Omar Abdullah Mohammed Al-Mashhadani

Micro-Frontends Integration Strategies: Breaking Boundaries

Metropolia University of Applied Sciences

Bachelor of Engineering

Mobile Solutions

Bachelor’s Thesis

13 May 2024

Abstract

Author:	Omar Al-Mashhadani
Title:	Micro-Frontends Integration Strategies: Breaking Boundaries
Number of Pages:	90 pages
Date:	13 May 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Lassi Sundqvist, EKE's Project Manager Amir Dirin, Senior Lecturer

In the face of a growing emphasis on microservices architecture in software development, this study investigates its potential application in frontend development, a term referred to as 'micro-frontends'. Various resources on frontend development and microservices architecture are analysed, with the intention of outlining their definitions, benefits, and drawbacks.

Furthermore, this study highlights the increasing demand on micro-frontends pattern solutions and presents a detailed demonstration of a conventional project's transition into a micro-frontends architecture.

This thesis focuses on the frontend infrastructure transformation of the EKE-Electronics company's web application. The goal is to enable smoother transitions between two technologies Angular.js and React.js within a single view frame by replacing the existing iframe solution with a more efficient micro-frontends approach.

The thesis discusses three important questions relevant to the field such as what motivates companies to adopt micro-frontends architectures, the advantages, and the challenges that may arise during the implementation of micro-frontend. The thesis involves utilising existing literature, case studies, and industry reports to identify motivations behind micro-frontends adoption to address the relevant questions. Benefits are analysed and compared with another architecture, emphasising scalability and modularity. Challenges in micro-frontends implementation strategies are analysed, with a focus on modern technologies.

As a result, a functional application was developed. It met user expectations as a proof of concept migrating a micro-frontends architecture to an existing system. Further development is needed to address the application's routing and authentication issues before it can go into production.

Keywords: micro-frontends, refactoring, integration, microservices architecture

Contents

List of Abbreviations

AMD:	Advanced Micro Devices
CSS:	Cascading Style Sheets
DDD:	Domain-Driven Design
HTML:	Hyper Text Markup Language
JS:	JavaScript
MFE:	Micro-Frontend
SEO:	Search Engine Optimisation
SPA:	Single Page Application
Track-CM:	Track Condition Monitoring
Train-CM:	Train Condition Monitoring

1 Introduction

With the increasing demand for efficient and user-centric web applications, businesses are searching for innovative architectural solutions to tackle the complexities of their systems.

This final year project was done at the request of EKE-Electronics/SmartVision, the leading provider of a Train Control and Management System (TCMS) and Condition Based Monitoring (CBM) with turnkey project management and safety control systems (EKE-Electronics Ltd, 2024). EKE has a web application, which is called Train-CM in this project. It is used internally at the company. In the frontend module there is a navigation bar containing two-tab buttons, and each one navigates to a separate application. One of the applications is built using the old web framework Angular.js and the other is built using the high-level object-oriented programming language Python (Python, 2024). Both are managed by iframe, which behaves like a container to manage the routing for both frameworks. This module needs a rebuild. It is therefore of EKE's interest to know if the micro-frontends pattern would be suitable to adopt when rebuilding the frontend module and replacing Python with a React.js application called Track-CM (React, 2024). The idea is to host Train-CM and Track-CM on one view by using a modern micro-frontend and compare it with the current traditional iframe approach.

In the case of EKE-Electronics company's Track-CM and Train-CM applications, this thesis explores the concept of micro-frontends and their integration process. The aim is to analyse the advantages and disadvantages of the micro-frontends compared to a monolith architecture on the frontend side and seamlessly integrating the architecture into the existing applications. One of the applications currently employs iframes to host distinct views developed with separated backend and frontend technologies built with Angular.js, resembling a Single Page Application (SPA) (AngularJS, 2021). However, the current approach of using iframe presents some challenges such as usability issues and some delay regarding the separate authentication required for each application (Smirnov, 2019).

The Condition Monitoring Track-CM is the other application developed using React.js. The application is a vital tool for EKE-Electronics company, enabling real-time monitoring and analysis of crucial systems and equipment. The Python application is going to be replaced with the Track-CM, which is built with React.js. At the end, both applications will be hosted by the micro-frontends application shell.

To address these challenges, a dedicated team will integrate the micro-frontends architecture into the existing application. This approach involves restructuring the application as a collection of small, independent frontend modules that can operate autonomously. This allows the team to enhance specific functionalities independently and efficiently, such as user authentication, data retrieval, and user interfaces with different technologies, combining them into one framework.

The main objective of this study is to analyse the impact of adopting micro-frontends on the performance and user experience of the Condition Monitoring applications. By utilising microservices for different technologies like Angular.js and React.js, the team aims to explore the potential of micro-frontends in managing multi-framework web applications, specifically for a smoother transition between different technologies within a single view frame.

