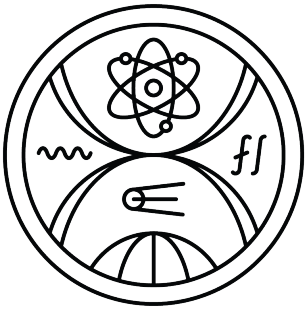
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS

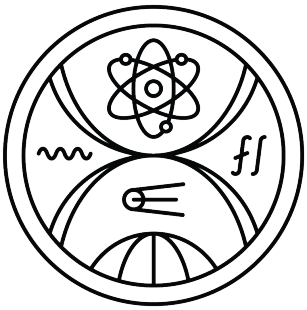


ANALYSIS, DESIGN AND IMPLEMENTATION OF MICRO-FRONTEND ARCHITECTURE

Diploma thesis

2025 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, 2025 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

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dátum zadania: 05.10.2023

Dátum schválenia: 05.10.2023 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

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

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

....

Student Supervisor

Declaration

I hereby declare that I have completed this thesis independently, without any assistance from third parties, and without using any sources or aids other than those explicitly cited.

.....

Bratislava, 2025 Bc. Pavol Repiský

Acknowledgement

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

Abstrakt

Táto práca sa zaoberá mikrofrontendmi vo viacvrstvových podnikových webových aplikáciách. V prvých kapitolách poskytuje prehľad súčasnej literatúry o architektúre mikrofrontendov a porovnáva sedem najčastejšie používaných prístupov k mikrofrontendom z viacerých hľadísk, ako sú rozšíriteľnosť, znovupoužiteľnosť, jednoduchosť, výkonosť, zdieľanie zdrojov, používateľská skúsenosť vývojára a samotné využitie. V závere prehľadovej časti identikujeme tri prístupy, ktoré sú vhodné pre podnikové aplikácie. Z týchto prístupov sme vybrali prístup, ktorý považujeme za najperspektívnejší, konkrétne Web Components, pre implementáciu prototypovej aplikácie. Vybraná aplikácia predstavuje zjednodušenú verziu nástroja na správu projektov. Práca opisuje návrh a implementáciu tejto prototypovej aplikácie. Na implementáciu sme zvolili framework Angular [1], ktorý patrí medzi najpoužívanejšie frameworky na vývoj moderných frontendov v podnikových webových aplikáciách. V závere práce vyhodnocujeme implementovanú aplikáciu z viacerých hľadísk, vrátane rozšíriteľnosti, znovupoužiteľnosti, zdieľania zdrojov a správy aplikačného stavu.

Kľúčové slová: Microfrontends, Web Components, architektúry webových aplikácií, Angular

Abstract

This thesis addresses microfrontends in multi-tier enterprise web applications. First, it reviews the current literature on microfrontend architecture and compares the seven most commonly used approaches to microfrontends from several aspects, such as extensibility, reusability, simplicity, performance, resource sharing, developer experience, and usage. It identifies three approaches that are suitable for enterprise applications. From these, it selects the approach, we consider to be the most promising, namely, Web Components, for the implementation of a prototypical application. The chosen application is a simplified version of a project management tool. The thesis describes the design and implementation of this prototypical application. For the implementation, we chose the Angular [1] framework, which is one of the most widely used frameworks for developing modern frontends in enterprise web applications. Finally, we evaluated the implemented application from several aspects, including extensibility, reusability, resource sharing, and application state management.

Keywords: Microfrontends, Web Components, web application architectures, Angular

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This chapter provides a brief introduction to microfrontends and the motivations behind this thesis. It also offers an 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 smaller, independent applications to better address issues such as complexity, maintainability, and deployability [2, 3, 4].

While microservices focus on the partitioning of the application layer, microfrontends deal with the partitioning of the frontend (presentation) layer of an application. Thus, despite some similarities, microservices and microfrontends address different sets of problems and use different techniques and technologies. Another important difference is that there are several accepted principles and patterns in microservices [5, 6], while in microfrontends, there is not yet consensus on which principles are most appropriate, nor is there a set of design patterns for microfrontends.

Several articles [4, 7, 8, 9] 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 one another. An overview of the basic approaches to microfrontends has been provided by Geers [3] in his book *Micro Frontends in Action*. This book not only summarizes the basic approaches to creating microfrontends and provides simple examples, but also compares 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 [2].

1.2 Aim

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

Subsequently, one of these prospective approaches will be selected to validate its suitability and correctness by implementing a simple prototype microfrontend application in an enterprise technology such as Angular [1] 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 microfrontends, 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 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 their brief history. It further 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 microfrontends adoption.

2.1 History of Microservices and Microfrontends

By the beginning of this millennium, some software systems had become so large that they were difficult to maintain. The most famous example is Amazon's [11] e-shop, 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 the build to the 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 generalized 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* [6]. In parallel, Chris Richardson created his website [18] in 2014. He later compiled the ideas from this site into a separate book, *Microservices Patterns* [5].

The term *Microfrontends* first appeared in the *ThoughtWorks Technology Radar*

magazine [19] at the end of 2016. Through six subsequent editions, it climbed from the "assess" and "trial" sections 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* [3]. In 2021, a second monograph by Luca Mezzalira, *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].

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 the 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. In contrast, vertical decomposition 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 tool 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 embrace 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 Introduction to Microfrontends

The main idea behind microfrontends is to extend the principles of microservices to the frontend side [8, 3]. In microfrontends, the presentation layer is split into smaller, more manageable pieces [8]. Jackson [9] describes microfrontends as: "An architectural style where independently deliverable frontend applications are composed into a greater whole." Each microfrontend can be developed, tested, and deployed independently while still appearing to customers as a single cohesive product [9].

As Geers [3] describes, microfrontends are not just an architectural approach but also an organizational one. Microfrontends can be combined with microservices on the backend, resulting in a system divided into several full-stack micro-applications [8]. Each such micro-application is autonomous, with its own continuous delivery pipeline and serving a specific business domain or feature [7]. Microfrontends introduce vertical teams instead of horizontal ones. Instead of being grouped by the development technologies they use, teams are grouped by application features [8]. Ideally, each team works on a single full-stack micro-application, developing it all the way from the presentation layer to its data layer. The team takes complete ownership of the feature, decentralizing decision-making and determining what technology to use. Figure 2.1, originally presented by Geers [20], shows a high-level diagram illustrating how an application can be divided into vertical microapplications in the context of a simple e-shop. Each microapplication is managed by a separate team, responsible for everything from the presentation layer to the persistence layer, with each team focused on a specific business objective or mission.

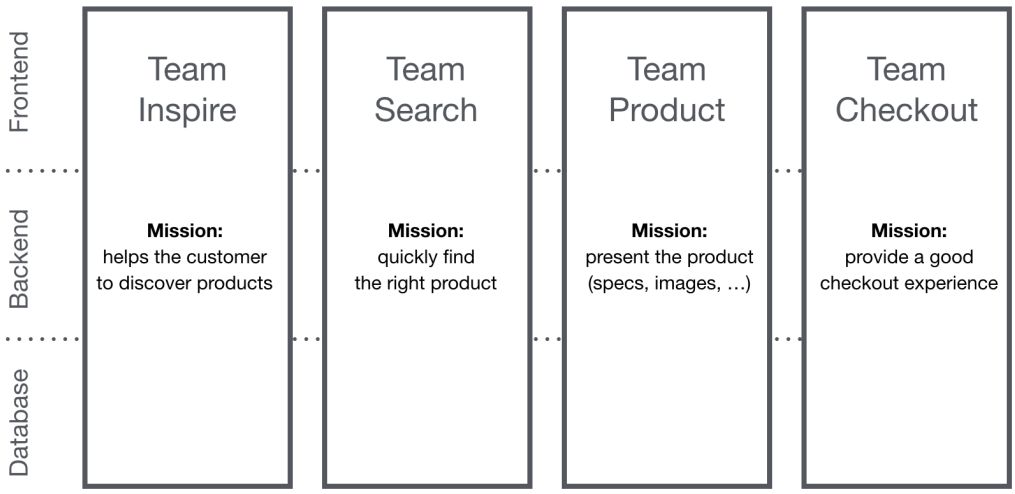


Figure 2.1: High-level architecture of a microfrontend-based application, source [20].

2.4 Microfrontends Challenges

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

2.4.1 Communication

In an ideal world, we would like microfrontends not to 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 (SPA), 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.

The appropriate method for managing communication varies depending on the project type and the 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 [2]. The second option is a variation of the previous one, where we utilize Custom Events [21, 2, 3]. Most browsers now also support the new *Broadcast Channel API* [22], which allows the creation of an event bus that spans across browser windows, tabs, and iframes [3]. Lastly, there are less flexible and secure options, such as communication via query strings [2] or utilizing web storage.

2.4.2 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 [2] describes, in the case of server-side composition (see Subsection 3.2.4), all the logic to build and serve the application happens on the server; therefore, routing must also be handled there. 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 content delivery network (CDN); however, this will not work for dynamic or personalized data. When using edge-side composition (see Subsection 3.2.5), the composition happens at the CDN level. The 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 [2].

There is also 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 [2]. Additionally, these approaches can also be combined, such as CDN and origin or client-side and CDN [2].

2.4.3 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 using 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 [4]. Additionally, each microfrontend can request its assets via an HTTP request inde-

pendently, or in the case of small assets, they can be bundled inside the microfrontend itself.

2.4.4 Reusability

Microfrontends introduce a lot of code redundancy, as many components must be implemented repeatedly across multiple microfrontends. 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 [23], which offers shared dependency management even for different dependency versions. However, there are currently no universal solutions, and each project must be assessed individually.

2.4.5 Styling Consistency and Isolation

Since all microfrontends are developed by autonomous teams, the user interface (UI) and user experience (UX) can differ significantly between 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 [7], 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 each of them. A more universal solution is to use a common style guide, such as Bootstrap [24] or Material Design [25], 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. 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.

2.4.6 Project Organization

A monolithic system can be easily stored in a single repository inside a Version Control System (VCS), but with microfrontends, this 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 [4].

As Pavlenko [4] explains, 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 through it. However, this increases the complexity of developer tools and maintenance [4].

The last approach is to have multiple repositories again, but manage them via the git-repo tool [26]. 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 eliminated, 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 [4].

2.5 Microfrontends Tradeoffs

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

2.5.1 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 [3, 8, 7]. Geers [3] 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 [3, 8]. Additionally, teams have much more freedom to make case-by-case decisions regarding individual parts of the product [9, 3], allowing them to become experts in their respective areas of the application [8].

