COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS

ANALYSIS, DESIGN AND IMPLEMENTATION OF MICRO-FRONTEND ARCHITECTURE

Diploma thesis

2024 Bc. Pavol Repiský

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS

ANALYSIS, DESIGN AND IMPLEMENTATION OF MICRO-FRONTEND ARCHITECTURE

Diploma thesis

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

Bratislava, 2024 Bc. Pavol Repiský

37862688

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

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

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

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

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

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

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

37862688

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

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

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

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

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

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

Supervisor:	RNDr. Ľubor Šešera, PhD.
Consultant:	Ing. Juraj Marák
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.
Assigned:	05.10.2023
Approved:	05.10.2023

prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

Acknowledgement

Tu môžete poďakovať školiteľovi, prípadne ďalším osobám, ktoré vám s prácou nejako pomohli, poradili, poskytli dáta a podobne.

Abstrakt

Slovenský abstrakt v rozsahu 100-500 slov, jeden odstavec. Abstrakt stručne sumarizuje výsledky práce. Mal by byť pochopiteľný pre bežného informatika. Nemal by teda využívať skratky, termíny alebo označenie zavedené v práci, okrem tých, ktoré sú všeobecne známe.

Kľúčové slová: jedno, druhé, tretie (prípadne štvrté, piate)

Abstract

Abstract in the English language (translation of the abstract in the Slovak language).

Keywords:

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a brief introduction to microfrontends and the motivation behind the research. It then presents the aim of the study. Finally, it acknowledges the scope and limitations of the study.

1.1 Background and Motivation

The landscape of web development is constantly changing. We've progressed from simple, static text-based pages to highly dynamic interfaces, largely thanks to CSS, JavaScript, and AJAX. Eventually leading to adoption of Single Page Applications (SPAs), powered by frontend libraries and frameworks like React and Angular, reducing the gap between web and desktop applications [11][9].

The backend side of web apllications has also transformed. As applications expand, their codebases grow larger, often leading to excessive coupling and a lack of comprehensive understanding of how the application functions. This complexity causes challenges related to maintenance and scalability. This resulted to a shift from traditional monolithic structures to modular microservices-based architectures. In this architecture style, applications are aplit into smaller, independent services that can be developed and deployed separately, which simplifies the maintainability and scalability aspects. The microservices architecture has gained popularity in recent years and has been embraced by major companies such as Amazon and Netflix [5]. However, while this approach has addressed the backend struggles, similar issues are now being encountered on the frontend side.

The term "Micro Frontends" was first introduced in ThoughtWorks Technology Radar [3] in 2016. It can be described as an extension of microservices to the frontend layer. In a Microfrontend architecture, the application is divided into multiple features, each owned by independent teams. These features are then seamlessly composed together to form a whole [6][4].

While the microservices architecture has been extensivly studied, understood and adopted, its counterpart on the frontend side remains under-explored and under-theorized. This study tries to reduce this gap by conducting an analysis and implementation of microfrontends within the context of modern enterprise-level web development.

1.2 Aim

This study aims to review the current literature on approaches for designing and developing web applications using microfrontend architecture. It seeks to conduct a comparative analysis of these approaches, focusing on their reusability, extendibility, resource sharing, and application state management. Special attention will be given to identifying approaches most suitable for enterprise-level application development. One such approach will be tested in a battle, by designing and developing a prototype microfrontend web application. Finally, conclusions will be drawn, and the selected approach will be evaluated based on the resulting application, providing a level of correctness and effectiveness of the chosen approach.

1.3 Scope and Limitations

This study focuses on exploring the concept of microfrontends and their application in modern web development, particularly in the context of enterprise-level applications. The scope encompasses: analysis of microfrontend approaches, their comparison, prototype design and development, and assessment of one chosen approach.

While this study aims to provide insights into microfrontend architecture, certain limitations must be acknowledged. Due to the breadth of the topic, the thesis will not cover every prominent aspect of microfrontend architecture. Furthermore, it will not cover every existing approach to implementing microfrontends. Instead, it focuses on those most commonly referenced in literature and industry practices. Findings and conclusions drawn from the prototype development may not be universally applicable to all scenarios. The prototype development will be constrained by time and technological resources available, impacting the complexity and scale of the application. The resulting application primarily functions as a proof of concept rather than a fully deployable solution for real-world use.

