

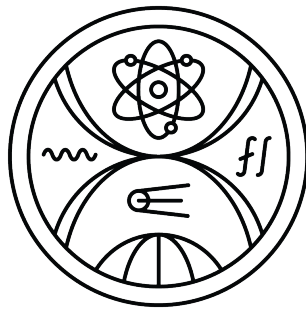
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS



ANALYSIS, DESIGN AND IMPLEMENTATION OF MICRO-FRONTEND ARCHITECTURE

Diploma thesis

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. Juraĵ Marák



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



THESIS ASSIGNMENT

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

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

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

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

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

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

Assigned: 05.10.2023

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

Acknowledgement

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

Abstrakt

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

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

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

Abstract

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

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

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

Contents

1	Introduction	1
1.1	Background and Motivation	1
1.2	Aim	2
1.3	Scope and Limitations	2
2	Microservices and Microfrontends	3
2.1	History of Microservices and Microfrontends	3
2.2	Microservices Principles	4
2.2.1	Componentize via Services	4
2.2.2	Model Around Business Domain	4
2.2.3	Design for Failure	5
2.2.4	Adopt a Culture of Automation	5
2.2.5	Decentralize Everything	5
2.2.6	Evolve Continuously	6
2.3	Introduction to Microfrontends	6
2.4	Microfrontends Challenges	7
2.4.1	Communication	7
2.4.2	Routing	8
2.4.3	Static Assets Serving	8
2.4.4	Reusability	9
2.4.5	Styling Consistency and Isolation	9
2.4.6	Project Organization	10
2.5	Microfrontends Tradeoffs	10
2.5.1	Advantages	10
2.5.2	Disadvantages	11
2.6	Microfrontends in Practice	12
3	Microfrontends in Detail	14
3.1	Basic Enablers of Microfrontends	14
3.1.1	Web Components	14
3.1.2	Custom Events	16

3.2	Composition Approaches	17
3.2.1	Link-based Composition	17
3.2.2	Composition via Iframes	18
3.2.3	Composition via Ajax	20
3.2.4	Server-side Composition	22
3.2.5	Edge-side Composition	24
3.2.6	Composition via Module Federation	25
3.2.7	Composition via Web Components	27
3.2.8	Summary	29
4	Design	30
4.1	Application introduction	30
4.2	Requirements	30
4.2.1	Functional Requirements	31
4.2.2	Non-functional Requirements	32
4.3	System Architecture	32
4.3.1	Application Shell	33
4.3.2	Users Microfrontend	33
4.3.3	Projects Microfrontend	33
4.3.4	Tasks Microfrontend	33
4.3.5	Dashboard Microfrontend	34
4.4	Tech Stack	34
4.4.1	Angular	34
4.4.2	TypeScript	34
4.4.3	Web Components	35
4.4.4	Browser Events	35
4.5	Graphical User Interface	35
4.5.1	Application Shell	35
4.5.2	Users Microfrontend	36
4.5.3	Projects Microfrontend	36
4.5.4	Tasks Microfrontend	37
4.5.5	Dashboard Microfrontend	38
5	Implementation	40
5.1	Overview	40
5.2	Communication	43
5.3	Routing	44
5.4	Styling	45
5.5	Resource sharing	45

5.6	Infrastructure	45
5.7	Testing	45
6	Conclusion	46
6.1	Implementation Results	46
6.2	Practical Implications	46
6.3	Recommendations for Future Work	46

List of Figures

2.1	Architectural overview of microfrontends	7
4.1	Application shell wireframe	36
4.2	Users microfrontend wireframe	37
4.3	Projects microfrontend wireframe	38
4.4	Tasks microfrontend wireframe	39
4.5	Dashboard microfrontend wireframe	39

List of Tables

3.1	Evaluation of link-based composition	19
3.2	Evaluation of composition via iframes	20
3.3	Evaluation of composition via Ajax	22
3.4	Evaluation of server-side composition	24
3.5	Evaluation of edge-side composition	25
3.6	Evaluation of composition via Module Federation	27
3.7	Evaluation of composition via Web-components	29
3.8	Comparison of different composition approaches	29

Chapter 1

Introduction

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

1.1 Background and Motivation

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

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

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

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

1.2 Aim

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

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

1.3 Scope and Limitations

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

Chapter 2

Microservices and Microfrontends

This chapter introduces the concept of microservices and microfrontends, starting with 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 microfrontend adoption.