Microfrontends also bring many of the benefits of microservices architecture to the frontend. The codebases are smaller, more understandable, and less complex [3, 9, 8]. Consequently, this can lead to shorter onboarding times for new developers [7]. Each microfrontend is isolated, independently deployable, and its failure does not affect the

rest of the system [7, 8, 9, 3].

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 [9, 8], and each microfrontend can potentially be developed using different technologies. This encourages developers to experiment with new tools and quickly adapt to changing requirements [7].

2.5.2 Disadvantages

However, microfrontends are not a universal solution. Having an application split into multiple parts across multiple teams, potentially using different technologies, naturally introduces a lot of redundancy. The redundancy can be in terms of actual code, where multiple teams implement the same functionality repeatedly, but also in terms of common dependencies, which, if not handled properly, will be included in multiple microfrontends. Furthermore, if teams are using different technologies or even just different versions, they must all be bundled individually [3, 7]. This all leads to a larger payload size, with the browser having to fetch more data, which can negatively affect the performance of the application [7, 8, 9]. Each team also needs to set up and maintain its own application server, build processes, and CI/CD pipelines.

As already mentioned in the challenges section 2.4, this architecture also complicates otherwise well-established techniques such as routing, communication, static asset serving, and styling, requiring them to be handled in a specialized way.

Microfrontends can potentially add a lot of unnecessary complexity at both the technical and organizational levels [7]. They require a significant amount of knowledge and analysis about a project before development begins [8, 7]. There are risks associated with developing in a standalone environment that is quite different from production, requiring extensive integration testing [9, 8, 7]. This differs from microservices, as they do not need to be integrated into a single application. Lastly, microfrontends could potentially lead to varying code quality if requirements are not clearly defined across all teams [8].

2.6 Microfrontends in Practice

This section presents a list of well-known medium and large companies that have adopted microfrontends as their main system for scaling their business further.

Amazon

The first noteworthy example is Amazon [11]. Although Amazon does not publicly share its internal architecture, according to Geers [3], several Amazon employees have reported that the e-commerce site has been using this architecture for quite some time. Amazon supposedly employs a UI composition technique that assembles different parts of the page before it is displayed to the customer.

IKEA

Another well-known e-commerce platform is IKEA [27]. IKEA's principal engineer, Gustaf Nilsson Kotte, shares in an interview [28] that they started experiencing the same problems with the frontend monolith as they had with the backend monolith. This led them to adopt the microfrontend architecture. They decided to use the Edge Side Includes (ESI) composition approach and introduced the concept of pages and fragments. Pages can contain ESI references to fragments. The fragments are self-contained, meaning they include everything they need, such as CSS and JavaScript, and are reused across multiple pages. Teams are responsible for a set of pages and fragments.

Zalando

The European fashion e-commerce platform Zalando [29] has also adopted microfrontends. Zalando even open-sourced its microfrontend framework, "Tailor.js" [30], which was later replaced by the Interface framework. Compared to Tailor.js, the Interface framework is based on similar concepts but is more focused on components and GraphQL instead of fragments [2].

Spotify

An example of an unsuccessful adoption is Spotify [14]. As Mezzalira [2] describes, their desktop application initially used iframes to compose the UI, communicating via a "bridge" for the low-level implementation made with C++. Spotify also attempted to use this approach when developing the web version of the Spotify player but abandoned it due to poor performance. Since then, they have reverted to a single-page application (SPA) architecture.

SAP

Next on the list is SAP [31]. SAP initially utilized iframes and eventually released its own microfrontend framework, "Luigi" [32], designed for creating enterprise applications

that integrate with SAP systems. It supports modern enterprise frontend frameworks such as Angular, React, Vue [33], and SAP's own SAPUI [2].

DAZN

The last on our list is DAZN [34], a sports streaming platform. DAZN has migrated its monolithic frontend to a microfrontend architecture [3]. The company focused on supporting not only the web but also multiple smart TVs and gaming consoles [2]. They chose a client-side approach to composition. As Mezzalira [2] describes, the platform uses a combination of SPAs and components orchestrated by Bootstrap. They have shared their experiences with microfrontends extensively and eventually even published a book on the topic [2].

Chapter 3

Microfrontends in Detail

This chapter provides a deeper exploration of the most commonly used microfrontend implementation approaches and their characteristics.

3.1 Basic Enablers of Microfrontends

In this section, we introduce key technologies that may be unfamiliar to the reader but are essential for implementing microfrontends and will be referenced frequently throughout the thesis.

3.1.1 Web Components

Web Components are a set of web platform APIs that allow developers to create reusable, encapsulated, and customizable HTML elements [35]. They are now supported by most major browsers and work across different frameworks and libraries [36]. We will now discuss the three main technologies that make up Web Components.

HTML Templates

HTML templates are an addition to the HTML language. They allow us to define reusable HTML structures that are not immediately rendered in the document. Instead, they remain inactive until they are cloned via JavaScript [35]. To create such a structure, we utilize the `<template>` element, which we can then clone and add to the DOM via JavaScript.

```
1  <template id="my-template">
2    <p>Some content goes here...</p>
3  </template>
4
5  let template = document.getElementById("my-template");
```

```
6   let clone = template.content.cloneNode(true);
7   document.body.appendChild(clone);
```

Listing 3.1: Example of using HTML template element

A `<slot>` element can be used inside the template as a placeholder for content passed from the outside. If no content is provided, a default fallback value can be specified. Templates and slots can then be used inside a custom element as the basis of its structure [35].

Custom Elements

Custom elements are a set of JavaScript APIs for creating custom HTML elements with specific behavior [35]. To register a custom element, we use the `customElements` global object and its `define` method, to which we pass a hyphenated tag name and a class that inherits from the `HTMLElement` class. Lifecycle hooks can then be leveraged inside the class to adjust its behavior, such as:

- `connectedCallback()` - this lifecycle hook is fired when the element is connected to the DOM,
- `disconnectedCallback()` - this lifecycle hook is fired when the element is disconnected from the DOM,
- `attributeChangedCallback(name, oldValue, newValue)` - this lifecycle hook is fired whenever one of the observed attributes is changed.

To define the content of a custom element, we assign an HTML string to its `innerHTML` property. The element can then be used inside an HTML document like any other element.

```
1   class MyElement extends HTMLElement {
2     constructor() {
3       super();
4     }
5
6     connectedCallback() {
7       console.log("Element connected to the DOM")
8     }
9   }
10
11  customElements.define(
12    "my-element",
13    class MyElement extends HTMLElement {}
```

```
14  );
15
16  ...
17  <my-element attribute="value"></my-element>
```

Listing 3.2: Example of creating and using a custom element

Shadow DOM

Lastly, Shadow DOM is a set of JavaScript APIs that enable us to attach an encapsulated "shadow" DOM subtree to an element, which is then rendered separately from the main document DOM [35]. The styles and scripts placed inside it do not leak out and affect the surrounding DOM, nor do outside styles and scripts leak into it. This way, we do not have to worry about collisions with other parts of the document [36]. To use Shadow DOM inside a custom element, we must first attach it by calling `attachShadow(options)` and specifying the mode (`open` or `closed`), then append whatever content we want to it.

```
1  class MyElement extends HTMLElement {
2    constructor() {
3      super();
4      let shadow = this.attachShadow({ mode: 'open' });
5      let div = document.createElement('div');
6      div.textContent = "Some content goes here...";
7      shadow.appendChild(div);
8    }
9  }
```

Listing 3.3: Example of using Shadow DOM in a custom element

Open Shadow DOM can be accessed via JavaScript using `element.shadowRoot`. Closed Shadow DOM cannot be accessed externally using `element.shadowRoot` [35].

3.1.2 Custom Events

Unlike built-in events like `click` or `keydown`, custom events do not natively exist in the browser but are created and dispatched manually via JavaScript. Custom events allow developers to define their own event names and pass any custom data [21]. The `CustomEvent` constructor is used to create such events, to which we pass an event name and an optional object where we can specify the data we want to send and the properties of the event, such as the following:

- `bubbles`: `true` - Allows the event to propagate upward to the parent element. This is set to `false` in custom events by default. Propagation can be stopped

using the `stopPropagation()` method.

- `cancelable: true` - Allows the event to be canceled via the `preventDefault()` method.
- `composed: true` - Allows the event to propagate from the shadow DOM to the real DOM. In this case, the `bubbles` property must also be set to `true` [37].

The event can then be dispatched on a specific element using the `dispatchEvent()` method and listened to using the `addEventListener()` method. Putting it all together, we would create a custom event as show in the example bellow.

```
let myEvent = new CustomEvent("my-event", {
  detail: {key: "value"},
  bubbles: true,
  cancelable: true,
  composed: false,
});

...
document.dispatchEvent(myEvent);

...
document.addEventListener("my-event", function (e) {
  console.log("My event value:", e.detail.key);
});
```

Listing 3.4: Example of creating and handling custom events

3.2 Composition Approaches

There are multiple approaches to implementing microfrontends. Some are as simple as linking between different applications, while others could fill an entire book. However, the important thing to note is that there is no single best approach; the most suitable one depends on the project requirements and the team's knowledge.

This section introduces seven of the most commonly mentioned approaches in the literature, outlining their advantages, disadvantages, and suitability. Based on both the reviewed literature and our own insights, each approach is rated on a scale from 1 (lowest) to 5 (highest) across the following aspects.

- **Extensibility**
  - 1: A tightly coupled, monolithic-like frontend where adding a new microfrontend requires modifying other parts of the codebase.

  - 5: A modularized approach where new microfrontends can be easily added without affecting existing ones.

- **Reusability**
  - 1: Microfrontends have minimal or no customization options.
  - 5: Each microfrontend is highly customizable to fit different environments.

- **Simplicity**
  - 1: Complex integration requiring custom solutions and extensive boilerplate code.
  - 5: A straightforward, declarative composition approach with minimal extra configuration.

- **Performance**
  - 1: Adding a new microfrontend significantly impacts performance.
  - 5: Performance remains comparable to or better than that of a SPA.

- **Resource Sharing**
  - 1: Each microfrontend loads its own separate versions of frameworks, styles, and assets.
  - 5: Common dependencies and assets can be efficiently deduplicated.

- **Developer Experience**
  - 1: A complex and unfamiliar local development setup.
  - 5: A streamlined development process similar to established web development techniques such as single-page applications (SPA) or server-side rendering (SSR).

