities and data are encapsulated within well-defined boundaries, which enables teams to work on individual microservices with minimal interference.

Decentralized data management is another important aspect of microservices architecture. Microservices typically manage their own data storage and access, avoiding a centralized data management system (Richardson 2018). This approach improves data consistency, resilience, and scalability.

Lastly, microservices promote polyglot development, allowing for the use of multiple programming languages, frameworks, and tools (Newman 2015). This flexibility enables developers to choose the most suitable technology for a particular service.

3.1.3 Principles of Microservices Architecture

Several principles underpin the microservices architecture, which contribute to its success in delivering scalable and maintainable applications. The single responsibility principle is central to microservices design, stating that each microservice should focus on a single functionality (Martin 2003). This focus promotes cohesion and simplifies the development and maintenance of individual services.

Another key principle is the combination of high cohesion and loose coupling. Microservices should exhibit high cohesion within their boundaries and be loosely coupled with other services (Fowler and Lewis 2014). This combination enables teams to make changes and deploy services independently without impacting the overall system.

API-driven communication is also an essential aspect of microservices architecture. Microservices communicate via well-defined APIs, typically using lightweight protocols such as REST or gRPC (Richardson 2018). This approach enables interoperability, versioning, and abstraction between services.

Resilience and fault tolerance are crucial principles in microservices architecture. The design should embrace the possibility of failure and incorporate patterns for resilience and fault

tolerance (Newman 2015). Techniques like circuit breakers, timeouts, and bulkheads help minimize the impact of failures on the overall system.

Finally, microservices should support continuous integration and continuous deployment (CI/CD) practices to facilitate rapid and reliable software development and deployment (Fowler and Lewis 2014). This approach allows teams to deliver new features and bug fixes quickly and consistently.

Overall, a microservices architecture is a software architectural style that structures an application as a collection of small, independently deployable services. Each service is focused on a specific task and communicates with other services through well-defined interfaces, typically using a lightweight mechanism such as an HTTP API.

The microservice approach is designed to make it easier to develop, test, and maintain applications by allowing developers to work on individual services in isolation and deploy them independently of the larger application. This can improve the flexibility and scalability of the application and make it easier to update or modify individual components without affecting the entire system.

Microservices are typically implemented using a variety of programming languages and technologies, and they may be deployed on different servers or in different environments. They are often used in conjunction with containerization technologies like Docker, which allow for easy deployment and scaling of individual services.

Overall, the goal of the microservice approach is to enable a more modular, agile, and scalable way of building and maintaining complex applications.

3.2 Used technologies

3.2.1 The .NET Framework: ASP.NET Core 6 and 7

The .NET framework, an open-source, cross-platform framework developed by Microsoft, has evolved over the years to support the development of modern, high-performance applications (Richter, 2012). With the introduction of ASP.NET Core, the framework has

become more modular, lightweight, and optimized for building microservices-based applications (Freeman, 2020). In this thesis, we will be focusing on utilizing the features of ASP.NET Core 6 and 7, which offer improved performance, enhanced security, and a simplified development experience (Microsoft, 2021a).

The choice of ASP.NET Core 6 and 7 is motivated by their support for the latest web standards, built-in Dependency Injection (DI) container, and advanced configuration options that simplify the development and deployment of microservices (Microsoft, 2021b). Furthermore, these versions of ASP.NET Core embrace the principles of microservices architecture by offering first-class support for containerization with Docker and Kubernetes, enabling better resource utilization and ease of deployment in cloud-native environments (Naylor, 2020).

ASP.NET Core 6 and 7 provide a robust set of tools and libraries for building microservices, including the integration with the Entity Framework Core, a powerful Object-Relational Mapper (ORM) for .NET, which simplifies data access and mapping between the application's domain model and the underlying databases. Additionally, the framework supports the development of asynchronous, non-blocking code using the `async/await` pattern, allowing for efficient resource utilization and improved application responsiveness (Richardson, 2018).

ASP.NET Core 6 and 7 also support the development of cross-platform applications, allowing developers to build and deploy microservices on Windows, Linux, and macOS environments. This cross-platform capability is enabled by the .NET runtime, which offers a consistent and high-performance execution environment across different platforms (Microsoft, 2021c). By leveraging this feature, microservices developed using ASP.NET Core can be easily deployed and scaled in diverse infrastructure setups, facilitating the adoption of cloud-native technologies.

The security features provided by ASP.NET Core are another critical aspect of building microservices-based applications. The framework offers built-in support for modern authentication and authorization protocols, such as OAuth 2.0 and OpenID Connect, ensuring secure communication between microservices and protecting sensitive data (Lock, 2021). Additionally, ASP.NET Core includes features for data protection, secure data transmission

with HTTPS, and cross-site request forgery (CSRF) prevention, further enhancing the security posture of the developed microservices (Microsoft, 2021d).

In summary, the features and enhancements in ASP.NET Core 6 and 7, including improved performance, enhanced security, and support for the latest web standards, provide a robust foundation for designing, developing, and deploying microservices-based applications. By leveraging the built-in tools and libraries for data access, containerization, and cross-platform development, developers can create scalable, maintainable, and high-performance applications that adhere to the principles of microservices architecture.

3.2.2 Front-end Web Framework: Angular

For the front-end web application, we will employ Angular, a popular and powerful web framework maintained by Google (Flanagan, 2020). Angular's component-based architecture, extensive ecosystem, and ability to integrate seamlessly with .NET Core make it an ideal choice for building sophisticated, responsive, and maintainable user interfaces for our microservices-based application (Hussain, 2018).

Angular's support for reactive programming, efficient change detection, and performance optimization techniques, such as Ahead-of-Time (AOT) compilation, influenced our choice of this framework (Google, 2021). Reactive programming in Angular is facilitated through the RxJS library, which provides a comprehensive set of tools and operators for creating, transforming, and composing asynchronous data streams (Larkin & Cross, 2020). This approach simplifies the management of complex state and data flow within the application, leading to more maintainable and scalable front-end code.

One of the key features of Angular is its robust support for building reusable and testable components. Angular's component-based architecture promotes a clear separation of concerns, enabling developers to create modular and self-contained components that can be easily tested and maintained. This modular structure aligns well with the microservices architecture, as it encourages developers to create small, focused, and reusable units of code that can be easily combined to form complex user interfaces.

Another advantage of Angular is its extensive ecosystem, which includes a wide range of third-party libraries, tools, and resources. These tools facilitate the rapid development and deployment of Angular applications by providing pre-built solutions to common problems and streamlining the development process. Notable examples include Angular Material, a collection of UI components that follow the Material Design guidelines (Google, 2021), and NgRx, a state management library built on reactive principles.

Angular also provides comprehensive support for various forms of testing, including unit testing, integration testing, and end-to-end testing. These testing capabilities are enabled by built-in tools such as TestBed, an API for testing Angular components and directives (Google, 2021), and Protractor, an end-to-end testing framework for Angular applications. By leveraging these testing tools, developers can ensure the quality and reliability of their Angular applications throughout the development lifecycle.

In addition to the aforementioned features, Angular offers a robust and extensible development toolchain, including the Angular CLI, a command-line interface for generating, building, and deploying Angular applications (Google, 2021). The Angular CLI streamlines the development process by automating repetitive tasks and enforcing best practices, which contributes to a more efficient and enjoyable development experience.

In conclusion, Angular's component-based architecture, reactive programming support, extensive ecosystem, and comprehensive testing capabilities make it an ideal choice for building the front-end user interfaces of our microservices-based application. By leveraging Angular's powerful features and tools, we can create a responsive, maintainable, and high-performance front-end that integrates seamlessly with the .NET Core back-end services.

3.2.3 Database: PostgreSQL

In our application, we will use two different databases to accommodate the diverse data storage requirements. The first choice is PostgreSQL, a powerful, enterprise-class open-source relational database system that offers advanced features such as full ACID compliance, support for complex data types, and extensibility (Obe & Hsu, 2015). We chose PostgreSQL due to its proven performance, reliability, and scalability for handling structured data.

PostgreSQL's robust transaction support and adherence to the SQL standard make it a reliable option for storing and managing critical data in a microservices-based application (Momjian, 2018). Furthermore, PostgreSQL provides advanced features such as stored procedures, triggers, and views, enabling developers to implement complex business logic and data processing tasks at the database level (Schönig, 2017).

The PostgreSQL ecosystem offers various tools and libraries for optimizing performance and simplifying administration tasks. For example, pgAdmin is a popular, open-source management tool for PostgreSQL that provides a graphical interface for managing and monitoring database objects and executing SQL queries. In addition, Npgsql, a high-performance .NET data provider for PostgreSQL, enables seamless integration with the .NET framework and Entity Framework Core. In our project we are going to use both pgAdmin and Npgsql as a tools to respectively manage and access the data in the database.

The choice of PostgreSQL as one of the databases for our microservices-based application is motivated by its advanced features, performance, and reliability. By leveraging the PostgreSQL ecosystem, we can build scalable and maintainable microservices that effectively handle structured data.

3.2.4 Database: MongoDB

The second database choice for our application is MongoDB, a popular NoSQL database designed for handling large volumes of unstructured or semi-structured data. It provides high availability, horizontal scaling, and a flexible data model, making it suitable for microservices that require schema flexibility and the ability to handle diverse data formats.

MongoDB stores data in a flexible, JSON-like format called BSON, which allows for efficient storage and querying of complex data structures such as arrays and nested documents (MongoDB, 2021). This flexibility makes MongoDB an ideal choice for microservices that deal with dynamic and evolving data models, as it eliminates the need for rigid schemas and complex data migration processes.

The MongoDB ecosystem provides various tools and libraries for optimizing performance, managing data, and integrating with the .NET framework. For instance, the

MongoDB Compass is a GUI for exploring and manipulating MongoDB data, while the MongoDB C#/.NET driver enables seamless integration with the .NET framework and support for LINQ queries (MongoDB, 2021).

MongoDB's support for horizontal scaling through sharding enables the database to distribute data across multiple nodes, providing a scalable solution for handling large volumes of data and high throughput loads (MongoDB, 2021). Additionally, MongoDB's built-in replication and automatic failover features ensure high availability and data durability, making it a suitable choice for mission-critical applications.

MongoDB's flexible data model, scalability, and high availability make it an appropriate choice for storing unstructured or semi-structured data in our microservices-based application. By leveraging the MongoDB ecosystem and its integration with the .NET framework, we can create efficient and scalable microservices that handle diverse data formats effectively.

3.2.5 API Gateway: Ocelot

As microservices communicate with each other and external clients through well-defined APIs, managing these interconnections can be challenging. To address this challenge, we will use Ocelot, a lightweight, extensible API gateway library designed for .NET Core (ThreeMammals, 2021). Ocelot provides a unified point of entry for external requests, enabling features such as routing, load balancing, authentication, and request aggregation (ThreeMammals, 2021). Incorporating Ocelot in our architecture simplifies communication between microservices and enhances the system's scalability and maintainability.

3.2.6 API Documentation: Swagger

Documenting APIs is a vital aspect of building and maintaining microservices-based applications, as it ensures that developers, testers, and other stakeholders can easily understand and interact with the APIs. To automate this process, we will use Swagger, a widely adopted tool for generating API documentation (Swagger, 2021). Swagger integrates with ASP.NET Core and provides an interactive user interface for exploring and testing API endpoints, making it easier for developers to work with the APIs throughout the intuitive UI interface.

3.2.7 Image Processing: Magick.NET

In our application, we will handle image processing tasks, such as resizing and optimizing downloaded images. This is needed to store user's files in a concise form and ensure the good speed of load. To achieve this, we will utilize Magick.NET, a powerful, open-source image processing library for .NET Core. Magick.NET provides an extensive set of features, including support for various image formats, image transformations, and optimizations. By integrating Magick.NET into our microservices-based application, we can efficiently process images and improve the overall performance and user experience.

3.2.8 Integrated Development Environment: Visual Studio

Visual Studio, developed by Microsoft, is a feature-rich and powerful Integrated Development Environment (IDE) for building applications using the .NET framework (Microsoft, 2021e). In this thesis, we will use Visual Studio as one of the primary development environments for building our microservices-based application, due to its extensive support for .NET, rich debugging capabilities, and seamless integration with various tools and platforms.

Visual Studio offers a comprehensive suite of tools for building, testing, and deploying .NET applications, including project templates, code generation tools, and support for various testing frameworks (Microsoft, 2021f). The IDE also provides first-class support for Git version control, enabling developers to track code changes, collaborate with team members, and manage application releases efficiently (Chacon & Straub, 2014).

One of the key features of Visual Studio is its powerful debugging and diagnostics capabilities, which enable developers to efficiently identify and resolve issues in their applications. Visual Studio's debugger supports advanced features such as conditional breakpoints, watch windows, and performance profiling, allowing developers to gain deep insights into the runtime behavior of their applications and optimize their code for better performance.

In addition to its core features, Visual Studio boasts a vibrant ecosystem of extensions and integrations, which enhances its capabilities and customizability. For instance, the Angular Language Service extension provides rich editing and debugging support for Angular

applications, while the PostgreSQL extension for Visual Studio simplifies database management tasks for PostgreSQL developers (Microsoft, 2021g).

In summary, Visual Studio's extensive support for .NET, powerful debugging capabilities, and seamless integration with various tools make it an ideal choice for building our microservices-based application. By leveraging the features and ecosystem of Visual Studio, we can create high-quality, maintainable, and performant applications using the .NET framework.