2.1 History of Microservices and Microfrontends

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

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

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

2.2 Microservices Principles

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

2.2.1 Componentize via Services

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

2.2.2 Model Around Business Domain

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

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

2.2.3 Design for Failure

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

2.2.4 Adopt a Culture of Automation

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

2.2.5 Decentralize Everything

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

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

2.2.6 Evolve Continuously

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

2.3 Introduction to Microfrontends

The main idea behind microfrontends is to extend the principles of microservices to the frontend side [7][2]. In microfrontends, the presentation layer is split into smaller, more manageable pieces [7]. Jackson [8] 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 [8].

As Geers [2] 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 a number of full-stack micro-applications [7], as shown in Figure 2.1. Each such micro-application is autonomous, with its own continuous delivery pipeline and serving a specific business domain or feature [6]. Microfrontends introduce vertical teams instead of horizontal ones. Instead of being grouped by the development technologies they use, teams are grouped by application features [7]. 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: Architectural overview of microfrontends

2.4 Microfrontends Challenges

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

2.4.1 Communication

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

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

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

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

2.4.3 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 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 dynam-

ically returns the resource location based on the context and state of the system [3]. Also each microfrontend can request its assets via an HTTP request independently, or in the case of small assets, they can be bundled inside the microfrontend itself.

2.4.4 Reusability

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

2.4.5 Styling Consistency and Isolation

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

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

2.4.6 Project Organization

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

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

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

2.5 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 [2][7][6]. Geers [2] mentions: "Reducing waiting time between teams is microfrontends' primary goal." Having all developers working on the same stack within a single team leads to fewer misunderstandings and much faster communication [2][7]. Additionally, teams have much more freedom to

make case-by-case decisions regarding individual parts of the product [8][2], allowing teams to become experts in their respective areas of the application [7].

Microfrontends also bring many of the benefits of microservices architecture to the frontend. The codebases are smaller, more understandable, and less complex [2][8][7]. Consequently, this can lead to shorter onboarding times for new developers [6]. Each microfrontend is isolated, independently deployable, and its failure does not affect the rest of the system [6][7][8][2].

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

2.5.2 Disadvantages

However, microfrontends are not a silver bullet. 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 [2][6]. 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 [6][7][8]. Each team also needs to set up and maintain its own application server, build process, 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 [6]. They require a significant amount of knowledge and analysis about a project before development begins [7][6]. There are risks associated with developing in a standalone environment that is quite different from production, requiring extensive integration testing [8][7][6]. 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 [7].

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 often publicly share its internal architecture, according to Geers [2], 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 [22]. IKEA’s principal engineer, Gustaf Nilsson Kotte, shares in an interview [23] 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 [24] has also adopted microfrontends. Zalando even open-sourced its microfrontend framework, “Tailor.js” [25], 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 [1].

Spotify

An example of an unsuccessful adoption is Spotify [14]. As Mezzalana [1] 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 [26]. SAP initially utilized iframes and eventually released its own microfrontend framework, “Luigi” [27], designed for creating enterprise applications that integrate with SAP systems. It supports modern enterprise frontend frameworks such as Angular, React, Vue, and SAP’s own SAPUI [1].

DAZN

The last on the list is DAZN [28], a sports streaming platform. DAZN has migrated its monolithic frontend to a microfrontend architecture [2]. The company focused on supporting not only the web but also multiple smart TVs and gaming consoles [1]. They chose a client-side approach to composition. As Mezzalana [1] describes, the platform uses a combination of SPAs and components orchestrated by a client-side agent called “Bootstrap”. They have shared their experiences with microfrontends extensively and eventually even published a book on the topic [1].

These are just a few of the most significant examples, but there are many more. And more and more companies are beginning to adopt this architecture.

Chapter 3

Microfrontends in Detail

This chapter provides a deeper exploration of microfrontends architecture, offering readers a solid foundation to understand not only what microfrontends are but also the key approaches to implementing them effectively.

3.1 Basic Enablers of Microfrontends

In this section, we introduce key technologies that may be unfamiliar to the reader but will be used 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 [29]. They are now supported by most major browsers and work across different frameworks and libraries [30]. We will now discuss the three main technologies that make up Web Components.

HTML Templates

HTML templates are a new 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 [29]. To create such a structure, we utilize the `<template>` element.

```
<template id="my-template">
  <p>Some content goes here...</p>
</template>
```

Then, we can clone the template and add it to the DOM via JavaScript.

```
let template = document.getElementById("my-template");
let clone = template.content.cloneNode(true);
document.body.appendChild(clone);
```

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 [29].