3.2.1 Link-Based Composition

The simplest approach to a microfrontend architecture in this list is the composition of multiple applications using hyperlinks. In this approach, the application is split into multiple microfrontends, which are developed completely separately by dedicated teams. Each application brings its own HTML, CSS, and JavaScript files and is deployed independently, typically on a different port under the same domain, ensuring it remains fully isolated from other applications in the system [3]. To compose the microfrontends, we simply interconnect them across different applications using standard anchor elements.

```
1   <a href="https://example.com/mfe2">View</a>
```

The teams must therefore share all URL patterns of their applications, which will be used to link their sites from other teams' applications [3]. Examples of such URL patterns might look like the following:

- **Team A - Microfrontend 1**
  URL pattern: `https://example.com/mfe1`
- **Team B - Microfrontend 2**
  URL pattern: `https://example.com/mfe2`.

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. Other applications can then read this file at runtime to construct the necessary URLs [3].

Advantages

This composition method offers a couple of unique advantages. It provides complete isolation between microfrontends like no other approach discussed here. Errors in one microfrontend are contained and do not affect any other parts of the system. The microfrontends can be deployed independently, and the project requires minimal extra configuration. The system can be easily expanded by adding new microfrontends [3].

Disadvantages

As Geers [3] mentions, the primary limitation of this approach is that it does not allow for the composition 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 microfrontends is also very complex.

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. It is typically combined with other techniques to create more robust solutions [3]. Our evaluation of this approach can be found in the Table 3.1.

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

Table 3.1: Evaluation of link-based composition

3.2.2 Composition via Iframes

An inline frame (iframe) is an old yet still widely used technique in web development. An iframe is an inline HTML element that represents a nested browsing context, allowing one HTML page to be embedded within another [38]. Each embedded context has its own document and supports independent URL navigation [38]. Compared to link-based composition, iframes allow multiple pages to be combined into a single unified view while maintaining almost the same level of loose coupling and robustness [2]. To add an iframe, we simply use the `<iframe>` tag, similar to an anchor `<a>` tag.

```
1  <iframe src="https://example.com/mfe1"></iframe>
```

Listing 3.5: Example of embedding a microfrontend using an iframe

The iframe can be further restricted using the `sandbox` attribute as follows:

- `<iframe sandbox src="..."></iframe>` - prevents the execution of JavaScript and form submissions.
- `<iframe sandbox="allow-scripts" src="..."></iframe>` - prevents form submissions but allows JavaScript execution.
- `<iframe sandbox="allow-forms" src="..."></iframe>` - prevents JavaScript execution but allows form submissions [2].

Its behavior and appearance can be customized using different attributes [38], and it can communicate with the host page through the `window.postMessage()` method.

Advantages

The biggest advantage of iframes is their excellent robustness and isolation, ensuring that styling and scripts do not interfere with each other. Iframes are also very easy to set up and work with. They are fully supported across all browsers and bring a lot of built-in security features [3, 9, 2].

Disadvantages

However, the main benefit is also the main drawback. It is impossible to share common dependencies across different iframes, leading to larger file sizes and longer download times [4]. It is also difficult to integrate them with each other, making communication, routing, and history management more complicated [9]. The host application must know the height of the iframe in advance to prevent scrollbars and whitespace, which can be fairly complicated on responsive websites [2], [3]. Each iframe creates a new browsing context, which requires a lot of memory and CPU usage. Numerous iframes on the same page can significantly degrade application performance [3]. Lastly, they perform poorly in terms of search engine optimization (SEO) and accessibility [3].

Suitability

Despite the relatively long list of disadvantages, iframes can still be a valid choice for some projects. Iframes shine when there is minimal communication between microfrontends, and the encapsulation of our system using a sandbox for each microfrontend is crucial. The best use cases for iframes are in desktop, Business-to-Business (B2B), and internal applications. However, other approaches should be preferred if performance, SEO, accessibility, or responsiveness are crucial factors [3, 2]. Our evaluation of this approach can be found in Table 3.2.

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: Evaluation of composition via iframes

3.2.3 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 [39]. 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 con-

tent 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 [3].

However, this approach introduces additional challenges, such as CSS conflicts in cases where multiple microfrontends use the same class names, leading to unintended overrides. To avoid this, all CSS selectors should be prefixed with the microfrontend name. Alternatively, tools such as SASS, CSS Modules, or PostCSS [40] can handle this for us. A similar issue can occur with scripts; to avoid it, we can wrap the scripts within Immediately Invoked Function Expressions (IIFEs) to limit the scope to the anonymous function and prefix global variables [3]. The composition can then be handled as follows.

```
<div id="mfe1" data-url="https://example.com/mfe1"></div>

<script>
  const element = document.getElementById("mfe1");
  const url = element.getAttribute("data-url");

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

Listing 3.6: Example of loading microfrontend using AJAX

Since all the microfrontends are included in the same 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. There are no longer 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 [3]. Additionally, this approach reduces initial load time since only essential content is loaded, while other microfrontends are loaded asynchronously later.

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, which can result in conflicts and overwriting among them. Lastly, there are no built-in lifecycle methods for the scripts, and these must be implemented from scratch [3].

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 [3]. Our evaluation of this approach can be found in Table 3.3.

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: Evaluation of composition via Ajax

3.2.4 Server-Side Composition

As Geers explains: "Server-side composition is typically performed by a service that sits between the browser and the actual application servers." After a user requests a page, the initial HTML is generated on the server. The `index.html` file contains common page elements and uses server-side includes (SSI) directives for places where microfrontends should be plugged in [9]. An example page would look as follows:

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Example page</title>
  </head>
  <body>
    <p>Common element</p>
    <!--# include virtual="/mfe2" -->
  </body>
</html>
```

Listing 3.7: Example of server-side composition using SSI

We would serve this file using a web server such as Nginx, which replaces directives with the contents of the referenced URL—microfrontends in our case—before passing the markup to the client. The configuration could look as follows [9].

```
# defines a group of servers that handle requests
upstream mfe1 {
  server team_mfe1:8081;
}
upstream mfe2 {
  server team_mfe2:8082;
}
server {
  listen 8080;
  ssi on; # enables server-side include feature
  index index.html; # all locations should render through
     index.html

  location /mfe2 {
    proxy_pass http://team_mfe2;
  }
  location / {
    proxy_pass http://team_mfe1;
  }
}
```

Listing 3.8: Example of Nginx configuration for server-side composition

Microfrontends can be deployed either as static assets or as dynamic assets, where an application server generates the templates for every user's request [2], with each having its own deployment pipeline. Communication is typically handled via APIs since the page must reload on all significant user actions. However, if some communication between microfrontends must happen on the client side, Mezzalira [2] suggests adding client-side JavaScript and using event emitters or custom events, keeping the microfrontends loosely coupled.

If possible, a CDN layer should be placed in front of the application server to cache as many requests as possible and reduce server load. Several frameworks, such as Podium, Mosaic, Puzzle.js, and Ara Framework, further simplify the implementation.

Advantages

Server-side composition is a proven, robust, and reliable technique. It offers excellent first-load performance, as the page is pre-assembled on the server, positively impacting search engine ranking. Maintenance is straightforward, and interactive functionality can be added via client-side JavaScript. It is also well-tested and well-documented [3, 2].

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 [3]. Furthermore, if web pages are not highly cacheable but are instead personalized per user request, scalability issues may arise due to excessive server load [7].

Suitability

This approach is ideal for pages that prioritize performance and search engine ranking, as it remains reliable and fully functional even without JavaScript [3]. This architecture is recommended for B2B applications with common modules reused across multiple views and teams consisting of full-stack or backend developers. However, it may not be optimal for pages requiring a high level of interactivity [2]. Our evaluation of this approach can be found in Table 3.4.

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

Table 3.4: Evaluation of server-side composition

3.2.5 Edge-Side Composition

Edge-side composition is typically performed at the CDN level. As Mezzalira [2] explains: “Edge-Side Includes (ESI) is a markup language used for assembling different HTML fragments into an HTML page and serving the final result to a client”. CDN providers such as Akamai and proxy servers like Varnish, Squid, and Mongrel all support ESI [3]. An edge-side composition solution is very similar to a server-side one; we swap the Nginx server with, for example, Varnish and use ESI directives instead of SSI. An edge-side include directive looks like the following:

```
1  <esi:include
2    src="https://example.com/mfe1"
3    alt="https://fallback.example.com/mfe1"
```

```
4   />
```

Listing 3.9: Example of edge-side composition using ESI

If the fragment from the `src` URL fails to load, the content from the `alt` URL will be used instead [3]. After the markup language is interpreted, the result is a completely static HTML page renderable by a browser [2].

Advantages

Since pages are composed at the edge, response times are reduced as content is served closer to users. This approach also avoids the server-side issue of sending too many requests to servers, handling high loads more efficiently [2].

Disadvantages

The first drawback of this technique is that ESI is not implemented uniformly across CDN providers or proxy servers. Therefore, adopting a multi-CDN strategy or porting application code from one provider to another could cause serious issues [7]. Another limitation is that ESI cannot be used for dynamic pages. A potential workaround could be adding client-side JavaScript, but this introduces additional complexity [2]. Additionally, due to the limited adoption of this approach, the developer experience is suboptimal, and it lacks documentation and tools compared to other solutions [2].

Suitability

This approach is typically used only for large static websites that are not personalized for individual users. According to Mezzalira [2], the IKEA catalog was implemented in some countries using ESI. Our evaluation of this approach can be found in Table 3.5.

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

Table 3.5: Evaluation of edge-side composition

3.2.6 Composition via Module Federation

This is a relatively new approach, made possible by the release of Webpack 5 [41]. Module Federation is Webpack's native plugin that allows chunks of JavaScript code to be loaded either synchronously or asynchronously [2]. As Mezzalira [2] explains, a Module Federation application consists of two parts.

- The host – Represents the main application that loads microfrontends and libraries. The Webpack configuration for the host could look as follows:

```
new ModuleFederationPlugin({
  name: "host",
  remotes: {
    mfe1: "https://example.com/mfe1/bundle.js",
    mfe2: "https://example.com/mfe2/bundle.js",
  },
});
```

Listing 3.10: Example of Module Federation host configuration

- The remote - Represents the microfrontend or library that will be loaded inside a host at runtime. The Webpack configuration for the remote could look as follows:

```
new ModuleFederationPlugin({
  name: "mfe1",
  filename: "bundle.js",
  shared: ["react", "react-dom"],
});
```

Listing 3.11: Example of Module Federation remote configuration

The composition takes place at runtime, which can occur either on the client side—with the application shell loading different microfrontends—or on the server side, using server-side rendering at the origin [2]. With Module Federation, it is very easy to expose different microfrontends. It also supports sharing external libraries across multiple microfrontends, ensuring they are loaded only once. If some microfrontends use different versions of the same libraries, this is handled automatically by wrapping them in different scopes, preventing conflicts [2]. Module Federation also supports bidirectional code sharing across microfrontends, though this should be avoided, as it goes against microfrontend principles. For communication, event emitters or custom events can be used.