2 Research Questions and Methodology

The goal of this study is to systematically map, review, and synthesise the state of art and practices of web frontend architectures and their field. This goal intended to provide an understanding of the reasons behind the attention given by practitioners and industrial companies to this architectural style. Additionally, it is aimed to highlight the benefits and issues linked to micro-frontends.

The uniqueness of the micro-frontends architecture enables exploring various research avenues since the amount of scientific investigation in this field is limited, especially for the legacy code of Angular.js. For this reason, research questions were defined to cover the most basic topics to get a comprehensive view of this subject but not to go too deep into details.

1. What motivates practitioners and industrial companies to adopt micro-frontends (MFE) architectures for web application development?

2. What are the key benefits and advantages of employing MFE in comparison to other web front-end architectures?

3. What challenges and issues may arise when implementing MFE, and how do practitioners address these concerns?

To address the first question, utilisation of existing literature, case studies, and industry reports are taken to identify the motivations behind the adoption of MFE architectures. Furthermore, an analysis will be conducted to compare the identified motivations and benefits with the industry trends and best practices considering scalability, flexibility, and maintenance advantages of MFE architecture.

For the second question, a review to the works of literature regarding the benefits of MFE architectures is undertaken. Additionally, an analysis is conducted to compare the benefits of traditional monolithic architectures and other frontend architectures, considering the scalability, modularity, and team independence aspects of MFE architectures, and highlight their advantages.

For the last question, the challenges and issues related to implementing MFE architectures will be addressed. The strategies and solutions proposed for addressing these challenges going to be analysed. The scalability, communication overhead, and technology compatibility issues and how practitioners can mitigate them are going to be considered.

In addition, the information and insights from the other theses, that can be addressed the research questions effectively going to be utilised.

3 Related Research

In this chapter, an overview on the related research into the MFEs and their results is taken. Before diving into the definition of microservices and micro-frontends, it is important to understand what the Monolith architects are, and what the challenges of traditional monolithic architectures and the motivations behind adopting a microservices approach are.

After looking at some theses, an overview of some of the existing techniques such as SPA, iframe and Module Federation is presented as general web techniques that are used for both monolith architectures and microservices. Additionally, other options do exist, but these are the most relevant to this work. The focus will therefore be directed towards these initially mentioned techniques to narrow the scope of the research.

The definition of a monolith is introduced with an overview of the architecture and its pros and cons. Then, the section presents micro-frontends definition, some scientific work, discussion, and the definitions of the attributes. Additionally, it outlines methods for software architecture evaluation with highlighting examples of some scientific work using these methods.

Next**,** an evaluation of the chosen architecture is provided. This is followed by a discussion of the methods used in this study. Subsequently, sections covering the theory necessary to understand the different approaches to implementing micro-frontends are presented. Following this, sections on the chosen implementation techniques are included. These techniques involve React.js and Angular.js frameworks, as well as Single-spa and Model-Federation for migrating to micro-frontends architecture. Finally, the framework concepts that are important for achieving the best results in this research are presented.

3.1 Theses of the Same Subject

In this section, the topics that are directly related to microservices and micro-frontends are going to be explored. Specifically, the two key aspects of these architectural paradigms are going to be explored in detail.

3.1.1 Microservices Implementation on Web Development

This thesis focuses on the evolution of a software architecture from a monolith to microservices, particularly exploring the concept of MFE as an alternative structure for client-side web applications. The study acknowledges the growing trend among large tech corporations, such as Netflix, LinkedIn, and Amazon, in adopting microservices to enhance flexibility and maintenance. The MFE, an extension of microservices to frontend development, is introduced to leverage the advantages of microservices. This approach involves segmenting a monolithic application into discrete components that can be developed and deployed independently by separate teams. The research aims to investigate micro-frontends in the context of migrating from a monolith to an MFE architecture, emphasising its benefits and implementation process.

The resulting project demonstrates the successful migration of a monolithic application to an MFE architecture. Despite some challenges in learning the migration process, the project achieves near-identical functionality to the original monolithic application. The micro-frontends applications are properly segregated, allowing independent development and deployment, enhancing the overall maintainability and flexibility of the system. Performance-wise, the migrated application performs similarly to the monolithic version, although the impact of cloud deployment on performance is noted. The study highlights the feasibility of MFE migration, particularly in smaller application settings, and emphasises the need for more clarity in migration methods. (Bui, 2021.)

3.1.2 Moderation Panel for Virtual Event Platform as a MFE Module

This thesis addresses the challenges posed by the COVID-19 pandemic to the event industry, leading to the transformation of Liveto Group Oy from organising physical events to developing a virtual event platform. The company's focus on creating a comprehensive virtual event solution required the development of an efficient moderation panel. This panel, operating with minimal moderator involvement, is developed as an independent micro-frontends module within the new architecture. The study aims to design and implement a user-friendly moderation panel that empowers moderators with tools to review chat feed messages, manage user interactions, and customise chat room settings.