Custom Elements

Custom elements are a set of JavaScript APIs for creating custom HTML elements with specific behavior [29]. To register a custom element, we use the `customElements` global object and its method `define`, to which we pass a hyphenated tag name and a class that inherits from the `HTMLElement` class.

```
customElements.define(
  "my-element",
  class MyElement extends HTMLElement {}
);
```

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.

```
<my-element attribute="value"></my-element>
```

Shadow DOM

Lastly Shadow DOM is a set of JavaScript APIs that enables us to attach an encapsulated “shadow” DOM subtree to an element, which is then rendered separately from the main document DOM [29]. 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 [30]. 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.

```
class MyElement extends HTMLElement {
  constructor() {
    super();
    let shadow = this.attachShadow({ mode: 'open' });
    let div = document.createElement('div');
    div.textContent = "Some content goes here...";
    shadow.appendChild(div);
  }
}
```

Open Shadow DOM can be accessed via JavaScript using `element.shadowRoot`. Closed Shadow DOM cannot be accessed externally using `element.shadowRoot` [29].

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 [31]. 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 following.

- **bubbles:** `true` - Allows the event to propagate upward to the parent element. This is set to `false` in custom events by default. The propagation can be stopped using `stopPropagation()` method.
- **cancelable:** `true` - Allows the event to be canceled via `preventDefault()` method.
- **composed:** `true` - Allows the event to propagate from the shadow DOM to the real DOM. In this case, the **bubbles** property must be set to `true` [32].

Putting it all together, we would create a custom event as follows.

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

```
    composed: false,  
  });
```

The event can then be dispatched on a specific element using `dispatchEvent()` method.

```
document.dispatchEvent(myEvent);
```

We can listen for the event using `addEventListener()` method.

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

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 six of the most commonly mentioned approaches in the literature, outlining their advantages, disadvantages, and suitability. In addition, we have rated each approach based on aspects of extensibility, reusability, simplicity, performance, resource sharing, and developer experience, drawing from both the reviewed literature and our own insights.

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 [2]. To compose the microfrontends, we simply interconnect them across different applications using standard anchor elements.

```
<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 [2]. 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 [2].

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 [2].

Disadvantages

As Geers [2] 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 [2]. Our evaluation of this approach can be found in the table 3.1.

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 [33]. Each embedded context

Aspect	Rating
Extensibility	4/5
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

has its own document and supports independent URL navigation [33]. 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 [1]. To add an iframe, we simply utilize the `<iframe>` tag, similar to an anchor `<a>` tag.

```
<iframe src="https://example.com/mfe2"></iframe>
```

The iframe can be further restricted via 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 [1].

Its behavior and appearance can be customized via different attributes [33], 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 [2][8][1].

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 [3]. It is also difficult to integrate them with each other, making communication, routing, and history management more complicated [8]. The host application must

know the height of the iframe in advance to prevent scrollbars and whitespace, which can be fairly complicated in responsive websites [1][2]. Every iframe creates a new browsing context, which requires a lot of memory and CPU usage. Numerous iframes on the same page can significantly worsen application performance [2]. Lastly, they perform poorly in terms of search engine optimization (SEO) and accessibility [2].

Suitability

Despite the relatively long list of disadvantages, iframes can still be a valid choice for some projects. Iframes shine when there is not much communication between microfrontends, and the encapsulation of our system using a sandbox for each micro-frontend is crucial. The best use cases for iframes are in desktop, B2B, and internal applications. However, other approaches should be preferred if performance, SEO, accessibility, or responsiveness are crucial factors [2][1]. 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 [34]. To use AJAX as an approach for microfrontends, each team must first expose their microfrontend on a specific endpoint. Next, we create a corresponding empty element in the host application and specify the URL in a data attribute from which the content should be downloaded. Finally, JavaScript code is needed to locate the element, retrieve the URL, fetch the content from the endpoint, and append it to the DOM [2]. 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 [35] 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 (IIFE) to limit the scope to the anonymous function and prefix global variables [2]. The composition can then be handled as follows.

```
<div id="mfe2" data-url="https://example.com/mfe2"></div>

<script>
  const element = document.getElementById("mfe2");
  const url = element.getAttribute("data-url");

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

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 [2]. This approach also reduces initial load time since only essential content is loaded, while other microfrontends are loaded later asynchronously.

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 between them. Lastly, there are no lifecycle methods built-in for the scripts, and these must be implemented from scratch [2].