Advantages

The biggest advantage of this approach is that it comes with a lot of tooling, solving most microfrontend challenges, such as lazy loading, library sharing (even with different versions), and microfrontend loading. It supports both client- and server-side rendering. Additionally, it benefits from the Webpack ecosystem, as other plugins can be used to further customize the bundles. It provides a high level of abstraction, making the developer experience almost as smooth as in a monolithic application.

Disadvantages

The main disadvantage of this approach is that it requires expertise in Webpack. Additionally, the ease of code sharing across projects can lead to a very complicated architecture if the team is not disciplined or experienced enough [2].

Suitability

This approach is suitable for most types of projects, as it supports both server- and client-side rendering. However, it is especially well-suited for environments where Webpack is the organizational standard [41]. Our evaluation of this approach can be found in Table 3.6.

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

Table 3.6: Evaluation of composition via Module Federation

3.2.7 Composition via Web Components

Another relatively new approach that is beginning to emerge—thanks to new HTML, JavaScript, and CSS standards—is composition via Web Components. This approach is somewhat of a variation of composition via Ajax. As Web Components were already described in detail in the previous section, this section will focus solely on the implementation.

First, a custom element must be created to represent the microfrontend by extending the `HTMLElement` class and attaching a Shadow DOM to it. To keep things simple, HTML templates will not be used.

```
class MFE1 extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  connectedCallback() {
    let message = this.getAttribute("message");
    this.render(message);
  }

  render(message) {
    this.shadowRoot.innerHTML = `
      <style>p { color: red; }</style>
      <p>${message}</p>
    `;
  }
}
```

Listing 3.12: Example of Web Component composition

We can utilize the `getAttribute(name)` method inside the custom element to obtain attribute values provided by the parent, which can further customize the appearance and behavior of the element. Afterwards, we need to register the element.

```
customElements.define('mfe-2', MFE2);
```

Listing 3.13: Example of Web Component registration

Finally, we can use it anywhere, just like a standard HTML element.

```
<mfe-2 message="Hello World"></mfe-2>
```

Listing 3.14: Example of Web Component usage

For communication, `CustomEvents` are typically used, as discussed in the previous section.

Advantages

Web Components are now a widely implemented web standard that offers strong isolation managed by the browser when the Shadow DOM is used. They provide many built-in lifecycle methods and can be used across different frameworks and libraries, such as Angular and React, making them easier to implement [3]. Web Components also offer a high level of reusability and flexibility.

Disadvantages

One issue with Web Components is that they are not fully supported in older browsers. Polyfilling can extend support for custom elements, but integrating the Shadow DOM is notably more challenging, and polyfills significantly increase the bundle size. Additionally, Web Components lack built-in state management. Handling SEO and accessibility can also be difficult [3][2].

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 ideal solution. Our evaluation of this approach can be found in Table 3.7.

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

Table 3.7: Evaluation of composition via Web-components

3.2.8 Summary

All of the approaches have their set of advantages and disadvantages. We have realized that there is no such thing as the best architecture, a suitable approach should always be selected based on the project specifications. However, there are some conclusions we can draw based on our analysis.

Table 3.8 combines all evaluations into a single, easy-to-read table, allowing for direct comparisons of all the mentioned approaches. Some of the names of the approaches have been shortened so that the table fits well on the page: "LBC" refers to Link-Based composition, "SSI" to Server-side composition, "ESI" to Edge-side composition, "MF" to Module Federation composition, and "WC" to Web Components composition.

What we can generally conclude from the table is that the performance ratings of Link-Based and Iframe compositions are very low, so if performance is a concern for a project, they should be avoided. Ajax offers midrange ratings for most of the aspects. Server-side and edge-side compositions offer good values in terms of extensibility, reusability, and performance, but they are more complex to implement and do

Aspect	LBC	Iframes	Ajax	SSI	ESI	MF	WC
Extensibility	3/5	4/5	4/5	4/5	4/5	4/5	4/5
Reusability	3/5	3/5	4/5	4/5	4/5	4/5	5/5
Simplicity	5/5	4/5	3/5	2/5	2/5	4/5	5/5
Performance	2/5	1/5	3/5	4/5	5/5	5/5	4/5
Resource sharing	1/5	2/5	3/5	4/5	3/5	4/5	3/5
Developer Experience	4/5	4/5	3/5	3/5	1/5	5/5	4/5

Table 3.8: Comparison of different composition approaches

not offer the best developer experience, especially edge-side composition. Compositions via Module Federation and Web Components offer both great results, with the former excelling in resource sharing and developer experience, and the latter in reusability and simplicity. In the case of Web Components, resource sharing is more difficult.

Enterprise-level application development typically involves building large-scale, complex systems that demand scalability, maintainability, strong security, and seamless integration across teams and technologies. Among the available microfrontend composition approaches, server-side rendering, Web Components, and Module Federation stand out as the best fit for enterprise environments. Server-side composition supports SEO and performance optimization, which are critical for enterprise applications. Web Components offer a framework-agnostic, standardized way to encapsulate and isolate features, enabling reusable, decoupled micro-application across teams. Module Federation allows dynamic loading and sharing of code across independently deployed applications, promoting team autonomy and reducing duplication. These approaches align well with the requirements of enterprise-level development.

3.2.9 Decision-Making

When choosing a composition approach for implementing a prototypical microfrontend application, we had to consider its requirements, which we defined in Chapter 4. The application contains multiple microfrontends presented in a unified view; therefore, the link-based composition was immediately ruled out. We also decided to avoid iframes due to the difficulties in communication between microfrontends and issues related to website responsiveness.

One of the most important characteristics of microfrontends is their isolation from the rest of the application, which is not truly achievable with Ajax composition. That is the reason we did not choose it. Our application requires a fairly high level of interactivity; thus, we decided to steer away from server-side and edge-side compositions.

The poor developer experience associated with these approaches also influenced our decision.

This left us with two remaining options: Module Federation and Web Components. Both approaches would have been suitable for our project. However, in the end, we decided to go with Web Components, as they are fully supported by default in Angular, our preferred technology. They are framework-agnostic, offer a high degree of isolation thanks to the Shadow DOM, and can be completely decoupled. Additionally, Module Federation requires extensive upfront knowledge of Webpack.

Chapter 4

Design

This chapter outlines the design considerations essential for implementing the resulting prototypical microfrontends application. It begins with a brief introduction to the application and its purpose. Subsequently, it examines the functional and non-functional requirements, followed by a dedicated section explaining the system architecture. Next, the technologies used for implementation are listed. Finally, graphical user interface mockups are presented and explained.

4.1 Application Introduction

One of the requirements for the application was to focus on the enterprise landscape. The application should be logically divisible into independent business concerns, striking a balance between being overly simplistic—making a microfrontends architecture irrelevant—and overly complex, making development unfeasible within the boundaries of this thesis. A well-known example that meets these criteria is a project management tool. However, a standard project management tool is quite large and includes many features. Therefore, only its core functionality will be implemented, sufficient for a proof-of-concept (PoC) application. At its core, the main purpose is to manage project users and their tasks, which will be the primary focus. The application will offer functionality for creating and managing users, managing tasks, linking tasks to users, and presenting this information in a clear and comprehensible manner. These functionalities will be discussed in more detail in the next section.

4.2 Application Requirements

This section presents the functional and non-functional requirements for the application, 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

This subsection provides a comprehensive list of all functional requirements for the application.

User Management

For simplicity, user registration and login will not be implemented. User roles will serve informational purposes only, as every user will have full access to all functionalities.

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

Task Management

A similar approach applies to tasks: all users will be able to manage any task.

- 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, priority, tag, due date, and description (M).
- Tasks can be assigned to users (M).

Dashboard

- A new users widget is displayed on the dashboard (M).
- A new tasks widget is displayed on the dashboard (M).
- The widgets must communicate with each other (M).
- The dashboard displays a monthly user count graph (C).
- The dashboard displays a monthly task count 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

This subsection presents a comprehensive list of all non-functional requirements for the application.

- The application is divided into several microfrontends (M).
- Each microfrontend is isolated to prevent cascading failures (M).
- The application can be easily scaled by adding new microfrontends (M).
- The application is intuitive and easy to use (S).
- The microfrontends are easily customizable (S).

4.3 System Architecture

This section explores the system architecture of the application, which adopts a microfrontend architecture to separate different business domains into distinct, independently developed, and deployed microfrontends. We present the various microfrontends and other components that make up the system, explaining the rationale behind these segmentation decisions. Figure 4.1 illustrates this architecture using a UML component diagram.

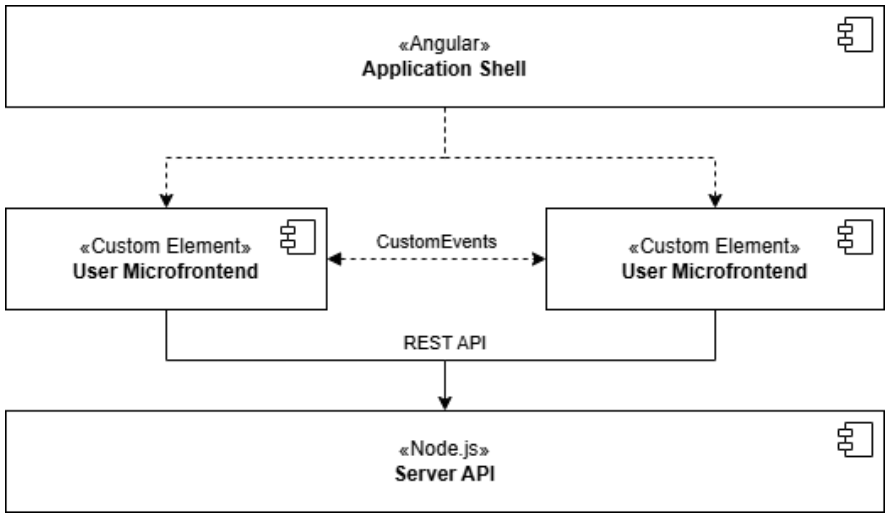


Figure 4.1: Component diagram of the application architecture

4.3.1 Application Shell

The Application Shell is the core of the application; it functions as an orchestrator for all microfrontends. Based on the current route, it loads and renders the appropriate microfrontends. It handles all global functionality—such as routing, theme settings, and language switching—in a centralized manner. It passes any necessary data down to the microfrontends, such as the current language (English or Slovak) or customization options. Common elements, such as navigation and settings, are also located here. Additionally, it renders the dashboard page by combining multiple microfrontends into a single unified view.