The research process encompasses two core phases: MFE framework exploration and moderation strategies. The implemented moderation panel allows moderators to efficiently review and manage chat messages, including accepting or declining messages and applying word/user blocklists. The panel offers customisation options at both global and chatroom-specific levels, providing flexibility in managing moderation settings. This pioneering MFE module serves as a pivotal component within Liveto’s new modular frontend architecture, contributing to the platform's enhanced interactivity and user experience. (Azizyan, 2022.)

3.2 General Web Techniques (iframe and SPA)

In this section, web techniques specifically relevant to the goal of the project are going to be explored in detail. This thesis aims to replace the proof-of-concept iframe application for EKE-Electronics Company with multiple single-page applications (SPAs) utilising the MFE architecture. To achieve this, the concepts of iframes and SPA going to be explored in detail.

3.2.1 Single Page Application (SPA)

A Single Page Application (SPA) is a type of web application that engages with the user by dynamically modifying the current page rather than loading entire new pages from the server. In a conventional multi-page application, each interaction or request usually involves loading a fresh page from the server, which can result in slower user experiences due to the time it takes to fetch and display new content. In Figure 1 below, an architectural diagram of an SPA and a monolithic backend application are displayed.

In contrast, a single-page application loads only one HTML page and dynamically refreshes its content as the user interacts with it. This is commonly achieved using JavaScript frameworks and libraries. The initial load of an SPA typically includes all necessary assets (HTML, CSS, and JS), while subsequent interactions trigger data retrieval and updates without requiring a complete page reload. (Mezzalira, 2021.)

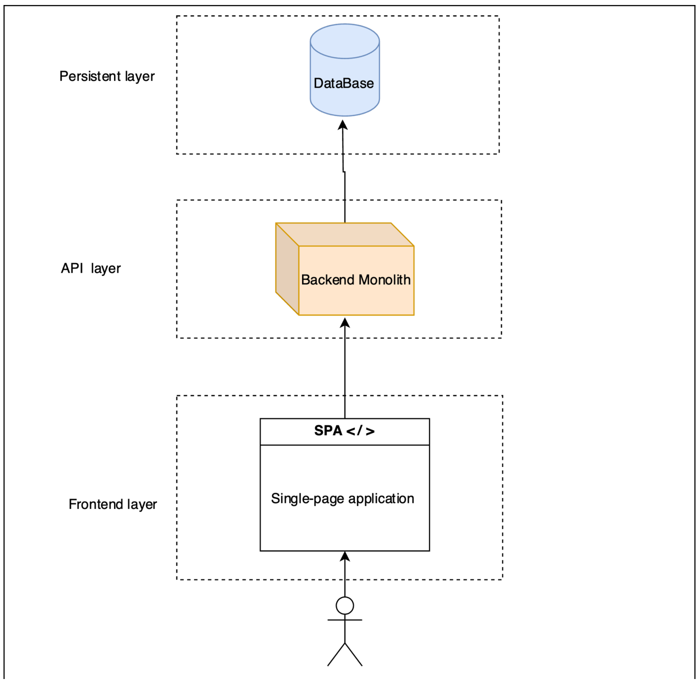


Figure 1: Architectural overview of an SPA frontend on top of a monolithic backend. (Mezzalira, 2021)

A list of advantages and disadvantages of SPA based on both Mezzalira’s (2021) and Newman's (2019, p. 103-104) perspectives can be described as follows:

Advantages:

- Efficient Resource Usage: It downloads the entire application logic upfront, ensuring efficient utilisation of resources during the user's session.
- Optimised Communication: It communicates with the backend through APIs, minimising round trips and enabling quick updates.
- Enhanced User Experience: It replicates the fluidity of desktop applications, providing a seamless and responsive user interface.
- Faster Interaction: Resources are loaded only once, resulting in speedier interactions and improved overall performance.
- Simplified Development: JavaScript frameworks simplify development by enabling the creation of dynamic user interfaces.

Disadvantages:

- Potential Performance Issues: There might be slow load times on less powerful devices, impacting user experience.
- Code Quality Concerns: Poor code implementation could necessitate extensive refactoring efforts for proper functionality.
- SEO Challenges: SPAs may face difficulties with search engine optimisation due to their single-page nature.
- JavaScript Dependency: Users with JavaScript disabled will not be able to use the application, limiting accessibility.
- Initial Load Time: Initial loading of resources may lead to slower page load times though later interactions are quicker.
- Complexity in Development: The unique architecture of SPAs introduces complexity, particularly in larger applications or when managed by distributed teams.

In summary, a Single-Page Application (SPA) is a web application that delivers a seamless user experience by dynamically updating content without requiring full page reloads. This is achieved using JavaScript frameworks, resulting in faster responsiveness and efficient resource usage. Both Newman's and Mazzelira's descriptions emphasise the elimination of page reloads, dynamic content updates, and the role of JS frameworks in creating SPAs.