Suitability

This approach is well-established, robust, and easy to implement. It is particularly well-suited for websites where markup is primarily generated on the server side. However, for pages that require a higher degree of interactivity or rely heavily on managing local state, other client-side approaches may be more suitable [2]. 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 [8]. 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>
```

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 [8]:

```
# 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;
}
}
```

Microfrontends can be deployed either as static assets or as dynamic assets, where an application server generates the templates for every user's request [1], 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 [1] 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 [2][1].

Disadvantages

For larger server-rendered pages, browsers may spend considerable time downloading markup rather than prioritizing essential assets. Additionally, technical isolation is limited, requiring the use of prefixes and namespacing to prevent conflicts. The local development experience is also more complex [2]. Furthermore, if web pages are not highly cacheable but are instead personalized per user request, scalability issues may arise due to excessive server load [6].

Suitability

This approach is ideal for pages that prioritize performance and search engine ranking, as it remains reliable and fully functional even without JavaScript [2]. 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 [1]. 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 Mezzalana [1] explains it: “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 [2].” 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:

```
<esi:include src="https://example.com/mfe2"
  alt="https://fallback.example.com/mfe2" />
```

If the fragment from the `src` URL fails to load, the content from the `alt` URL will be used instead [2]. After the markup language is interpreted, the result is a completely static HTML page renderable by a browser [1].

Advantages

Since we are composing the pages at the edge, response times are reduced as content is served closer to users. It also avoids the server-side problem of too many requests to servers, handling high loads more efficiently [1].

Disadvantages

The first drawback of this technique is that ESI is not implemented in the same way by each CDN provider or proxy server. Therefore, a multi-CDN strategy, as well as porting application code from one provider to another, could cause serious issues [6]. Another problem is that ESI cannot be used for dynamic pages. A potential solution to this problem could be adding client-side JavaScript, but this brings extra complexity [1]. Additionally, due to the poor adoption of this approach, the developer experience is not the best, and it lacks documentation and tools compared to other solutions [1].

Suitability

This approach is typically only used for large static websites that are not personalized for each user. According to Mezzalira [1], 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 fairly new approach, made possible by the release of Webpack 5 [36]. Module Federation is Webpack's native plugin, allowing chunks of JavaScript code to load synchronously or asynchronously [1]. As Mezzalira [1] 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",
```

```
    },  
  });
```

- 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"],  
});
```

The composition takes place at runtime, which can either be on the client side, with the application shell loading different microfrontends, or on the server side, with server-side rendering at the origin [1]. 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 [1]. It also supports bidirectional code sharing across microfrontends, but this should be avoided as it goes against microfrontend principles. For communication, we could use event emitters or custom events.

Advantages

The biggest advantage of this approach is that it comes with a lot of tooling, solving most of the microfrontend challenges, such as lazy-loading, library sharing (even with different versions), microfrontend loading, and more. It supports both client and server-side rendering. Additionally, it benefits from the Webpack ecosystem, as other plugins can be used to customize the bundles even further. It provides a very 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 Webpack expertise. Additionally, the simplicity of code sharing across projects can lead to a very complicated architecture if the team is not disciplined or skilled enough [1].

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 standard within the organization [36]. 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

A 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, we will only focus on the implementation part.

First, we need to create a custom element, which will represent our microfrontend by extending the `HTMLElement` class and attaching a Shadow DOM to it. To keep things simple, we will not utilize HTML templates.

```
class MFE2 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>
  ‘;
}
}
```

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);
```

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

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

For communication, we typically utilize `CustomEvents`, which were also discussed in the previous chapter ??.

Advantages

Web Components are now a widely implemented web standard that offers strong isolation managed by the browser when we use the Shadow DOM. Web Components provide many built-in lifecycle methods and can be used across different frameworks and libraries, such as Angular and React, making them even easier to implement [2]. They offer a great level of reusability and flexibility.

Disadvantages

One issue with Web Components is that they are not fully supported in all 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 mechanisms. Finally, handling SEO and accessibility can also be challenging [2][1].

Suitability

Web Components are an excellent choice for multitenant environments and are well-suited for building interactive, app-like applications. However, for applications that prioritize SEO or require compatibility with legacy browsers, Web Components may not be the most suitable option [2][1]. Our evaluation of this approach can be found in table 3.7.