3.2.9 Integrated Development Environment: VS Code

Another development environment we will use for building our microservices-based application is Visual Studio Code (VS Code), a lightweight, cross-platform, and open-source code editor developed by Microsoft (Microsoft, 2021h). VS Code has gained widespread popularity among developers due to its extensibility, performance, and support for various programming languages and platforms, including .NET, Angular, and Node.js (Microsoft, 2021i).

VS Code's extensibility is one of its key strengths, as it allows developers to customize and enhance the editor's capabilities through a vast collection of extensions available in the VS Code Marketplace (Microsoft, 2021j). For example, the C# extension for VS Code provides rich support for .NET development, including IntelliSense, code navigation, and debugging, while the Angular Essentials extension pack offers a curated set of tools and extensions for building Angular applications (Microsoft, 2021k).

In addition to its extensibility, VS Code offers built-in features that facilitate efficient software development, such as Git integration, a powerful terminal, and a flexible settings system that enables developers to fine-tune the editor's behavior to their preferences (Microsoft, 2021l). Moreover, VS Code's cross-platform support allows developers to work on Windows, Linux, or macOS, ensuring a consistent development experience across different platforms.

VS Code also provides a rich set of tools for collaborating with team members and sharing code, such as Live Share, which enables real-time collaboration and pair programming within the editor (Microsoft, 2021m). This feature simplifies the process of code reviews,

debugging sessions, and knowledge sharing among developers, fostering a more collaborative and efficient development process.

3.2.10 Entity Framework

Entity Framework (EF) is a popular Object-Relational Mapping (ORM) framework for .NET applications that simplifies data access by allowing developers to interact with databases using strongly typed objects instead of raw SQL queries. This section will provide a brief overview of Entity Framework and its significance in the application development process, along with references to relevant sources.

Entity Framework streamlines the process of mapping between database tables and the application's domain objects, automating many repetitive tasks associated with data access (Microsoft, 2020a). By abstracting database-related operations, EF enables developers to focus on writing clean and maintainable code while reducing the likelihood of errors in SQL queries (Gupta & Misra, 2015, p. 61).

Some key benefits of using Entity Framework include:

Enhanced productivity through code generation and reduced boilerplate code (Julia Lerman & Miller, 2019, p. 11).

Improved maintainability and readability, as developers work with domain objects rather than raw SQL queries.

Support for various database systems, including PostgreSQL and MongoDB, through the use of Entity Framework Core providers (Microsoft, 2020b).

In the context of the application, Entity Framework plays a crucial role in managing data access and ensuring smooth communication between the application and the databases. The application leverages Entity Framework Core, the latest version of the framework, which offers improved performance and cross-platform support (Microsoft, 2021a).

3.2.11 C# Language

C# is a modern, versatile, and object-oriented programming language developed by Microsoft as part of the .NET framework (Hejlsberg et al., 2003). It combines features from

languages such as C++, Java, and Visual Basic, providing powerful capabilities for a wide range of applications, from desktop software to web services and mobile apps (Ferron, 2020). Some of the key advantages of C# include type safety, object-oriented programming, garbage collection, and Language Integrated Query (LINQ) support, which streamline the development process and improve code maintainability (Albahari & Albahari, 2020).

3.2.12 TypeScript Language

TypeScript is a statically typed superset of JavaScript, developed by Microsoft to address some of the limitations and challenges associated with JavaScript, particularly in large-scale application development (Bierman et al., 2014). TypeScript adds optional static typing to JavaScript, which can help catch potential errors early in the development process and improve code maintainability and readability (Vanderkam, 2019). TypeScript is designed to be compatible with existing JavaScript code and can be easily integrated into JavaScript projects. One of the primary advantages of TypeScript is its strong typing system, which enables better tooling support, including features such as autocompletion, refactoring, and error detection (Microsoft, 2021).

3.3 Best Practices for Building Microservices Architectures

3.3.1 Main practices.

Building applications using microservices architecture involves several best practices that contribute to the development of scalable, maintainable, and resilient systems. In this section, we outline key best practices for building applications using microservices architecture, drawing on the insights gained from the literature review and interviews with software developers and software architects. These best practices are applicable to the technologies used in our project, such as .NET, Angular, Ocelot, PostgreSQL, and MongoDB.

Domain-Driven Design

Domain-driven design (DDD) is a software development approach that focuses on defining clear boundaries between different parts of the system based on the underlying business domain (Evans, 2003). Applying DDD principles in microservices architecture ensures that each microservice corresponds to a specific business capability or domain, promoting a clear separation of concerns and facilitating maintainability and scalability (Newman, 2015).

API Design and Contract-First Development

Designing APIs that are consistent, well-documented, and easy to consume is crucial for effective communication between microservices (Newman, 2015). Adopting a contract-first approach to API development, where the API contract is defined before the implementation, ensures that microservices adhere to a consistent interface, simplifying integration and reducing the likelihood of errors (Newman, 2015). Tools like Swagger can be used to create and maintain API documentation, facilitating communication between developers and improving the overall development process (Swagger, 2021).

Decentralized Data Management

Each microservice should manage its own data store to ensure data consistency and autonomy (Newman, 2015). This approach allows each microservice to choose the most appropriate data storage technology based on its requirements, such as using MongoDB for Post.WebApi and PostgreSQL for Uni.WebApi in our project. Decentralized data management also prevents data contention and performance bottlenecks that may arise from sharing a single data store across multiple microservices (Richardson, 2018).

Resilience and Fault Tolerance

Microservices should be designed to handle failures gracefully and maintain functionality even in the face of partial system failures (Richardson, 2018). Implementing fault-tolerant communication patterns, such as retries, timeouts, and circuit breakers, can help ensure the stability and resilience of the overall system (Newman, 2015). In addition, monitoring and logging mechanisms should be in place to detect and diagnose issues early, enabling proactive resolution and minimizing the impact on the system (Richardson, 2018).

Continuous Integration and Deployment

Adopting continuous integration and deployment practices is essential for maintaining the agility and responsiveness of microservices-based applications (Humble & Farley, 2010). By automating testing, code reviews, and deployment processes, development teams can ensure the consistent quality and reliability of their microservices, and quickly adapt to changing requirements and market conditions (Humble & Farley, 2010).

Security

Securing microservices involves implementing authentication, authorization, and data protection mechanisms at the API level, as well as ensuring secure communication between microservices (Newman, 2015). Utilizing tools like OAuth 2.0 and OpenID Connect for implementing secure access control and user authentication can help protect sensitive data and prevent unauthorized access to the system.

Keeping services small and focused.

Each microservice should have a narrow, well-defined scope and should be responsible for a single task or function. This helps to keep the service simple and maintainable.

Use loose coupling.

Microservices should be designed to be as independent as possible, with minimal dependencies on other services. This makes it easier to develop, test, and deploy individual services independently.

Use API gateways.

An API gateway is a layer that sits between the client and the microservices and acts as a reverse proxy. It can help to improve security, optimize performance, and provide other benefits such as rate limiting and caching.⁵ Monitor and log extensively: Proper monitoring and logging is crucial for understanding how a microservice-based application is performing and for debugging issues that may arise.

Use a resilient architecture.

Microservices should be designed to be resilient to failures, with features such as circuit breakers, retries, and fallbacks to help ensure that the application can continue to function even if one or more services fail.

3.3.2 Example project structure:

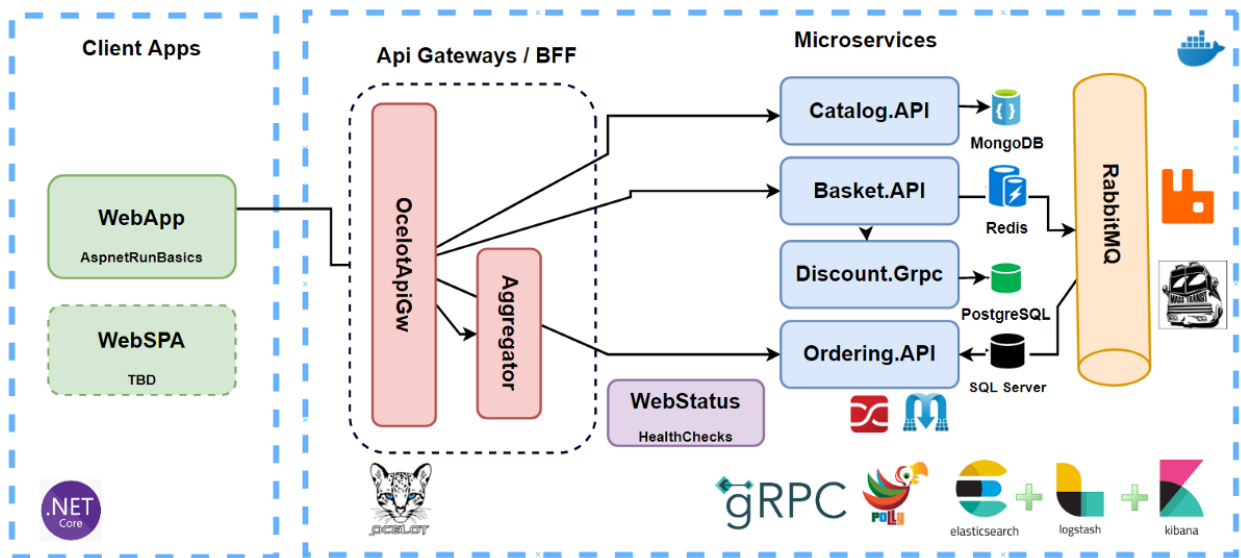


Figure 1. An example of the project structure. Source: (Ozkaya, 2019)

Here is one possible structure for a project that uses a microservices architecture:

1. Client applications: These are the applications that the end users will interact with, such as web or mobile apps.

Client applications are the applications that end users interact with to access the functionality provided by the system. These could be web apps, mobile apps, or other types of applications.

Client applications typically communicate with the backend services through APIs. The client application makes API requests to a gateway service, which routes the requests to the appropriate backend service and returns the response to the client.

Client applications are typically designed to be decoupled from the backend services, meaning that they are not directly dependent on the implementation details of the backend services. This can make it easier to change or update the backend services without affecting the client applications.

Client applications are designed to provide a user-friendly interface to the functionality provided by the system, while the backend services handle the underlying business logic and data management.

2. Gateway service: This is a service that acts as the entry point for all external requests to the system. It is responsible for routing requests to the appropriate service, handling authentication and authorization, and providing other security and routing functionality.

Some specific responsibilities of a gateway service might include:

- Routing requests to the appropriate service: The gateway service receives incoming requests and forwards them to the appropriate service based on the request path or other routing criteria.
- Load balancing: The gateway service can distribute incoming requests among multiple instances of a service to improve reliability and scalability.
- Service discovery: The gateway service can use a service registry to discover the available services in the system and route requests to them.
- Security: The gateway service can handle tasks such as authentication and authorization to ensure that only authorized users can access the system.
- Caching: The gateway service can cache results from downstream services to improve performance.

Overall, the gateway service is responsible for managing the flow of requests and responses between the client applications and the backend services.

3. Microservices: These are the individual services that make up the system. Each service is designed to be small and focused, with a single, well-defined responsibility.

Microservices are small, independent services that each have a single, well-defined responsibility. Each microservice is designed to be modular and scalable and to work with other microservices to form a larger application.

Some key characteristics of microservices include:

- Loose coupling: Microservices are designed to be independently deployable and to have minimal dependencies on other services. This allows them to be developed, tested, and deployed independently.
- Single responsibility: Each microservice is designed to have a single, well-defined responsibility, such as handling a specific data type or performing a specific business function.
- Automation: Microservices are typically built using automated processes and tools, such as continuous integration and deployment pipelines, to enable rapid development and deployment.
- Scalability: Microservices are designed to be horizontally scalable, meaning that they can be scaled by adding more instances of the service as needed to handle increased load.

Overall, microservices are a way of building applications as a set of small, independent services that work together to provide a complete application. This can improve the modularity, scalability, and maintainability of an application compared to a monolithic architecture.

4. Database(s): The system may include one or more databases to store data used by the services.

A database is a system for storing and managing data used by the services. Each service may have its own database or may share a database with other services.

There are several different types of databases that might be used in a microservices architecture, including:

- Relational databases: These are databases that use a tabular structure to store data and support SQL for querying the data. Examples include MySQL, PostgreSQL, and Microsoft SQL Server.
- NoSQL databases: These are databases that use a variety of data models, such as key-value, document, or graph, and do not support SQL. Examples include MongoDB, Cassandra, and Redis.
- In-memory databases: These are databases that store data in memory rather than on disk and are optimized for high performance. Examples include Redis and Memcached.

Which type of database to use will depend on the specific requirements of the services and the data they need to store. Some services may require the scalability and flexibility of a NoSQL database, while others may need the transactional support and consistency of a relational database.

The choice of a database is an important consideration, as it can have significant implications for the performance, scalability, and reliability of the system.

5. **Build and deployment tools:** These are the tools and processes used to build, test, and deploy the services and client applications.

Each microservice is developed, built, and deployed independently of the other microservices. This means that each microservice will typically have its own build and deployment process. There are many different tools that can be used to build and deploy microservices, and the choice of tool will depend on the specific needs of the microservice and the environment in which it will be deployed.

3.4 Transitioning from Monolith to Microservices

One of the primary challenges organizations face when adopting a microservices architecture is the transition from a monolithic application. Starting a new project with a microservices architecture can be difficult, particularly for developers unfamiliar with the paradigm. However, the long-term benefits of adopting a microservices approach often outweigh the initial challenges, as it leads to improved maintainability, scalability, and fault tolerance. This section will explore the process of breaking a monolithic application into microservices, focusing on how to separate the application into modules and discuss strategies for managing data during the transition.

