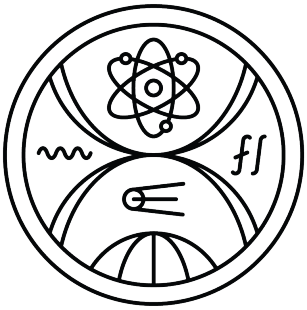
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS

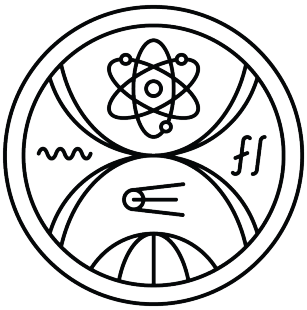


ANALYSIS, DESIGN AND IMPLEMENTATION OF MICRO-FRONTEND ARCHITECTURE

Diploma thesis

2024 Bc. Pavol Repiský

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS



ANALYSIS, DESIGN AND IMPLEMENTATION OF MICRO-FRONTEND ARCHITECTURE

Diploma thesis

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

Bratislava, 2024 Bc. Pavol Repiský

37862688

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

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

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

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

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

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

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

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

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

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

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

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

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

Assigned: 05.10.2023

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

Acknowledgement

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 [10][8].

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].

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.

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

When considering the architecture of a micro-frontends application, there are several options to choose from. Some are as straightforward as employing a link, while others could fill a whole book. However, there is not a one-size-fits-all solution, it is all about finding the optimal balance for each project. This section presents the five primary approaches, outlining their advantages, disadvantages, and use cases, ranging from the most straightforward to more complex ones.

2.3.1 Composition via Links

In this simple, traditional architectural pattern, the application is decomposed into independent standalone pages - microfrontends, which are then interconnected through hyperlinks. Each page comes with its own HTML and CSS and is served directly via the team's applications. All teams must share URL patterns, which will be used to reference their pages from other teams' pages. Examples of such patterns include:

- Team A - Order details page
  URL pattern: `http://example.com/order/<order-id>`
- Team B - Invoice details page
  URL pattern: `http://example.com/invoice/<invoice-id>`

A common scenario, where this technique can be used involves a standalone cart check-out pages being linked from various other pages.

```
<html>
   <head>
       <title>Order Details</title>
```

```
        <link rel="stylesheet" href="static/page.css">
    </head>
    <body>
        <!--Page content goes here -->
        <h1>Order Details</h1>
        <p>Details of the order....</p>

        <!-- Link to the invoice details page -->
        <a href="http://example.com/invoice/3">View Invoice</a>

        <!-- More page content -->
    </body>
</html>
```

Advantages

This simple approach has several advantages. Its most significant benefit lies in its easy setup; the project can be bootstrapped in a matter of minutes. Additionally, it provides complete isolation between microfrontends, eliminating any direct communication between them. Instead, they interact solely through hyperlinks. Any errors occurring within one microfrontend are isolated, preventing them from affecting the rest of the system.

Disadvantages

The most notable drawback of this approach is its inability to combine different microfrontends into a unified view. Users are required to navigate between them by clicking on respective hyperlinks, which can lead to a bad user experience.
Moreover, common elements such as headers must be individually implemented and maintained in each microfrontend, resulting in duplication of code and effort and potential inconsistencies across the application.

Suitability

Although microfrontend composition via links provides a foundational strategy, it is typically not employed on its own in modern website development due to its limitations in seamlessly combining microfrontends within a single page. Instead, it is often complemented with other techniques. [6][9]

2.3.2 Composition via Iframes

Iframes are a well-established technique in web development. Essentially, an iframe is an HTML element that represents a nested browsing context, allowing the embedding of another HTML page into the current one. Compared to composition via links, the transition to iframes requires minimal changes—all that's needed is to include an iframe tag and specify the URL. Thus, teams only need to share URL patterns with each other. [2]
[Code sample here]

Advantages

Iframes offer the same loose coupling and robustness as link composition. They are straightforward to set up and provide strong isolation against scripts and styles leaking out. Additionally, they are universally supported by all browsers by default. [4][6][7]

Disadvantages

However, the main advantage of iframes also presents a significant disadvantage. 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, managing history, and implementing deep-linking quite challenging.