Chapter 2

Analysis

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

2.1 Web Application Architectures

2.1.1 Monolithic Applications

A monolithic application represents a traditional architectural style where all components are interconnected and tighly coupled, forming a single, unified codebase, that operates independently from other applications. It couples all of the business concerns together. When some changes are made to any part of the application, it necessitates rebuilding and redeploying the entire codebase. Development is typically horizontally partitioned into frontend, backend, and database layer teams, thereby decomposing the code into three distinct layers. The monolith can be scalled up, by running multiple its instances behind a load-balancer.

Monolithic applications are easier to debugg, test, deploy and monitor, given that everything is encompassed within a single codebase. Using a singular API, as opposed to multiple ones, can potentially enhance performance, and the initial project configuration is small. However, this architecture comes with numerous disadvantages. The tightly coupled nature of large monolithic codebases makes them challenging to fully understand, resulting in more complex and slow development processes. Individual components cannot be scaled independently, and an error in one such component can affect the entire application. Moreover, transitioning to new technologies is nearly impracticable as it necessitates rewriting the entire application.

Thus, the monolithic approach is best suited for simple, small-scale applications that do not anticipate frequent code changes or evolving scalability requirements such as content-based websites.

2.1.2 Server-side Rendered Application

Server-side rendered applications are web applications in which the majority of the HTML content is generated on the server before it is sent to the user's browser. Which can also include fetching data from APIs, composing components together, and applying styling. This is in contrast to client-side rendered applications, where the browser generates the HTML content after receiving data from the server.

Server-side rendering offers improved performance by delegating rendering to the server. It also provides easy indexation by search engines and better accessibility. A server-side rendered application speeds up initial page load time. However, there are also some trade-offs to consider when using SSR. Applications may experience higher server load since HTML must be rendered on the server for each request, and therefore, higher cost. Managing application state can be more complex, and a number of third-party libraries and tools are incompatible with SSR. Additionally, SSR may cause slower page rendering in case of frequent server requests.

Server-side rendering is ideal for static HTML site generation, which does not require too many requests, or for projects with a focus on SEO and accessibility, such as blogging platforms.

2.1.3 Single Page Applications

Single page application or SPA for short, is a modern architecture approach to the development of client-side rendered web applications in which all essential resources are downloaded during the initial page load. As users interact with the application, the DOM undergoes dynamic updates via JavaScript and HTTP requests to fetch data from the server, eliminating the need for full page reloads. Frameworks like React, Angular, and Vue have streamlined SPA development by offering robust tools for managing application state, routing, and data binding.

The key adavtanges of SPAs are a dynamic, app-like user experience, enhanced performance and reduced server load. The obvious downside of this architecture is a longer initial load time, which can be improved by lazy loading. Furthermore, the codebase may quiclky expand expand and become challenging to maintain. Multiple teams working on the same codebase, but different areas of the application can result in conflicts and communication overhead.

SPAs are best suited for highly responsive and interactive applications such as editors or design tools [12].

2.1.4 Microservices

Microservices is an architectural style for developing web applications as a suite of loosely coupled, independently deployable small services. Each service is responsible for a specific business concern and can be developed, deployed, and scaled independently of other services. Each of these services typically uses its own databases and communicate with other services using lightweight mechanisms, such as an HTTP resource API. They rely on automated processes mainly for deployment, and each service can be written in a different programming language.

Microservices offer enhanced team productivity by splitting into small, focused teams to concentrate on particular services and their development, deployment, and maintenance without being burdened by the complexities of the entire system. This also results in quicker deployment times. As the services are independent of each other, errors in one service does not affect entire system. The system can be easily scaled by adding new services or multiple instances of existing ones. They are technology-agnostic, and teams can experiment with new features and technologies, which can lead to innovations. Because of the smaller codebases, developers can better understand the functionality of particular services. On the other hand, microservices require a lot of complex upfront configuration. Since the services are independent, it's hard to manage communication between them, leading to difficult integration testing and debugging. As microservices communicate over a network, this can cause longer response times. Microservices require a high amount of automation and a complex infrastructure, which can lead to increased costs and operational overhead.