3.4.1 The Challenge of Starting with Microservices

Starting with microservices architecture can be a daunting task, especially for teams that are inexperienced with the concepts and practices associated with the approach. Microservices require a higher level of coordination and understanding of distributed systems than monolithic applications, making it difficult for some organizations to embrace the change. Nevertheless,

microservices architecture can be an essential step for projects that need to scale and adapt over time, as it enables teams to independently develop, deploy, and manage individual components of the system.

3.4.2 Separating a Monolith into Modules

The first step in transitioning from a monolithic application to a microservices architecture is identifying and separating the application into individual modules. This process typically involves analyzing the existing application's structure and functionality, identifying logical boundaries between components, and defining a set of services that can be developed and deployed independently. Some key considerations during this process include:

- Analyze the application's domain: Understand the business requirements, processes, and data flows within the application. This understanding will guide the identification of natural boundaries between services and facilitate the decomposition of the monolith.
- Define cohesive and loosely coupled services: Each service should encapsulate a specific set of functionalities that are related to a single business capability or domain. The services should be designed to minimize dependencies and maximize the separation of concerns.
- Prioritize the separation: Identify the most critical or problematic areas of the monolith and prioritize their separation into microservices. This approach allows the team to gradually transition the application while minimizing the risk of disruption.

3.4.3 Data Management Strategies

Managing data during the transition from monolith to microservices is one of the most critical aspects of the process. The following strategies can help teams effectively handle data during the transition:

- Use separate tables: Initially, separate tables can be created for each microservice within the existing monolithic database. This approach enables teams to isolate the data for each microservice and minimize the impact on the remaining monolith during the transition.

- Migrate to separate databases: As the transition progresses, teams can gradually migrate each microservice to use its dedicated database. This step provides greater autonomy and reduces the risk of data contention or bottlenecks between microservices.
- Leverage shared database patterns: In cases where some data must be shared between microservices, teams can use patterns such as the Shared Database or Database per Service to manage data access and consistency.

3.4.4 Modular monolith architecture

By researching the existing ideas, we came up with some practices that can allow to mitigate the pain of separating boundaries on the later stages of application. At the current moment there is no current mainstream terminology on this topic, although in online discussions terms like “microlith” or “monoservices” can be found. In our opinion the term “microlith” is more accurate at describing the next set of recommendations which can help to curb the pain from separating tightly coupled application. It involves initially building application in a more modular way without causing of significant increase of complexity and loss of development agility.

Following recommendations were developed by researching different repositories on GitHub (Grzybek, 2019) and interviews with fellow developers who were working with different commercial projects which involve microservices architecture.

The recommendations that we came up with are:

1. Planning beforehand some context boundaries that may arrive in development of the application.
2. Creating a common solution with already separated project by the boundaries we defined at step
3. Copy-paste the infrastructure to each one of those projects. Make them use the same database and logging tools. This way it would not hinder the developers ability of developing the application as fast as possible.
4. Do not create direct references from one Web API project to another. Instead, create new .proj projects which are going to be used as a facade to the projects we separated in the

second step. This would be basically an implementation of a facade pattern on the infrastructure level. Those “façade projects” should have a single responsibility of redirecting the calls to the respective controller methods of the project.

5. Include the references to “façade projects” in other services that need to use their controller methods. This way we ensure that there would be no tight coupling and it would be easy to separate the projects completed on the later stages of development.

6. But still it would not eliminate all problems, because the coupling will remain on the database level. To leverage this problem the developers should try adding prefixes to the data tables which would mean the project to which the database belongs. Then during development try to avoid using the databases intended for other projects.

Transitioning from a monolithic application to a microservices architecture is a complex and challenging process. By separating the application into modules, focusing on data management strategies, and prioritizing the separation of critical components, organizations can successfully navigate the transition and realize the long-term benefits of a microservices approach.

4 Practical part

In this section, we present the practical implementation of a new application built using the server-client architecture, leveraging the technologies and best practices described in the previous sections. The application comprises a server-side component, constructed using microservices architecture, and a web client, which is a Single Page Application (SPA) built with the latest Angular (14 at the moment of starting development).

4.1 Application Functionality and Features

The application developed in the practical part of this thesis is designed to facilitate university-related communication and provide a platform for managing and sharing educational resources. This section outlines the key features and functionalities of the application, which include user registration and authentication, role-based behavior, profile management, and content creation and discovery.

4.1.1 User Registration and Authentication

The application employs a robust and secure user registration and authentication system to ensure that only authorized individuals can access its features and content. This section provides a more detailed overview of the registration and authentication process, including the use of JSON Web Tokens (JWT) and local storage for session management.

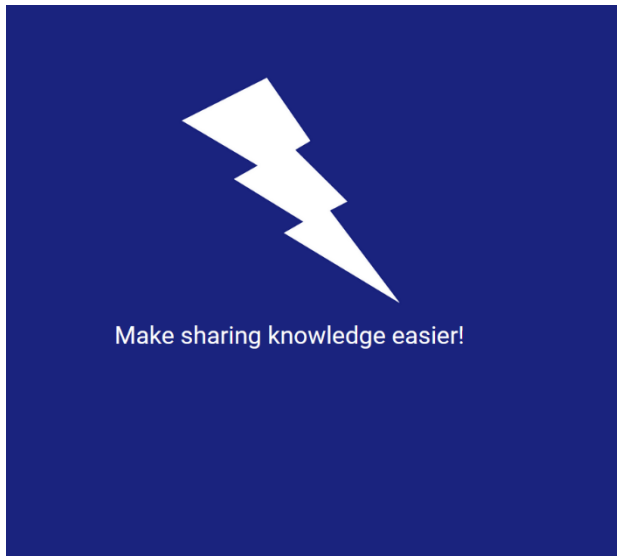
A white rectangular form titled "Sign In". It contains two input fields: "Email *" with a purple asterisk icon and "Password *" with a red eye icon. A link "Forgot password" is positioned to the right of the password field. Below the fields are two buttons: a dark blue "Sign In" button and a black "Sign Up" link.

Figure 2. Sign in page screenshot. Source: Author

To begin using the application, users must first create an account by completing the registration process. During registration, users are prompted to provide their name, email address, and password. The application validates the entered information, ensuring that the email address is unique and adhering to security best practices for password strength and storage, such as hashing and salting.

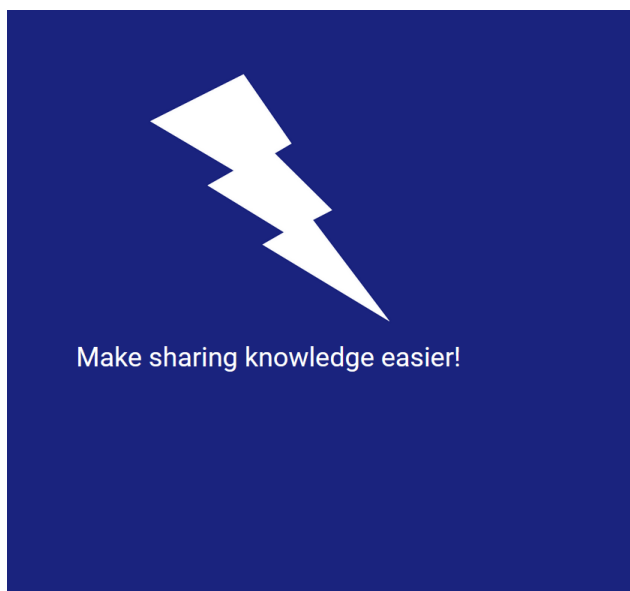
A white rectangular form titled "Sign up". It contains four input fields: "Username *" with a validation note "From 5 to 32 characters. You can use only a-z, 0-9, dots and underscores"; "Email *"; "Password *"; and "Confirm password *". Below the fields are two buttons: a dark blue "Sign up" button and a black "I've an account" link.

Figure 3. Registration page screenshot. Source: Author

Once the registration is successful, users can log in to the application by entering their email address and password. The authentication process involves verifying the entered credentials against the stored user data. If the credentials are valid, the application generates a JSON Web Token (JWT) and sends it to the client (Jones, 2015).

JSON Web Tokens are an industry-standard method for representing claims securely between two parties (Jones, 2015). In the context of the application, JWTs are used to encode the user's identity and role information, ensuring that subsequent requests to the server can be securely authenticated without the need to re-enter credentials. JWTs have a predefined expiration time, after which users must re-authenticate to obtain a new token.

To manage user sessions, the application stores the JWT in the browser's local storage. Local storage is a web storage API that allows the storage of key-value pairs in a web browser, with the data being accessible across multiple sessions. By storing the JWT in local storage, the application can maintain user sessions across page reloads and browser restarts, providing a seamless and convenient user experience.

It is important to note that storing JWTs in local storage has potential security implications, as the tokens may be vulnerable to Cross-Site Scripting (XSS) attacks if an attacker can inject malicious scripts into the application (OWASP, 2021). To mitigate this risk, the application should implement appropriate security measures such as Content Security Policy (CSP) and proper input validation and output encoding to prevent XSS attacks (OWASP, 2021).

4.1.2 Role-Based Behavior

The application implements role-based behavior to enforce access control and ensure that users can only perform actions appropriate for their role. The roles available within the application include regular users and administrators. Validators are responsible for validating posts submitted by users, ensuring the content is accurate and relevant. Administrators have the highest level of privileges, including the ability to delete any post and manage user roles.

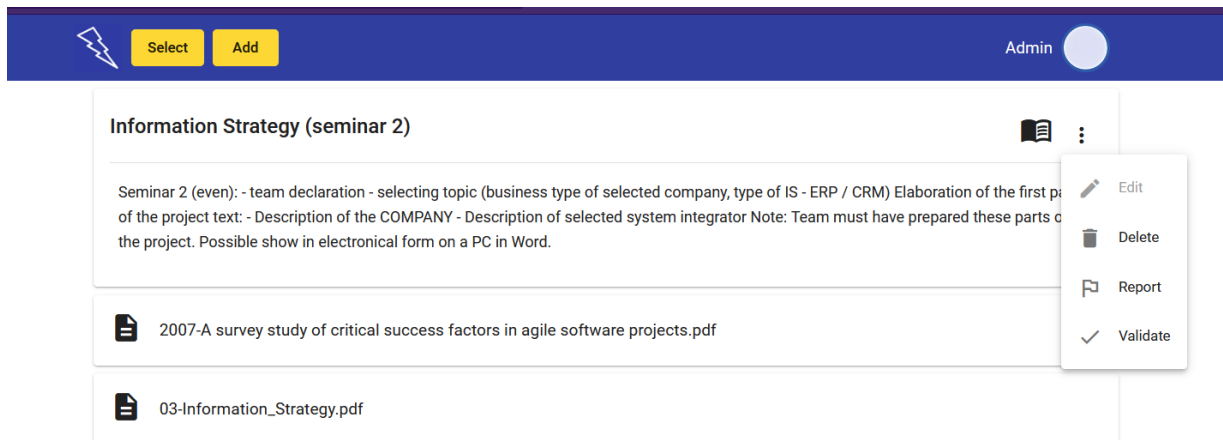


Figure 4. Screenshot of role-based behavior. Source: Author

4.1.3 Content Creation: "Add" Page

The "Add" page of the application is designed to facilitate a seamless and intuitive process for users to create new posts containing educational resources or information. In this section, we delve deeper into the user experience and functionality offered by the "Add" page, highlighting its key features and explaining the rationale behind its design.

Add

Subject *
Informatics 0 – Doe

Title *
Information Strategy (seminar 2)

example 5-500 characters

Course * Semester *
2 2

Group *
B-2021

Type *
Material



Description placeholder
Seminar 2 (even):

- team declaration
- selecting topic (business type of selected company, type of IS - ERP / CRM)

Elaboration of the first part of the project text:


- Description of the COMPANY
- Description of selected system integrator


Note: Team must have prepared these parts of the project. Possible show in electronical form on a PC in Word.






example

Upload file


2007-A survey study of critical success factors in agile software projects.pdf




03-Information_Strategy.pdf



Add

Figure 5. Screenshot of the "Add" page. Source: Author

When creating a new post, users are presented with a user-friendly form, which consists of several fields that capture the essential information associated with the post. These fields include:

- **Title:** This field captures the main subject or theme of the post, providing a concise and informative summary that allows users to quickly understand the content and purpose of the post.
- **Description:** The description field provides users with an opportunity to elaborate on the content of the post, offering detailed information and context that may not be fully conveyed by the title. This field supports long text entries and various

text formatting options, enabling users to create comprehensive and well-structured descriptions.

- Student's group: This field is used to specify the student group for which the post is relevant, allowing the application to categorize and filter posts based on the target audience. By organizing posts according to student groups, the application ensures that users can easily find content tailored to their specific needs and interests.
- Semester number: The semester number field helps further categorize the post within the context of the academic timeline, enabling users to discover content that is relevant to their current stage of study.
- Course number: Users can specify the course number associated with the post, which allows the application to organize content according to the specific courses and subjects being taught at the university.
- Type of post: This field provides users with a choice between three different post types – example, task, or material. By categorizing posts by type, the application can facilitate efficient and targeted searches, allowing users to find the specific resources they need more easily.

The structured approach to content creation on the "Add" page ensures that all posts are organized and easy to navigate, promoting a consistent and user-friendly experience. Additionally, the design of the "Add" page encourages users to provide detailed and accurate information, which helps maintain the overall quality and usefulness of the content on the platform.

In addition to these fields, the "Add" page also provides users with the option to attach files to their post, supporting a wide range of formats such as documents, images, and media files. This feature allows users to supplement their textual content with visual and interactive elements, enhancing the overall learning experience for other users who access the post.

