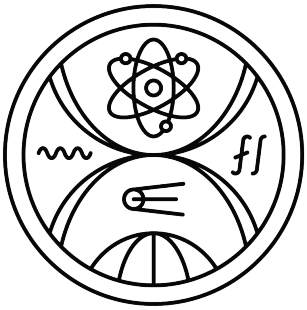
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS

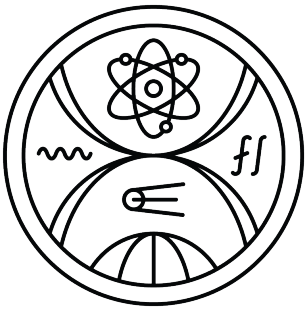


ANALYSIS, DESIGN AND IMPLEMENTATION OF MICRO-FRONTEND ARCHITECTURE

Diploma thesis

2024 Bc. Pavol Repiský

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS



ANALYSIS, DESIGN AND IMPLEMENTATION OF MICRO-FRONTEND ARCHITECTURE

Diploma thesis

Study program:	Applied Computer Science
Branch of study:	Computer Science
Department:	Department of Computer Science
Supervisor:	RNDr. Ľubor Šešera, PhD.
Consultant:	Ing. Juraj Marák

Bratislava, 2024 Bc. Pavol Repiský

37862688

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Pavol Repiský
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Analysis, Design and Implementation of Micro-frontend Architecture
Analýza, návrh a implementácia mikrofrontendovej architektúry

Anotácia: Mikrofrontendy predstavujú ďalší logický krok vo vývoji architektúry webových aplikácií. Tento prístup si však vyžaduje zvýšenie zložitosti architektúry a vývoja projektu. Problémy ako smerovanie, opätovná použiteľnosť, poskytovanie statických aktív, organizácia úložiska a ďalšie sú stále predmetom značnej diskusie a komunita ešte musí nájsť riešenia, ktoré dokážu efektívne spustiť projekt a riadiť výslednú zložitosť. Aj keď boli navrhnuté a diskutované niektoré prístupy, existuje veľké množstvo poznatkov a potenciálu na objavenie nových prístupov.

Cieľ: Preskúmajte existujúcu literatúru o prístupoch k návrhu a vývoju webových aplikácií pomocou mikro-frontend architektúry.
Porovnajte existujúce prístupy z hľadiska opätovnej použiteľnosti, rozšíriteľnosti, zdieľania zdrojov a správy stavu aplikácií.
Identifikujte prístupy, ktoré sú najvhodnejšie pre vývoj podnikových aplikácií, potom navrhnite a implementujte prototypovú mikrofrontendovú aplikáciu pomocou jedného vybraného prístupu.

Literatúra:
https://www.researchgate.net/publication/351282486_Micro-frontends_application_of_microservices_to_web_front-ends
https://www.angulararchitects.io/blog/micro-apps-with-web-components-using-angular-elements/
https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1570726&dswid=5530
https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1778834&dswid=-4588
https://www.scientificbulletin.upb.ro/rev_docs_arhiva/reze1d_965048.pdf

Vedúci: RNDr. Ľubor Šešera, PhD.
Konzultant: Ing. Juraj Marák
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Pavol Repiský
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Analysis, Design and Implementation of Micro-frontend Architecture

Annotation: Micro-frontends represents the next logical step in the development of a web-application architecture. However, this approach necessitates an increase in the complexity of the project architecture and development. Issues such as routing, reusability, static asset serving, repository organization, and more are still the subject of considerable discussion, and the community has yet to find any solutions that can effectively bootstrap a project and manage the resulting complexity. While there have been some approaches proposed and discussed, there is a great deal of knowledge and potential for new approaches to be discovered.

Aim: Review existing literature about approaches to design and development of web applications using micro-frontend architecture.
Compare existing approaches from aspects of reusability, extendibility, resource sharing and application state management.
Identify approaches best suited for enterprise application development, then design and implement a prototypical micro-frontend application using one selected approach.

Literature:
https://www.researchgate.net/publication/351282486_Micro-frontends_application_of_microservices_to_web_front-ends
https://www.angulararchitects.io/blog/micro-apps-with-web-components-using-angular-elements/
https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1570726&dswid=5530
https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1778834&dswid=-4588
https://www.scientificbulletin.upb.ro/rev_docs_arhiva/reze1d_965048.pdf

Supervisor:	RNDr. Ľubor Šešera, PhD.
Consultant:	Ing. Juraj Marák
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.

Assigned: 05.10.2023

Approved: 05.10.2023 prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

Acknowledgement

I would like to thank RNDr. Ľubor Šešera, PhD., for his supervision, willingness, and valuable advice. I am also very thankful to Ing. Juraj Marák for his patience, guidance, and all the time he dedicated to me during the preparation of my thesis.

Abstrakt

Táto práca podrobne analyzuje niekoľko najpoužívanejších prístupov k mikrofrontendom a porovnáva ich na základe aspektov, ako sú rozšíriteľnosť, opätovná použiteľnosť, jednoduchosť, výkon, zdieľanie zdrojov a skúsenosti vývojárov. Každý z týchto prístupov je dôkladne posúdený z hľadiska jeho výhod, nevýhod a použiteľnosti. Jeden z týchto prístupov, konkrétne web-compoents, bol zvolený na implementáciu proof-of-concept aplikácie pre manažovanie projektov vo frameworku Angular. Počas jej vývoju sme narazili na niekoľko bežných problémov, ako sú kompozícia, zdieľanie zdrojov, smerovanie a izolácia, ktoré sme vyriešili a riešenia uvádzame priamo v práci.

Výsledky implementácie naznačujú, že webové komponenty sú v efektívnym a dobrým prístupom k implementácii mikrofrontendovej architektúry. Prinášajú však niekoľko výziev, z ktorých hlavnými sú riešenie spoločných závislostí a komunikácia medzi mikroaplikáciami. Úspešné adoptovanie tejto architektúry si preto vyžaduje dôkladné plánovanie. Zistenia tejto práce poskytujú cenné poznatky o týchto výzvach a ich riešeniach. Okrem toho zjednodušujú rozhodovanie vývojárom a organizáciám, ktoré zvažujú túto architektúru pre svoje projekty.

Kľúčové slová: Microfrontends, Web-components, architektúry webových aplikácií, Angular

Abstract

This paper analyzes in detail several of the most widely used approaches to micro-frontends and compares them based on aspects such as extensibility, reusability, simplicity, performance, resource sharing, and developer experience. Each of these approaches is thoroughly assessed in terms of its advantages, disadvantages and usability. One of these approaches, namely web-compoints, was chosen to implement a proof-of-concept project management application in the Angular framework. During its development, we encountered several common issues such as composition, resource sharing, routing, and isolation, which we solved and will present solutions directly in the thesis.

The implementation results suggest that web components are an efficient and good approach to implement a micro-frontend architecture. However, they present several challenges, the main ones being the sharing of common dependencies and communication between micro-applications. Successful adoption of this architecture therefore requires careful planning. The findings of this work provide valuable insights into these challenges and their solutions. In addition, they simplify decision making for developers and organizations considering this architecture for their projects.

Keywords: Microfrontends, Web-components, web application architectures, Angular

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a brief introduction to microfrontends and the motivations behind this thesis. It also offers a brief overview of the existing literature on microfrontends. Then it presents the aim of the thesis and finally, it acknowledges its scope and limitations.

1.1 Background and Motivation

Microfrontends are one of the most promising directions in frontend development of modern software applications. They are based on the same idea as the more well-known microservices: splitting a software application into several separate smaller, independent applications to better address issues such as complexity, maintainability and deployability. [1][2][3]

While microservices focus on the partitioning the application layer, microfrontends deal with the partitioning of the frontend (presentation) layer of an application. Thus, despite some similarities, microservices and microfrontends address a different set of problems and use different techniques and technologies. Another important difference is that there are several accepted principles and patterns in microservices [4][5]. In microfrontends, there is not yet a general consensus on which of the principles is most appropriate, nor is there a set of design patterns for microfrontends.

Several articles [3][6][7][8] have been written on different approaches to microfrontends. However, each of these articles explains the approaches only briefly and does not analyze them in depth nor compare them with each other. An overview work on the basic approaches to microfrontends has been provided by Geers [2] in his book *Micro Frontends in Action*. This book not only summarized the basic approaches to creating microfrontends and provided simple examples, but also compare them at a general

level. On the other hand, the examples given in the book are too simple for enterprise applications, and even the comparison of approaches is left at a rather general level. The same can be said about the book *Building Micro-Frontends* by Mezzalira [1].

1.2 Aim

The aim of this thesis is to review the current literature on existing approaches to creating microfrontends, and analyze in depth and compare these approaches from the perspective of enterprise applications in aspects such as reusability, extensibility, resource sharing, and application state management.

Subsequently, one of these prospective approaches will be selected to validate its suitability and correctness by implementing a simple prototype microfrontend application implemented in an enterprise technology such as Angular [9] or React [10]. Finally, conclusions will be drawn and the chosen approach will be evaluated.

1.3 Scope and Limitations

Due to the breadth of the topic of microfrotends, the thesis will not cover all the important aspects of this architecture. Furthermore, it will not cover all possible existing approaches to implementing microfrontends, focusing on selecting those that are most commonly mentioned in the literature and used in practice.

The resulting application will serve as a proof of the validity of the chosen approach, not as a fully deployable solution for real-world use. The development of the application will be limited by time and available resources, which will affect its complexity and scope. The findings and conclusions drawn may not be generally applicable to all scenarios.

Chapter 2

Microservices and Microfrontends

This chapter introduces the concept of microservices and microfrontends, starting with theirs brief history. It futher explores the fundamental principles of microservices and how they apply to microfrontends. It also describes the challenges, advantages, and disadvantages of microfrontends. Finally, it concludes with real-world examples of microfrontend adoption.