3.2.2 Inline Frame (iframe)

An iframe, short for inline frame, is an HTML element that enables the embedding of another HTML document within the current one. This creates a distinct web page loaded within the existing page context. (Mezzalira, 2021.)

There are some advantages and disadvantages to iframe. A list to cover both as follows:

Advantages:

- Seamless Integration: Iframes facilitate effortless integration of external content into the site.
- Content Isolation: By loading iframe content separately, isolation is achieved, enhancing security and mitigating conflicts with other scripts.
- Dynamic Display: Iframes dynamically loaded content like ads or social media feeds is employed without necessitating a complete page reload.
- Precise Control: It enhances security by providing fine-grained control over the content that runs within each iframe.
- Micro-frontends Encapsulation: Iframes with sandboxes is utilised to encapsulate micro-frontends, thereby minimising complexities, and memory usage.
- Prevent Leakage: Data confinement within iframes is ensured while enabling controlled inter-iframe communication.
- Dependency Efficiency: Redundant downloading of dependencies for micro-frontends in desktop applications is prevented. (Mezzalira, 2021.)

Disadvantages:

- SEO Implications: iframes can pose challenges for search engine crawling and content indexing, impacting site SEO rankings.
- Performance Impact: The utilisation of iframes can introduce additional HTTP requests, potentially slowing down overall page loading times.
- Accessibility Concerns: Users dependent on assistive technologies like screen readers may encounter navigation difficulties within iframes.
- Security Vulnerabilities: iframes can be exploited to load malicious content or execute clickjacking attacks.
- Suitability Constraints: While advantageous, iframes may not be ideal for every project, particularly for mobile or public-facing applications due to accessibility and performance limitations.
- Load Time Variation: iframes might exhibit slower loading speeds compared to alternative implementations, especially on low-end devices.
- Interaction Limitation: Highly interactive or real-time applications may not be optimally represented by iframes. (Mezzalira, 2021.)

3.3 Monolith Architecture

In this section, an overview of the monolithic architecture is provided to explore various types and discuss some of the architectural challenges. This foundational understanding will prepare for a deeper exploration of microservices and the MFE architecture in the upcoming sections.

3.3.1 Definition of Monolith

A monolith is a “unit of deployment” that refers to a specific type of software architecture. In this architecture, all the functionality of a system is tightly coupled and must be “deployed as a single process” (Newman, 2020, p. 12). This means that any changes or updates to the system require deploying the entire codebase, including all its components and dependencies “with few complications and risks”. (Mezzalira, 2021.)

3.3.2 Monolith Types

There are three types that can be defined as a monolith:

- Single-Process System: In this type of a monolith, all the code and functionality run within a single process. This can lead to challenges in scalability and maintainability since any increase in load or new features necessitates scaling up the entire monolithic application.
- Distributed Monolith: This variant of a monolith involves breaking down the codebase into multiple processes or components. However, despite the distribution, these components remain tightly coupled and dependent on each other. Consequently, deploying changes to one component often requires deploying the entire distributed monolith.
- Third-Party Black-Box Systems: In some cases, a monolith may include third-party components or modules that are integrated tightly into the system. These black-box components become integral to the functioning of the monolith, making it necessary to deploy them together with the rest of the codebase. (Newman, 2019, p. 12-14.)

3.3.3 Challenges of Monolithic Architectures

In traditional monolithic architectures, an entire application is built as a single, large codebase. While this approach might be manageable for smaller projects, it becomes increasingly challenging as the application grows and complexity over time. Some common challenges of monolithic architectures include:

- Scalability: Scaling a monolithic application can be difficult as the entire application needs to be replicated, even if only certain components require more resources.

- Maintainability: As the codebase grows, it becomes harder to maintain and make changes without unintended side effects, leading to longer development cycles.
- Deployment Bottlenecks: Since the entire application is deployed as a single unit, making frequent updates can be risky and time-consuming.
- Technology Diversity: It becomes challenging to use different technologies or programming languages within the same monolith, limiting flexibility and innovation. (Newman, 2019, p. 15.)

3.4 Microservices

In this section, an overview of the microservices architecture in general from a concept and an architectural perspective is presented. Some of the architectural key motivations and disadvantages going to be discussed. This foundational understanding will prepare us for a deeper exploration of the MFE architecture in the upcoming sections.

3.4.1 Definition of Microservices

Given the context of the challenges posed by monolithic architectures, The definition of microservices can be as follows:

From a concept perspective: The concept of a microservices is an architectural decision that diverges from the traditional approach of building a single, extensive application known as a monolith. Instead, it involves breaking down or restructuring the application into smaller independent services, each with its own distinct role within the system “allowing teams full ownership and independent evolution of the codebase” (Mezzalira 2021). At its most basic level, this can be described as having individual services that are responsible for specific tasks. In this way, every service has a purpose for existing and can undergo changes based on their designated role.