The file attachment functionality is designed to be user-friendly and efficient. Users can easily upload files by clicking on an "Attach File" button, which opens a file browser allowing them to select the desired files from their local storage. The application provides real-time

feedback on the progress of the upload and displays a list of attached files, enabling users to manage their attachments and ensure that all necessary files are included.

By supporting various file formats and allowing users to attach multiple files to their posts, the application caters to diverse learning styles and preferences, promoting the sharing of comprehensive and engaging educational resources.

To further enhance the user experience and streamline the content creation process, the "Add" page also includes features such as input validation, tooltips, and auto-suggestions. Input validation helps prevent the submission of incomplete or improperly formatted information, while tooltips provide context and guidance to users as they complete the form. Auto-suggestions can assist users in selecting appropriate values for fields such as student groups, courses, and subjects, ensuring that posts are consistently categorized and easily discoverable.

4.1.4 Content Discovery: Select Page

The "Select" page plays a crucial role in the application by enabling users to efficiently discover and navigate through the wealth of educational resources and information available on the platform. In this section, we provide a more in-depth look at the various features and design considerations that contribute to an effective and user-friendly content discovery experience on the "Select" page.

One of the primary goals of the "Select" page is to provide users with a straightforward and intuitive method for filtering and sorting posts based on a variety of criteria. To achieve this, the "Select" page incorporates a multi-level filtering system that allows users to progressively narrow down the list of available posts according to their specific needs and interests.



Figure 6. Screenshot of the selection of country and city. Source: Author

The filtering process includes the following steps:

Country: Users can first select the country in which the university is located. This initial filtering step helps to eliminate irrelevant content and ensures that users are only presented with posts that are directly applicable to their geographical context.

City: After selecting a country, users can further refine their search by choosing a specific city. This additional filtering step helps to further narrow down the list of available posts, making it easier for users to find content that is relevant to their local area.

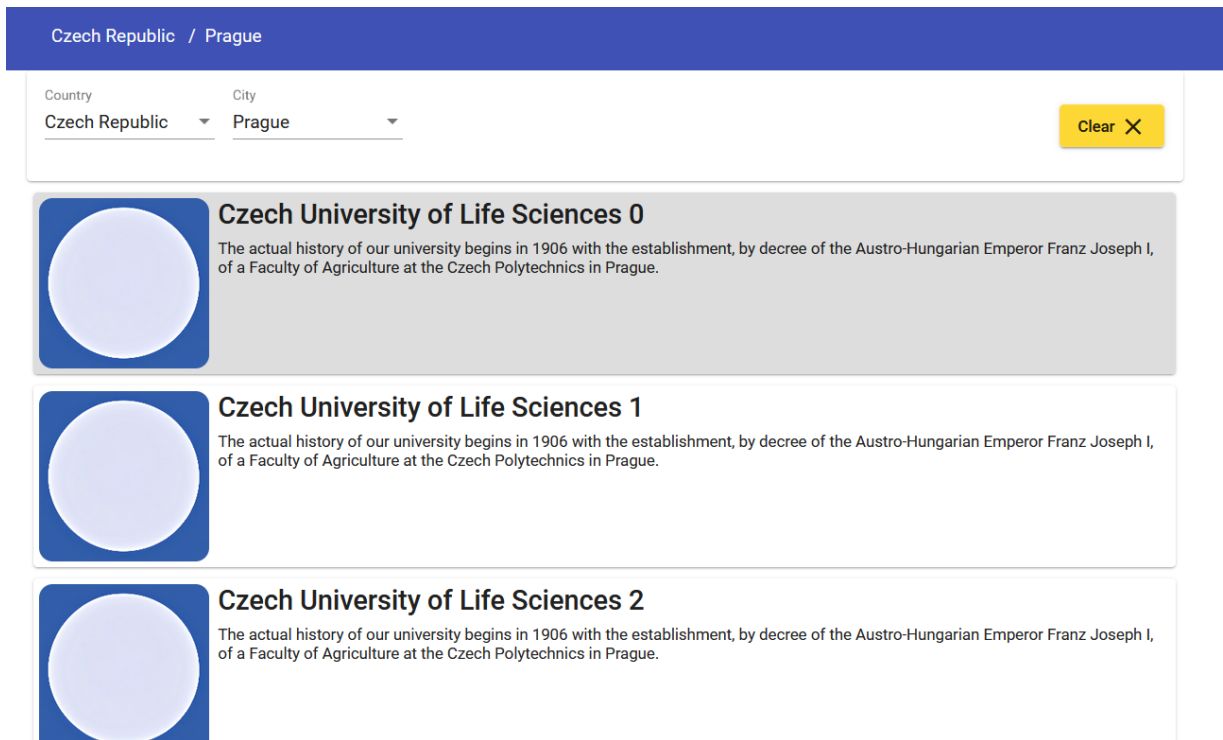


Figure 7. Screenshot of selection of university. Source: Author

University: With the country and city selected, users can then choose the specific university for which they are searching for content. This filtering step ensures that users are presented with posts that are directly related to their chosen institution, providing them with a tailored and focused content discovery experience.

Faculty: Next step of navigation is selection a faculty. This step has very similar UI controls as the selection of the university.

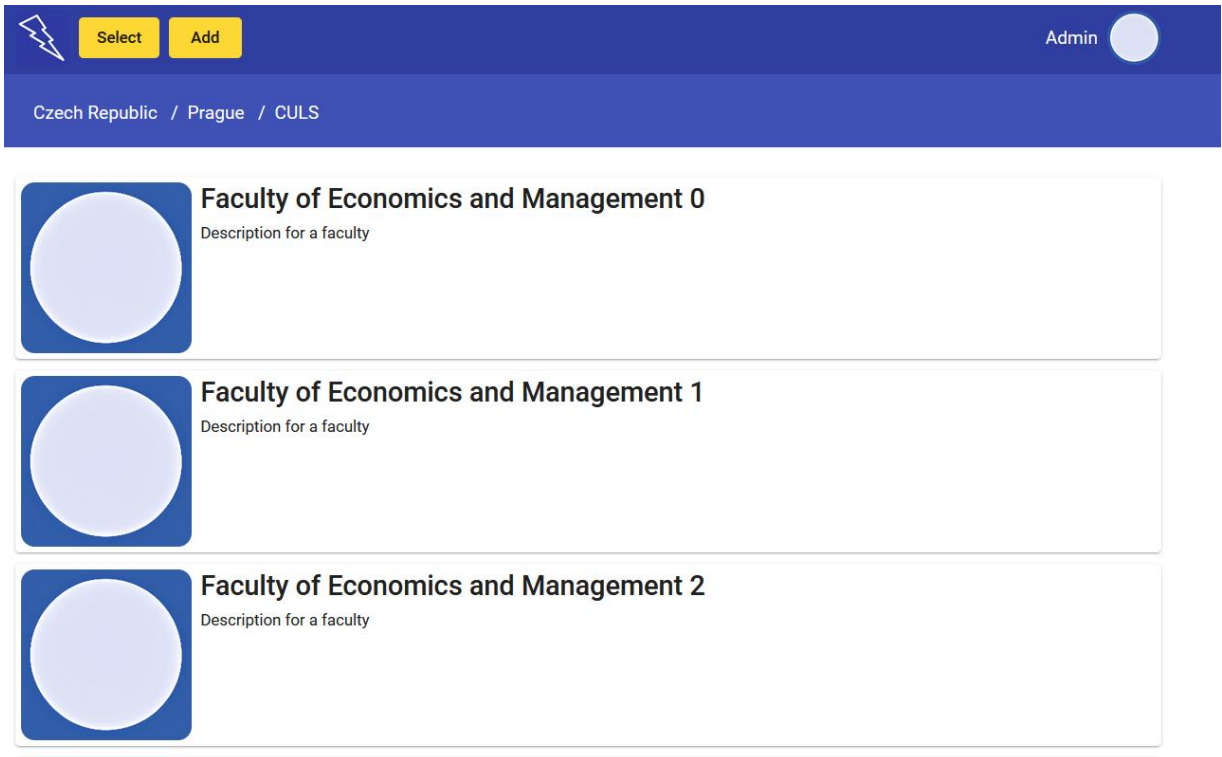


Figure 8. Screenshot of selecting a faculty. Source: Author

Subject: Finally, users can filter posts based on the subject or course to which the content pertains. By organizing posts according to subjects, the "Select" page helps users find content that is directly applicable to their area of study, further improving the efficiency and effectiveness of the content discovery process.

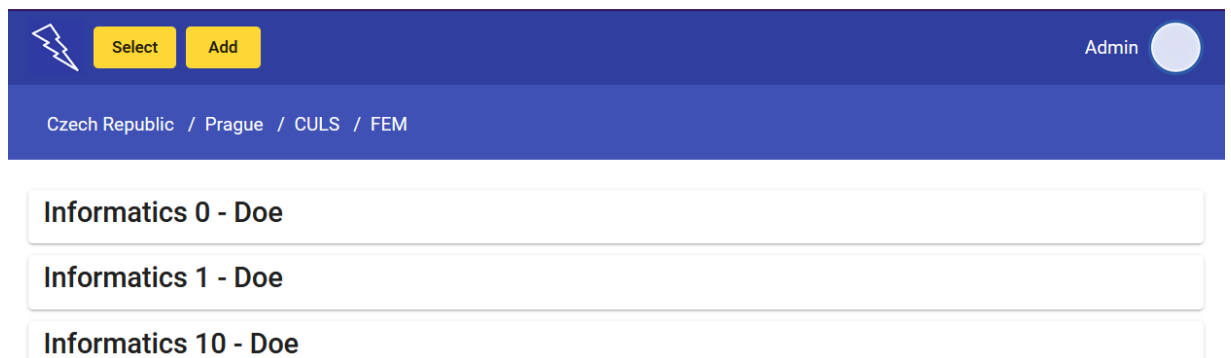


Figure 9. Screenshot of selecting a subject process. Source: Author

Once users have specified their filtering criteria, the "Select" page displays a list of posts that match the selected parameters. These posts are organized by semester and student group, making it easy for users to browse and locate the content they need. Additionally, users can sort

the list of posts based on various factors, such as date, popularity, or relevance, allowing them to quickly identify the most valuable and up-to-date resources.

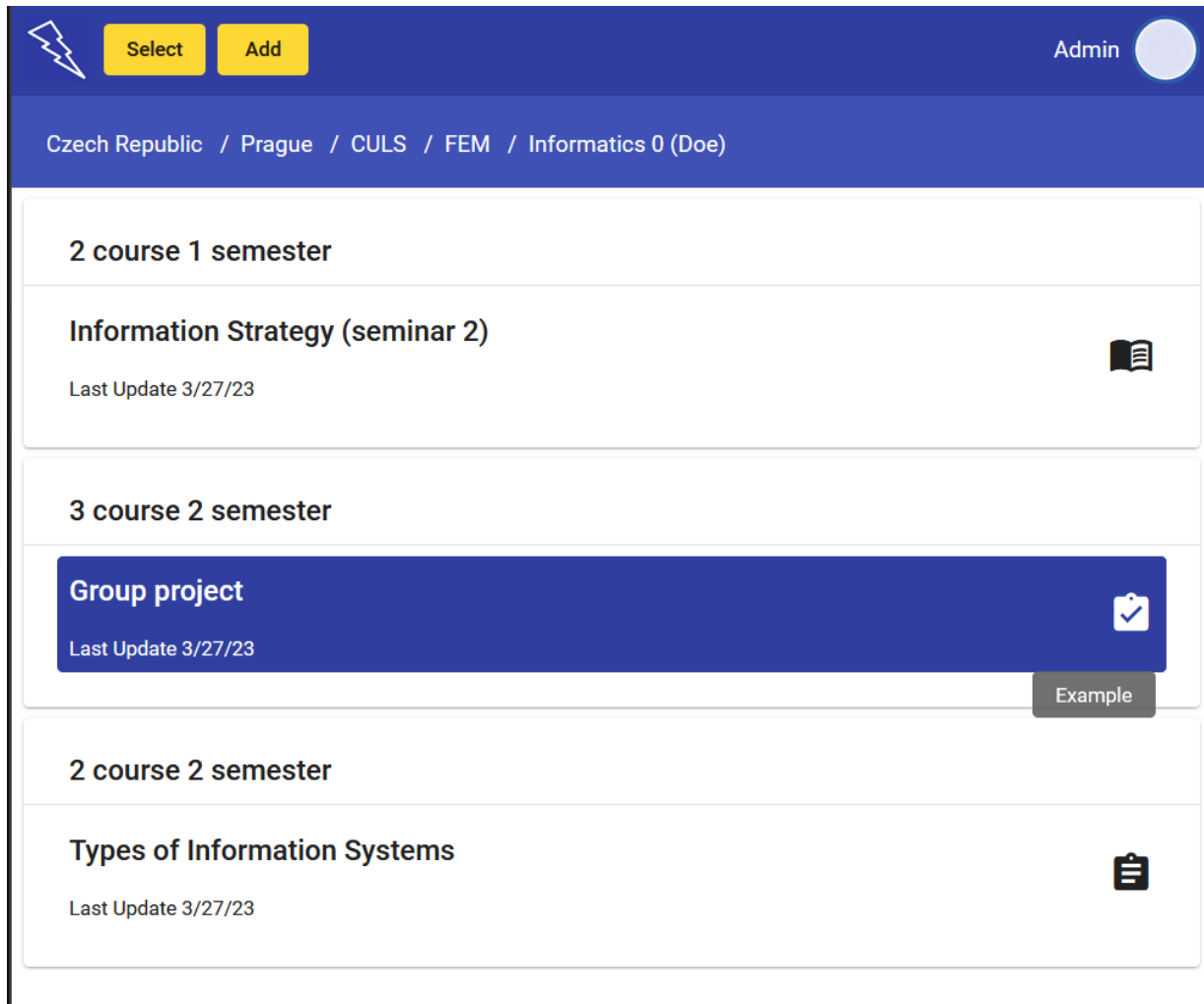


Figure 10. Screenshot of selecting a post process. Source: Author

To complement the filtering and sorting functionality, the "Select" page also features an intuitive and visually appealing user interface that promotes a seamless and enjoyable content discovery experience. The user can easily clear the selection of the country and city. Also, user has an ability to go at any level of hierarchy by using an element called “breadcrumbs” in the top part of the page. The design of the page incorporates clear and consistent visual cues, such as icons, color coding, and typography, which help users to quickly understand and navigate the available filtering options and content categories.