The reasoning behind this is clear: we wanted to keep all common elements in one place and handle global functionality centrally. This greatly reduces code redundancy and improves maintainability. The Application Shell only needs to know the URLs where the microfrontends are hosted, as well as their HTML element names and attributes, and it can be developed and deployed independently.

4.3.2 User Microfrontend

This microfrontend is primarily responsible for user management functionality within the application. It handles user management through CRUD operations (Create, Read, Update, Delete) and displays users in a table format. It is a standalone page within the application, rendered independently by the Application Shell. It receives several attributes from the Application Shell: the first is `language`, which the microfrontend should use (default: English); the second is `theme`, either light or dark; and the third is a boolean attribute, `compact`, which defaults to `false`. If this attribute is set to `true`, the microfrontend switches to a compact mode, displaying only a simple list of new users and a monthly user count graph for dashboard purposes, allowing it to be displayed alongside other microfrontends.

Clicking on a user in the list selects them; clicking again deselects them. This action triggers an event to notify other microfrontends about the selection. Additionally, the microfrontend listens for events related to task selection or deselection—in such cases, it displays only users assigned to the selected task.

By following this approach, the user management microfrontend remains completely isolated from the rest of the system. It is reusable, customizable, and serves a single business domain. The only information it needs to function correctly is the names of the events it should listen to.

4.3.3 Task Microfrontend

This microfrontend is primarily responsible for task management. Similar to the user management microfrontend, it supports task CRUD operations (Create, Read, Update, Delete) and displays tasks using a Kanban board. It functions as a standalone page within the application and is rendered independently by the Application Shell. It receives the same attributes from the Application Shell as the user microfrontend: `language`, `theme`, and `compact`.

When the `compact` mode is enabled, the microfrontend displays only a simplified view, including a list of new tasks and a monthly task count graph, making it suitable for use within the dashboard. Clicking on a task selects or deselects it, triggering an event to notify other microfrontends of the selection. Additionally, this microfrontend listens for user selection and deselection events. When a user is selected, it filters the displayed tasks to show only those assigned to the selected user.

The rationale for this design follows the same principles as those applied to the user management microfrontend: modularity, reusability, and isolation of concerns.

4.3.4 Backend

In a real-world microfrontend-based application, each microfrontend would access data through microservices. However, given the scope and constraints of this thesis and extensive existing research on microservices and backend architectures, our focus will remain on the frontend implementation. As a result, we will use a simple, lightweight monolithic backend to support the application's functionality.

4.4 Tech Stack

This section outlines the technologies used in the implementation of the prototypical microfrontend application.

Angular

The primary framework used for application development is Angular 18, the most recent version available at the time of writing. Angular is a robust, platform-agnostic web development framework developed by Google, designed for building scalable, maintainable, and high-performance single-page applications (SPAs). In this project, Angular is used to develop both the Application Shell and the individual microfrontends. The choice of Angular is motivated by its strong adoption in enterprise environments and its built-in support for Web Components.

TypeScript

TypeScript serves as the main programming language for the application. As the standard language for Angular development, TypeScript extends JavaScript by introducing static typing, which helps catch errors during compile time rather than at runtime. This enhances reliability, reduces bugs, and improves code maintainability. Its rich tooling ecosystem and type safety make it particularly well-suited for enterprise-level applications, further justifying its selection for this project.

Web Components

To ensure that each microfrontend operates independently and can be seamlessly integrated into the Application Shell, Web Components are employed. In this implementation, each microfrontend is exposed as a custom element. The Shadow DOM is used to encapsulate styles and avoid CSS conflicts, thereby enforcing strict isolation and reusability across microfrontends.

Custom Events

Communication between the various microfrontends is facilitated using browser-native `CustomEvent` objects. This event-driven approach enables decoupled microfrontends to share data and notify each other of interactions (such as selections) without creating tight dependencies. It ensures a flexible, scalable, and modular system architecture.

Node.js

The backend of the application is implemented using Node.js, a lightweight, event-driven JavaScript runtime environment optimized for scalable network applications. To streamline backend development, Node.js is used in conjunction with Express.js—a minimal and flexible framework that simplifies the creation of RESTful APIs. This backend setup enables rapid prototyping and allows the development effort to remain focused on the frontend aspects of the application.

4.5 Graphical User Interface

This section presents and explains the wireframes of the application. These wireframes provide a visual overview of the layout and user interface design across various parts of the system, highlighting how users will interact with the application. Each wireframe corresponds to a specific microfrontend or the Application Shell and illustrates its core functionalities and structural components.

Application Shell

Figure 4.2: Wireframe of the Application Shell interface

As illustrated in Figure 4.2, the Application Shell features a side panel that includes the application logo and navigation links for accessing each microfrontend, as well as the dashboard page. A settings button is located at the bottom of the side panel; when clicked, it opens a modal containing radio buttons for toggling the application's language (English/Slovak) and theme (light/dark). The main content area, positioned adjacent to the side panel, dynamically renders the active page—either a single microfrontend or the composed dashboard view.

User Microfrontend

As illustrated in Figure 4.3, the User Microfrontend interface features a table in which each row represents a user. Key user details such as name, email, phone number, role, and status are displayed in respective columns. The final column contains an action button that opens a dropdown menu with options to view, edit, or delete the user. Upon selecting any of these actions, a modal window is triggered to perform the desired operation.

Above the table, a set of filter tabs allows users to quickly view subsets of users based on predefined roles (admins, project managers, developers, designers and testers). At the very top of the page is a top bar, which includes the page title, a subtitle, and a button labeled "Add User". Clicking this button opens a modal window for creating a new user.

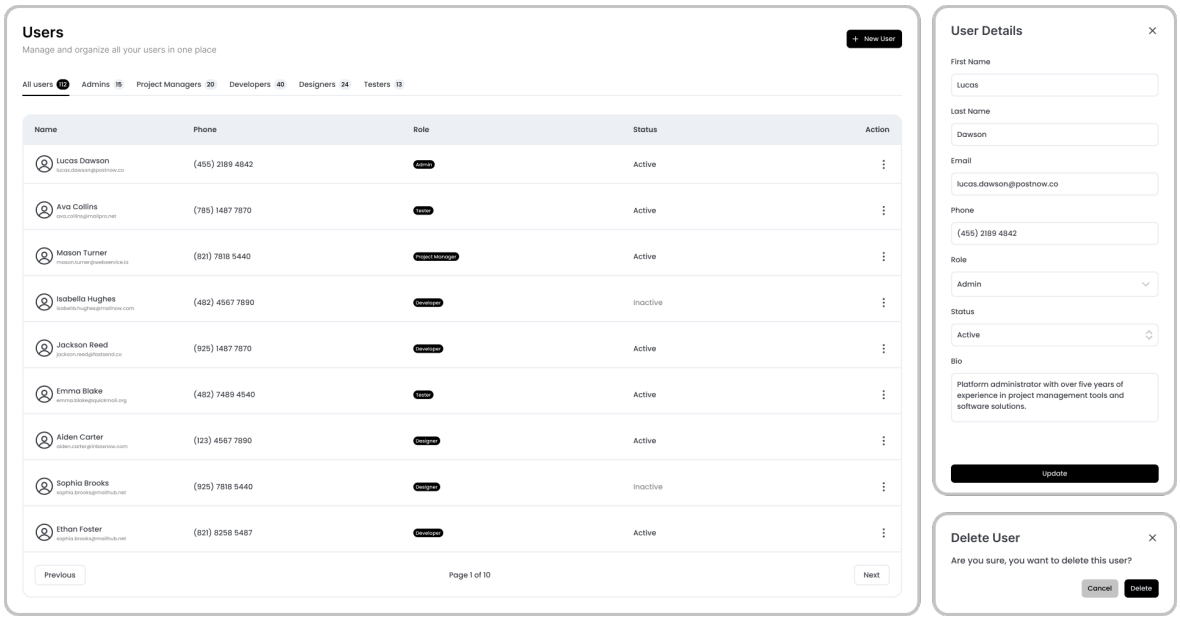


Figure 4.3: Wireframe of the User Microfrontend interface

Task Microfrontend

Figure 4.4 illustrates the Task Microfrontend, which centers around a Kanban board. This board categorizes tasks into distinct stages such as *Backlog*, *In-Progress*, *In-Review*, and *Done*. Each task is visualized as a card that displays key metadata including the title, tag, assignees, and more. A three-dot menu in the top-right corner of each card allows users to view, edit, or delete the task via modal dialogs.

Each stage column features a button for creating tasks directly within that category. At the top of the interface is a top bar containing the page title, a subtitle, and a "New Tas" button that opens a dedicated modal for task creation.

Dashboard

Figure 4.5 shows the Dashboard, which combines the User and Task Microfrontends in compact modes, along with components from the Application Shell. Positioned at the top of the layout is the global top bar, rendered by the Application Shell. Below the top bar, the Task Microfrontend displays the monthly task count graph and a widget listing new tasks. The lower portion of the dashboard features the User Microfrontend, which displays a monthly user count graph and a widget for new users. These microfrontends are rendered in compact mode and interact with each other as discussed in the section 4.3.