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

Table 3.7: Evaluation of composition via Web-components

3.2.8 Summary

As we can see, all of the approaches have their set of advantages and disadvantages. We have realized that there is no such thing as the best architecture, only the lesser of the worse options, based on the project specifications. Each approach can be a good choice for certain projects. The table 3.8 combines all tables into a single, easy-to-read one, allowing for direct comparisons of all the mentioned approaches. The names of the approaches have been shortened so that the table fits well on the page.

Aspect	Link	Iframes	Ajax	Server	Edge	MF	WC
Extensibility	4/5	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	4/5
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

Chapter 4

Design

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

4.1 Application introduction

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

4.2 Requirements

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

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

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

4.2.1 Functional Requirements

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

User Management

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

Project Management

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

Task Management

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

Dashboard

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

Settings

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

4.2.2 Non-functional Requirements

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

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

4.3 System Architecture

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

4.3.1 Application Shell

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

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

4.3.2 Users Microfrontend

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

4.3.3 Projects Microfrontend

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

4.3.4 Tasks Microfrontend

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

4.3.5 Dashboard Microfrontend

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

4.4 Tech Stack

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

4.4.1 Angular

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

4.4.2 TypeScript

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

4.4.3 Web Components

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

4.4.4 Browser Events

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

4.5 Graphical User Interface

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

4.5.1 Application Shell

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



Figure 4.1: Application shell wireframe

4.5.2 Users Microfrontend

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

4.5.3 Projects Microfrontend

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

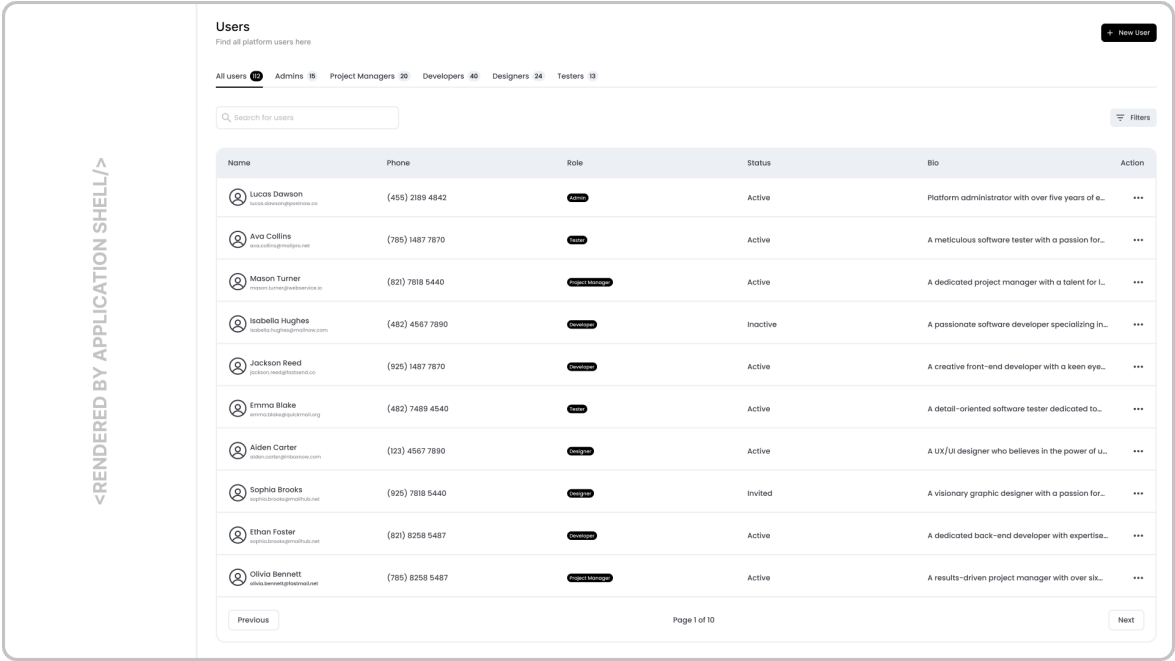


Figure 4.2: Users microfrontend wireframe

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

4.5.4 Tasks Microfrontend