From an architectural perspective: “[M]icroservice architecture is based on multiple collaborating microservices referred to as service-oriented architecture

(SOA)". It makes sense to avoid a massive, complex codebase that has been built over a span of 15 years. Instead, developers can opt for a more practical approach by adding focused and targeted services on the side. This allows them to circumvent the inherent complexity of their legacy code when making changes. By breaking down the application into smaller, manageable components, microservices enable faster development cycles, more flexible deployment, and improved maintainability. Microservices are referred to as "independent deployable services modelled around a business domain, using networks to communicate with each other". (Newman, 2019, p. 1.)

Few organisations exclusively rely on microservices; many still maintain monolithic systems as their core infrastructure while incorporating newer microservices at the periphery. The monolith serves as the central system that keeps everything running smoothly while additional functionalities are added through microservices at various stages throughout the expanding system. It is simply a choice, and one does not have to strictly adhere to one method or the other.

This analogy can be likened to the solar system of the earth, where the sun represents the monolith at the centre and an increasing number of microservices act like planets orbiting around it. As the system expands outward with new features and functionalities, these smaller modularised services provide flexibility and ease of management without compromising stability. (Very and Bell Main, 2023.)

In contrast, modern architectures like microservices promote loose coupling, enabling developers to work on smaller, independent components and have better experience by working with preferred technology. This approach fosters agility, scalability, and easier maintenance, making it increasingly popular in contemporary software development practices.

3.4.2 Motivations for Microservices

Microservices offer an alternative approach to tackle the challenges posed by monolithic architectures. Some key motivations for adopting microservices include:

- Modularity: microservices promote modularity by breaking down the application into smaller, independent services, making it easier to develop, maintain, and understand.
- Scalability: Each microservice can be scaled independently based on its specific resource requirements, allowing for more efficient resource utilisation.
- Faster Deployment: With microservices, smaller and focused codebases can be deployed independently, enabling faster and more frequent updates.
- Technology Flexibility: Different microservices can be built using diverse technologies, enabling teams to use the best tools for specific tasks.
- Team Autonomy: Microservices facilitate team autonomy since each team can work on a specific microservice, making development and decision-making more streamlined.

(Newman, 2019, p. 6; Mezzalira, 2021.)

3.4.3 Disadvantages of Utilising Microservices

There are some pitfalls to working with microservices. “It brings many benefits to the table but can bring many drawbacks as well” (Mezzalira, 2021). These benefits and drawbacks are listed below:

- Capital investment is required to ensure that automation, observability, and monitoring are in place to effectively manage a decentralised system.
- An incorrect definition of the boundaries of microservices can result in strong interdependencies between services, necessitating their deployment together on each occasion.

- When this situation occurs across multiple services, it can lead to a complex and unwieldy system that is challenging to enhance due to its intricate nature.
- The complexities associated with implementing a services architecture may outweigh its benefits and become more burdensome than advantageous.
- It is crucial to adopt microservices only, when necessary, rather than haphazardly follow the trend simply because it represents the newest and most popular approach.

3.5 Micro-frontends (MFE)

In this section, the intricacies of micro-frontends (MFE) architecture going to be explored. The core concepts, explore popular techniques and approaches, and draw comparisons with microservices is going to be examined. Additionally, various approaches to determine the most suitable one for our project are explored. Lastly, we’ll explore effective testing strategies within the context of MFE architecture.

3.5.1 Definition of Micro-frontends

The concept of micro-frontends can be defined as the division of a business domain into autonomous and independently deliverable components, each owned by a single team. The main idea is to divide the application into smaller pieces where each one is responsible for doing very specific functionalities. In this case, teams can take full ownership and independently evolve their respective codebases (see Figure 2). A suitable micro-frontends strategy could be achieved by different factors of consideration, such as project requirements, organisational structure, and developer expertise. The main principles of underlining micro-frontends which include representing distinct business domains, maintaining an autonomous codebase, enabling independent deployment, and ensuring single-team ownership must be considered. Micro-frontends present numerous advantages in terms of flexibility and adaptability, but making informed decisions requires careful consideration of these factors. Figure 2 shows an overview of the MFEs architecture with end-to-end teams' ownership. (Mezzalira, 2021.)