2.1 History of Microservices and Microfrontends

By the beginning of this millennium, some software systems had become so large that they became difficult to maintain. The most famous example is *Amazon*'s [11] eshop, which went into critical condition in 2002. Hundreds of developers worked on the system. Although the system was divided into layers and components, these components were tightly interconnected. The development of a new version of the system was slow and deployment took on the order of weeks. Amazon then came up with a key solution: splitting the monolithic application into separately deployable services and creating an automated pipeline from build to deployment of the application [12].

Amazon's example was later followed by other companies such as *Netflix* [13], *Spotify* [14], *Uber* [15] and others [16]. In 2011, the first stand-alone workshop on this new approach to architecture was held near Venice. The workshop called this new architectural style *Microservices* [17]. In 2014, James Lewis and Martin Fowler wrote a blog in which they generalised the ideas from the workshop. In the article, they defined what Microservices are and specified their basic characteristics [17]. The article popularized Microservices in the general professional community. In 2015, Sam Newman wrote the first book on microservices, *Building Microservices* [5]. In parallel, Chris Richardson created his website [18] in 2014. He later compiled the ideas from this site into a separate book *Microservices Patterns* [4].

The term *Microfrontends* first appeared in the *ThoughtWorks Technology Radar* magazine [19] at the end of 2016, through 6 subsequent editions it has climbed from the Asses, Trial to the Adopt section. They described it as the application of microservices concepts to the frontend. Ideas from this magazine were further developed by Cam Jackson in an article *Micro Frontends* published on the Martin Fowler website. Another major milestone was the creation of a website on the topic of micro frontends by Michael Geers' [20] in 2017. Later in 2021, he compiled this site into a comprehensive, monograph, *Micro Frontends in Action* [2]. In 2021, a second monograph by Luca Mezzaliro, *Building Micro-frontends*, was published, further exploring this architecture. From the history, we see that Microfrontends are following a very similar path to microservices.

2.2 Microservices Principles

This section summarizes the basic principles of microservices architecture based on Martin Fowler's article [17] and Sam Newman's book [5].

2.2.1 Componentize via Services

The primary way of componentizing microservices applications is by breaking them down into services. Applications built this way aim to be as decoupled and cohesive as possible, where each service acts more like a filter in the Unix sense—receiving a request, applying some logic, and producing a response. The services communicate using REST protocols or a lightweight message bus. For this to work in practice, we must create a loosely coupled system where one service knows very little or nothing about others. This combination of related logic into a single unit is known as cohesion. The higher the cohesion, the better the microservice architecture.

2.2.2 Model around Business Domain

Traditionally, applications were split into technical layers such as the Presentation Layer, Business Layer, and Data Access Layer, with each of these layers being managed as a separate service by its own team. The problem with this approach is that making even a small change often cascades across multiple layers and teams, resulting in several deployments. This challenge worsens as more layers are added. This is known as horizontal decomposition. Opposite to this is vertical decomposition, which focuses on finding service boundaries that align with business domains. This results in services with names reflecting the system's functionalities and exposing the capabilities that

customers need. Therefore, changes related to a specific business domain tend to affect only the corresponding service boundary rather than multiple services. The team owning the service becomes an expert in that domain. Additionally, this approach ensures loose coupling between services and enhances team autonomy.

2.2.3 Design for Failure

Microservices aim to enhance the fault tolerance and resilience of an application by isolating services to prevent the failure of one service from cascading to others. Microservices should be designed with the assumption that any service can fail at any time. To ensure this, different patterns are utilized, such as the circuit breaker pattern. If a microservice fails repeatedly, the circuit breaker pattern allows the system to temporarily cut off communication with the problematic service, thereby preventing repeated communication attempts with the failing service and avoiding potential performance issues and application-wide failures. This enables other services to function properly without disruption, improving overall system resilience.

2.2.4 Adopt a Culture of Automation

Microservices add a lot of additional complexity because of the number of different moving pieces. Embracing a culture of automation is one way to address this, and front-loading effort to create automation tooling makes a lot of sense. One such tooling is automated testing at various levels (unit, integration, end-to-end) to ensure code quality is maintained. Another key aspect of automation is continuous integration and continuous deployment (CI/CD), where code changes are automatically built, tested, and deployed to production. Infrastructure as Code (IaC) is another important aspect. It enables teams to define and manage infrastructure via code, ensuring that environments are consistent, easily reproducible, and scalable. Automation enhances productivity, speeds up development, minimizes errors and frees teams from repetitive tasks.

2.2.5 Decentralize Everything

One of the problems with centralized governance is the tendency to standardize on a single technology, which may not be the best fit for every problem, as there may be better choices available. Microservices aim to avoid this by encouraging developers to use different technologies and frameworks based on what they believe will be the best tool for a given service. Each service typically also manages its own data storage, whether through different instances of the same technology or entirely different systems. Teams take complete ownership of their services through decentralized decision-making,

becoming domain experts in both the service and the related business domain they are supporting. By decentralizing everything—from decision-making and development to infrastructure and data management—we create systems that are more scalable, resilient, and adaptable.

2.2.6 Evolve Continuously

Microservices embraces the idea that software systems should keep evolving to meet new requirements, technologies, and business needs. This aligns well with the architecture, as each service is built and deployed independently and therefore can evolve at its own pace without requiring redesigns of the entire system. Additionally, new features can be added and plugged in as completely new services. This allows organizations to adapt quickly to changing requirements and grow with the needs of the business, ensuring long-term sustainability.

2.3 Microfrontends Challenges

This section presents inevitable challenges that need to be addressed when building microfrontends applications.

2.3.1 Communication

In an ideal world, we would want microfrontends to not communicate with each other at all. However, in real-world applications, that is not the case, especially when multiple microfrontends are on the same page. In traditional architectures, such as Single Page Applications, components communicate via direct parent-to-child data binding (e.g., props), child-to-parent callbacks, or shared state management solutions. However, this type of direct communication is not possible between microfrontends, as they are decoupled and independent.

There are several options for handling communication, with the most suitable one depending on the project type and chosen composition approach. Here are the most commonly mentioned ones. The first option is to inject an event bus, to which all microfrontends can send and receive events. To inject the event bus, we need the microfrontend container to instantiate it and inject it into all of the page's microfrontends [1]. The second option is a variation of the previous one, where we utilize Custom Events [1][2]. Most browsers now also support the new *Broadcast Channel API* [21], which allows the creation of an event bus that spans across browser windows, tabs, and iframes [2]. Lastly, there are less flexible and secure options, such as communication

via query strings [1] or utilizing web storage.

2.3.2 Routing

As microfrontends should be independent, isolated, and developed without prior knowledge of how they will be deployed, the next challenge is how to handle application routing, including the management of the currently active microfrontend. The solution will again depend on the chosen composition approach.

As Mezzalira [1] describes it, in the case of server-side composition, all the logic to build and serve the application happens on the server; therefore, routing must also be handled here. When a user requests a page, the server retrieves and combines different microfrontends to generate the final page. Since every request goes back to the server, it must be able to keep up with all the requests and scale well. This scaling issue can be improved by storing copies of the pages on a CDN; however, this will not work for dynamic or personalized data. When using edge-side composition, the composition happens at the CDN level. CDN knows which microfrontends to combine to serve the page based on the current URL. This is done through a technique called transclusion, where different microfrontends are stitched together. These two types of routing are best for applications that need to change views based solely on the current URL and for teams with strong backend skills [1].

Finally, there is the possibility of using client-side routing. In this case, the application shell will handle all routing and load different microfrontends based on the user state (such as whether the user is authenticated or not) and the current URL. This is a perfect approach for more complex routing based on different factors beyond just the URL and for teams with stronger frontend skills [1]. Additionally, the approaches can also be combined, such as CDN and origin or client-side and CDN [1].

2.3.3 Static Assets Serving

Static asset serving can also present a challenge in microfrontends due to each microfrontend having its own autonomous deployment. One issue arises from potential namespace conflicts, where multiple microfrontends may use the same filenames, leading to unintended overwrites or collisions. This can be avoided by strict naming conventions. Another issue is security concerns, such as Cross-Origin Resource Sharing (CORS) issues, which require careful configuration of asset access policies. Finally, microfrontends deployed on different domains or environments may face difficulties in locating and serving static files.

Potential solutions include a predefined deployment structure, where all static files are placed in specific directories, or creating a system-wide shared function that dynamically returns the resource location based on the context and state of the system [3]. Also each microfrontend can request its assets via an HTTP request independently, or in the case of small assets, they can be bundled inside the microfrontend itself.

2.3.4 Reusability

Microfrontends introduce a lot of code redundancy, as many components must be implemented in multiple microfrontends repeatedly. This can be somewhat mitigated by using shared dependencies. However, this introduces another issue: without proper handling, each microfrontend might bundle the same dependency, increasing the overall bundle size and load time. Additionally, if microfrontends use different technologies or versions, inconsistencies across the libraries may occur. Some libraries provide solutions to the first issue, such as Webpack's Module Federation, which offers shared dependency management even for different dependency versions. However, there are currently no no universal solutions, and each project must be assessed individually.

2.3.5 Styling Consistency and Isolation

Since all microfrontends are developed by an autonomous team, the user interface (UI) and user experience (UX) can differ significantly in each one of them. Therefore, there needs to be a common design system to ensure UI and UX consistency across all microfrontends. As mentioned in Peltonen's study [6], one possible approach is to use a shared CSS stylesheet. However, this would mean that all applications depend on a single common resource, which goes against the principle of loose coupling. Another option is to use a common component library, but if the microfrontends are developed using different technologies, the library must be available for in of them. A more universal solution is to use a common style guide, such as Bootstrap or Material Design, which helps maintain a consistent, though not identical, look and feel across the entire application.