Microservices are best suited for building large and complex software applications with large teams, particularly those expected to expand, scale, and change.

2.2 Microfrontends In Detail

2.2.1 Overview

overview of microfrontends.

2.2.2 Strengths

List all the benefits of microfrontends.

2.2.3 Drawbacks

List all the negatives of microfrontends.

2.2.4 Case Studies

Mention some notable companies such as Zalando, Upwork, and Dazn that have adopted microfrontends, and discuss their experiences with this approach.

2.3 Composition approaches

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

2.3.1 Link-based composition

One of the simplest approaches to a microfrontend architecture is the composition of multiple applications using hyperlinks. In this method, each such application is developed by a dedicated team that creates its own HTML, CSS, and JavaScript files. Each application is deployed independently, typically on a different port under the same domain, ensuring it remains fully isolated from other applications in the system. To facilitate navigation between applications, teams must share their URL patterns that will be used to link their sites from other teams' applications.[6][10] Example of such URL patterns might look like the following.

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

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

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

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

Advantages

This composition method offers several benefits. It provides complete isolation between microfrontends, eliminating direct communication between them. Errors in one microfrontend are contained and do not affect other parts of the system. The microfrontends can be deployed independently, and the project requires minimal configuration. The system can be easily expanded by adding new microfrontends [6].

Disadvantages

The primary limitation of this approach is that it does not allow for the integration of multiple microfrontends on a single page. Users must click through links and wait for page loads when navigating between different sections, potentially worsening the user experience. Additionally, common elements, such as headers, need to be reimplemented and maintained separately within each microfrontend, leading to code duplication, maintenance overhead, and potential inconsistencies. Resource sharing between such applications is also very complex. [6]

Suitability

While microfrontend composition via links provides a basic integration strategy, it is rarely used as the sole approach in modern web development due to its limitations in resource sharing between microfrontends. It is typically combined with other techniques to create more robust solutions. [6]

2.3.2 Composition via iframes

Iframes are an old yet still widely used technique in web development. Essentially, an iframe is an inline HTML element that represents a nested browsing context, allowing one HTML page to be embedded within another. Each embedded context has its own document and supports independent URL navigation [2]. Compared to link-based

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

Table 2.1: Characteristics assessment of link-based composition

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

Iframes can communicate with the host page through the postMessage() method and are supported across all browsers [6][7][8][4]. Adding an iframe is as simple as including an HTML tag, and its behavior can be customized with additional attributes. This time, to compose the microfrontends, we would use the iframe element. The difference is that the linked microfrontend would be displayed directly within the application that linked it.

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

Advantages

The biggest advantages of using iframes are their excellent robustness and isolation in terms of styling and scripts not interfering with each other. Iframes are also very easy to set up and work with, and as already mentioned, are fully supported by all major browsers. [6][7][8][4]

Disadvantages

However, the main advantage of iframes also comes with a cost. It is impossible to share common dependencies across different iframes, leading to larger file sizes and longer download times. Communication between iframes is restricted to the iframe API, which is cumbersome and inflexible. These limitations make tasks such as routing and history management challenging. Additionally, the host application must know the height of the iframe in advance to prevent scrollbars and whitespace, which can be particularly tricky in responsive designs. Using numerous iframes on the same page can significantly degrade application performance. Lastly, iframes perform poorly in terms of search engine optimization (SEO) and accessibility. [6][7][8][4]

Suitability

Despite the numerous disadvantages, iframes can still be the most suitable choice in some cases. Iframes shine when there is not much communication between micro-frontends and the encapsulation of our system using a sandbox for every micro-frontend is crucial. The best use cases for iframes are in desktop, B2B, and internal applications. They should be avoided if performance, SEO, accessibility, or responsiveness are crucial factors.[6][8]

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

Table 2.2: Characteristics assessment of composition via iframes

2.3.3 Composition via Ajax

Asynchronous JavaScript and XML (AJAX) is a technique that enables fetching content from a server through asynchronous HTTP requests. It then uses the retrieved content to update parts of the website without requiring a full-page reload [1]. To use AJAX as an approach for microfrontends, each team must first expose their microfrontend on a specific endpoint. Next, we create a corresponding empty element in the host application and specify the URL in a data attribute from which the content should be downloaded. Finally, JavaScript code is needed to locate the element, retrieve the URL, fetch the content from the endpoint, and append it to the DOM.