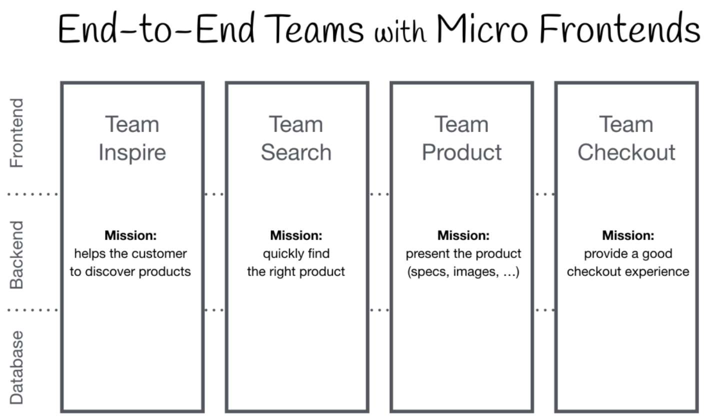

End-to-End Teams with Micro Frontends
Frontend
Backend
Database
Team Inspire
Mission: helps the customer to discover products
Team Search
Mission: quickly find the right product
Team Product
Mission: present the product (specs, images, ...)
Team Checkout
Mission: provide a good checkout experience

Figure 2: Micro-frontends architectural with end-to-end teams' ownership. (Mezzalira, 2021)

3.5.2 Micro-Frontends Composition Strategies

When designing a micro-frontends application, early architectural decisions play a crucial role in shaping the project's trajectory. The fundamental choice revolves around the technical perspective for micro-frontends, presenting two primary options:

- Horizontal Split: This approach involves integrating multiple micro-frontends onto a single page. However, it necessitates effective coordination among different teams, each responsible for a specific section of the user interface.
- Vertical Split: In this scenario, individual teams take charge of distinct business domains like authentication or payment processes, so the principles of Domain-Driven Design (DDD) come into play, ensuring a more cohesive structure.
  (Mezzalira, 2021)

These architectural decisions are shown in Figure 3, which provides a visual representation of the horizontal and vertical splits in micro-frontends applications, highlighting the coordination among teams and the alignment with DDD principles.

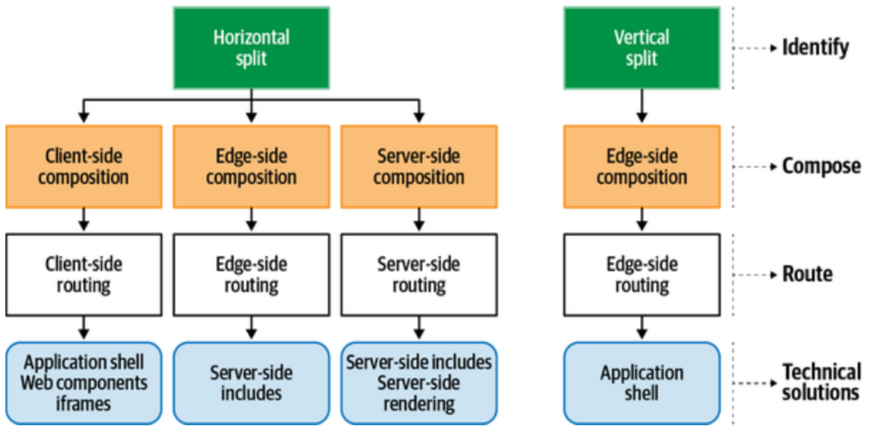


Figure 3: The micro-frontends decision framework. (Mezzalira, 2021)

In a horizontal split setup, a key concern is establishing communication between micro-frontends. One effective method is utilising an event emitter, injected into each micro-frontend. This approach ensures the autonomy of each micro-frontend and facilitates independent deployments. By emitting events, micro-frontends can trigger appropriate reactions in other subscribed micro-frontends. Custom events are also an option, requiring them to bubble up to the window level to be heard by fellow micro-frontends. This entails all micro-frontends listening to all window-level events, either dispatching events directly to the window object or propagating them upwards for communication. (Mezzalira, 2021.)

The vertical split configurations need a clear understanding of information sharing among micro-frontends. Regardless of the split approach, considerations arise for communicating view changes. Techniques like passing variables through query strings or using URLs to transmit concise data (with the new view retrieving supplementary information from the server) are viable. Alternatively, leveraging web storage mechanisms such as session storage for temporary or local storage for more enduring data can facilitate information exchange among micro-frontends. (Mezzalira, 2021.)

Figure 4 provides a visual comparison between horizontal and vertical splitting, showing the differences in communication strategies and architectural layouts.