Another styling challenge that needs to be addressed is style isolation within each microfrontend to prevent styles from one microfrontend overriding styles in another. This issue is particularly relevant for microfrontends composed on the client-side since server-side microfrontends are composed before reaching the browser and can be better isolated. There are two main solutions to this problem. The first involves using unique class naming conventions, such as prefixing class names with the microfrontend's name. Alternatively, there are libraries that handle this automatically. The second solution is utilizing the Shadow DOM, particularly when using web-components for composition.

2.3.6 Project Organization

A monolithic system can be easily stored in a single repository inside a Version Control System (VCS), but with microfrontends, it becomes more complex. Generally, there are three types of repositories that can be utilized: mono-repository, multi-repository, and multi-repository with the git-repo tool [3].

As Pavlenko [3] puts it, the simplest way is to use a mono-repository and place all microfrontends in a single repository, structured into folders. The problem with this approach is that each developer must download the entire codebase, even if they are only working on one microfrontend. This can be quite large in the case of complex projects. Additionally, branch checkouts and synchronizations can take a lot of time. The second approach is to have separate repositories for each microfrontend and potentially also for common parts. Microfrontends would then be published as packages in a private package registry and linked to the project via it. However, this increases the complexity of developer tools and maintenance [3].

The last approach is to again have multiple repositories, but manage them via the git-repo tool. The developer provides a configuration file to the tool, which describes which repositories should be downloaded and how they should be structured. The tool then performs the necessary actions. With this approach, developers can download only the necessary parts of the codebase, and the issue of slow checkouts is also removed, as synchronization happens only between the chosen repositories. Additionally, it does not require a package registry. However, the complexity is still higher than in the case of a mono-repository [3].

2.4 Microfrontends Advantages and Disadvantages

This section lists the most frequently mentioned advantages and disadvantages of microfrontends in the literature.

2.4.1 Advantages

There are numerous reasons why large companies are adopting microfrontend architecture, but the most prominent one is the increased development speed and reduced time to market due to cross-functional teams [2][7][6]. Geers [2] mentions: "Reducing waiting time between teams is microfrontends' primary goal." Having all developers working on the same stack within a single team leads to fewer misunderstandings and much faster communication [2][7]. Additionally, teams have much more freedom to

make case-by-case decisions regarding individual parts of the product [8][2], allowing teams to become experts in their respective areas of the application [7].
Microfrontends also bring many of the benefits of microservices architecture to the frontend. The codebases are smaller, more understandable, and less complex [2][8][7]. Consequently, this can lead to shorter onboarding times for new developers [6]. Each microfrontend is isolated, independently deployable, and its failure does not affect the rest of the system [6][7][8][2].

Microfrontends encourage changes and experimentation with new technologies, which is especially important in the frontend space, where technologies evolve rapidly. Microfrontends provide a way to upgrade only specific parts of the application instead of rewriting the entire system at once [8][7], and each microfrontend can potentially be developed using different technologies. This encourages developers to experiment with new tools and quickly adapt to changing requirements [6].

2.4.2 Disadvantages

2.5 Real-World Applications of Microfrontends

Chapter 3

Analysis

This chapter constitutes a significant part of the study. Initially, it offers a concise overview of other, common web application architectures. Subsequently, it explores the details of microfrontend architecture, explaining its specific aspects. Lastly, it conducts a comprehensive comparison of various approaches to this architecture.

3.1 Web application architectures

3.1.1 Monolithic applications

Monolithic architecture is one of the oldest and most traditional web architectures. In this architecture, all required components are tightly coupled within a single, unified system that operates independently of other applications. It integrates all business concerns into one structure. All components are dependent on each other and usually cannot run or even compile independently. The system traditionally uses a single shared database. From the operating system's point of view, a monolithic application runs as a single process within the application server's environment. Monolithic applications can be scaled up by running multiple instances behind a load balancer. Examples of technologies commonly used to build monolithic applications are Spring Boot, Ruby on Rails or Laravel.

The most significant advantage of this architecture is its simplicity. Since it runs as a single process, it is much easier to test, deploy, debug, and monitor. The project configuration and setup are quick and straightforward. All data is stored in a single database, eliminating the need for synchronization. All communication occurs via intra-process mechanisms, which avoids delays and other communication problems, typically resulting in better performance. It is also cheaper to host the application.

However, as the company grows, so does the monolithic application. It eventually

becomes so large that few, if any, people fully understand how the entire application works. This leads to slower development processes and a higher likelihood of bugs. Changes in one components may cause unexpected behavior in parts of the system, resulting in a cascade of errors and a failure of a single component impacts the entire system. Developers must choose a technology at the start and are essentially locked into it for the application's lifetime, as switching technologies would require a complete rewrite of the application. When changes are made to any part of the application, it is necessary to rebuild and redeploy the entire application. Due to their size, monoliths often have longer startup times. As the application grows, the number of developers increases, often leading to unequal workforce utilization and productivity losses.

Therefore, monolithic applications are best suited for smaller, less complex applications that do not anticipate frequent code changes or evolving requirements. Examples of these are content-based websites. [22][23]

3.1.2 Single-page applications

A Single-page Application (SPA) is a popular approach to the development of client-side rendered web applications. In this architecture, all essential resources are typically downloaded during the initial page load. As users interact with the application, the DOM is dynamically updated via JavaScript and HTTP requests to fetch necessary data from the server, completely eliminating the need for full page reloads. In addition, routing is fully managed on the client side, meaning that each time a user requests a view change, the URL is rewritten in a meaningful way to enhance the user experience. SPAs also allow developers to decide how to split the application logic between the server and the client. For example, a 'fat client' and a 'thin server' architecture implies that most of the logic is performed on the client side in the browser, while the server side is used primarily as a persistence layer. Alternatively, a 'thin client' and a 'fat server' approach implies that most of the logic is executed on the server, with the client simply displaying the content in a meaningful way. There are numerous frameworks, libraries, and tools available for building SPAs, the most popular being Angular, React, and Vue. These frameworks abstract DOM manipulation and provide lifecycle methods.

The key advantage of SPAs is their dynamic, app-like user experience, as there are no full page reloads, simulating the experience of native applications for mobile devices or desktops. This allows users to seamlessly navigate between different parts of the application without long waiting times. Additionally, SPAs place a much lower load on the server.

On the other hand, the obvious disadvantage is the longer initial load time because the entire application is downloaded upfront, instead of only the resources needed for the initial view. However, this can be mitigated with techniques such as code splitting or lazy loading. Another disadvantage is organizational: SPAs can quickly grow large, and if they are managed by different, especially distributed, teams working on the same codebase, different areas of the application may end up with inconsistent approaches and decisions. This can result in communication overhead between teams. Furthermore, as the codebase grows, team productivity may drastically slow down.

SPAs are best suited for highly responsive and interactive applications with smaller data volumes, such as social networks or SaaS platforms. [1][6]

3.1.3 Microservices

Microservices architecture can be defined as an approach to developing web applications, where each component runs in its own process and communicates through lightweight mechanisms, often an HTTP resource API. Each service is built around specific business capabilities and is independently developed, testable, deployable through fully automated deployment processes, and scalable. Each service can be written in a different programming language, has its own storage, and is managed by its own team. Usually, each service has a dedicated team, enabling the system to be developed in parallel. This approach places great emphasis on continuous delivery and deployment, service isolation, and decentralized decision-making.

Microservices solve many problems associated with monolithic architectures. Due to the smaller size of services, they are less complex, easier to understand, and simpler to onboard new developers. Changes in one service do not affect the rest of the system, so only the modified service needs to be rebuilt and redeployed. Services are faster to reboot and start. Each service can be scaled independently based on its usage, rather than scaling the entire system. Teams can choose different technologies based on their preferences. Since each team is responsible for its own service, there is less communication overhead, and the codebase is more consistent.

However, this approach also comes with certain challenges. Projects are much harder to initiate and require significant upfront configuration. Developers need a broader range of skills and knowledge. Since the services are independent, managing communication between them can be challenging, making end-to-end testing and debugging more difficult. Communication between services over a network can result in longer response

times and potential delays. Additionally, microservices require complex infrastructure, which can lead to increased costs and operational overhead.

Therefore, microservices are best suited for large and complex web applications with large teams, especially those expected to expand, scale, and undergo frequent changes. [17][6]

3.2 Microfrontends in detail

3.2.1 Overview

Microfrontends is an architectural and organizational approach to building web applications, essentially extending the concepts of Microservices to the frontend. The main idea behind it is to break down a monolithic codebase into multiple small, isolated, and loosely coupled applications, which are then combined into one. Each application can be developed, run, and deployed independently, focusing on a single feature or business sub-domain.

In traditional web applications, teams are typically split based on the technologies they use, such as frontend and backend teams. However, microfrontends aim to divide the application vertically, meaning the teams are cross-functional, with members from all departments. These teams develop the application from its presentation layer to its data layer and are fully responsible for the feature. Teams can choose the technologies they want to use based on their preferences and project requirements. Typically, each microfrontend would be an autonomous application with its own dedicated backend and continuous delivery pipeline. Each microfrontend can be built with completely different implementation techniques, even though they together compose a single web application.

A microfrontend can either be an entire page or just a "fragment"—an element that typically appears on multiple pages, such as the header or footer. The process of putting all the microfrontends together is called composition or integration. There are several ways to do this, and a comparison of various approaches will be presented in the next section. These approaches are separated into two categories: server-side and client-side, based on where the composition occurs. Other concepts that form the foundation of microfrontends are:

- **Code isolation:** The runtime should not be shared, and applications should be self-contained.