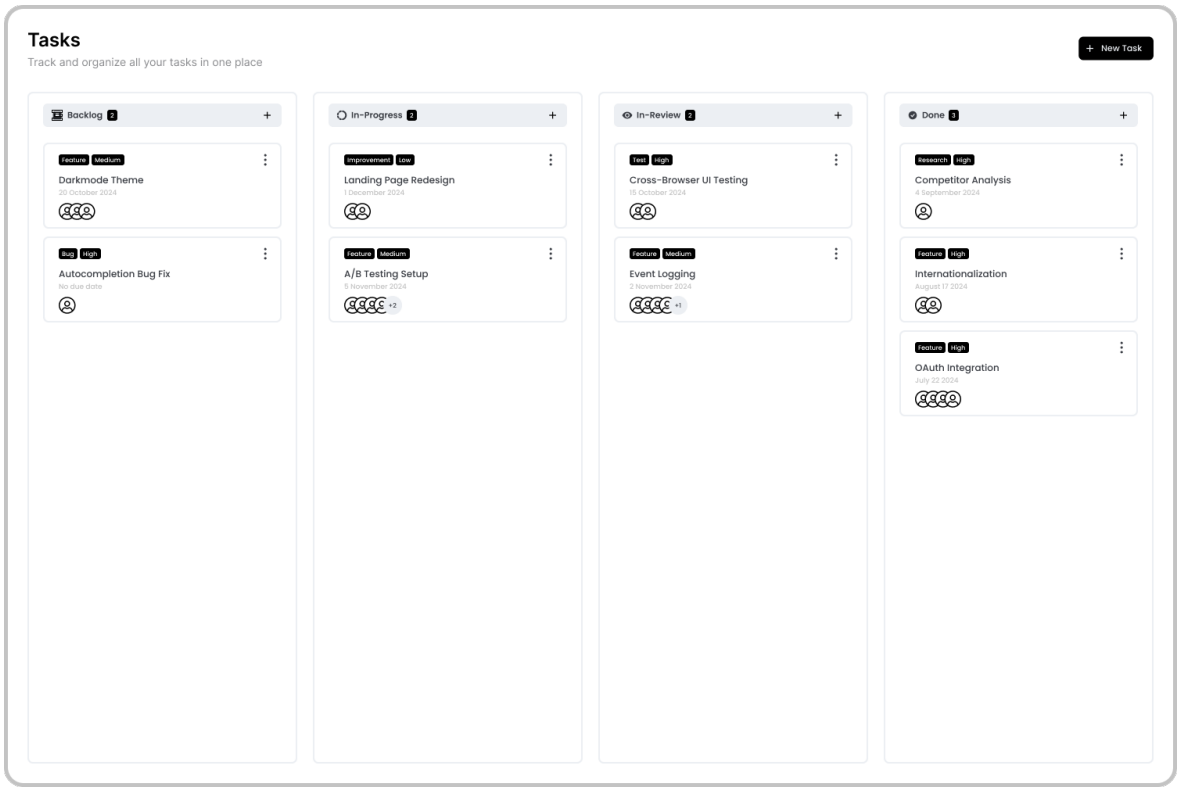


Figure 4.4: Wireframe of the Task Microfrontend interface

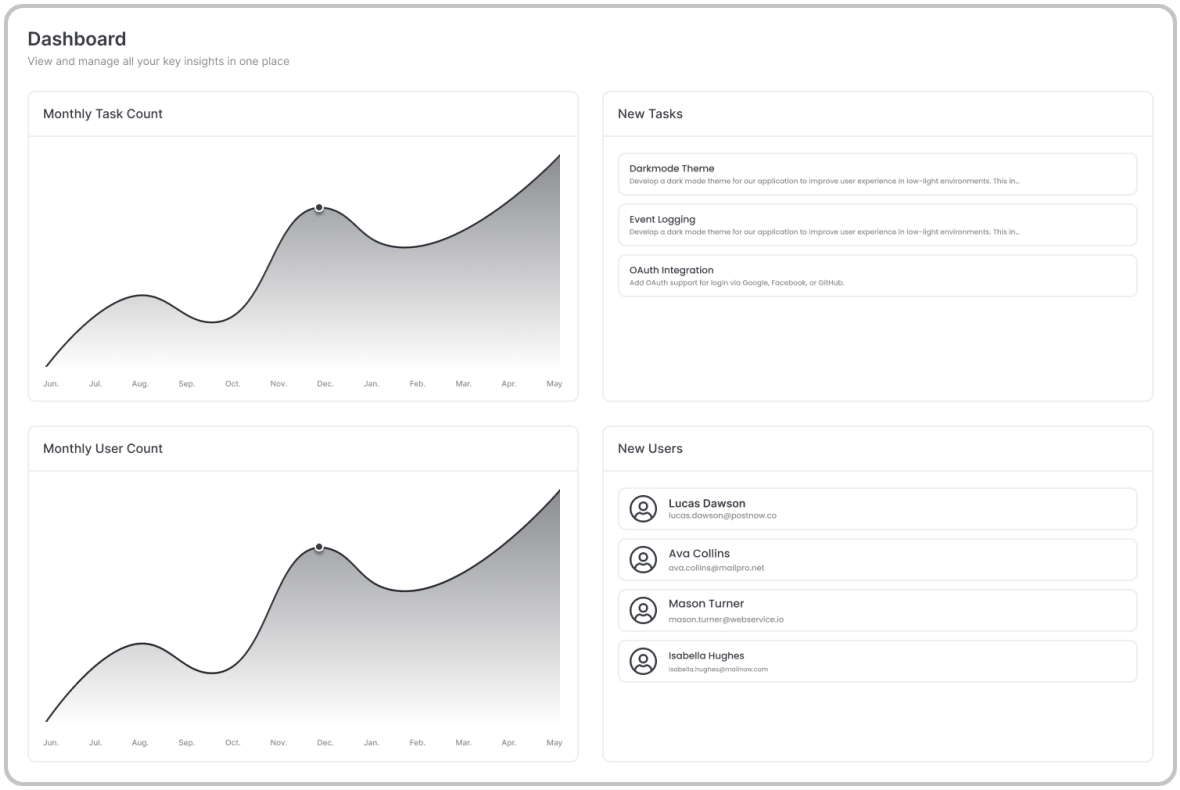


Figure 4.5: Wireframe of the Dashboard interface

Chapter 5

Implementation

In this chapter, we present the implementation of a prototypical project management application based on a microfrontends architecture using Web Components and Angular. We describe the development process in detail, highlighting the key implementation decisions, challenges encountered, and the solutions adopted to address them. The chapter is structured into several sections, covering an overall implementation overview, communication and composition, routing mechanisms, and application styling.

5.1 Overview

The application was developed using Angular 18 and is composed of three separate Angular projects. The first project is the `application-shell`, which serves as the container for all microfrontends. The second project, `user-management`, represents the user microfrontend, and the third project, `task-management`, represents the task microfrontend.

For practical purposes, all these projects were generated within the same Angular workspace. This approach was chosen due to the absence of distinct teams working on the individual projects. In a real-world scenario, each microfrontend would be developed as a separate Angular project to facilitate independent development and deployment. The workspace was created using the standard Angular CLI command as follows.

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

Afterward, the application-shell project was generated using the standard method.

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

After generating the application, Bootstrap and `@ng-bootstrap/ng-bootstrap` were installed and configured in the standard way, along with `@ngx-translate`, a library for internationalization support. Interfaces were then designed using placeholder compo-

nents for the microfrontends. Support for theme switching (light/dark) and language switching (Slovak/English) was added, with preferences saved in local storage.

Microfrontends were generated in the same way, but with the use of the `-prefix` option, as follows.

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

The development of microfrontends followed a similar approach to the Application Shell, with one key difference: the way the application is bootstrapped. Depending on the environment, the bootstrapping process varies. The microfrontends were then built and statically served using the `http-server` package, each on a different port via the command line:

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

To avoid repeatedly typing these commands, they were added as script commands in `package.json`. The composition and loading of microfrontends will be further explained in the next section.

The microfrontends communicate with a backend, which is a standard Node.js server using Express.js. Each microfrontend has been assigned its own URL prefix, such as `/users` for the User Microfrontend and `/tasks` for the Task Microfrontend. The microfrontends interact with the shared backend using Angular's `HttpClient` for CRUD operations. We are not using a database; instead, data is stored only at runtime. In a real-world application, each microfrontend that requires a backend would ideally have its own microservice and dedicated database. However, due to the scope of this thesis and its focus on the frontend aspects of microfrontends, we chose a simplified approach to avoid unnecessary complexity.

5.2 Composition

For the application composition to work correctly, we had to utilize a few special techniques. First, we modified how the microfrontends are built. There are two types of environments in which the microfrontends can operate. The first is the local development environment, where the microfrontend is built as a standard Angular application for easier debugging and development. The second is the embedded environment, where the microfrontend must be built as a custom element.

To build an Angular application as a custom element, we used the following piece of code that demonstrates the custom element bootstrapping process:

```
const app = await createApplication(appConfig);
const customElement = createCustomElement(EntryComponent,
   {
```

```
3     injector: app.injector,
4   });
5   customElements.define(environment.customElementName,
        customElement);
```

Listing 5.1: Custom element bootstrapping process in Angular

Instead of specifying the `AppComponent` as the component we want to build, we create a special component called `EntryComponent`. This component handles all inputs from the Application Shell and renders the `AppComponent`.

However, when we build the application this way, we end up with two JavaScript files (`polyfills.js`, `main.js`) and a CSS file (`style.css`). Having multiple files complicates the loading of microfrontends in the Application Shell. To avoid this, we use Webpack to bundle the scripts and styles into a single file. Finally, we statically serve this file, as explained in the previous section.

Now, let's look at how microfrontends are loaded in the Application Shell. When a user navigates to a microfrontend route, the `LoadMicrofrontendGuard` ensures that the required JavaScript bundle is loaded before route activation. It first extracts the `bundleUrl` from the route data and then uses the `MicrofrontendRegistryService` to load the microfrontend. The service checks if the microfrontend is already loaded; if not, it injects a script tag into the document with the provided `bundleUrl` to load it. In the case of a unified microfrontends view, we do not use the guard. Instead, we directly call the registry service for each microfrontend. If a microfrontend fails to load, we skip rendering it and simply move on to the next one.

5.3 Communication

Two types of communication occur within our application; this section presents both of them.

Applicaition Shell to Microfrontend

The first type of communication is from the Application Shell down to a microfrontend. This occurs when we need to pass attributes to the microfrontend, such as the current language or whether it should be rendered in compact mode. It also occurs when an attribute changes to a new value and the microfrontend needs to be informed of it. Since microfrontends are built as custom elements, we can pass any number of attributes as needed. However, proper handling must be implemented within the microfrontend. Therefore, the corresponding teams must define a contract beforehand, specifying which attributes can be passed between them.

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

```
<user-management compact='true'></user-management>
```

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

```
@Input() compact: boolean = false;
```

For attributes that can change over time, such as the language, we can create a directive to avoid repeatedly writing subscriptions to observables for each microfrontend. The language directive is defined in the Application Shell, as shown in the code below.

```
@Directive({
  standalone: true,
  selector: '[microfrontendLanguage]',
})
export class MicrofrontendLanguageDirective implements
   OnInit, OnDestroy {
  private destroy$ = new Subject<void>();

  constructor(
    private element: ElementRef,
    private translateService: TranslateService
  ) {}

  ngOnInit(): void {
    this.translateService.onLangChange
      .pipe(
        map((event) => event.lang),
        startWith(
          this.translateService.currentLang ?? this.
             translateService.defaultLang
        ),
        takeUntil(this.destroy$)
      )
      .subscribe((language) => {
        this.element.nativeElement.language = language;
      });
  }

  ngOnDestroy(): void {
    this.destroy$.next();
  }
}
```

Listing 5.2: Angular directive for synchronizing language changes with microfrontends

And it is used on the microfrontend elements like this:

```
<user-management microfrontendLanguage></user-management>
```

It is important to note that for attributes that can change over time, we need to define an `onChange` listener in the microfrontend's entry component.