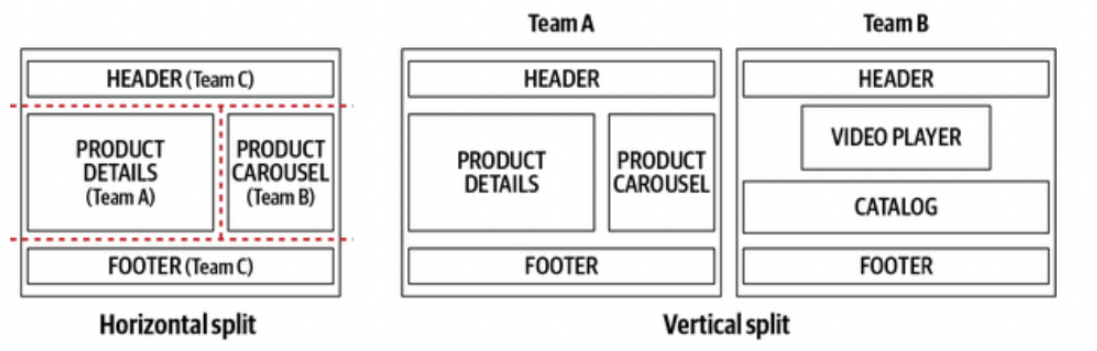
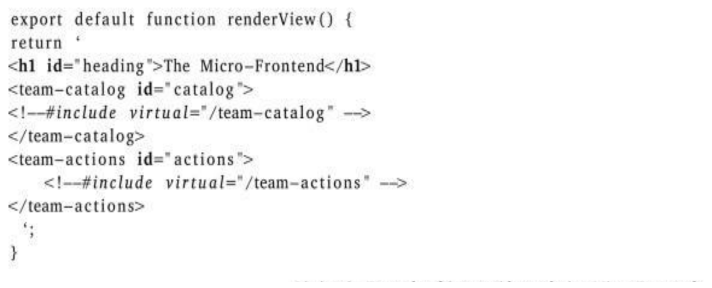


Figure 4: Horizontal versus vertical splitting. (Mezzalira, 2021)

Three different approaches can be used for composing micro-frontends applications.

The first approach is Server-Side Integration (SSI). This strategy employs server-side technology to seamlessly incorporate the HTML content of one micro-frontend into another. This is achieved through dynamic inclusion using a templating engine or similar server-side tools.

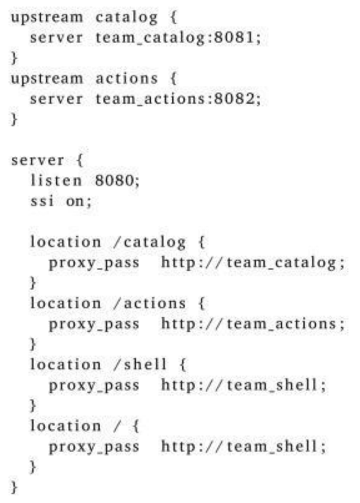
When using server-side to render micro-frontends, the web server takes the place of a command like “<!--#include virtual=”/team-catalog” →” as shown in Listing 1.



```
export default function renderView() {
return `
<h1 id="heading">The Micro-Frontend</h1>
<team-catalog id="catalog">
<!--#include virtual="/team-catalog" -->
</team-catalog>
<team-actions id="actions">
    <!--#include virtual="/team-actions" -->
</team-actions>
  `;
}
```

Listing 1: Example of server-side rendering micro-frontends. (Mezzalira, 2021)

It does this by replacing the instruction with the actual content from the specified URL before delivering the page to the user’s browser. These URLs are often set in the server’s settings, as illustrated in Listing 2. This specific example employs a technique called Server-Side Include (SSI), but there are alternative methods available. For instance, the Edge-Side Includes (ESI) method is a viable choice for achieving server-side rendering in micro-frontends, which is going to be discussed in the next bullet point.



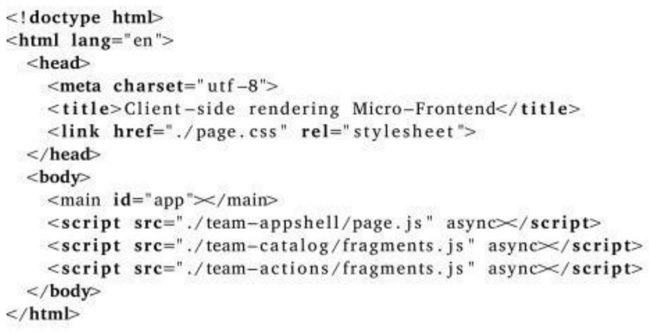
```
upstream catalog {
  server team_catalog:8081;
}
upstream actions {
  server team_actions:8082;
}

server {
  listen 8080;
  ssi on;

  location /catalog {
    proxy_pass  http://team_catalog;
  }
  location /actions {
    proxy_pass  http://team_actions;
  }
  location /shell {
    proxy_pass  http://team_shell;
  }
  location / {
    proxy_pass  http://team_shell;
  }
}
```

Listing 2: Example of proxy server config. (Mezzalira, 2021)

The second approach is Client-Side Integration (CSI). This method harnesses client-side tools like JS to blend the HTML content of one micro-frontend into another. Utilising JS libraries or frameworks, the content from one MFE is dynamically loaded into the context of another.

Another option entails employing a combination of iframes to load numerous micro-frontends. An iframe, functioning as an HTML element, is harnessed on the client-side to embed an additional HTML document within the present one. This technology provides a means of isolating disparate domains, mitigating the possibility of conflicts between codebases maintained by separate teams. (Mezzalira, 2021.)