- **Native browser API preference:** For communication purposes, the native browser's APIs should be preferred.
- **Focus on automation:** A strong focus is placed on automation to speed up the development process.
- **Hiding implementation details:** Contracts should be defined upfront between teams to avoid reliance on implementation details that may change.
- **Decentralized governance:** Decisions are moved away from a one-size-fits-all approach and into the hands of the teams. approach

3.2.2 Benefits

There are numerous reasons why large companies are adopting microfrontends architecture, but in most cases, the number one reason is to increase development speed and decrease time to market. The reason why development in microfrontends architecture is much faster is due to cross-functional teams. All the people involved in creating a feature are in the same team, resulting in significantly less communication overhead, without unnecessary waiting for responses. Additionaly, due to their different perspectives, they come up with more creative and effective solutions for the tasks.

Microfrontends also bring most of the benefits of microservices architecture to the frontend side. The application is split into smaller pieces (microfrontends), resulting in a smaller, more understandable, and less complex codebase with shorter onboarding times. Each microfrontend is isolated, therefore independently deployable, and its failure does not affect the rest of the system. The implementation decisions are decentralized to the teams working on the features, giving them a stronger sense of ownership of the given feature.

Microfrontends encourage changes and experimentation with new technologies, which is very important, especially in the frontend space, where technologies change rapidly. What was considered state-of-the-art in frontend development last year might now be deprecated. Microfrontends provide a way to upgrade only small parts one by one instead of rewriting the entire application at once. It is also easier to migrate old applications since the architecture allows for a new application to run side by side with the old one. Lastly, teams deliver features directly to the customer instead of relying on an additional team, increasing customer focus. [2][7]

3.2.3 Drawbacks

However, everything comes with a cost. Having an application split into multiple parts across teams, potentially using different technologies, naturally introduces a lot of code redundancy. This can lead to larger file sizes and, consequently, longer download times, so performance should be closely monitored for this reason. Each team also needs to set up and maintain its own application server, build process, and continuous integration pipelines.

A bug in a library used across multiple microfrontends must be fixed in each one. If not managed properly, common libraries may be shipped in multiple microfrontends, further increasing the bundle size. This architecture also complicates routing and communication within the system, requiring them to be handled in a specialized manner.

Lastly, it introduces new challenges, such as orchestrating the microfrontends (e.g., determining when each microfrontend should be displayed) and maintaining UI/UX consistency across the entire system. To address the second one, a design system should be developed.[2][3]

3.2.4 Use cases

Microfrontends make scaling projects easier and development faster. However this may not apply, for small projects with just a few people, where communication is not a significant issue, in this case microfrontends may introduce more drawbacks than benefits. Large, complex projects with many people involved gain the most advantages from this architecture. Below are examples of projects that have adopted this architecture.

The first noteworthy example is **Amazon**. Several employees have reported that the e-commerce site has been using this architecture for quite some time. It uses a UI integration technique to assemble parts of the page before it is displayed. Other e-commerce companies, such as the well-known **IKEA** and the European fashion e-commerce platform **Zalando**, have also adopted microfrontends. Zalando even open-sourced their microfrontends framework Tailor.js, later replacing it with the Interface framework. Another prominent adopter is **Spotify**. Their desktop application initially used iframes to compose the UI, but this approach was abandoned due to poor performance and was replaced by a SPA. Similarly, **SAP** also utilized iframes and released their own microfrontends framework, designed for creating enterprise applications that integrate with SAP systems. Lastly, **DAZN**, a sports streaming platform, migrated its monolithic frontend to a microfrontends architecture. DAZN focused on supporting not only the web but also multiple smart TVs and gaming consoles.[2] [1]

3.3 Composition approaches

There are several primary approaches to implementing a microfrontend application. Some are as straightforward as using a link, while others take a bit more time to understand. The most important thing to remember is that there isn't a single best solution for all projects. Each project needs to be thoroughly analyzed from different perspectives to choose the most suitable approach. This section presents five primary approaches, outlining their advantages, disadvantages, use cases, and an assessment of their characteristics to make the choice easier.

3.3.1 Link-based composition

One of the simplest approaches to a microfrontend architecture is the composition of multiple applications using hyperlinks. In this method, each such application is developed by a dedicated team that creates its own HTML, CSS, and JavaScript files. Each application is deployed independently, typically on a different port under the same domain, ensuring it remains fully isolated from other applications in the system. To facilitate navigation between applications, teams must share their URL patterns that will be used to link their sites from other teams' applications.[2] Example of such URL patterns might look like the following.

- **Team Project - Project page**
  URL pattern: `http://localhost:3000/projects/<project-id>`
- **Team Task - Task page**
  URL pattern: `http://localhost:3001/tasks/<task-id>`

Rather than exchanging these URL patterns verbally, which would require informing all affected teams and potentially redeploying their applications when URLs change, a better solution is to maintain a centralized JSON file containing all URL patterns for each team, which can then other applications read at runtime to construct the necessary URLs. To compose the microfrontends, we simply interconnect them across different applications using standard anchor elements. As an example, let's consider an online store application that consists of a catalog microfrontend and product microfrontends. The link from the catalog to a specific product would look as follows.

```
<!-- Using an anchor tag to link to a product within the catalog page
-->
<a href="http://localhost:3001/products/123">View Product #123</a>
```

Advantages

This composition method offers several benefits. It provides complete isolation between microfrontends, eliminating direct communication between them. Errors in one microfrontend are contained and do not affect other parts of the system. The microfrontends can be deployed independently, and the project requires minimal configuration. The system can be easily expanded by adding new microfrontends [2].

Disadvantages

The primary limitation of this approach is that it does not allow for the integration of multiple microfrontends on a single page. Users must click through links and wait for page loads when navigating between different sections, potentially worsening the user experience. Additionally, common elements, such as headers, need to be reimplemented and maintained separately within each microfrontend, leading to code duplication, maintenance overhead, and potential inconsistencies. Resource sharing between such applications is also very complex. [2]

Suitability

While microfrontend composition via links provides a basic integration strategy, it is rarely used as the sole approach in modern web development due to its limitations in resource sharing between microfrontends. It is typically combined with other techniques to create more robust solutions. [2]

Aspect	**Score**
Extensibility	4/5
Reusability	2/5
Simplicity	5/5
Performance	3/5
Resource sharing	1/5
Developer experience	4/5

Table 3.1: Characteristics assessment of link-based composition

3.3.2 Composition via iframes

Iframes are an old yet still widely used technique in web development. Essentially, an iframe is an inline HTML element that represents a nested browsing context, allowing one HTML page to be embedded within another. Each embedded context has its own document and supports independent URL navigation [24]. Compared to link-based

composition, iframes allow multiple pages to be combined into a single unified view while maintaining the same loose coupling and robustness.

Iframes can communicate with the host page through the postMessage() method and are supported across all browsers [2][8][1][3]. Adding an iframe is as simple as including an HTML tag, and its behavior can be customized with additional attributes. This time, to compose the microfrontends, we would use the iframe element. The difference is that the linked microfrontend would be displayed directly within the application that linked it.

```
<!-- Using an iframe to include a specific product directly within the
catalog page -->
<iframe src="http://localhost:3001/products/123"></iframe>
```

Advantages

The biggest advantages of using iframes are their excellent robustness and isolation in terms of styling and scripts not interfering with each other. Iframes are also very easy to set up and work with, and as already mentioned, are fully supported by all major browsers. [2][8][1][3]

Disadvantages

However, the main advantage of iframes also comes with a cost. It is impossible to share common dependencies across different iframes, leading to larger file sizes and longer download times. Communication between iframes is restricted to the iframe API, which is cumbersome and inflexible. These limitations make tasks such as routing and history management challenging. Additionally, the host application must know the height of the iframe in advance to prevent scrollbars and whitespace, which can be particularly tricky in responsive designs. Using numerous iframes on the same page can significantly degrade application performance. Lastly, iframes perform poorly in terms of search engine optimization (SEO) and accessibility. [2][8][1][3]

Suitability

Despite the numerous disadvantages, iframes can still be the most suitable choice in some cases. Iframes shine when there is not much communication between micro-frontends and the encapsulation of our system using a sandbox for every micro-frontend is crucial. The best use cases for iframes are in desktop, B2B, and internal applications. They should be avoided if performance, SEO, accessibility, or responsiveness are crucial factors.[2][1]

Aspect	Score
Extensibility	4/5
Reusability	3/5
Simplicity	4/5
Performance	1/5
Resource sharing	2/5
Developer experience	4/5

Table 3.2: Characteristics assessment of composition via iframes

3.3.3 Composition via Ajax

Asynchronous JavaScript and XML (AJAX) is a technique that enables fetching content from a server through asynchronous HTTP requests. It then uses the retrieved content to update parts of the website without requiring a full-page reload [25]. To use AJAX as an approach for microfrontends, each team must first expose their microfrontend on a specific endpoint. Next, we create a corresponding empty element in the host application and specify the URL in a data attribute from which the content should be downloaded. Finally, JavaScript code is needed to locate the element, retrieve the URL, fetch the content from the endpoint, and append it to the DOM.