Compared to composition via iframes, accessibility and SEO are no longer issues. However, this approach introduces additional challenges, such as CSS conflicts if two microfrontends use the same class names, which can lead to unintended overrides. To avoid this, all CSS selectors should be namespaced. Numerous tools, such as SASS, CSS Modules, or PostCSS, can handle this automatically. A similar issue can occur with scripts; in that case, we can encapsulate scripts within Immediately Invoked Function Expressions (IIFE) to limit the scope to the anonymous function and again namespace global variables. [6] The composition-related code would look something like this.

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

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

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

Advantages

Since all the microfrontends are included in the same Document Object Model (DOM), they are no longer treated as separate pages, as was the case with iframes. This eliminates the SEO and accessibility issues associated with iframe-based composition. Additionally, there are no issues with responsiveness, as the microfrontends will be loaded as standard HTML elements, which can be styled as needed. This approach also provides greater flexibility for error handling; if the fetch request fails, for example, a direct link to the standalone page can be provided. [6]

Disadvantages

One obvious disadvantage is the delay before the page is fully loaded. Since microfrontends must be downloaded, parts of the page may appear a bit later, which can worsen the user experience. Another significant issue is the lack of isolation between microfrontends. Lastly, all lifecycle methods for the scripts must be implemented from scratch. [6]

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. [6]

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

Table 2.3: Characteristics assessment of composition via Ajax

2.3.4 Server-side composition

Server-side composition is typically managed by a service positioned between the browser and the application servers, often using tools like Nginx. This service assembles the view by aggregating various microfrontends and constructing the final page before delivering it to the browser. There are two primary approaches to achieving this.

Server-side Includes (SSI)

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

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

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

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

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

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

- Tailor: A Node.js library developed by Zalando
- Podium: Built upon Tailor by Finn.no, Podium extends its capabilities and provides additional features. [6]

Advantages

Server-side composition is proven, robust, and reliable technique. It offers excellent first-load performance, as the page is pre-assembled on the server, providing a positive impact on search engine ranking. Maintenance is straightforward, and interactive functionality can be added via client-side JavaScript. It also well-tested and well-documented. [6][8]

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. [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. However, it may not be optimal for pages requiring a high level of interactivity. It is also well-suited for B2B applications with numerous modules that are reused across different views. [6][8]

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

Table 2.4: Characteristics assessment of server-side composition

2.3.5 Composition via Web-components

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

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

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

HTML Templates
HTML Templates allow for the definition of markup templates that can be reused and instantiated multiple times within a document. These templates are not rendered until they are explicitly activated. [6][8]