4.1.5 View post page

The "View Post" page is a crucial component of the application, designed to enable users to view and interact with individual posts and their associated data. This page displays all relevant post information and provides access to attached files, such as documents, images, and media. Additionally, it offers various management options depending on the user's role and permissions. For instance, admins can verify posts submitted by students, ensuring the quality and accuracy of content. This section will outline the features and functionality of the "View Post" page in more detail.

The "View Post" page presents users with a comprehensive view of the post's data, including title, description, student's group, semester number, course number, and type of post (example, task, or material). By displaying all pertinent information in a clear and organized manner, the page allows users to easily digest and understand the content of the post.

One of the key features of the "View Post" page is its ability to grant users access to all files attached to a post. By providing a simple and intuitive interface for downloading or viewing documents, images, and media, the "View Post" page ensures that users can fully engage with the post's content and any supporting material.

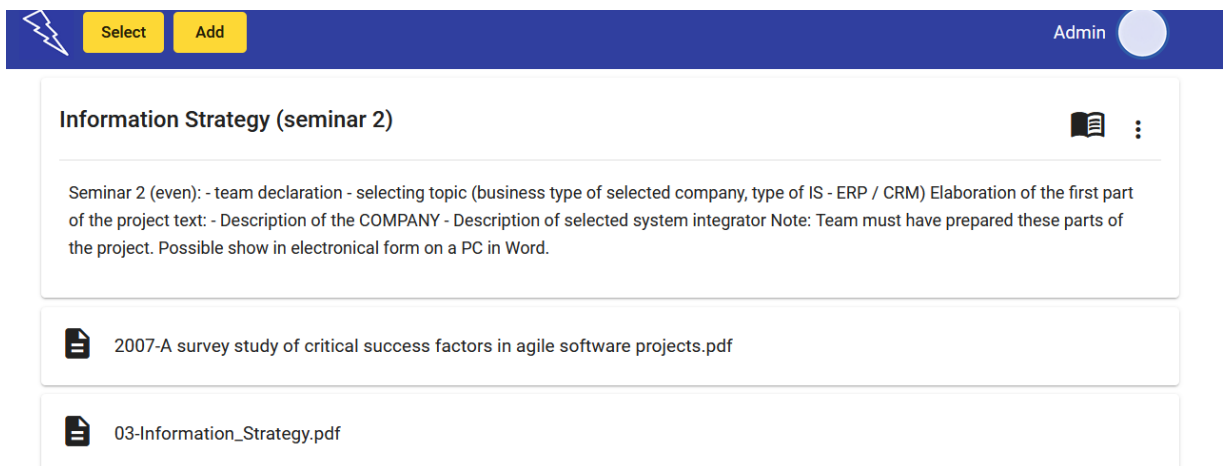


Figure 11. Screenshot of "View post" page. Source: Author

A unique feature of the "View Post" page for admins is the ability to verify posts submitted by students. This functionality allows admins to review and approve content, ensuring

that it meets the platform's quality standards and guidelines. By incorporating a verification process, the application promotes a higher level of accuracy and reliability in the content shared among users.

4.1.6 Profile Page: Displaying User Information and Posts

The Profile Page is an integral part of the application, designed to showcase a user's basic information, profile image, and a list of their posts. By providing users with a personalized space within the platform, the Profile Page enhances user engagement and promotes content discovery. This section will briefly outline the features and functionality of the Profile Page.

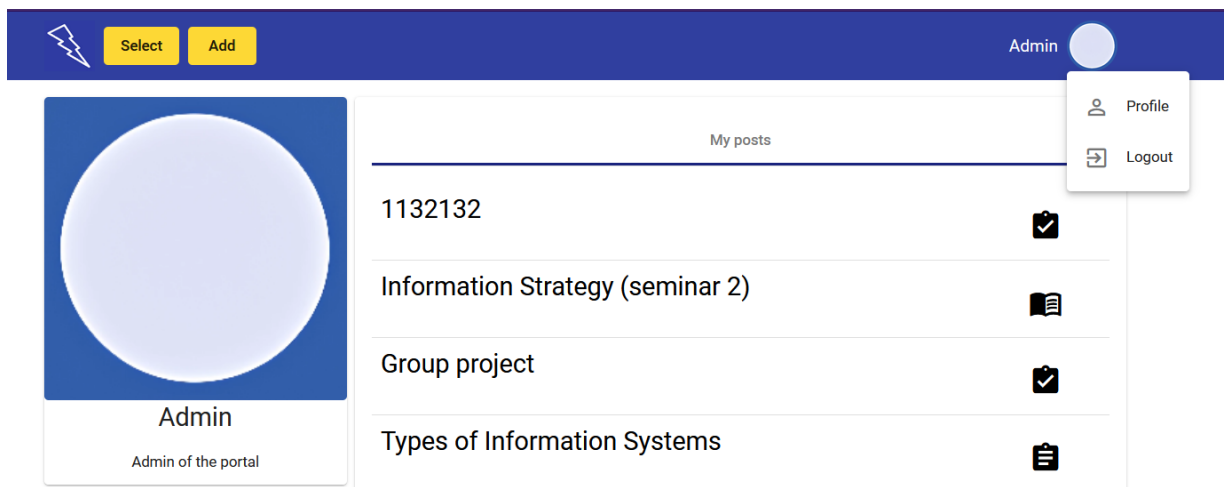


Figure 12. Screenshot of "Profile" page. Source: Author

At the core of the Profile Page is the display of the user's basic information, such as their name, profile image, and any other relevant details. This personalization creates a sense of identity within the application and allows users to quickly access their own information.

One of the primary features of the Profile Page is the presentation of the user's posts. By displaying a list of posts created by the user, the Profile Page enables easy content navigation and discovery, allowing users to revisit their contributions or explore the posts of others. This functionality fosters an environment of knowledge sharing and collaboration among users.

In conclusion, the Profile Page serves as a central hub for users to access their personal information, profile image, and the content they have contributed to the platform. By providing

these features in a user-friendly and accessible manner, the Profile Page enhances the overall user experience and promotes engagement within the application.

4.2 The process of building the application.

As mentioned above, it is a relatively hard task to build an application that uses microservices architecture from scratch. Usually, the task of the new project is to build the software as soon as possible to receive the fast feedback from users. This Agile approach has become a mainstream approach in most startup projects.

For our application we used the same approach and built the project as a monolith at first. Then we started separating it in microservices. The monolith application had PostgreSQL as the only database source. We utilized the approach of creating “microlith” described in theoretical part of thesis.

Structure after building monolith

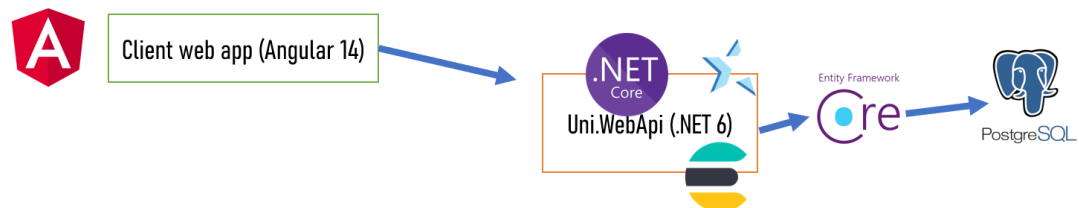


Figure 13. Structure of the project after building a monolith. Source: Author

Firstly, we decided on boundaries of the application separating the logic into two big groups:

1. Managing users, universities, and logistic data.
2. Managing posts and their content. Secondly, we created two projects for those needs called “Uni” and “Posts”.

For the third step, the façade projects for respecting services were created. We added “Posts.Facade” project as a reference to “Uni” project and “Uni.Facade” project as a reference to the “Posts” project.

Also, separate database tables with “Post_” prefixes were created: “Post_Posts” and “Post_Files”. Which were be heavily used for the “Posts” project. All other tables were primary used for “Uni” project.

This allowed us to easily separate those two projects into two completely independent microservices at the final step of our development, that can be independently scaled.

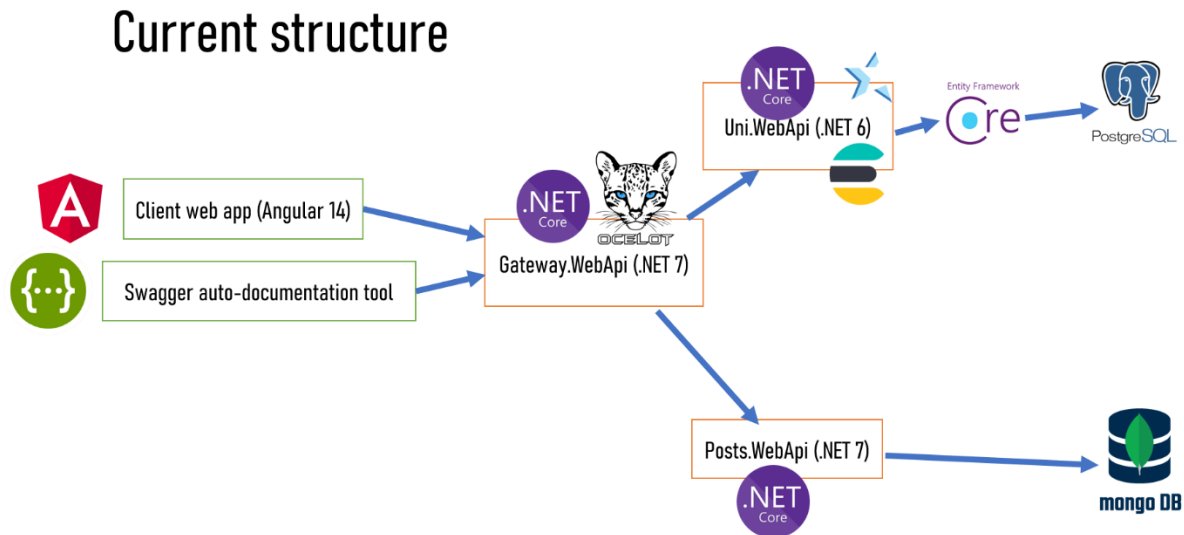


Figure 14. Structure of the project after separating it into microservices. Source: Author

In the end, we got a nice application separated in modules.

4.3 Overview of the application: Client side

The Angular application developed as part of this thesis is designed to provide a user-friendly and efficient platform for managing educational resources and information. The application incorporates all the features mentioned in the previous sections, with each feature implemented as a separate module to promote modularity, maintainability, and scalability. Additionally, the application utilizes the NgRx library to manage the state and enhance the