Another drawback is that the outer document must precisely know the height of the iframe to prevent scrollbars and whitespace, which can be particularly tricky in responsive designs. While there are JavaScript libraries available to address this, iframes remain performance-heavy. Using numerous iframes on the same page can significantly degrade page speed. Furthermore, iframes are detrimental to accessibility and search engine optimization (SEO), as each iframe is considered a separate page by search engines. [6][4][7]

Suitability

Despite the numerous disadvantages associated with iframes, they can be a suitable choice in specific scenarios. For instance, consider the Spotify desktop application, which incorporates iframes for certain parts of its functionality. In this context, concerns about responsiveness, SEO, and accessibility are not much of an issue.

In conclusion, iframes are not ideal for websites where performance, SEO, or accessibility are crucial factors. However, they can be a suitable option for internal tools

due to their simple setup. [6]

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 leverages the retrieved content to update specific parts of a webpage without requiring a full page reload (AjaxDocs).[1]

When considering microfrontends, the transition from traditional approaches isn't overly complex. Each microfrontend must be exposed as a fragment endpoint and subsequently fetched via an Ajax call, then dynamically inserted into an existing element within the document.

However, this approach introduces unique challenges. Notably, CSS conflicts may arise, particularly if two microfrontends utilize the same class names, leading to undesired overrides. To mitigate such conflicts, it's imperative to namespace all CSS selectors. Various tools, such as SASS, CSS Modules, or PostCSS, facilitate this process automatically.

Additionally, JavaScript isolation is crucial. A commonly employed technique involves encapsulating scripts within Immediately Invoked Function Expressions (IIFE), thereby confining the code's scope to the anonymous function. In instances where global variables are necessary, they can be declared explicitly or managed through namespacing. [6]

Advantages

With all microfrontends integrated into the same Document Object Model (DOM), they are no longer treated as separate pages, as was the case with iframes. Consequently, concerns such as setting precise heights become obsolete. Furthermore, SEO (Search Engine Optimization) and accessibility are no longer issues since they operate within the context of the final assembled page.

This approach also facilitates the implementation of fallback mechanisms in case JavaScript fails, such as providing a link to a standalone page. Moreover, it supports flexible error handling strategies in the event of fetch failures. [6]

Disadvantages

Asynchronous loading, while powerful, can introduce delays in content rendering, resulting in parts of the page appearing later, thereby potentially degrading the user experience. Additionally, a notable concern within this architecture is the lack of isolation between microfrontends, which is a crucial aspect of microfrontends.

Interactions that trigger AJAX calls may suffer from latency issues, particularly in scenarios with poor network connectivity. Furthermore, this approach do not inherently provide lifecycle methods, unless implemeted manually. [6]

Suitability

This approach is well-established, robust, and easy to implement. It's particularly well-suited for websites where the markup is primarily generated on the server side. However, for pages that demand extensive interactivity or rely heavily on managing local state, client-side approaches may prove to be more suitable alternatives.

2.3.4 Server-side Composition

Server-side composition is typically orchestrated by a service positioned between the browser and application server. This service is responsible for assembling the view by aggregating various microfrontends and constructing the final page before delivering it to the browser. There are two primary approaches to accomplish this:

Server-side Includes (SSI): In this approach, a server-side view template is created, incorporating SSI include 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.

Edge-Side Includes (ESI): ESI is a specification that standardizes the process of assembling markup. Similar to SSI, ESI involves including directives within the server-side view template. However, ESI typically requires the involvement of content delivery network providers like Akamai or proxy servers such as Varnish. The web server replaces ESI directives with content from the referenced URLs before transmitting the page to the client.