Compared to composition via iframes, accessibility and SEO are no longer issues. However, this approach introduces additional challenges, such as CSS conflicts if two microfrontends use the same class names, which can lead to unintended overrides. To avoid this, all CSS selectors should be namespaced. Numerous tools, such as SASS, CSS Modules, or PostCSS, can handle this automatically. A similar issue can occur with scripts; in that case, we can encapsulate scripts within Immediately Invoked Function Expressions (IIFE) to limit the scope to the anonymous function and again namespace global variables. [2] The composition-related code would look something like this.

```
<!-- Initial element with an id and data-url attribute within the
catalog page -->
<div id="product/123" data-url="http://localhost:3001/products/123">
</div>

<!-- Appending the content of the microfrontend inside the
empty element -->
<script>
   const productElm = document.getElementById("product/123");
```

```
   const url = productElm.getAttribute("data-url");

   window
      .fetch(url)
      .then(res => res.text())
      .then(html => {
         productElm.innerHTML = html;
      });
</script>
```

Advantages

Since all the microfrontends are included in the same Document Object Model (DOM), they are no longer treated as separate pages, as was the case with iframes. This eliminates the SEO and accessibility issues associated with iframe-based composition. Additionally, there are no issues with responsiveness, as the microfrontends will be loaded as standard HTML elements, which can be styled as needed. This approach also provides greater flexibility for error handling; if the fetch request fails, for example, a direct link to the standalone page can be provided. [2]

Disadvantages

One obvious disadvantage is the delay before the page is fully loaded. Since microfrontends must be downloaded, parts of the page may appear a bit later, which can worsen the user experience. Another significant issue is the lack of isolation between microfrontends. Lastly, all lifecycle methods for the scripts must be implemented from scratch. [2]

Suitability

This approach is well-established, robust, and easy to implement. It is particularly well-suited for websites where markup is primarily generated on the server side. However, for pages that require a higher degree of interactivity or rely heavily on managing local state, other client-side approaches may be more suitable. [2]

3.3.4 Server-side composition

Server-side composition is typically managed by a service positioned between the browser and the application servers, often using tools like Nginx. This service assembles the view by aggregating various microfrontends and constructing the final page

Aspect	Score
Extensibility	4/5
Reusability	4/5
Simplicity	3/5
Performance	3/5
Resource sharing	3/5
Developer experience	3/5

Table 3.3: Characteristics assessment of composition via Ajax

before delivering it to the browser. There are two primary approaches to achieving this.

Server-side Includes (SSI)

n this approach, a server-side view template is created using Server-Side Includes (SSI) directives. These directives specify URLs from which content should be fetched and included in the final page. The web server replaces these directives with the actual content from the referenced URLs before sending the page to the client. Additionally, fallback content can be defined in case the include fails. So the SSI directive used to compose the microfrontends would look something like the following.

```
<!-- Fallback element, which will be displayed in case the include
fails -->
<!--# block name="product_fallback" -->
   <a href="http://localhost:3001/products/123">View Product #123</a>
<!--# endblock -->

<!-- Using SSI directive, which will be replaced by the microfrontend
content -->
<!--#include virtual="/products/123" stub="product_fallback" -->
```

Edge-Side Includes (ESI)

Edge Side Includes (ESI) is a specification that standardizes the process of assembling markup. Similar to Server-Side Includes (SSI), ESI uses directives within the server-side view template to include content. However, ESI typically involves content delivery network (CDN) providers like Akamai or proxy servers such as Varnish. The server replaces ESI directives with content from the referenced URLs before delivering the page to the client. An ESI directive look as follows:

```
<esi:include src="http://localhost:3001/tasks/123"  />
```

Other Approaches

Various libraries and frameworks simplify server-side composition. Two notable examples include

- Tailor: A Node.js library developed by Zalando
- Podium: Built upon Tailor by Finn.no, Podium extends its capabilities and provides additional features. [2]

Advantages

Server-side composition is proven, robust, and reliable technique. It offers excellent first-load performance, as the page is pre-assembled on the server, providing a positive impact on search engine ranking. Maintenance is straightforward, and interactive functionality can be added via client-side JavaScript. It also well-tested and well-documented. [2][1]

Disadvantages

For larger server-rendered pages, browsers may spend considerable time downloading markup rather than prioritizing essential assets. Additionally, technical isolation is limited, requiring the use of prefixes and namespacing to prevent conflicts. The local development experience is also more complex. [2]

Suitability

This approach is ideal for pages that prioritize performance and search engine ranking, as it remains reliable and fully functional even without JavaScript. However, it may not be optimal for pages requiring a high level of interactivity. It is also well-suited for B2B applications with numerous modules that are reused across different views. [2][1]

Aspect	**Score**
Extensibility	4/5
Reusability	4/5
Simplicity	2/5
Performance	5/5
Resource sharing	4/5
Developer experience	3/5

Table 3.4: Characteristics assessment of server-side composition

3.3.5 Composition via Web-components

A relatively new approach that is starting to arise, thanks to new HTML standards, is composition via web-components, which is somewhat of a variation on AJAX composition. Web-components consist of three main technologies that can be combined to create custom, reusable elements with encapsulated functionality, preventing any conflicts in code or styles when reused across applications. These core technologies are as follows.

Custom Elements
Custom Elements extend HTML, allowing developers to define their own custom HTML elements with unique behavior and functionality. They provide callbacks that enable interaction with the external environment, essentially serving as wrappers for microfrontends.

Shadow DOM
The Shadow DOM provides encapsulation by hiding a custom element's implementation details from the rest of the document, allowing for the creation of self-contained components. This is achieved by attaching an encapsulated shadow DOM tree to an element, rendering it separately from the main document DOM.

HTML Templates
HTML Templates allow for the definition of markup templates that can be reused and instantiated multiple times within a document. These templates are not rendered until they are explicitly activated. [2][1]

To create a product custom element, we would extend HTML elemnt class like follows.

```
<!-- Definition of the custom element -->
class ProductComponent extends HTMLElement {
    constructor() {
        super();
        this.attachShadow({ mode: 'open' });
    }

    connectedCallback() {
        this.render();
    }

    render() {
```

```
        this.shadowRoot.innerHTML = `
            <style>
                h3 {
                margin: 0 0 10px 0;
                font-size: 1.2em;
                }
                p {
                margin: 0 0 5px 0;
                color: #555;
                }
            </style>
            <div>
                <h3>Product A</h3>
                <p>Description for Product A</p>
                <strong>$10</strong>
            </div>
        `;
    }
}

<!-- Registration of the custom element -->
customElements.define('product-component', ProductComponent);

<!-- Usage of the custom element -->
<product-component></product-component>
```

Advantages

Web-components are a widely implemented web standard that offer strong isolation managed by the browser, making applications more robust. Web Components can be used across different frameworks and libraries, such as Angular and React. They inherently support lazy loading and code splitting, enhancing performance. Additionally, Web Components enable the encapsulation of complex functionality into reusable building blocks and provide a convenient way for components to communicate with one another. [2][1]

Disadvantages

One common criticism of Web-components is their reliance on JavaScript for functionality. They are not fully supported in all older browsers; while polyfilling can extend

support for custom elements, integrating the shadow DOM is notably more challenging, and polyfills significantly increase the bundle size. Additionally, Web-components lack built-in state management mechanisms, complicating integration with existing solutions. Finally, handling search engine optimization can also be challenging. [2][1]

Suitability

Web-components are an excellent choice for multitenant environments and are well-suited for building interactive, app-like applications. However, for applications that prioritize SEO or require compatibility with legacy browsers, Web Components may not be the most suitable option. [2][1]

Aspect	**Score**
Extensibility	4/5
Reusability	4/5
Simplicity	4/5
Performance	4/5
Resource sharing	3/5
Developer experience	4/5

Table 3.5: Characteristics assessment of composition via Web-components

Chapter 4

Design

This chapter offers an insightful exploration of the design considerations essential for the resultant prototypical microfrontends application. It begins with a brief introduction, outlining the application's core purpose and objectives. Subsequently, it examines its functional and non-functional requirements. Following that, a dedicated section elaborates on how the the application will be split into micro-frontends and outlines the teams that will be established for this purpose. Subsequently, the system architecture is discussed, followed by a dedicated section on the technologies used for implementation. Finally, graphical user interface mockups are presented.

4.1 Application introduction

The resulting prototypical application should be focused on the on the enterprise landscape. It should be logically divisible into independent business concerns, striking a balance where it's not overly simplistic for microfrontends architecture to lose its relevance, yet not overly complex to develop within the boundaries of the thesis. After careful consideration, a project management tool appears to meet all the criteria. Its main purpose is to aid both companies and individuals in managing their projects and tasks. It will offer functionality for creating and managing users, setting up projects, linking tasks between projects and users, managing task and project states and presenting such information in an easily comprehensible manner. These functionalities will be discussed in more detail in upcoming sections.

4.2 Requirements

The requirements for the application are categorized by priority as follows:

- (M) Must: Crucial for the system's operation.

- (S) Should: Highly recommended, though not mandatory.
- (C) Could: Optional for implementation.

4.2.1 Functional Requirements

Here is a comprehensive list of all functional requirements for the application.

User Management

- Users can create, update, and delete other users. (M)
- Users are displayed in a list view. (M)
- Users can set and update the following attributes for a user: name, email, phone number, role, status, bio. (M)
- Users can be filtered by: name, email, phone number and role. (S)

Project Management

- Users can create, update, and delete projects. (M)
- Projects are displayed in a list view. (M)
- Users can set and update the following project attributes: name, company, status, summary. (M)
- Users can be linked to projects as members. (M)
- Projects can be filtered by: name, company, status, and members (S)

Task Management

- Users can create, update, and delete tasks. (M)
- Tasks are displayed on a Kanban board. (M)
- Users can set and update the following task attributes: title, status, tags, due date, priority, and description. (M)
- Users can be linked to tasks as assignees. (M)
- Users can link tasks to projects. (M)
- Tasks can be filtered by: title, project, and assignees. (S)

Dashboard

- User's active, new, and completed project statistics are displayed on the dashboard. (M)
- User's active, new, and completed task statistics are displayed on the dashboard. (M)
- The dashboard contains widgets for the user's projects and tasks. (M)
- The dashboard includes a widget of new users. (M)
- The dashboard displays a task completion graph. (C)

Settings

- Users can switch between light and dark themes. (C)
- Users can switch between Slovak and English languages. (C)

4.2.2 Non-functional Requirements

Here is a comprehensive list of all non-functional requirements for the application:

- The application is divided into several microfrontends. (M)
- Each microfrontend is isolated from the others to prevent cascading failures. (M)

4.3 System Architecture

In this section, we will delve into the system architecture of the application, which adopts a microfrontend architecture to separate different domains into distinct, independently developed, and deployed microfrontends. The application is entirely frontend-focused, built using Angular, and follows a modular design to ensure scalability, maintainability, and flexibility. This section will discuss the role of the application shell, the individual microfrontends, and how they interact with each other. Each of these modules will be managed by dedicated imaginary teams to reflect the independence and separation of concerns inherent in microfrontend architectures.