overall performance and responsiveness of the application. Although, when people talk about microservices, they usually talk about server-side applications, there are ways to separate the client side on microservices too. However, usually, it provides less benefits than utilizing microservices architecture on the server side of the application. The more mainstream approach to microservices would be to separate the client side on independent modules that would get uploaded using lazy loading.

```
import { AuthenticationGuard } from './core/modules/authentication/services/guards/authentication.guard';

const routes: Routes = [
  {
    path: 'authorization',
    component: AuthComponent,
  },
  {
    path: 'registration',
    component: RegisterComponent,
  },
  {
    path: 'profile',
    canActivate: [AuthenticationGuard],
    loadChildren: () => import('./modules/profile/profile.module').then(m => m.ProfileModule),
  },
  {
    path: 'select',
    canActivate: [AuthenticationGuard],
    loadChildren: () => import('./modules/search/search.module').then(m => m.SearchModule),
  },
  {
    path: 'post',
    loadChildren: () => import('./modules/post/post.module').then(m => m.PostModule),
  },
  {
    path: '***',
    redirectTo: 'authorization',
    pathMatch: 'full',
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { relativeLinkResolution: 'legacy' })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Figure 15. Source code of Routing for modules in Angular 14 app. Source: Author

The created application's architecture is organized into the following modules:

1. **Authentication Module:** This module is responsible for handling user registration, login, and JWT-based authentication, ensuring that only authorized users can access the application's features and content.
2. **Profile Module:** The Profile module is designed to display user profile, displaying personal information such as name and profile image. Also, it allows to
3. **Add Module:** This module facilitates the creation of new posts, providing users with a structured form to input relevant information, including title, description, student's group, semester number, course number, and post type. Additionally, it supports the attachment of various file types such as documents, images, and media files.
4. **Select Module:** The Select module enables users to discover and navigate content by filtering posts based on criteria such as country, city, university, and subject. The module presents the results in a list grouped by semester and student's group, providing an organized and intuitive browsing experience.
5. **NgRx Store:** The application incorporates the NgRx library, which is an implementation of the Redux pattern for Angular applications. NgRx helps manage the application state in a predictable and efficient manner by using a unidirectional data flow and a centralized store (NgRx, 2021). The store is responsible for managing the state of various application features, such as authentication, user profiles, and content.

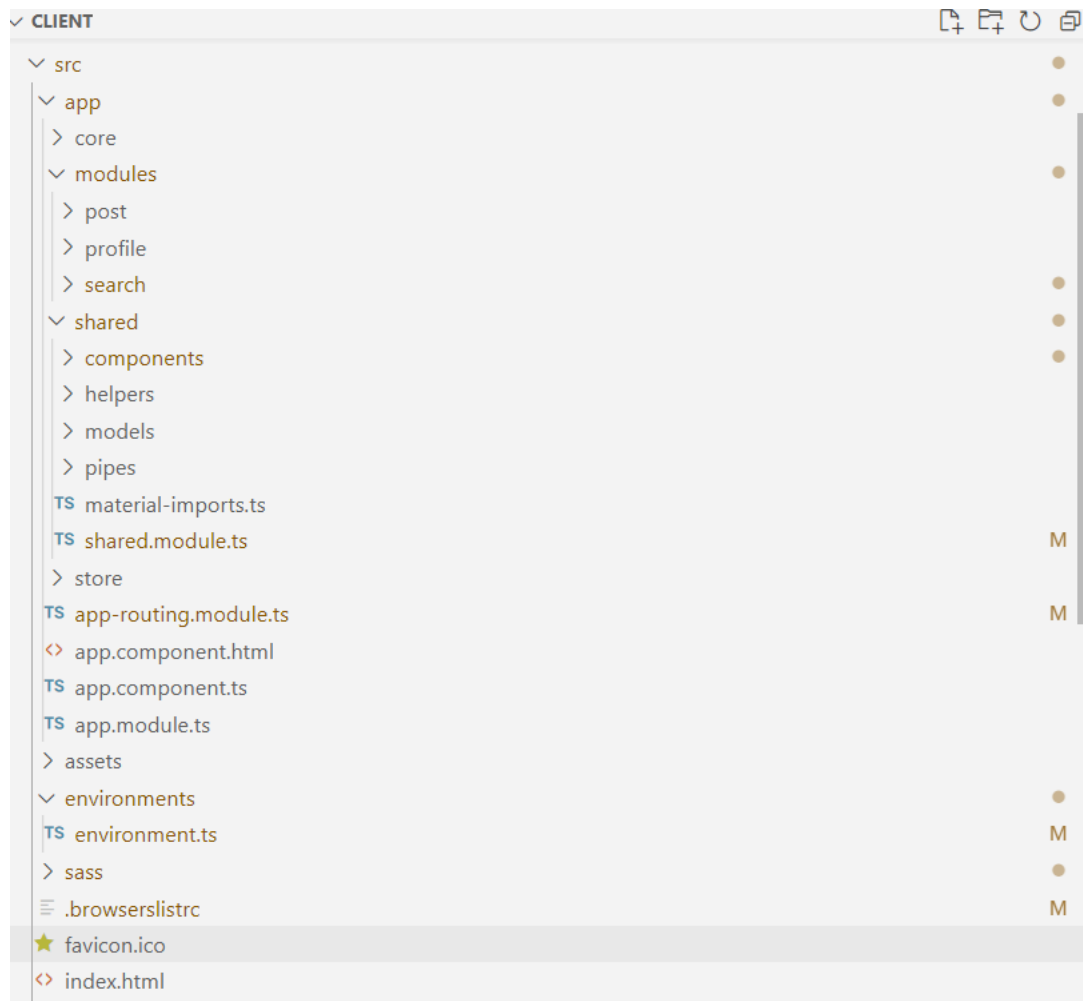


Figure 16. Screenshot of the Client project structure in VS Code. Source: Author

By implementing features as separate modules, the application adheres to the principles of modularity and separation of concerns, which promotes maintainability and scalability in the long run. Furthermore, the use of NgRx for state management ensures that the application remains performant and responsive, even as the complexity and size of the application grow.

4.4 Overview of the Application - Server Side

The server-side of the application is built using a microservices architecture, which breaks the application down into smaller, independent services that communicate with each other through APIs. This approach improves scalability, maintainability, and fault tolerance, as

each service can be developed, deployed, and managed independently. The server-side application is composed of three primary services: Gateway.WebApi, Post.WebApi, and Uni.WebApi.

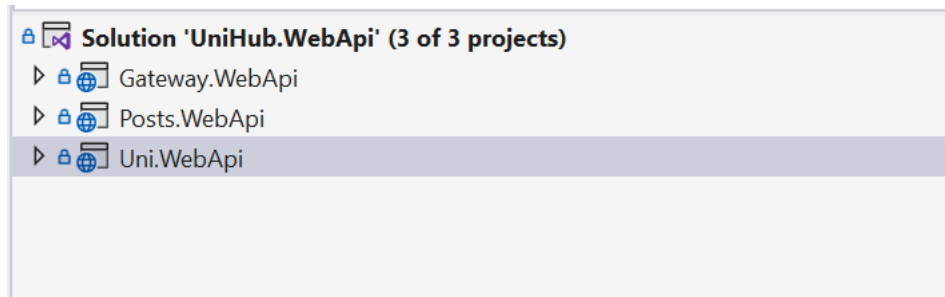


Figure 17. Screenshot of the final version of project architecture in Visual Studio. Source: Author

This section will provide an in-depth overview of each service, their roles, and their interactions within the server-side architecture.

The server-side application architecture leverages the strengths of both the .NET framework and the chosen databases, MongoDB, and PostgreSQL, to create a robust and scalable solution. By using a microservices approach and separating the application into distinct services, the application can more easily adapt to changing requirements and grow with the project's needs.

4.4.1 Uni.WebApi

Uni.WebApi is a dedicated microservice responsible for managing university-related logic and user management within the application. This service encapsulates functionalities such as user profiles, student groups, courses, and user authentication. By segregating these features into a separate service, the application's architecture remains modular and maintainable. This section will provide a comprehensive overview of the Uni.WebApi microservice, its responsibilities, and its interactions within the microservices architecture.

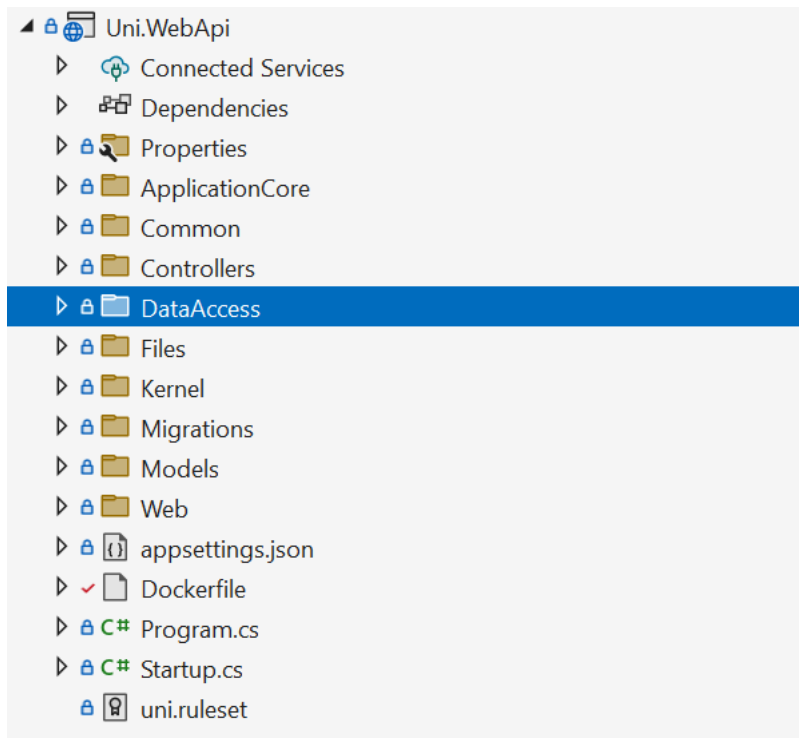


Figure 18. Screenshot of the Uni.WebApi project structure in Visual Studio. Source: Author

On the picture you can see that the microservice is carefully separated into layers. It respects a typical 3-layered architecture of building services, where the service is divided into 3 parts: Web, Application and Data Access.

“ApplicationCore” folder consist of Services that manage courses, universities etc. The "ApplicationCore" folder in the Uni.WebApi solution contains the core functionality and business logic required for the application to operate. It consists of three subfolders: "Constants", "Helpers", and "Services". The "Constants" subfolder contains files that define constants and configuration settings used throughout the application. The "Helpers" subfolder provides utility functions and classes that are used across the application, while the "Services" subfolder contains the core business logic, organized into separate files or classes based on their functionality. By centralizing this functionality in a single location, developers can maintain a clean and organized codebase that is easier to modify and maintain. Also, it includes interfaces to those services, because the application respects Dependency Inversion principle of SOLID principles.

```

2 references
public async Task<ServiceResult<UserDto>> UpdatePasswordAsync(int userId, UpdatePasswordRequest request)
{
    User user = await this._unitOfWork.UserRepository.GetSingleAsync(up => up.Id == userId);

    if (user == null)
    {
        return ServiceResult<UserDto>.Fail(EOperationResult.EntityNotFound,
            "User doesn't exist");
    }

    if (!string.IsNullOrEmpty(request.NewPassword))
    {
        if (!Authenticate.Verify(request.CurrentPassword, user.PasswordHash))
        {
            return ServiceResult<UserDto>.Fail(EOperationResult.ValidationError,
                "Current password is incorrect");
        }

        user.PasswordHash = Authenticate.Hash(request.NewPassword);
    }

    await this._unitOfWork.CommitAsync();

    return ServiceResult<UserDto>.Ok(this._mapper.Map<User, UserDto>(user));
}

```

Figure 19. Example of the method in the Service file. Source: Author

“Common” folder includes options for authorization, files management and different constants that are configured at the start of application.

“Controllers” folder basically consists of controllers that expose REST API to different consumers. Each controller is responsible for handling HTTP requests and returning HTTP responses. Controllers map HTTP requests to Service calls and return data in the appropriate Response model.

The "DataAccess" folder in the Uni.WebApi solution is an essential component of the application's architecture, responsible for handling data access and database-related tasks. The folder contains repositories and other files related to the database. Repositories are used to encapsulate the logic for interacting with the database. Each repository corresponds to a particular entity in the data model and provides methods for querying, inserting, updating, and deleting data from that entity. Repositories abstract away the details of the underlying data storage mechanism and provide a simple, easy-to-use interface for working with data. Files folder is the place where user's media, documents and images are getting uploaded. It also includes default pictures.

```
2 references
public async Task<IEnumerable<Subject>> GetSubjectsByFacultyWithTeachersAsync(int facultyId, int skip, int take)
{
    return await _dbContext.Subjects
        .Include(s => s.Teacher)
        .Where(s => s.FacultyId == facultyId)
        .OrderBy(s => s.Title)
        .Skip(skip).Take(take)
        .ToListAsync();
}
```

Figure 20. Example of the method in a repository class. Source: Author

“Kernel” folder includes database Entities. Entities in the "Kernel" folder are the starting point for building the data model used by the application. They represent the core concepts and relationships of the domain model, and serve as the foundation for the rest of the application's functionality.

The "Migrations" folder in the Uni.WebApi solution contains database migrations created by Entity Framework. Our usage of Entity Framework follows a "Code-first" approach, which means that we define the data model in code first, and then use Entity Framework to create the corresponding database schema and migrations. In other words, we create C# classes that represent our data model, including entities and relationships between them, and then Entity Framework generates the database schema and migrations based on those classes. This approach allows us to focus on the domain model and the business logic of our application, while Entity Framework handles the low-level details of database schema and migrations. The "Migrations" folder contains a series of migration files, each of which represents a change to the database schema. These migrations are applied to the database in order, allowing us to keep the database schema in sync with the data model defined in code. This approach makes it easy to evolve the database schema over time as the requirements of the application change.

	Id [PK] integer	CreatedAtUtc timestamp without time zone	FullTitle character varying (100)	ShortTitle character varying (7)	Description text
1	1	2023-03-26 16:02:25.55457	Czech University of Life Sciences 0	CULS	The actual history of our
2	2	2023-03-26 16:02:25.55457	Czech University of Life Sciences 1	CULS	The actual history of our
3	3	2023-03-26 16:02:25.55457	Czech University of Life Sciences 2	CULS	The actual history of our
4	4	2023-03-26 16:02:25.55457	Czech University of Life Sciences 3	CULS	The actual history of our
5	5	2023-03-26 16:02:25.55457	Czech University of Life Sciences 4	CULS	The actual history of our
6	6	2023-03-26 16:02:25.55457	Czech University of Life Sciences 5	CULS	The actual history of our
7	7	2023-03-26 16:02:25.55457	Czech University of Life Sciences 6	CULS	The actual history of our
8	8	2023-03-26 16:02:25.55457	Czech University of Life Sciences 7	CULS	The actual history of our
9	9	2023-03-26 16:02:25.55457	Czech University of Life Sciences 8	CULS	The actual history of our
10	10	2023-03-26 16:02:25.55457	Czech University of Life Sciences 9	CULS	The actual history of our

Total rows: 16 of 16 Querv complete 00:00:00.277

Figure 21. Stored universities in PostgreSQL database. Source: Author

The "Models" folder in the Uni.WebApi solution is a critical component of the application's architecture. It contains models for all layers of the application, including Request models, Service models, and Response models. Request models represent the data that is received by the application through HTTP requests. These models are used to validate and sanitize input data, and to map the data to the appropriate Service model. Service models represent the domain entities and business logic of the application. These models are used by the Services in the "Services" folder to operate on data, perform calculations, or implement algorithms. Response models represent the data that is returned by the application in response to HTTP requests. These models are used to map Service models to a format that can be easily consumed by clients of the API. By separating the models into separate folders based on their use case, we can ensure that each model is responsible for a specific aspect of the application's functionality. This makes the code easier to understand and maintain, and also allows us to reuse models across multiple parts of the application.

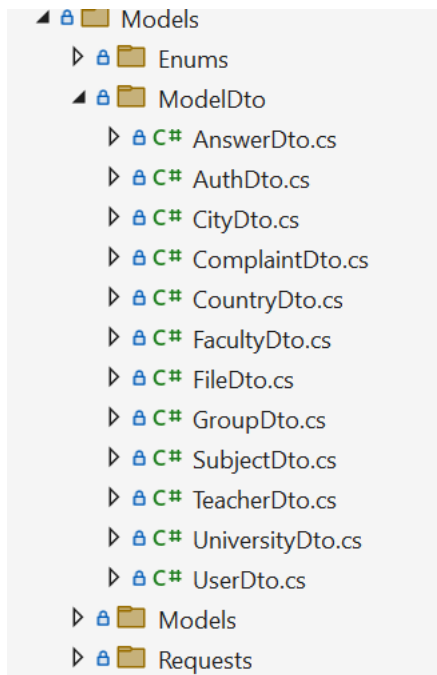


Figure 22. Models folder structure. Source: Author

Uni.WebApi manages user profiles, which store essential information such as name, email address, and user image. This service also handles user registration, allowing new users to create accounts and log in to the application. Uni.WebApi ensures that user data is securely stored and retrieved, maintaining the privacy and integrity of user information.

Uni.WebApi is responsible for managing student groups and courses, enabling the application to organize and display information relevant to each group and course. By handling these entities within a dedicated service, the application can maintain a clear separation of concerns and simplify the process of updating or modifying group and course information.

Uni.WebApi manages user authentication, utilizing JWT authorization to securely validate user credentials and grant appropriate access permissions based on user roles. By storing JWT tokens in local storage, the application can efficiently manage user sessions and ensure that users are authenticated for subsequent requests. Role-based access control enables the application to provide different levels of access and privileges to users, such as administrators having the ability to validate posts or delete any post.