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

```
<!-- Definition of the custom element -->
class ProductComponent extends HTMLElement {
```

```
    constructor() {
        super();
        this.attachShadow({ mode: 'open' });
    }

    connectedCallback() {
        this.render();
    }

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

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

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

Advantages

Web-components are a widely implemented web standard that offer strong isolation managed by the browser, making applications more robust. Web Components can be used across different frameworks and libraries, such as Angular and React. They inherently support lazy loading and code splitting, enhancing performance. Additionally, Web Components enable the encapsulation of complex functionality into reusable building blocks and provide a convenient way for components to communicate with one another. [6][8]

Disadvantages

One common criticism of Web-components is their reliance on JavaScript for functionality. They are not fully supported in all older browsers; while polyfilling can extend support for custom elements, integrating the shadow DOM is notably more challenging, and polyfills significantly increase the bundle size. Additionally, Web-components lack built-in state management mechanisms, complicating integration with existing solutions. Finally, handling search engine optimization can also be challenging. [6][8]

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. [6][8]

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

Table 2.5: Characteristics assessment of composition via Web-components

Chapter 3

Design

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

3.1 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.

3.2 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.

3.2.1 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)

3.2.2 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)

3.3 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.

3.3.1 Application Shell

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

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

3.3.2 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.

3.3.3 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.

3.3.4 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").

3.3.5 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.

3.4 Tech Stack

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

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.4 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.

3.5 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.

3.5.1 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 3.1: Application shell wireframe

3.5.2 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.

3.5.3 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,

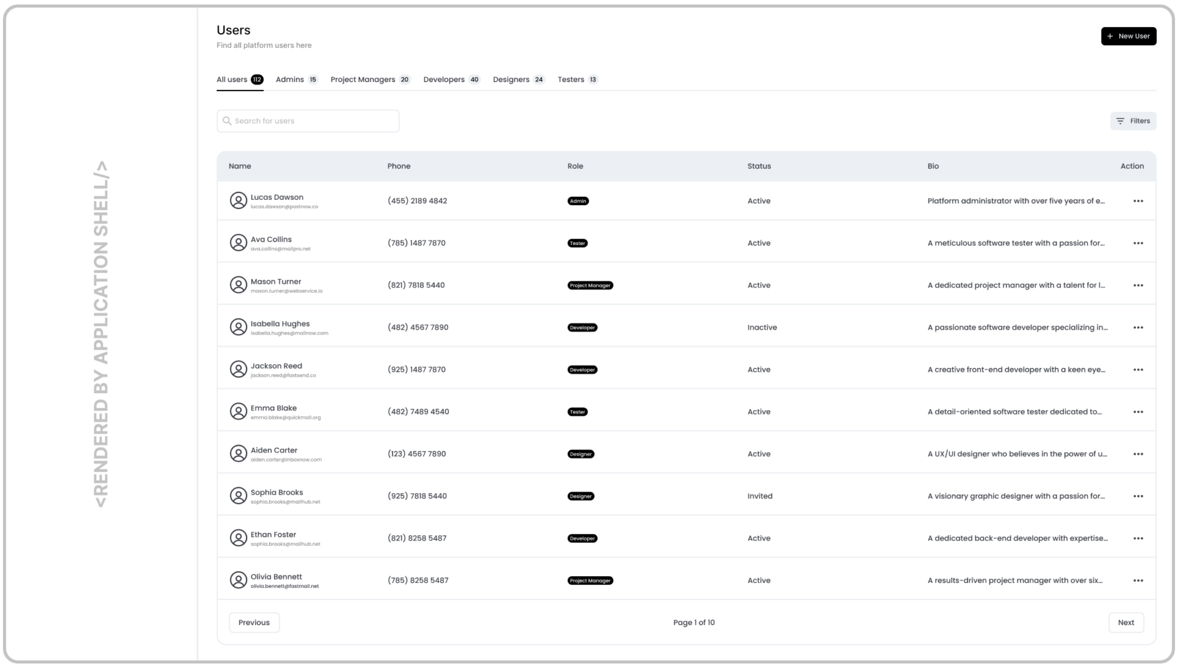

Figure 3.2: Users microfrontend wireframe

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.

3.5.4 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.

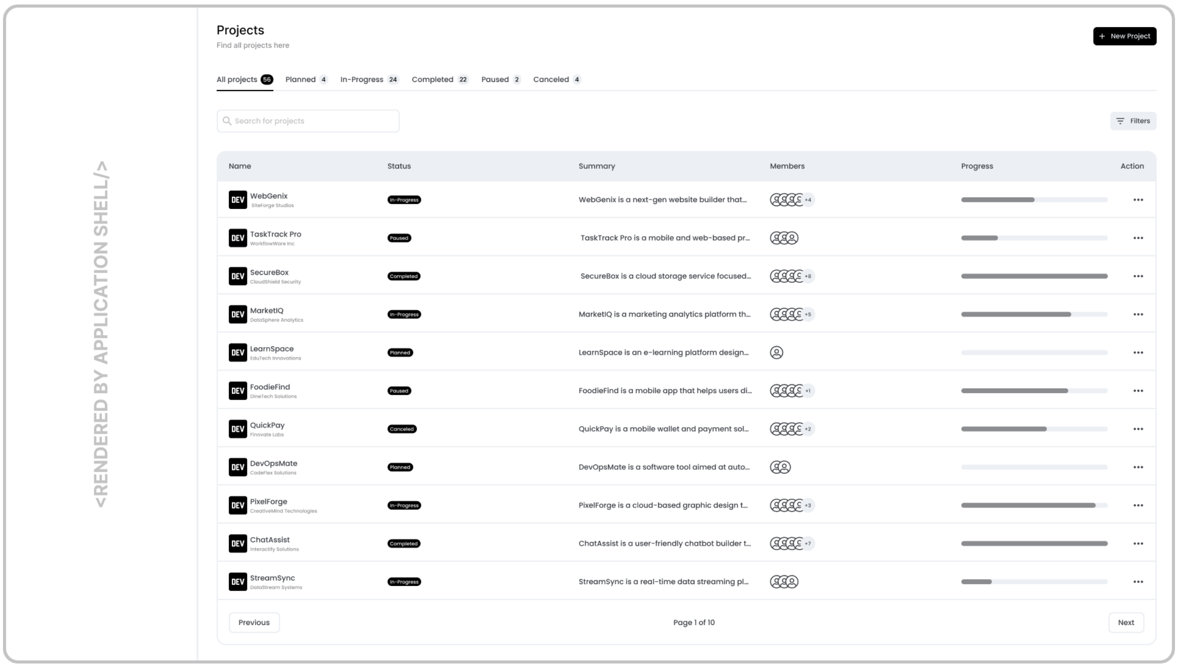

Figure 3.3: Projects microfrontend wireframe

3.5.5 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.

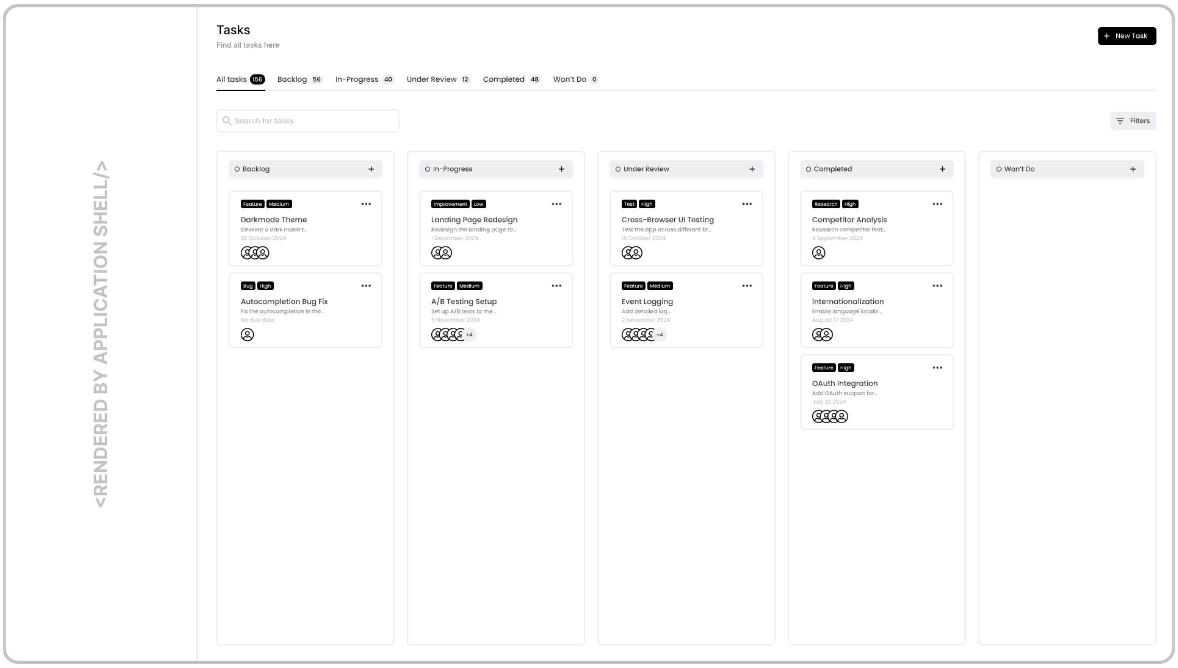

Figure 3.4: Tasks microfrontend wireframe

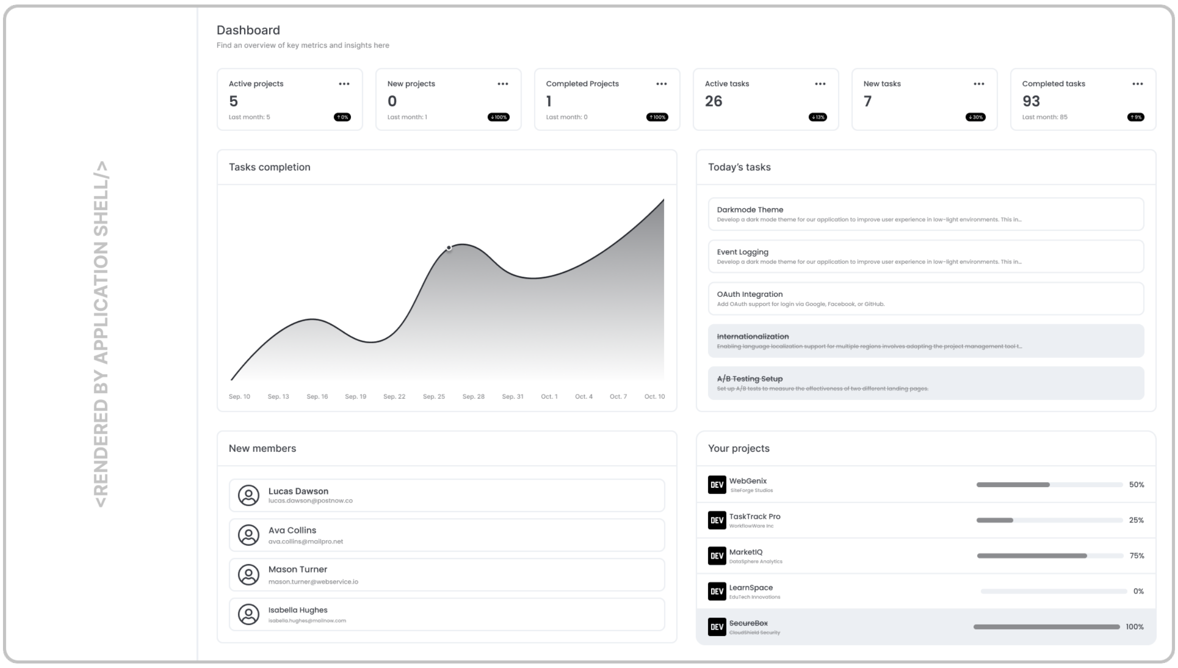

Figure 3.5: ashboard microfrontend wireframe

Chapter 4

Implementation

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

4.1 Overview

- Describe the application implementation process.
- Highlight key considerations and the overall approach to implementing micro-frontends.

4.2 Styling & Sharing