Microfrontend to Microfrontend

The second type of communication primarily involves communication between microfrontends, but it can also be used to communicate from the shell to a microfrontend or vice versa. This type of communication is handled via the browser CustomEvents. In each microfrontend, we create an event service with methods to listen for and emit events, as shown in the code below.

```
private emit(eventName: string, data: any) {
  const customEvent = new CustomEvent(eventName, {
    detail: data,
    bubbles: true,
    composed: true,
  });

  window.dispatchEvent(customEvent);
}

private on(eventName: string, callback: (data: any) =>
   void) {
  const windowListener = (event: CustomEvent) => callback(
     event.detail);
  window.addEventListener(eventName, windowListener as
     EventListener);

  return () => {
    window.removeEventListener(eventName, windowListener
       as EventListener);
  };
}
```

Listing 5.3: Event service implementation for microfrontend communication using CustomEvents

The microfrontend teams must define a contract beforehand, describing what types of events their microfrontend will listen to and emit, as well as what types of data will be passed through those events. We can then define those listeners and emitters in

whichever components are needed.

5.4 Routing

Routing is completely handled in the Application Shell, 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 whether the microfrontend can be loaded via a route guard. If it can, we then lazy-load the microfrontend by injecting a script tag with the provided `bundleUrl`, as explained in the previous section.

Each microfrontend that should be rendered as a separate page has its own route definition, which looks, for example, like the following:

```
export const USER_MANAGEMENT_ROUTES: Routes = [
  {
    path: '**',
    canActivate: [LoadMicrofrontendGuard],
    component: UserManagementHostComponent,
    data: {
      bundleUrl: 'http://localhost:4201/bundle.js',
      compact: false,
    },
  },
];
```

Listing 5.4: Route configuration for the User Microfrontend

And the host component, which is used in the route definition, would then look like the following:

```
@Component({
  selector: 'user-management-host',
  standalone: true,
  imports: [MicrofrontendLanguageDirective],
  template: `<user-management microfrontendLanguage></user
     -management>`,
  schemas: [CUSTOM_ELEMENTS_SCHEMA],
})
export class UserManagementHostComponent {}
```

Listing 5.5: Host component for the User Microfrontend

In the case of multiple microfrontends on the same page, we define the page and the route in the Application Shell like any other page and load the microfrontends directly

using the `MicrofrontendRegistryService`.

In our case, we do not have a microfrontend with multiple pages. However, if that were the case, we would propagate the full route from the Application Shell down to the microfrontend via an attribute. The microfrontend would then have its own router and handle the route internally. In the event of a route change triggered within the microfrontend (rather than the application shell), we would propagate the new route to the Application Shell, which would handle it as necessary, pass it back to the microfrontend, and only then would the route change within the microfrontend.

5.5 Styling

To keep the user interface consistent across the Application Shell and microfrontends, we use Bootstrap for styling in each of them. This ensures that each part of the application has a uniform appearance, unless we significantly modify the Bootstrap classes.

Another issue we had to address is style isolation, to prevent class name collisions between microfrontends. For example, if the same class name is used in two different microfrontends, one should not override the other. To avoid this, we utilize the Shadow DOM. This is very straightforward in Angular — we simply set the encapsulation to `ViewEncapsulation.ShadowDom` in the microfrontend's entry component.

However, for this to work properly with Bootstrap, we also had to change how Bootstrap is imported. Instead of importing Bootstrap in the styles section of the angular.json file, we directly imported it into the main styles file of the microfrontend. We then link this file in the `styleUrls` property of the microfrontend's entry component, allowing each microfrontend to have its own isolated, separate DOM tree. When we put all the information for this and previous sections together, the entry component for the User Microfrontend would look like the following:

```
1  @Component({
2    selector: 'user-entry',
3    standalone: true,
4    imports: [AppComponent],
5    template: '<user-root [compact]="compact"></user-root>',
6    styleUrl: '../../../styles.scss',
7    encapsulation: ViewEncapsulation.ShadowDom,
8  })
9  export class EntryComponent implements OnChanges {
10   @Input() compact = false;
11   @Input() language?: 'en' | 'sk';
12
```

```
13      constructor(private translateService: TranslateService)
            {}
14
15      ngOnChanges(changes: SimpleChanges): void {
16        if (changes['language'] && this.language) {
17          this.translateService.use(this.language);
18        }
19      }
20    }
```

Listing 5.6: Entry component implementation with Shadow DOM encapsulation and language support for the User Microfrontend

Chapter 6

Evaluation Results

This chapter analyzes and evaluates the implemented prototypical application in terms of reusability, extensibility, resource sharing, and application state management, with each aspect being discussed in its own section.

6.1 Reusability

Reusability refers to the ability of software components — in our case, microfrontends — to be used across multiple applications or within different parts of the same application with minimal modifications.

In our resulting application, even though it is relatively simple and small, we achieved a high degree of reusability. Each microfrontend functions not only as a standalone page — such as for user or task management — that can be reused across other projects but also integrates within the same application as dashboard widgets.

Furthermore, our application leverages Web Components, which allow components to be reused in any environment or framework. Well-designed boundaries and domains for the microfrontends also enable their reuse across different applications. For example, the user microfrontend can be used in almost any application that supports user roles, as user management is an essential part of most systems.

However, our implementation also introduces a challenge to reusability, as microfrontends cannot be directly styled externally. This is due to two factors: first, they come with their own dependencies and do not share any with the parent application; second, they utilize the Shadow DOM, which isolates their DOM tree, making it inaccessible from the outside.

6.2 Extensibility

Extensibility refers to the ability of a system to be enhanced with new functionality or modifications without requiring significant changes to existing components. Our microfrontend architecture is designed with extensibility in mind.

In our application, extensibility is achieved mainly through microfrontend architecture that supports both internal evolution and external expansion. New features can be added, or existing ones modified, with minimal impact on the rest of the system.

This extensibility takes several forms. First, changes within a single microfrontend are well-isolated due to their strong encapsulation. Internal modifications—such as updating business logic or UI elements—do not affect other parts of the application. Even interface-level changes, such as introducing new input properties or emitting additional events, remain manageable as long as communication contracts are respected. These changes typically require only the addition of new listeners in the corresponding microfrontends or the passing of additional attributes from the application shell.

Second, our architecture is designed to make the addition of entirely new features straightforward. New functionality is typically introduced as a separate microfrontend, allowing teams to develop and deploy independently. Since microfrontends are standard Angular applications that are wrapped into custom elements only at build time, development remains familiar and efficient. New microfrontends can be easily added to the Application Shell by declaring their routes and defining their host components. One of the technical achievements of our approach is the ability to run multiple Angular applications concurrently within the same browser environment—something that was traditionally considered infeasible.

However, extensibility comes with trade-offs. The most critical issue is application size. Each microfrontend includes its own dependencies, which leads to linear growth in the total bundle size. For example, the bundled size of the User Microfrontend is about 1.21MB, consisting of:

- `styles.css` - 272KB (22.5%) - primarily Bootstrap styles,
- `polyfills.js` - 35KB (2.9%) - for legacy browser support,
- `main.js` - 904KB (74.6%) - actual application logic.

Similarly, the Task Microfrontend is about 1.25MB in total, with:

- `styles.css` - 260KB (20.9%),
- `polyfills.js` - 35KB (2.8%),
- `main.js` - 952KB (76.3%).

On average, a microfrontend in our application contributes approximately 1.21MB. With this size, when using the Angular framework, it would be possible to build a real-world application consisting of roughly 15 to 30 microfrontends. The theoretical number could be even higher, since Chromium-based browsers have a per-tab limit of up to 500MB, which—given the size of our microfrontends—translates to around 400 microfrontends. The real limitation, however, would be performance. Based on our experience, at around 20-30MB total (which corresponds to 15-30 of our microfrontends), we have not observed a significant drop in performance yet. However, the initial load time would already be quite long, and it would require the use of optimization techniques such as lazy loading and asset sharing.

6.3 Resource Sharing

Resource sharing, in our context, refers to the ability of microfrontends to efficiently utilize and share common assets, dependencies, and services to minimize redundancy and improve performance. This is one of the areas where our implementation is most lacking.

Although each microfrontend and the Application Shell use the same version of most dependencies, each one still includes its own copy of them. This is a trade-off we have accepted in exchange for the strong isolation provided by Web Components. Using the `source-map-explorer` package [42], we can analyze the space usage of our bundles through source maps.

Figure 6.1 shows the results of running this analysis on the `main.js` file of the User Microfrontend, while Figure 6.2 presents the results for the `main.js` file of the Task Microfrontend. In both cases, dependencies (i.e., `node_modules`) occupy about 70% of the total bundle size, whereas the actual application code accounts for only about 30%. Angular core dependencies comprise approximately 35% to 40% of the entire file size. Another significant portion is taken up by the `chart.js` library, which contributes around 20% and is used in the dashboard for graph visualizations. The remaining dependencies are relatively small.

Based on this analysis, we can conclude that our application would significantly benefit from sharing Angular core dependencies across microfrontends, as well as the `chart.js` library. However, we were unable to address this issue due to time limitations. One potential solution is to combine the Web Components approach with the Module Federation Webpack plugin—especially since we are already using Webpack.

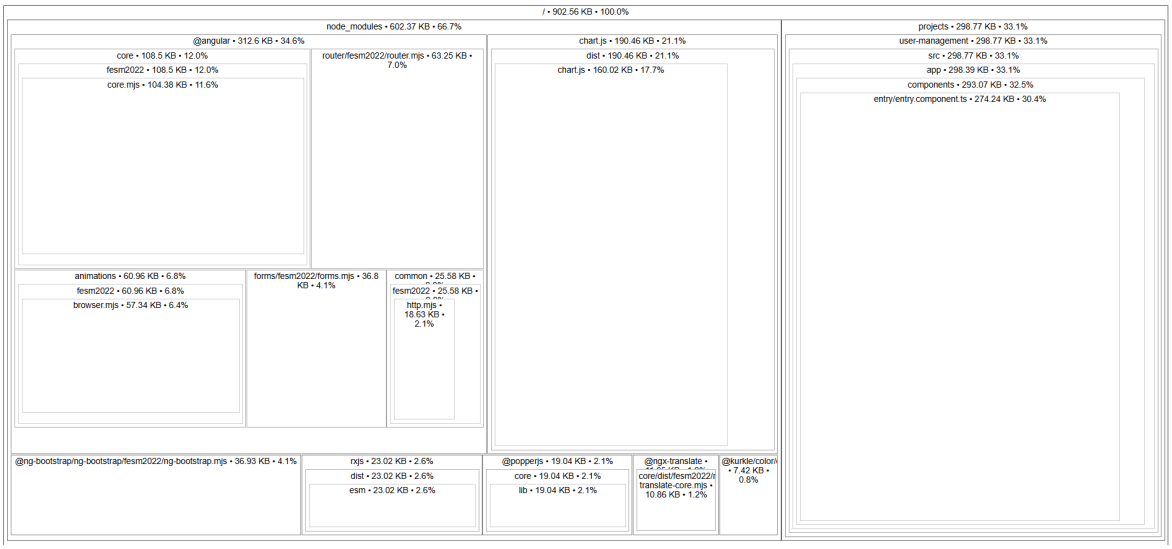


Figure 6.1: Space usage of the `main.js` file in the User Microfrontend

6.4 Application State Management

Application state management refers to the synchronization of data across different parts of the application. In a microfrontend architecture, managing the application state becomes more complex due to the independent nature of each microfrontend.