```

public static void AddJwtAuth(this IServiceCollection services, IConfiguration configuration)
{
    services.Configure<TokenOptions>(configuration.GetSection("Token"));
    var tokenOptions = configuration.GetSection("Token").Get<TokenOptions>();

    JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear(); // remove default claims
    services
        .AddAuthentication(options =>
            {
                options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
                options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
                options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
            })
        .AddJwtBearer(cfg =>
            {
                cfg.TokenValidationParameters = new TokenValidationParameters
                {
                    ValidateIssuer = tokenOptions.ValidateIssuer,
                    ValidateLifetime = tokenOptions.ValidateLifetime,
                    ValidateIssuerSigningKey = tokenOptions.ValidateIssuerSigningKey,
                    ValidateAudience = tokenOptions.ValidateAudience,
                    ClockSkew = TimeSpan.Zero,

                    ValidIssuer = tokenOptions.Issuer,
                    ValidAudience = tokenOptions.Audience,
                    IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(tokenOptions.IssuerSecurityKey))
                };
            });
}

```

Figure 23. Adding JWT Authorization to a pipeline. Source: Author

Uni.WebApi uses PostgreSQL as its primary database, a powerful and reliable open-source relational database management system (RDBMS). PostgreSQL's strong ACID compliance and support for complex data types make it a suitable choice for handling the structured data associated with university entities and user profiles. By using PostgreSQL for Uni.WebApi, the application can leverage its performance and scalability benefits while maintaining data integrity and consistency.

```

public UnitOfWork(UniHubDbContext dbContext,
    IFacultyRepository facultyRepository,
    IFileRepository fileRepository,
    IUniversityRepository universityRepository,
    ISubjectRepository subjectRepository,
    IUserRepository userRepository,
    ICountryRepository countryRepository,
    ICityRepository cityRepository,
    ITeacherRepository teacherRepository,
    IRefreshTokenRepository refreshTokenRepository,
    IGroupRepository groupRepository)
{
    RefreshTokenRepository = refreshTokenRepository;
    FacultyRepository = facultyRepository;
    FileRepository = fileRepository;
    UniversityRepository = universityRepository;
    SubjectRepository = subjectRepository;
    UserRepository = userRepository;
    CountryRepository = countryRepository;
    CityRepository = cityRepository;
    TeacherRepository = teacherRepository;
    GroupRepository = groupRepository;

    this._dbContext = dbContext;
}

15 references
public async Task CommitAsync()
{
    await this._dbContext.SaveChangesAsync();
}
}

```

Figure 24. Unit of work pattern implementation. Source: Author

One of the key features of Uni.WebApi is its ability to handle file attachments, allowing users to attach documents, images, and media to their posts. The service provides support for nearly unlimited file attachments, ensuring that users can include as much relevant content as needed to support their posts. Uni.WebApi manages the storage, retrieval, and processing of these files, maintaining the application's performance and ensuring that file data is securely stored.

Uni.WebApi leverages the MagickImage library to adjust picture sizes for the downloaded files. This powerful image processing library enables the application to efficiently handle image resizing and manipulation, ensuring that images are optimized for display within the application. By utilizing the MagickImage library, Posts.WebApi can provide a seamless and efficient user experience when managing images within posts.

```
2 references
public async Task<ServiceResult<string>> UploadImageAsync(IFormFile imageFile)
{
    string extension = Path.HasExtension(imageFile.FileName)
        ? Path.GetExtension(imageFile.FileName)
        : string.Empty;

    if (!this._imageExtensions.Contains(extension))
    {
        return ServiceResult<string>.Fail(EOperationResult.ValidationError, "Invalid extension");
    }

    string imageName = $"image_{Guid.NewGuid()}";

    string relativeFolderPath = $"{this._fileOptions.UploadFolder}/{this._fileOptions.InnerFolders.ImagesFolder}";
    string fullPath = Path.Combine(this._hostingEnvironment.ContentRootPath, relativeFolderPath);

    string fileName = $"{imageName + extension}";
    fullPath = Path.Combine(fullPath, fileName);

    if (imageFile.Length > ImageFileMaxSize)
    {
        using (MagickImage image = new MagickImage(imageFile.OpenReadStream()))
        {
            MagickGeometry size = new MagickGeometry(PixelSize) { IgnoreAspectRatio = false };
            image.Resize(size);
            image.Write(fullPath);
        }
    }
    else
    {
        using (var fileStream = new FileStream(fullPath, FileMode.Create))
        {
            await imageFile.CopyToAsync(fileStream);
        }
    }

    string urlPath = Path.Combine(this._urlOptions.ServerUrl, relativeFolderPath, fileName);

    return ServiceResult<string>.Ok(urlPath);
}
2 references
```

Figure 25. Adding picture code implementation. Source: Author

4.4.2 Gateway.WebApi

Gateway.WebApi serves as the API gateway for the application, acting as an intermediary between clients and the underlying microservices, such as Posts.WebApi and Uni.WebApi. By utilizing Ocelot, a lightweight and flexible API gateway library, Gateway.WebApi handles the routing and redirection of requests, load balancing, and communication between services. This section will provide a detailed overview of the Gateway.WebApi microservice, its primary responsibilities, and its role within the microservices architecture.

```

1  {
2  "ReRoutes": [
3  {
4  "DownstreamPathTemplate": "/v1/{url}",
5  "DownstreamScheme": "http",
6  "DownstreamHostAndPorts": [
7  {
8  "Host": "localhost",
9  "Port": 5000
10 }
11 ],
12 "UpstreamPathTemplate": "/v1/{url}"
13 },
14 {
15 "DownstreamPathTemplate": "/postsService/{url}",
16 "DownstreamScheme": "http",
17 "DownstreamHostAndPorts": [
18 {
19 "Host": "localhost",
20 "Port": 5008
21 }
22 ],
23 "UpstreamPathTemplate": "/postsService/{url}"
24 }
25 ],
26 "DangerousAcceptAnyServerCertificateValidator": true,
27 "GlobalConfiguration": {
28 "BaseUrl": "http://localhost:5004"
29 }
30 }
31 }
32 }

```

Figure 26. Ocelot configurations. Source: Author

One of the main functions of Gateway.WebApi is to route and redirect client requests to the appropriate microservices. Ocelot is employed to configure the routing rules, ensuring that requests are accurately directed to either Posts.WebApi or Uni.WebApi based on the request's path and parameters. By centralizing request routing through Gateway.WebApi, the application can maintain a clear separation of concerns and simplify communication between clients and services.

Gateway.WebApi acts as the primary communication layer between clients and the Posts.WebApi and Uni.WebApi microservices. By routing requests to the appropriate service based on the request's content and context, Gateway.WebApi enables seamless interaction between the various components of the application. This centralized communication model simplifies the overall architecture and promotes maintainability and modularity.

4.4.3 Post.WebApi

Posts.WebApi is a specialized microservice designed to handle post-related logic within the application. It is responsible for managing and organizing posts, including creating, updating, deleting, and retrieving posts. By isolating post-related functionalities within a dedicated service, the application's architecture remains modular, maintainable, and scalable.

This section will provide an in-depth overview of the Posts.WebApi microservice, its primary responsibilities, and its interactions within the microservices architecture.

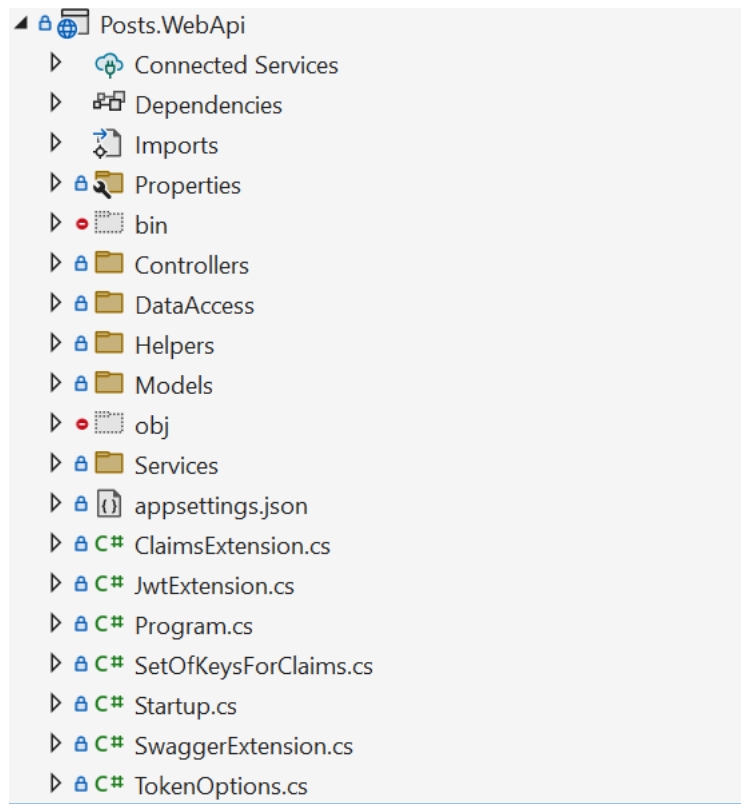


Figure 27. Screenshot of the Posts.WebApi project structure in Visual Studio. Source: Author

Posts.WebApi follows a similar structure as Uni.WebApi. It consists of similar concepts. However, it contains slightly less complex infrastructure compared to Uni.WebApi. It uses the capabilities of Mondo DB. All core “entities” of database files are quite complex and have relatively a lot of logic inside of them following the best practice of not using Anemic Domain Models, which is considered an anti-pattern by Martin Fowler (Fowler, 2003).

Posts.WebApi is responsible for managing and organizing posts, which can be examples, tasks, or materials. Each post contains essential information such as title, description, student's group, semester number, course number, and type of post. Posts.WebApi enables the application to efficiently store, retrieve, and display post information, ensuring that the data remains organized and easily accessible.

```
2 references
public async Task<IEnumerable<Post>> GetPostsBySubjectAsync(int subjectId,
    int groupId = 0, int? semester = 0, int? course = 0, int skip = 0, int take = 0)
{
    var posts = await this._mongoDatabase.GetCollection<Post>(typeof(Post).Name).Find(p => p.SubjectId == subjectId)
        .SortByDescending(s => s.ModifiedAtUtc)
        .Skip(skip)
        .Limit(take)
        .ToListAsync();

    IEnumerable<Post> postEnumerable = posts;

    Func<Post, bool> predicate = p => p.SubjectId == subjectId;

    var postsQuery = this._mongoDatabase.GetCollection<Post>(typeof(Post).Name).Find(p => p.SubjectId == subjectId && p.Course == cour:
    postsQuery = postsQuery.SortByDescending(s => s.ModifiedAtUtc);

    if (skip != 0)
    {
        postsQuery = postsQuery.Skip(skip);
    }

    if (take != 0)
    {
        postsQuery = postsQuery.Limit(take);
    }

    return await postsQuery.ToListAsync();
}

2 references
public async Task<IEnumerable<PostBySemesterGroup>> GetInitialGroupedPostsBySubjectAsync(int subjectId,
    string title = "", int groupId = 0, int? semester = 0, int? course = 0, EPostValueType? valueType = null,
```

Figure 28. Working with Mongo.DB. Source: Author

Posts.WebApi uses MongoDB as its primary database, a flexible and scalable open-source NoSQL database management system. MongoDB's document-oriented storage model is well-suited for handling the complex and varied data associated with posts and file attachments. By using MongoDB for Posts.WebApi, the application can take advantage of its high-performance capabilities and support for horizontal scaling, ensuring that the service remains performant and scalable as the application grows.