Other Approaches: Various libraries and frameworks exist, which simplifies the 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

One of the most significant advantages is the excellent first load performance, as the page is pre-assembled on the server, thereby minimizing stress on the user's device. This approach strongly supports progressive enhancement, allowing for additional interactivity to be seamlessly incorporated via client-side JavaScript. Moreover, this technique has been in existence for a considerable period, benefiting from extensive testing, documentation, and refinement. Consequently, it is recognized for its reliability, ease of maintenance, and positive impact on search engine ranking. [6]

Disadvantages

For larger server-rendered pages, browsers may spend considerable time downloading markup instead of prioritizing necessary assets. Moreover, technical isolation is lacking, necessitating the use of prefixes and namespacing to prevent conflicts. Pure server-side solution may notbe optimal for highly interactive applications, requiring combination with other techniques. [6]

Suitability

This approach is ideal for pages prioritizing performance and search engine ranking, as it excels in reliability and functionality even in the absence of JavaScript. However, it may not be optimal for pages requiring instantaneous responses to user input, as it may not offer the level of interactivity needed in such scenarios. [6]

2.3.5 Composition via Web Components

Web components encompass a collection of JavaScript APIs that enable the creation of custom, reusable, and encapsulated HTML elements. This suite of technologies comprises four main components:

Custom Elements Custom Elements represent an extension of HTML elements, empowering developers to define their own custom HTML elements with unique behavior and functionality. They provide callbacks, which can be used to interact with the external environment.

Shadow DOM Shadow DOM provides encapsulation by hiding the implementation details of a custom element from the rest of the document, enabling 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 enable the definition of markup templates that can be reused and instantiated multiple times within a document without rendering until activated.

Advantages

Web Components are widely implemented web standard, they offer string isolation by default making the applications more robust. Web Components can be used across different frameworks and libraries. They provide lazy loading and code splitting by default enhacing performance. They provide a way to encapsulate complex functionality into reusable building blocks.

Disadvantages

One common criticism of Web Components is their reliance on JavaScript for functionality. They are not fully supported in all older browsers, polyfilling can extend support for custom elements, integrating Shadow DOM is notably more challenging. Additionally, Web Components lack built-in state management mechanisms, complicating integration with existing solutions. Lastly, their SEO friendliness is often questioned, presenting a potential drawback for web visibility and indexing.

Suitability

Web components excel in constructing interactive, app-like applications that require seamless integration of user interfaces developed by different teams onto a single screen. Additionally, they serve as an ideal choice when supporting multitenant environments is required. However, for applications prioritizing SEO or requiring compatibility with legacy browsers, web components may not be the most suitable option.

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 primarily focus on the enterprise landscape. It should be logically divisible into somewhat 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 to them and users to tasks, 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.

Project Management

- Users can create, update, and delete projects. (M)
- Projects are displayed in an easy-to-comprehend manner. (M)
- Users can plan projects by specifying start and end dates. (S)
- Project states can be set by users. (S)
- Projects can be viewed on a timeline. (C)

Task Management

- Users can create, update, and delete tasks. (M)
- The task listing page presents all tasks along with their respective information. (M)
- Users can link tasks to projects. (M)
- Task states can be set by users. (S)
- Tasks can be viewed on a kanban board. (C)

Collaboration Features

- Project owners can invite other users to join projects via email. (S)
- Project owners can remove project members. (S)

Dashboard

- A dashboard-like page contains information about new members, active tasks, and active projects. (M)

3.2.2 Functional Requirements

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

- The application must be divided into several microfrontends. (M)
- Each microfrontendis isolated from others to prevent cascading failures. (M)
- The microfrontends comunicate via custom events. (M)

3.3 Decomposition

3.3.1 Microfrontends

3.3.2 Teams

3.4 System Architecture

3.5 Tech Stack

3.6 Graphical User Interface

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

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

[10] 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.