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

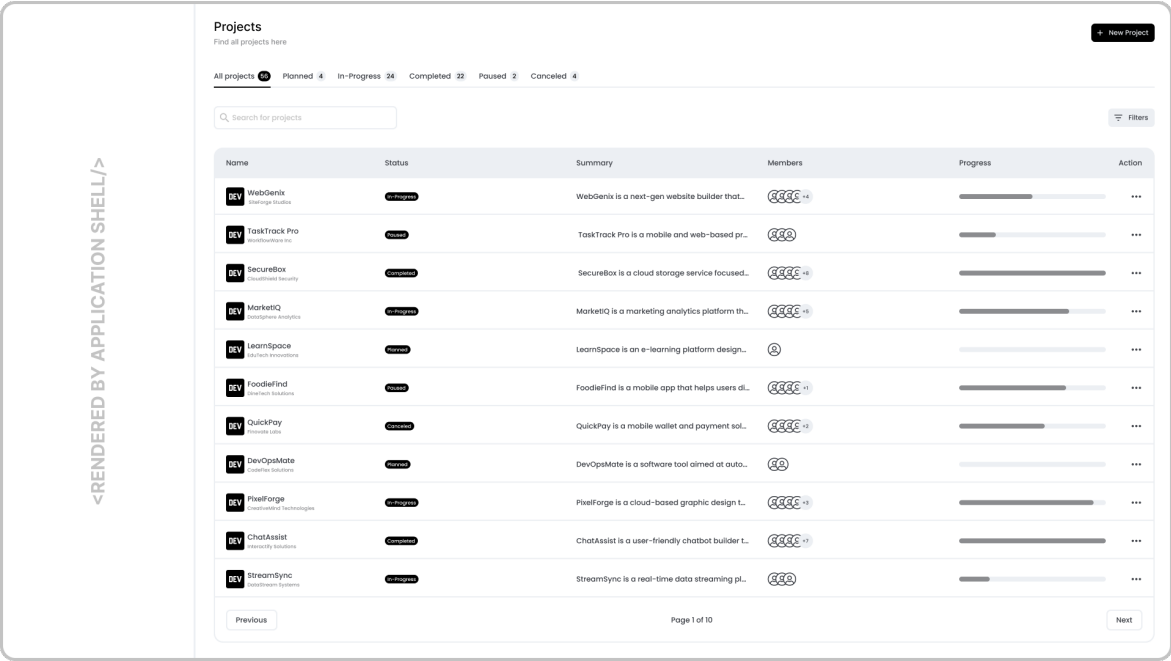


Figure 4.3: Projects microfrontend wireframe

4.5.5 Dashboard Microfrontend

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

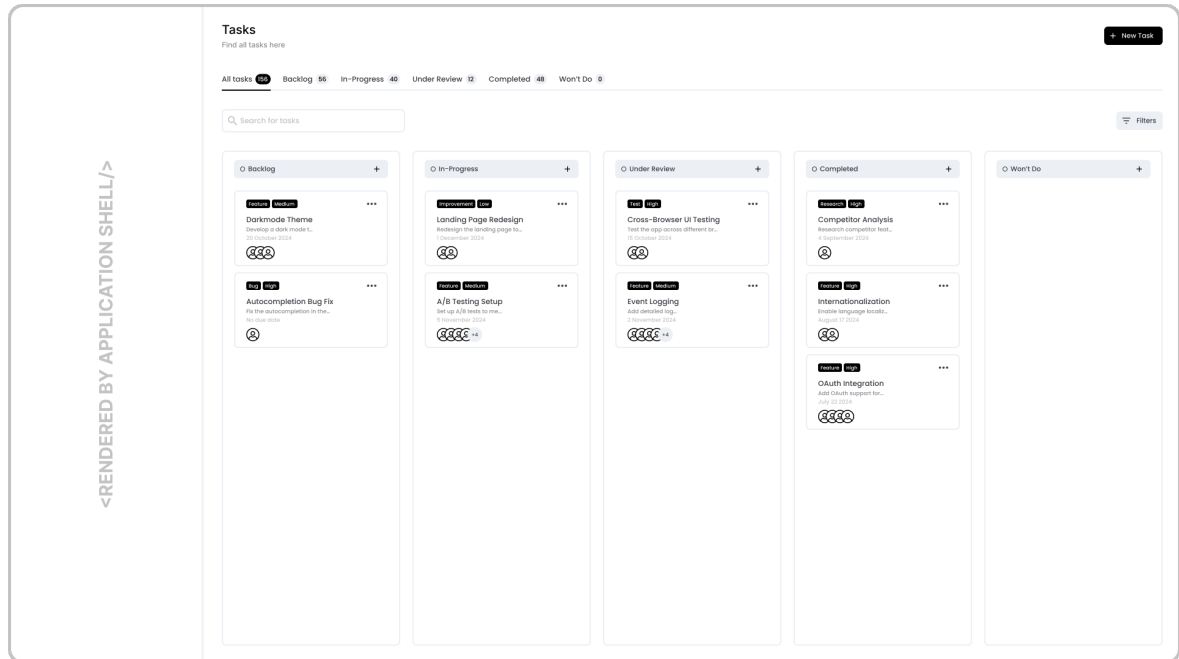


Figure 4.4: Tasks microfrontend wireframe

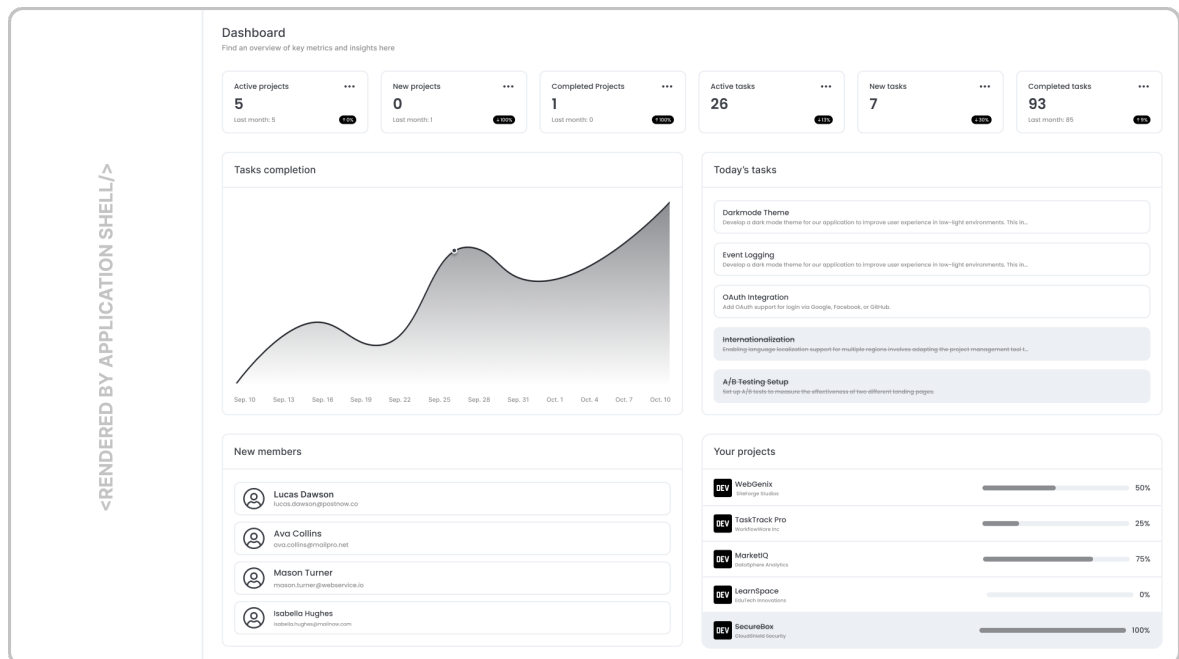


Figure 4.5: ashboard microfrontend wireframe

Chapter 5

Implementation

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

5.1 Overview

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

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

This was followed by generating the application-shell itself:

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

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

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

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

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

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

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

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

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

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

initializeApp();
```

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

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

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

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

};

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

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