In our application, there is not much state to manage aside from the current language and theme. To handle these, we use a combination of local storage and attributes, which are passed from the Application Shell to the microfrontends. Both states are managed centrally in the Application Shell using `BehaviorSubjects`. User preferences are stored in local storage, and upon initialization or any change, the language is passed as attributes to the microfrontends. The theme is managed globally in the Application Shell.

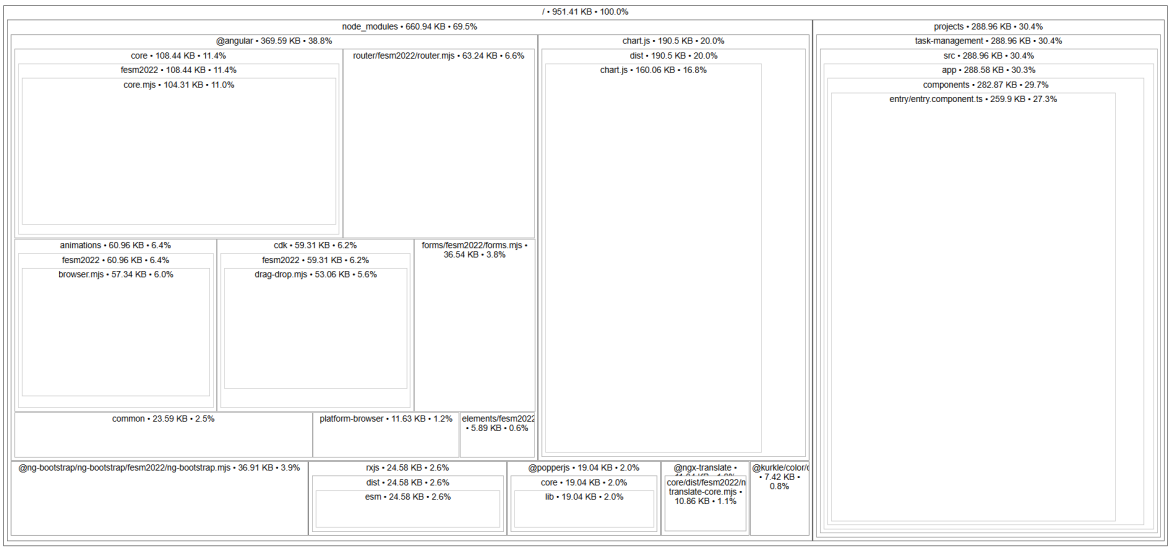


Figure 6.2: Space usage of the `main.js` file in the Task Microfrontend

Chapter 7

Conclusion

This chapter revisits the primary objectives of the thesis and discusses its key findings. It then explores the practical implications of the results and outlines potential areas for future research.

7.1 Results

The purpose of this thesis was to review the current literature on microfrontend architecture and explore various approaches to its design and implementation. The goal was to compare these approaches in terms of reusability, extensibility, resource sharing, and application state management. Furthermore, the thesis aimed to identify the approaches best suited for enterprise application development and to design and implement a prototypical microfrontend application using one of these approaches.

Based on the reviewed literature, we conducted a comprehensive analysis of microfrontend architecture—its challenges, advantages, disadvantages, practical use cases, and implementation approaches. We compared seven of the most frequently referenced approaches in the aspects of extensibility, reusability, simplicity, performance, resource sharing, and developer experience, listing their advantages and disadvantages and identifying the environments in which they perform best. Among these, we identified three approaches as most suitable for enterprise environments: server-side composition, composition via Module Federation, and composition using Web Components. We selected the Web Components-based approach for implementing the prototypical application and explained the reasoning behind this choice.

We defined the aim of our prototypical application, specified its requirements, designed its microfrontend-based system architecture, and analyzed various technologies best suited to our use case. Based on this analysis, we selected a set of implementation technologies and created the application's wireframes. Using this design, we developed the prototypical application in Angular 18 with Web Components. Common challenges

in microfrontend development—such as communication, composition, routing, and styling—were addressed, and corresponding solutions are provided in this thesis. The resulting application was evaluated in terms of reusability, extensibility, resource sharing, and application state management. The complete source code is freely available in our GitHub repository: `https://github.com/PavolRepisky/microfrontends`.

7.2 Practical Implications

We believe that the findings of this thesis provide valuable insights for organizations and individuals considering the adoption of microfrontend architecture in their projects. Our analysis of various implementation approaches and their evaluation aims to assist decision-makers in selecting the most suitable approach for their specific use cases.

Using the chosen approach, we successfully developed a fully functional prototypical microfrontend application that demonstrated key advantages in terms of reusability and extensibility, as outlined in Chapter 6. The implementation of this application validated the feasibility of employing a Web Components-based approach within an enterprise environment, showcasing its effectiveness in addressing common microfrontend challenges such as communication, composition, routing, and styling. Given these results, we consider this approach a viable and scalable option for real-world applications.

7.3 Recommendations for Future Work

Although this thesis covered many aspects and challenges of microfrontend architecture implemented via Web Components in Angular, there are still areas that were not addressed due to time limitations. We outline several of these areas for potential further research within this approach:

- Performance optimization techniques, such as lazy loading, code splitting, and resource sharing (primarily related to common dependencies).
- State management for large-scale applications.
- Integration testing solutions.
- CI/CD pipeline strategies for independent microfrontend deployment.
- Versioning and repository organization.

Investigating these areas could help expand the adoption of microfrontend architecture in modern software development.

Bibliography

[1] Google, "Angular - the modern web developer's platform." URL: `https://angular.dev/` (visited on 01/03/2025).

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

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

[4] 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.

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

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

[7] 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.

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

[9] C. Jackson, "Micro frontends." URL: `https://martinfowler.com/articles/micro-frontends.html` (visited 05/12/2024), 2019.

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

[11] Amazon, "Amazon official website." URL: `https://www.amazon.com` (visited 09/02/*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.

[13] Netflix, "Netflix official website." URL: `https://www.netflix.com` (visited 09/02/2024).

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

[15] Uber, "Uber official website." URL: `https://www.uber.com` (visited 09/02/2024).

[16] A. Kwiecień, "10 companies that implemented the microservice architecture and paved the way for others." URL: urlhttps://www.cloudflight.io/en/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/ (visited 18/01/2024), 2019.

[17] M. Fowler and J. Lewis, "Microservices." URL: `https://martinfowler.com/articles/microservices.html` (visited 04/12/2024), 2014.

[18] C. Richardson, "Microservices.io." URL: `https://microservices.io` (visited 04/12/2024).

[19] Thoughtworks, "Technology radar: Micro frontends." URL: `https://www.thoughtworks.com/radar/techniques/micro-frontends` (visited) 10/09/2024, 2016.

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

[21] "Mdn web docs: Custom event." URL: `https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent` (visited: 23/02/2025).

[22] "Broadcast channel api." URL: `https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API` (visited 17/02/2025).

[23] Webpack, "Module federation." URL: `https://module-federation.io/` (visited 05/04/2025).

[24] Bootstrap, "Bootstrap." URL: `https://getbootstrap.com/` (visited 05/04/2025).

[25] M. Design, "Material design." URL: `https://m3.material.io/` (visited 05/04/2025).

[26] Google, "Repo - the multiple git repository tool." URL: `https://gerrit.googlesource.com/git-repo/` (visited 05/04/2025).

[27] IKEA, "Ikea official website." URL: `https://www.ikea.com/` (visited 07/02/2024).

[28] J. Stenberg, "Experiences using micro frontends at ikea." URL: `https://www.infoq.com/news/2018/08/experiences-micro-frontends/` (visited 24/08/2024), 2018.

[29] Zalando, "Zalando official website." URL: `https://www.zalando.com/` (visited 11/02/2024).

[30] Zalando, "Tailor.js: A streaming layout service for front-end microservices." URL: `https://github.com/zalando/tailor` (visited 15/08/2024).

[31] SAP, "Sap official website." URL: `https://www.sap.com/` (visited 11/04/2024).

[32] SAP, "Luigi: The enterprise-ready micro frontend framework." URL: `https://luigi-project.io/` (visited 11/04/2024).

[33] Vue.js, "Vue.js - the progressive javascript framework." URL: `https://vuejs.org/` (visited 05/04/2025).

[34] DAZN, "Dazn: Live and on demand sports streaming." URL: `https://www.dazn.com/` (visited 05/02/2024).

[35] "Mdn web docs: Web components." URL: `https://developer.mozilla.org/en-US/docs/Web/API/Web_components` (visited 22/02/2025).

[36] R. Eisenberg, "About web components." URL: `https://eisenbergeffect.medium.com/about-web-components-7b2a3ed67a78` (visited 20/01/2025), 2023.

[37] J. James, "Custom events in javascript: A complete guide." URL: `https://blog.logrocket.com/custom-events-in-javascript-a-complete-guide/` (visited 15/01/202), 2021.

[38] "Mdn web docs: The inline frame element." URL: `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe` (visited 30/11/2024).

[39] "Mdn web docs: Ajax." URL: `https://developer.mozilla.org/en-US/docs/Glossary/AJAX` (visited 26/11/2024).

[40] PostCSS, "Postcss: a tool for transforming css with javascript." URL: `https://postcss.org/` (visited 24/02/2025).

[41] D. Taibi and L. Mezzalira, "Micro-frontends: Principles, implementations, and pitfalls," *ACM SIGSOFT Software Engineering Notes*, vol. 47, pp. 25–29, 09 2022.

[42] "Source map explorer - analyze and debug space usage through source maps."