4.3.1 Application Shell

The application shell is the core of the application, functioning as the orchestrator for all microfrontends. It provides critical infrastructure and functionality, including routing, internationalization (i18n), and data communication. This shell is responsible for rndering idividual microfrontends, based on the route the user navigates to. It uses Angular Router to determine which microfrontend should be displayed at any given time, and loads them as standalone JavaScript bundles fetched from separate servers.

The shell maintains the master routing configuration of the entire application. Each microfrontend has its own routing module, but the shell takes care of routing at the highest level, enabling deep linking and lazy loading of microfrontends as needed. And it passes essential data to the microfrontends, such as: current route or current language. Additionaly the shell listens for and dispatches Browser Events to facilitate communication between microfrontends. The application shell will be deployed as a separate module.

4.3.2 Users Microfrontend

This microfrontend is responsible for user management functionality within the system. It is designed as a standalone module that can be deployed independently, and its key functionality revolves around managing users through CRUD operations (Create, Read, Update, Delete). The users are displayed in a list view for easy management.

4.3.3 Projects Microfrontend

The Projects Microfrontend handles project management, providing functionality to create, update, view, and delete projects. Like the Users module, the projects are presented in a list view format. Additionally, this microfrontend is responsible for assigning users to projects.

4.3.4 Tasks Microfrontend

The Tasks Microfrontend offers task management features, utilizing a Kanban board to display and organize tasks. This microfrontend enables the creation, updating, and deletion of tasks, and supports assigning users to tasks as well as associating tasks with specific projects. The Kanban board provides a visual representation of task progression, allowing users to move tasks between different stages (e.g., "Backlog", "In Progress", "Completed").

4.3.5 Dashboard Microfrontend

The Dashboard Microfrontend serves as the homepage of the application, providing an overview of relevant statistics and key metrics across users, projects, and tasks. It presents a consolidated view, offering users a high-level summary of the application's current state. The dashboard aggregates data from other microfrontends to display statistics such as the number of active projects, tasks in progress, and users. It serves as a passive consumer of data, with minimal interaction beyond displaying information to the user.

4.4 Tech Stack

In this section, we discuss the technologies that will be utilized in the development of the project.

4.4.1 Angular

The primary framework for the application development will be Angular 18, the latest version of Angular available at the time of writing. Angular is a powerful, platform-agnostic web development framework created by Google that enables developers to build scalable, maintainable, and performant single-page applications (SPAs). It comes with built-in tools for handling routing, form validation, state management, and more. In this project, Angular will be used to develop both the application shell and the microfrontends, leveraging its modular architecture. The choice of Angular stems from its wide adoption in enterprise applications, providing robust support and a mature ecosystem. Additionally, Angular Router will handle routing between microfrontends, while HttpClient will manage the loading of microfrontend bundles from their respective servers.

4.4.2 TypeScript

TypeScript will be the primary programming language for the project, as it is the standard language used in Angular development. TypeScript is a superset of JavaScript that adds static types, enabling developers to catch errors at compile time rather than at runtime. This helps reduce bugs and makes the code more reliable and easier to maintain. The additional type safety and tooling support offered by TypeScript make it a popular choice in the enterprise landscape, and this is one of the key reasons why it was chosen for the project.

4.4.3 Web Components

Web Components will be leveraged in this project to ensure that each microfrontend operates independently and can be integrated seamlessly into the application shell. Web Components are a set of standardized APIs that allow developers to create reusable, encapsulated HTML elements. In this project, each microfrontend will be exposed as a custom element, ensuring interoperability between microfrontends. To avoid CSS conflicts and ensure style encapsulation, the Shadow DOM will be used for each Web Component. While Web Components are not fully supported by all browsers, polyfills will be employed where necessary to ensure compatibility.

4.4.4 Browser Events

To facilitate communication between the various microfrontends and the application shell, standard browser events will be utilized. Browser events provide a lightweight and efficient way to send and receive messages between different parts of the application, ensuring that microfrontends remain decoupled but can still share essential data when needed. This event-driven approach will serve as the primary method for cross-microfrontend communication, ensuring a flexible and scalable architecture.

4.5 Graphical User Interface

In this section, we will present and describe the wireframes of the application. These wireframes illustrate the layout and structure of the user interface across different parts of the system, providing a visual representation of how users will interact with the application. Each wireframe focuses on a specific microfrontend or application shell, showcasing its core functionality and design elements.

4.5.1 Application Shell

The application shell design features a side panel that contains the logo, standard navigation links for each microfrontend, and a settings button. The main content area next to the side panel serves as the placeholder where microfrontends will be dynamically rendered based on the user's navigation. When the user clicks the settings button, a modal window opens, allowing them to switch languages and toggle between light and dark themes. To keep the document concise, the wireframe for the settings modal is not included, as it would take up additional space.

Figure 4.1: Application shell wireframe

4.5.2 Users Microfrontend

The Users Microfrontend consists of a table where each row represents a user, displaying key details such as name, email, and role. The last column of the table is dedicated to an action button, which opens a dropdown menu with options to view, edit, or delete a user. Upon selecting any of these actions, a modal window appears, allowing users to execute the desired operation. As with other modals in the application, they are not included in this document to conserve space. Directly above the table is a search bar and a filter button, enabling users to refine the displayed records further. Additionally, a set of tabs allows for quick filtering of users based on their roles (e.g., Admin, Developer, Tester). At the very top, the topbar includes the page title, a brief subtitle, and a button to add a new user. Clicking this button also opens a modal specifically designed for user creation.

4.5.3 Projects Microfrontend

The projects microfrontend follows a similar structure to the users microfrontend. It features a table where each row represents a project, displaying key details such as project name, associated company, tags, and other relevant information. The last column of the table is reserved for an action button, which opens a dropdown menu with options to view, edit, or delete the project. Above the table, there is a search bar and a filter button to help users further refine the list of projects. Additionally, a set of tabs allows for quick filtering of projects based on their status (e.g., Planned,

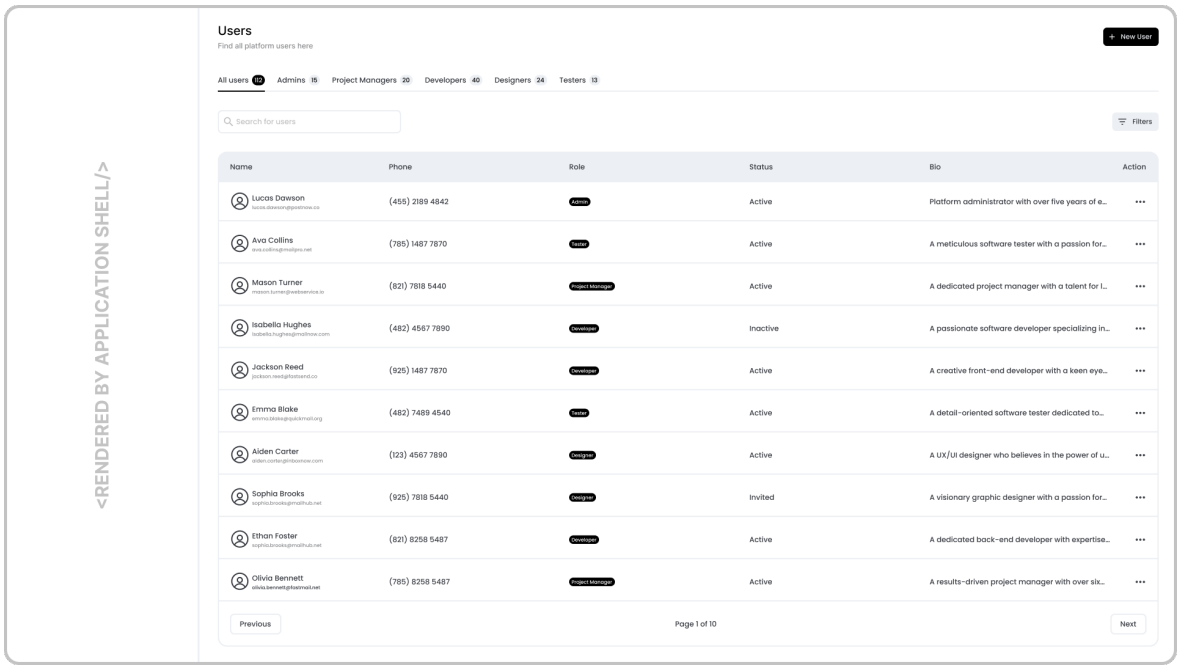


Figure 4.2: Users microfrontend wireframe

In-Progress, Completed), enabling users to quickly locate projects in different stages of development. At the top of the page, the topbar displays the page title, a brief subtitle, and a button to add a new project. Similar to the user management interface, clicking the "Add Project" button opens a modal designed for creating a new project.