```
_id: ObjectId('6421edb6475f3d07dd34fa59')
CreatedAtUtc: 0001-01-01T00:00:00.000+00:00
Title: "Information Strategy (seminar 2)"
Description: "Seminar 2 (even):
            - team declaration
            - selecting topic (business type..."
Course: 2
Semester: 0
GivenAt: null
LastVisit: 2023-03-27T19:25:42.139+00:00
ModifiedAtUtc: 2023-03-27T19:25:42.139+00:00
DeletedAtUtc: null
ValidatedAtUtc: null
PostValueTypeId: 3
GroupId: 2
UserId: 1
SubjectId: 1
Files: Array
```

```
_id: ObjectId('6421f34d475f3d07dd34fa5a')
CreatedAtUtc: 0001-01-01T00:00:00.000+00:00
Title: "Group project"
```

Figure 29. Saved post in MongoDB. Source: Author

5 Results and discussions

The development of a university-related application using microservices architecture, as described in previous sections, has provided valuable insights into the process and best practices for creating a "microlith" application. In this section, we will discuss the results obtained from the implementation of microservices architecture in the application, the set of recommendations that have been established, and potential ways to improve the application further.

Based on the practical experience gained from developing the university-related application, several recommendations for building a "microlith" have been identified. These guidelines can serve as a starting point for developers and organizations looking to transition from a monolithic application to a microservices-based architecture.

The university-related application has successfully implemented the microservices architecture, with the Gateway.WebApi, Posts.WebApi, and Uni.WebApi services working together to deliver a seamless user experience. By adhering to the recommendations listed above and leveraging technologies such as .NET, Angular, and Ocelot, the application has demonstrated the benefits of microservices in terms of scalability, maintainability, and flexibility.

Despite the successful implementation of microservices, there is always room for improvement. Some potential enhancements for the university-related application include:

- **Enhancing security:** Implementing more robust security measures, such as rate limiting, and improving user authentication and authorization mechanisms.
- **Expanding functionality:** Adding new features, such as real-time notifications, to further enhance the user experience and encourage engagement within the application.
- **Optimizing performance:** Continuously monitoring the application's performance and implementing performance optimization techniques, such as caching, to ensure optimal user experience.

- Streamlining deployment: Adopting containerization technologies, such as Docker and Kubernetes, to simplify deployment and management of microservices (Microsoft, 2021e).

6 Conclusions

Throughout this thesis, we have explored the microservices architecture and its implementation in the context of a university-related application using the .NET framework and Angular. By examining the best practices for building applications using this architecture and discussing various technologies and tools, such as Ocelot, Swagger, and Entity Framework, we have gained valuable insights into the benefits and challenges associated with the microservices approach.

The practical implementation of the application demonstrated the effectiveness of using microservices architecture, .NET framework, and Angular for building scalable and maintainable server-client applications. By adhering to best practices and leveraging the features provided by these technologies, we were able to develop a responsive, high-performance application that met the requirements of managing university-related information and facilitating communication.

The development of the university-related application using microservices architecture has offered valuable insights and demonstrated the benefits of transitioning to a "microlith" application. By following the recommendations provided and continuously seeking ways to improve the application, developers and organizations can harness the power of microservices to create scalable, maintainable, and flexible applications.

This thesis has showcased the potential of microservices architecture in creating scalable, maintainable, and flexible applications, highlighting the importance of adopting best practices and leveraging appropriate technologies. As the software development landscape continues to evolve, the microservices architecture will undoubtedly play a critical role in shaping the future of application development and empowering developers to build more robust and efficient solutions.

7 References

1. ALBAHARI, J., & Albahari, B. *C# 9.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, 2020. ISBN 978-1-492-05561-3.
2. BIERMAN, G., Abadi, M., & Torgersen, M. *Understanding TypeScript*. In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014). ACM. DOI: 10.1145/2660193.2660195.
3. CHACON, S., & STRAUB, B.. *Pro Git*. Apress, 2014. ISBN 978-1484200773.
4. DRAGONI, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Communications of the ACM*, 60(6), 80-90. DOI: 10.1145/3075565.
5. EVANS, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional; 1st edition (August 20, 2003). ISBN-13: 978-0321125217.
6. FERRON, J. *Learning C# by Developing Games with Unity 2020: An enjoyable and intuitive approach to getting started with C# programming and Unity*. 2020 Packt Publishing. ISBN 978-1-83921-799-5.
7. FLANAGAN, D. (2020). *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language (7th ed.)*. O'Reilly Media. ISBN 978-1491952023.
8. FOWLER M, *Anemic Domain Model* [online] 25 November 2003 <https://martinfowler.com/bliki/AnemicDomainModel.html>
9. Freeman, A. *Pro ASP.NET Core 3: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages*. Apress. (2020). ISBN 978-1-4842-4810-9.
10. Google (2021). Angular - Introduction. [online] Available at: <https://angular.io/docs> [Accessed 28 March 2023].
11. GRZYBEK K. “modular-monolith-with-ddd” Repository [online] 23 August 2022 <https://github.com/kgrzybek/modular-monolith-with-ddd#11-purpose-of-this-repository> Accessed 23 August 2019
12. HEJLSBERG, A., Wiltamuth, S., & Golde, P. *The C# Programming Language*. Addison-Wesley, 2003. 978-0321334435

13. HORSDAL, Christian. *Microservices in .NET Core: with examples in Nancy* 1st Edition. Manning; 1st edition (February 3, 2017). ISBN-13: 978-1617293375
14. HUMBLE, J., FARLEY, D. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Professional, 2010. ISBN: 978-0321601919.
15. Hussain, A. (2018). *Angular: From Theory to Practice*. [online] 21 July 2018. <https://github.com/JooYoo/Books-Angular/blob/master/angular-from-theory-to-practice.pdf>
16. ImageMagick Studio LLC (2021). *ImageMagick: Convert, Edit, or Compose Bitmap Images*. [online]. Available at: <https://imagemagick.org/>. [Accessed 28 March 2023].
17. ImageMagick Studio LLC (2021). *ImageMagick: Convert, Edit, or Compose Bitmap Images*. [online] Available at: <https://imagemagick.org/>. [Accessed 28 March 2023].
18. JONES, M., BRADLEY, J., & SAKIMURA, N. *JSON Web Token (JWT)*. Internet Engineering Task Force. [online] 2015. <https://tools.ietf.org/html/rfc7519>. Accessed 28 Marth 2023
19. Jwt.io (2021). *JSON Web Tokens - jwt.io*. [online] Available at: <https://jwt.io/>. [Accessed 28 March 2023].
20. LEWIS J, FOWLER, M. (2014). *Microservices*. [online]. 25 March 2014. <https://martinfowler.com/articles/microservices.html>. Accessed 28 Marth 2023.
21. Lock, A. (2019). *ASP.NET Core in Action*. Manning Publications, 2019. ISBN • 9781617298301
22. MARTIN, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education. ISBN 0-13-597444-5.
23. Microsoft (2021). *TypeScript Handbook*. Retrieved from <https://www.typescriptlang.org/docs/handbook/intro.html>
24. Microsoft (2021a). *Introduction to ASP.NET Core*. [Online]. Available at: <https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0>. Accessed 28 March 2023.

25. Microsoft (2021b). Announcing .NET 7 Preview 1. [online]. Available at: <https://devblogs.microsoft.com/dotnet/announcing-net-7-preview-1/>. [Accessed 28 March 2023].
26. Microsoft (2021c). Blazor: Build client web apps with C#. [online]. Available at: <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>. Accessed 28 March 2023
27. Microsoft (2021d). Visual Studio: Write your code fast, debug and diagnose with ease, test often, and release with confidence. [online] Available at: <https://visualstudio.microsoft.com/>. [Accessed 28 March 2023].
28. Microsoft (2021e). Visual Studio Code: Code editing redefined and optimized for building and debugging modern web and cloud applications. [online] Available at: <https://code.visualstudio.com/>. [Accessed 28 March 2023].
29. Microsoft (2021f). Entity Framework Core. [online] Available at: <https://docs.microsoft.com/en-us/ef/core/>. [Accessed 28 March 2023].
30. Microsoft (2021g). ASP.NET Core Identity. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity>. [Accessed 28 March 2023].
31. Microsoft (2021h). JSON Web Tokens (JWT) in ASP.NET Core. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/jwt-bearer>. [Accessed 28 March 2023].
32. Microsoft (2021i). Role-based authorization in ASP.NET Core. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles>. [Accessed 28 March 2023].
33. Microsoft (2021j). Local Storage in Blazor. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/blazor/state-management>. [Accessed 28 March 2023].
34. Microsoft (2021k). ASP.NET Core Razor Pages. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/razor-pages>. [Accessed 28 March 2023].
35. Microsoft (2021l). Middleware in ASP.NET Core. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware>. [Accessed 28 March 2023].

36. Microsoft (2021m). ASP.NET Core Web API: Create web APIs with ASP.NET Core. [online] Available at: <https://dotnet.microsoft.com/apps/aspnet/apis>. [Accessed 28 March 2023].
37. MOMJIAN, B. PostgreSQL: *Introduction and Concepts*. Addison-Wesley Professional, 2010. ISBN 0-201-70331-9.
38. MongoDB Inc. (2021). MongoDB - The database for modern applications. [online] Available at: <https://www.mongodb.com/> [Accessed 28 March 2023].
39. NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, February 2015. ISBN 978-1-491-95035-7.
40. NgRx (2021). NgRx: Reactive State for Angular. [online] Available at: <https://ngrx.io/>. [Accessed 28 March 2023].
41. NgRx (2021). NgRx: Reactive State for Angular. [online]. Available at: <https://ngrx.io/>. [Accessed 28 March 2023].
42. OBE, R., & HSU, L. PostgreSQL: Up and Running. O'Reilly Media, November 2014. ISBN 978-1-449-39368-9.
43. OWASP. (2021). Cross-Site Scripting (XSS). [online] Available at: <https://owasp.org/www-community/attacks/xss/>. [Accessed 28 March 2023].
44. OZKAYA M. [online] 14 April 2019 <https://github.com/aspnetrun/run-aspnetcore-microservices>. Accessed 28 Marth 2023.
45. PostgreSQL Global Development Group (2021). PostgreSQL: The World's Most Advanced Open Source Database. [Online]. Available at: <https://www.postgresql.org/> [Accessed 28 March 2023].
46. RICHARDSON, C. *Microservices Patterns: With examples in Java*. O'Reilly Media. (2018). ISBN 978-1-491-93684-6.
47. Richter, J. *CLR via C# (4th ed.)*. Microsoft Press. (2012). ISBN 978-0-7356-6745-7.
48. ROSENBERG, D. *Effective TypeScript: 62 Specific Ways to Improve Your TypeScript*. O'Reilly Media. 2019. ISBN-13 : 978-1492053743.
49. SCHÖNIG, H.-J. (2017). *Mastering PostgreSQL 9.6: An expert guide to implementing advanced PostgreSQL 9.6 functionalities and applications*. Packt Publishing, 2017. ISBN: 9781783555352.

50. Swagger (2021). Swagger: Simplify API development for users, teams, and enterprises with the Swagger open source and professional toolset. [online]. Available at: <https://swagger.io/>. [Accessed 25 March 2023].
51. Threefold (2021). Ocelot: .NET API Gateway. [online] Available at: <https://ocelot.readthedocs.io/en/latest/> [Accessed 28 March 2023].
52. WHITESELL, Sean. Pro Microservices in .NET 6: With Examples Using ASP.NET Core 6, MassTransit, and Kubernetes. Apress; 1st ed. edition (January 2, 2022). ISBN-13:978-1484278321

8. List of figures.

Figure 1. An example of the project structure. Source: (Ozkaya, 2019)	29
Figure 2. Sign in page screenshot. Source: Author.....	37
Figure 3. Registration page screenshot. Source: Author	37
Figure 4. Screenshot of role-based behavior. Source: Author	39
Figure 5. Screenshot of the "Add" page. Source: Author	40
Figure 6. Screenshot of the selection of country and city. Source: Author	42
Figure 7. Screenshot of selection of university. Source: Author	43
<i>Figure 8. Screenshot of selecting a faculty. Source: Author</i>	<i>44</i>
Figure 9. Screenshot of selecting a subject process. Source: Author	44
Figure 10. Screenshot of selecting a post process. Source: Author	45
Figure 11. Screenshot of "View post" page. Source: Author	46
Figure 12. Screenshot of "Profile" page. Source: Author.....	47
Figure 13. Structure of the project after building a monolith. Source: Author.....	48
Figure 14. Structure of the project after separating it into microservices. Source: Author	49
.....
Figure 15. Source code of Routing for modules in Angular 14 app. Source: Author ..	50
Figure 16. Screenshot of the Client project structure in VS Code. Source: Author	52
Figure 17. Screenshot of the final version of project architecture in Visual Studio. Source: Author	53
Figure 18. Screenshot of the Uni.WebApi project structure in Visual Studio. Source: Author	54
Figure 19. Example of the method in the Service file. Source: Author.....	55
Figure 20. Example of the method in a repository class. Source: Author	56
Figure 21. Stored universities in PostgreSQL database. Source: Author	57
Figure 22. Models folder structure. Source: Author	58
Figure 23. Adding JWT Authorization to a pipeline. Source: Author.....	59
Figure 24. Unit of work pattern implementation. Source: Author.....	60
Figure 25. Adding picture code implementation. Source: Author.....	61
Figure 26. Ocelot configurations. Source: Author.....	62

Figure 27. Screenshot of the Posts.WebApi project structure in Visual Studio. Source: Author	63
Figure 28. Working with Mongo.DB. Source: Author	64
Figure 29. Saved post in MongoDB. Source: Author	65