- 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.

4.3 Routing

- How was the routing issue resolved

- Which technologies are utilized for both internal and external routing

4.4 Cross-application communication

- Discuss when micro-frontends must communicate with each other.
- Explain the techniques utilized for communication.
- Discuss how was tight coupling avoided.

4.5 Versoning & 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

4.6 Testing

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

Chapter 5

Conclusion

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

5.1 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.

5.2 Practical Implications

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

5.3 Recommendations for Future Work

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

[?]

Bibliography

[1] Ajax. *MDN Web Docs.*

[2] <iframe>: The inline frame element. *MDN Web Docs.*

[3] Micro frontends. *ThoughtWorks Technology Radar*, 5 2020.

[4] Swati Megha Andrei Pavlenko, Nursultan Askarbekuly and Manuel Mazzara. Micro-frontends: application of microservices to web front-ends. 05 2020.

[5] Martin Fowler and James Lewis. Microservices. 03 2014.

[6] Michael Geers. *Micro Frontends in Action.* Manning, 2020.

[7] Cam Jackson. Micro frontends. 06 2019.

[8] Luca Mezzalira. *Building Micro-Frontends.* O'reilly, 2021.

[9] Anna Montelius. An exploratory study of micro frontends. 2021.

[10] okmttdhr. Micro frontends patterns. *dev.to*, 2020.

[11] Nilesh Savani. The future of web development: An in-depth analysis of micro-frontend approaches. *International Journal of Computer Trends and Technology*, pages 65–69, 11 2023.

[12] Luca Mezzalira Severi Peltonen and Davide Taibi. Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review. *Information and Software Technology*, 136:106571, 2021.