4.5.4 Tasks Microfrontend

The Tasks Microfrontend is slightly different from the other sections, as its main component is a Kanban board that organizes tasks into various stages, such as Backlog, In-Progress, and Under Review. Each task is represented by a card that displays essential details such as the task title, tags, assignees, and more. In the upper-right corner of each task card, there is a three-dot button that opens a dropdown menu with options to view, edit, or delete the task. Each stage on the Kanban board includes a button for creating new tasks directly within that stage. Above the Kanban board, there is a search bar and a filter button to help users refine the displayed tasks. Additionally, a set of tabs allows for quick filtering of tasks based on their current status, helping users focus on tasks that are relevant to their needs at the moment. At the top of the page, the topbar contains the page title, a brief subtitle, and a button to add a new task. As with other parts of the application, clicking this button opens a modal designed specifically for task creation.

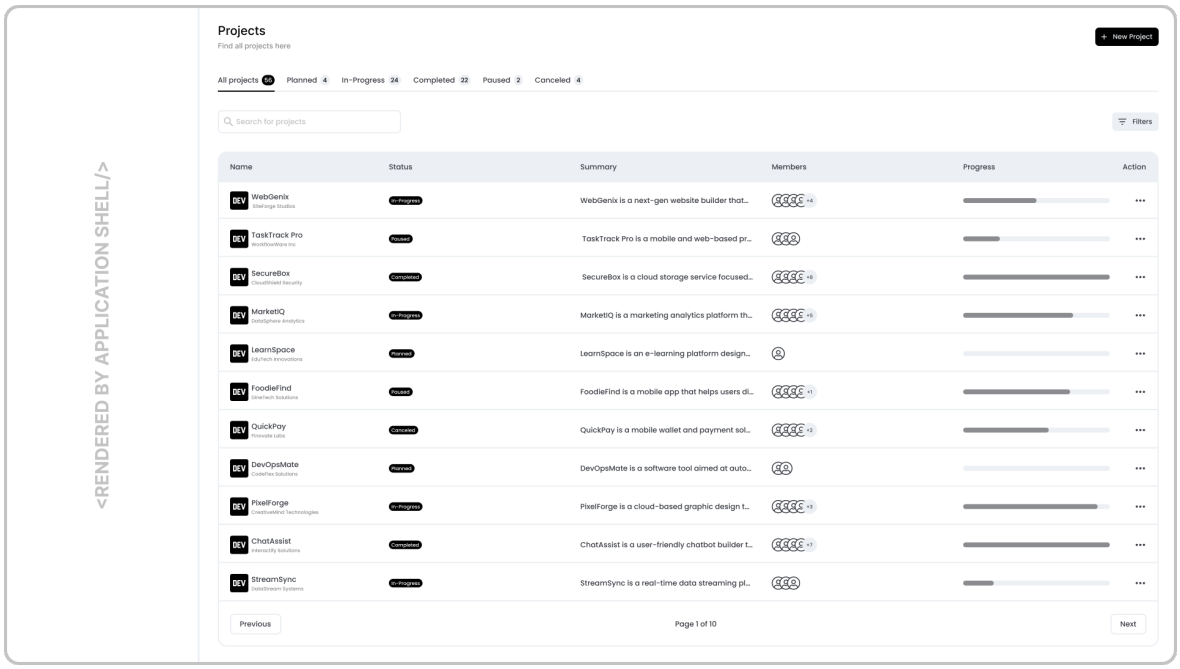


Figure 4.3: Projects microfrontend wireframe

4.5.5 Dashboard Microfrontend

The Dashboard microfrontend provides an overview of key metrics and insights for the current user. At the top of the dashboard, there is a row of summary cards displaying quick statistics for the past month, compared to the previous month. These metrics include: number of active projects, number of new projects, number of completed projects, total number of active tasks, number of new tasks, number of completed tasks. Below this summary, a task completion graph visually represents the number of tasks completed each day over the last month. This graph allows users to track productivity trends at a glance. To the right of the graph is a list of today's tasks, providing a convenient view of tasks assigned to the user that are due or in progress for the current day. Underneath the graph is a list of new members, showing users who have recently joined the system. To the right of this section is a list of the user's projects, displaying each project and its current progress, allowing users to quickly assess the status of their ongoing work.

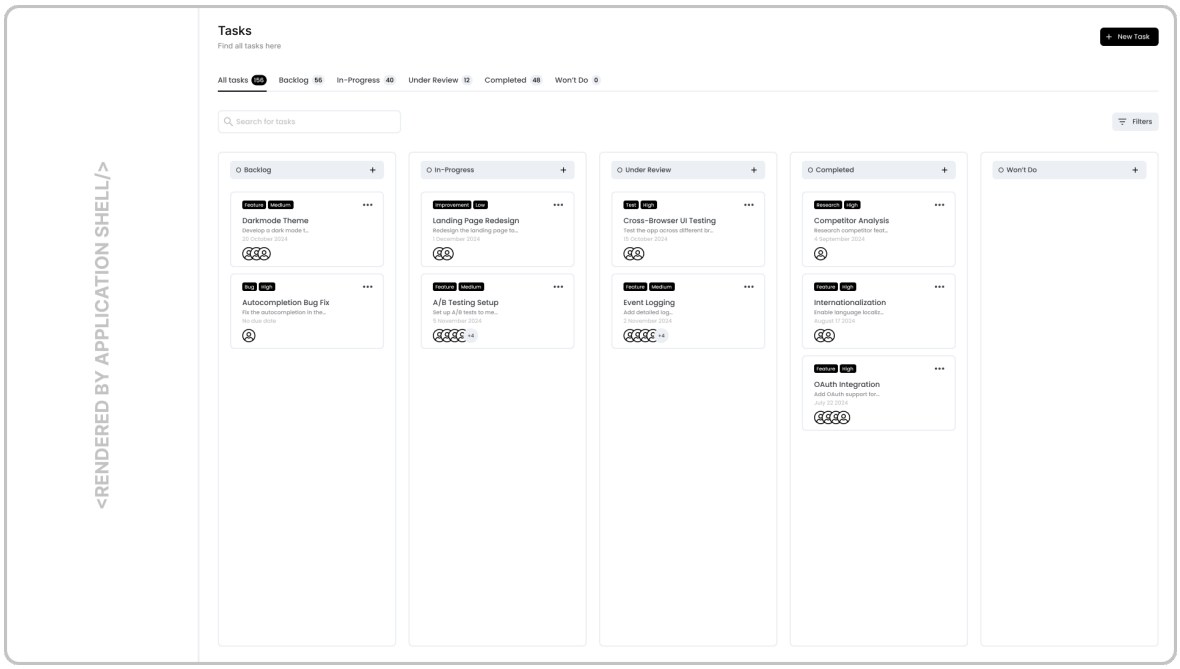


Figure 4.4: Tasks microfrontend wireframe

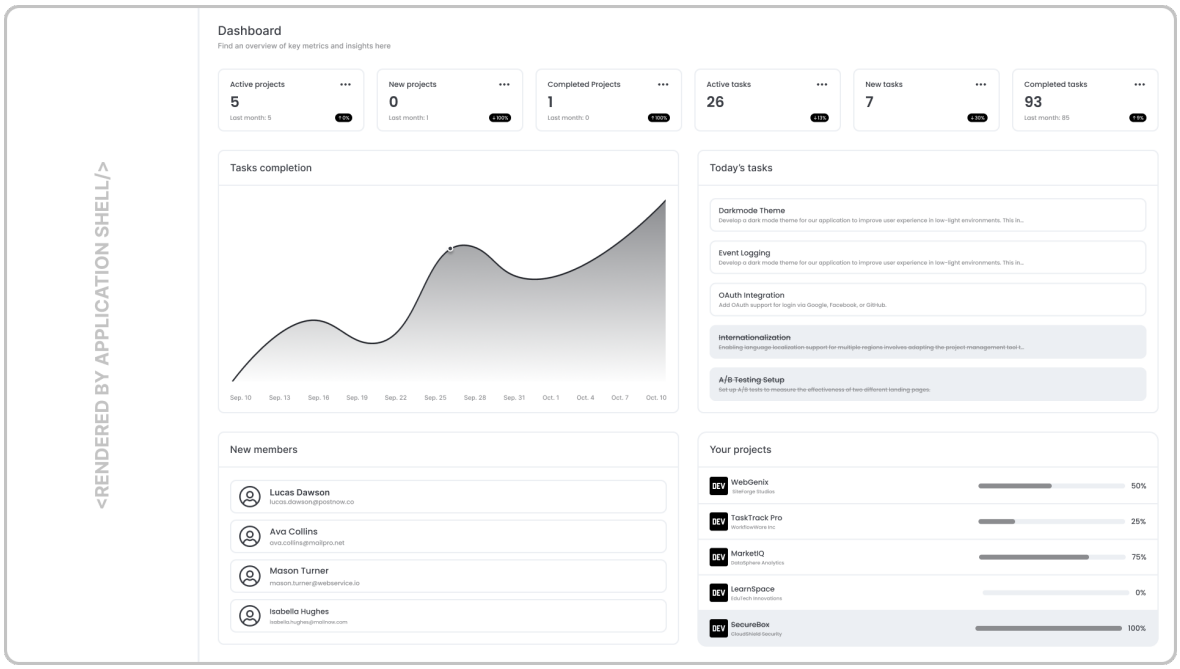


Figure 4.5: ashboard microfrontend wireframe

Chapter 5

Implementation

Provide a description outlining the purpose and content of this chapter, detailing the topics being discussed herein.

5.1 Overview

The core of the application consists of four Angular applications within the same workspace. The first application developed was the application-shell. The process started by generating an empty Angular workspace via the command line:

```
ng new project-management-tool --create-application=false
```

This was followed by generating the application-shell itself:

```
ng generate application application-shell
```

Afterwards, Bootstrap was installed and configured in the standard way, along with @ngx/translate, a library for internationalization support. Interfaces were then designed using placeholder components for microfrontends. Support for theme (light, dark) and language (Slovak, English) switching was added, with preferences saved in local storage. The application-shell was developed as a standard Angular application.