```

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

```

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

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

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

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

5.2 Communication

There are two types of inter-application communication occurring.

Application-shell to microfrontend

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

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

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

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

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

And handle it as needed.

Microfrontend to microfrontend

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

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

It can be listened to as follows:

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

5.3 Routing

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

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

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

5.4 Styling

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

5.5 Resource sharing

5.6 Infrastructure

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

5.7 Testing

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

Chapter 6

Conclusion

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

6.1 Implementation Results

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

6.2 Practical Implications

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

6.3 Recommendations for Future Work

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

[?]

Bibliography

- [1] L. Mezzalana, *Building Micro-Frontends*. O'Reilly Media, Inc., 2021.
- [2] M. Geers, *Micro Frontends in Action*. Manning Publications, 2020.
- [3] A. Pavlenko, N. Askarbekuly, S. Megha, and M. Mazzara, “Micro-frontends: application of microservices to web front-ends,” *Journal of Internet Services and Information Security*, vol. 10, pp. 49–66, 2020.
- [4] C. Richardson, *Microservices Patterns: With examples in Java*. Manning Publications, 2018.
- [5] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 2021.
- [6] S. Peltonen, L. Mezzalana, and D. Taibi, “Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review,” *Information and Software Technology*, vol. 136, p. 106571, 2021.
- [7] A. Montelius, “An exploratory study of micro frontends,” Master’s thesis, Linköping University, Software and Systems, 2021.
- [8] C. Jackson, “Micro frontends.” <https://martinfowler.com/articles/micro-frontends.html>, 2019. Accessed: 2024-05-12.
- [9] Google, “Angular - the modern web developer’s platform.” <https://angular.dev/>, 2025. Accessed: 2025-02-01.
- [10] Meta, “React - a javascript library for building user interfaces.” <https://react.dev>, 2024. Accessed: 2024-02-09.
- [11] Amazon, “Amazon official website.” <https://www.amazon.com>, 2024. Accessed: 2024-02-09.
- [12] R. Brigham and C. Liquori, “Devops at amazon: A look at our tools and processes.” Presentation at AWS re:Invent, 2015. Accessed: 2024-02-09.
- [13] Netflix, “Netflix official website.” <https://www.netflix.com>, 2024. Accessed: 2024-02-09.

- [14] Spotify, “Spotify official website.” <https://www.spotify.com>, 2024. Accessed: 2024-02-09.
- [15] Uber, “Uber official website.” <https://www.uber.com>, 2024. Accessed: 2024-02-09.
- [16] A. Kwiecień, “10 companies that implemented the microservice architecture and paved the way for others.” <https://www.cloudflight.io/en/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way>, 2019. Accessed: 2024-01-18.
- [17] M. Fowler and J. Lewis, “Microservices.” <https://martinfowler.com/articles/microservices.html>, 2014. Accessed: 2024-04-04.
- [18] C. Richardson, “Microservices.io,” 2024. Accessed: 2024-03-15.
- [19] “Thoughtworks ’technology radar: Micro frontends’.” <https://www.thoughtworks.com/radar/techniques/micro-frontends>, 2016. Accessed: 2024-09-10.
- [20] M. Geers, “Micro-frontends - extending the microservices idea to frontend development.” <https://micro-frontends.org>, 2017. Accessed: 2024-08-26.
- [21] “Broadcast channel api.” https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API, 2025. Accessed: 2025-02-17.
- [22] IKEA, “Ikea official website.” <https://www.ikea.com/>, 2024. Accessed: 2024-02-07.
- [23] J. Stenberg, “Experiences using micro frontends at ikea.” <https://www.infoq.com/news/2018/08/experiences-micro-frontends/>, 2018. Accessed: 2024-08-24.
- [24] Zalando, “Zalando official website.” <https://www.zalando.com/>, 2024. Accessed: 2024-02-11.
- [25] Zalando, “Tailor.js: A streaming layout service for front-end microservices.” <https://github.com/zalando/tailor>, 2022. Accessed: 2024-08-15.
- [26] SAP, “Sap official website.” <https://www.sap.com/>, 2024. Accessed: 2024-04-11.
- [27] SAP, “Luigi: The enterprise-ready micro frontend framework.” <https://luigi-project.io/>, 2024. Accessed: 2024-04-11.
- [28] DAZN, “Dazn: Live and on demand sports streaming.” <https://www.dazn.com/>, 2025. Accessed: 2025-01-05.

- [29] “Web components.” https://developer.mozilla.org/en-US/docs/Web/API/Web_components, 2025. Accessed: 2025-02-22.
- [30] R. Eisenberg, “About web components.” <https://eisenbergeffect.medium.com/about-web-components-7b2a3ed67a78>, 2023. Accessed: 2025-02-20.
- [31] M. W. Docs, “Customevent.” <https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent>, 2025. Accessed: 2025-02-23.
- [32] J. James, “Custom events in javascript: A complete guide.” <https://blog.logrocket.com/custom-events-in-javascript-a-complete-guide/>, 2021. Accessed: 2025-01-15.
- [33] “<iframe>: The inline frame element.” <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe>, 2024. Accessed: 2024-11-30.
- [34] “Ajax.” <https://developer.mozilla.org/en-US/docs/Glossary/AJAX>, 2024. Accessed: 2024-11-26.
- [35] PostCSS, “Postcss: a tool for transforming css with javascript.” <https://postcss.org/>, 2025. Accessed: 2025-02-24.
- [36] D. Taibi and L. Mezzalira, “Micro-frontends: Principles, implementations, and pitfalls,” *ACM SIGSOFT Software Engineering Notes*, vol. 47, pp. 25–29, 09 2022.