Microfrontends were generated in the same way, but with the use of the –prefix option, such as:

```
ng generate application user-management -prefix=user
```

Microfrontends were further developed similarly to the application-shell. However, they differ in how the application is bootstrapped. Depending on the environment, the application is bootstrapped differently. For local standalone microfrontend development,

the application is bootstrapped in the standard way using the main AppComponent. However, for the embedded version inside the application-shell, a special entry component is used instead. This component is registered as a custom element rather than being directly bootstrapped. The entry component acts as a wrapper for the application and handles all attributes received from the application shell. The code for this part looks as follows:

```
const CUSTOM_ELEMENT_NAME = 'user-management';

async function initializeApp(): Promise<void> {
   try {
      if (!customElements.get(CUSTOM_ELEMENT_NAME)) {
         if (document.querySelector('user-root')) {
         await bootstrapApplication(AppComponent, appConfig);
         } else {
         const app = await createApplication(appConfig);

         const customElement = createCustomElement(EntryComponent, {
            injector: app.injector,
         });

         customElements.define(CUSTOM_ELEMENT_NAME, customElement);
         }
      }
   } catch (error) {
      console.error('Failed to initialize application:', error);
   }
}

initializeApp();
```

However, when Angular applications are built, they consist of multiple JavaScript files. Therefore, bundling them into a single JavaScript file is necessary. For this purpose, Webpack is used with the following configuration:

```
const path = require("path");
const fs = require("fs");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");

const deleteFiles = (files) => {
```

```
    files.forEach((file) => {
      try {
        if (fs.existsSync(file)) {
          fs.unlinkSync(file);
          console.log(`Deleted: ${file}`);
        }
      } catch (err) {
        console.error(`Error deleting ${file}:`, err);
      }
    });
  };

  const inputFiles = [
    path.resolve(__dirname, "../../dist/user-management/browser/polyfills.js"),
    path.resolve(__dirname, "../../dist/user-management/browser/styles.css"),
    path.resolve(__dirname, "../../dist/user-management/browser/main.js"),
    path.resolve(__dirname, "../../dist/user-management/browser/scripts.js"),
  ];

  module.exports = {
    entry: {
      "bundle.js": inputFiles,
    },
    output: {
      filename: "[name]",
      path: path.resolve(__dirname, "../../dist/user-management/browser"),
    },
    module: {
      rules: [
        {
          test: /\.css$/i,
          use: ["style-loader", "css-loader"],
        },
      ],
    },
    plugins: [new CleanWebpackPlugin()],
    infrastructureLogging: {
      level: "info",
    },
```

```
  plugins: [
    {
      apply: (compiler) => {
        compiler.hooks.done.tap("DeleteInputFilesPlugin", () => {
          console.log("Cleaning up input files...");
          deleteFiles(inputFiles);
        });
      },
    },
  ],
};
```

The microfrontends are then statically served using http-server, each on a different port via the command line, for example:

```
npx http-server dist/user-management/browser -p 4201
```

To load them in the application-shell, a script is appended to the DOM tree upon route request using lazy loading. The URL points to the respective microfrontend's JavaScript file. For the user-management microfrontend, the URL would be:

```
http://localhost:4201/bundle.js
```

5.2 Communication

There are two types of inter-application communication occurring.

Applicaition-shell to microfrontend
The first type is communication from the application shell to a microfrontend. This type of communication occurs when we need to pass some attributes to the microfrontend, such as the current route or language. It also occurs when an attribute has changed to a new value, and the microfrontend needs to be rendered. Since the microfrontends are built as custom elements, we can pass any number of attributes as needed. However, there must be proper handling implemented in the microfrontend. Therefore, the application-shell teams and microfrontend teams must define a contract beforehand, describing which types of attributes can be passed between them.

To pass an attribute, we simply specify it in the HTML as follows:

```
<user-management language='en'></user-management>
```

In the microfrontend's entry component, we would then write:

```
@Input() language: string;
```

And handle it as needed.

Microfrontend to microfrontend

The second type of communication involves communication between microfrontends or from a microfrontend to the application shell. This type of communication is handled via browser Custom Events. A custom event can be created and dispatched like this:

```
const customEvent = new CustomEvent(eventName, { data: ... });
window.dispatchEvent(customEvent);
```

It can be listened to as follows:

```
window.addEventListener(eventName, (event) => {
   console.log('Custom event received:', event.data);
});
```

5.3 Routing

Routing is primarily handled in the application shell, for the most part using the standard approach via @angular/router. When a route for a component inside the application shell is requested, we simply load the component in the standard way. If a route for a microfrontend is requested, we first check if the microfrontend can be loaded via a route guard. If it can, we then lazy load the microfrontend by appending a script that points to the correct URL. This script subsequently renders the microfrontend.

If a microfrontend has multiple routes of its own, this requires specific handling. In such cases, a route change listener in the application shell will detect route changes. When a route is changed, the application shell will pass an updated attribute for the route to the microfrontend, causing it to re-render. The microfrontend will have its own router, which uses the route information from the application shell to load the correct page.

The microfrontend itself must also listen for its own route changes. If the route is changed within the microfrontend, it will inform the application shell via CustomEvents. The application shell will then handle the routing accordingly and pass the new route to the microfrontend.

5.4 Styling

- Describe the challenge posed by CSS in a micro-fronted architecture, where styles are global, inherit, and cascade without the support of a module system, namespacing, or encapsulation.
- Highlight the necessity of ensuring that each micro frontend doesn't conflict with others regarding CSS properties.
- Explain the approach taken to address these challenges, emphasizing the need for consistency in the graphical user interface (GUI) across all micro frontends.
- Discuss the implementation of a shared UI component library as a solution to promote consistency and streamline development efforts.
- Describe how was static assets sharing managent.

5.5 Resource sharing

5.6 Infrastructure

- Discuss the type of repository employed (Mono-repo/Multi-repos) and reasons behind its selection.
- Enumerate all automated workflows which were utilized.
- Highlight any additional tools employed
- Describe the deployment process

5.7 Testing

- Detail the types of tests utilized.
- Explain the testing process, including any automation implemented.
- List the testing tools utilized.

Chapter 6

Conclusion

Provide a description outlining the purpose and content of this chapter, detailing the topics being discussed herein.

6.1 Implementation Results

- Presents the findings and outcomes from the implementation and analysis of microfrontend architectures.
- Cover key metrics, performance indicators, and notable observations to support the thesis's conclusions.

6.2 Practical Implications

Discuss the circumstances under which microfrontends are suitable and when they may not be the optimal solution.

6.3 Recommendations for Future Work

Outline potential areas for future research and improvement in microfrontend architectures.

[?]

Bibliography

[1] L. Mezzalira, *Building Micro-Frontends*. O'Reilly Media, Inc., 2021.

[2] M. Geers, *Micro Frontends in Action*. Manning Publications, 2020.

[3] A. Pavlenko, N. Askarbekuly, S. Megha, and M. Mazzara, "Micro-frontends: application of microservices to web front-ends," *Journal of Internet Services and Information Security*, vol. 10, pp. 49–66, 2020.

[4] C. Richardson, *Microservices Patterns: With examples in Java*. Manning Publications, 2018.

[5] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 2021.

[6] S. Peltonen, L. Mezzalira, and D. Taibi, "Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review," *Information and Software Technology*, vol. 136, p. 106571, 2021.

[7] A. Montelius, "An exploratory study of micro frontends," Master's thesis, Linköping University, Software and Systems, 2021.

[8] C. Jackson, "Micro frontends." `https://martinfowler.com/articles/micro-frontends.html`, 2019. Accessed: 2024-05-12.

[9] Google, "Angular - the modern web developer's platform." `https://angular.dev/`, 2025. Accessed: 2025-02-01.

[10] Meta, "React - a javascript library for building user interfaces." `https://react.dev`, 2024. Accessed: 2024-02-09.

[11] Amazon, "Amazon official website." `https://www.amazon.com`, 2024. Accessed: 2024-02-09.

[12] R. Brigham and C. Liquori, "Devops at amazon: A look at our tools and processes." Presentation at AWS re:Invent, 2015. Accessed: 2024-02-09.

[13] Netflix, "Netflix official website." `https://www.netflix.com`, 2024. Accessed: 2024-02-09.

[14] Spotify, "Spotify official website." `https://www.spotify.com`, 2024. Accessed: 2024-02-09.

[15] Uber, "Uber official website." `https://www.uber.com`, 2024. Accessed: 2024-02-09.

[16] A. Kwiecień, "10 companies that implemented the microservice architecture and paved the way for others." `https://www.cloudflight.io/en/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-wa` 2019. Accessed: 2024-01-18.

[17] M. Fowler and J. Lewis, "Microservices." `https://martinfowler.com/articles/microservices.html`, 2014. Accessed: 2024-04-04.

[18] C. Richardson, "Microservices.io," 2024. Accessed: 2024-03-15.

[19] "Thoughtworks 'technology radar: Micro frontends'." `https://www.thoughtworks.com/radar/techniques/micro-frontends`, 2016. Accessed: 2024-09-10.

[20] M. Geers, "Micro-frontends - extending the microservices idea to frontend development." `https://micro-frontends.org`, 2017. Accessed: 2024-08-26.

[21] "Broadcast channel api." `https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API`, 2025. Accessed: 2025-02-17.

[22] N. Salaheddin and N. Ahmed, "Microservices vs. monolithic architectures (the differential structure between two architectures)," *MINAR International Journal of Applied Sciences and Technology*, vol. 4, pp. 484–490, 2022.

[23] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20357–20374, 2022.

[24] "<iframe>: The inline frame element." `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe`, 2024. Accessed: 2024-11-30.

[25] "Ajax." `https://developer.mozilla.org/en-US/docs/Glossary/AJAX`, 2024. Accessed: 2024-11-26.