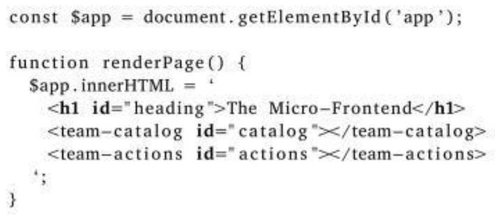
Client-Side rendering micro-frontends enables the division of the web page into micro-frontends. For example, the Listing 3 below demonstrating an HTML file that contains the remotes and host tags for the micro-frontends.



```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Client-side rendering Micro-Frontend</title>
    <link href="./page.css" rel="stylesheet">
  </head>
  <body>
    <main id="app"></main>
    <script src="./team-appshell/page.js" async></script>
    <script src="./team-catalog/fragments.js" async></script>
    <script src="./team-actions/fragments.js" async></script>
  </body>
</html>
```

Listing 3: Example of client-side rendering an HTML file of micro-frontends. (Mezzalira, 2021)

The HTML file need a JS code to render the micro-frontends as shown in the Listing 4 below.



```
const $app = document.getElementById('app');

function renderPage() {
  $app.innerHTML = `
    <h1 id="heading">The Micro-Frontend</h1>
    <team-catalog id="catalog"></team-catalog>
    <team-actions id="actions"></team-actions>
  `;
}
```

Listing 4: Client-side rendering JS code to render micro-frontends. (Mezzalira, 2021)

In the given instance, page.js functions as the application shell, orchestrating the assembly of all loaded fragments beneath it and inserting the resulting content into the main tag. Both the application shell and the accompanying applications operate within the browser environment.

The third approach is Edge-Side Integration (ESI). Here, cutting-edge technologies like content delivery networks (CDNs) or advanced edge caching systems are employed. The HTML content of one micro-frontend is merged into another through specialised markup, such as ESI, which dictates both the content to be merged and the conditions dictating its integration. (Newman, 2019, p. 100; Mezzalira, 2021.)

In Figure 5, an overview of the three micro-frontend composition approaches is provided in a diagram form.

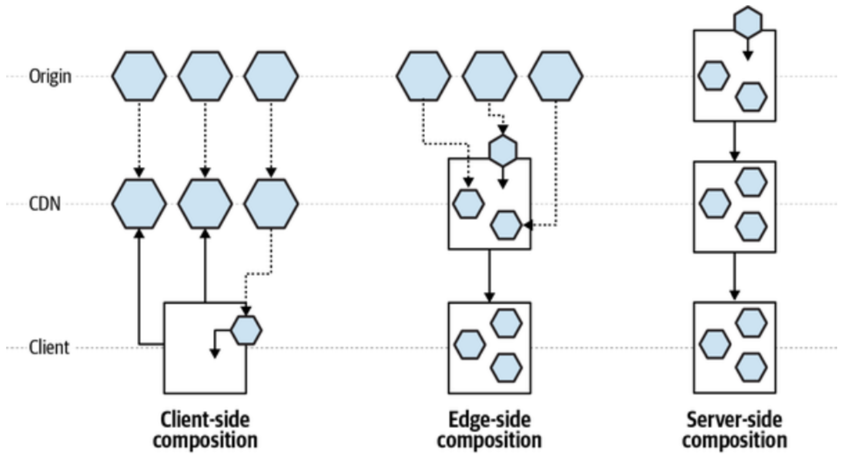


Figure 5: Micro-frontends composition diagram. (Mezzalira, 2021)

3.5.3 Microservices vs Micro-frontends

Microservices and micro-frontends are distinct strategies for organising software development based on business and functional boundaries. Microservices involve breaking down a large application into self-contained, autonomous services that operate independently. Each service corresponds to a specific business capability, communicates via APIs, and can be developed, deployed, and scaled autonomously. (See Figure 6.)

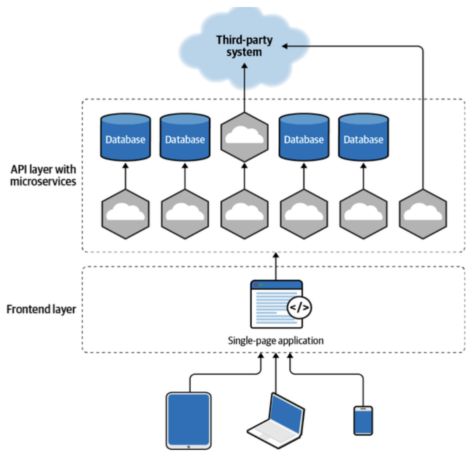


Figure 6: Microservices with monolith frontend. (Mezzalira, 2021)

On the other hand, micro-frontends pertain to web development and entail dividing a web application into discrete, autonomous frontend modules. These modules handle distinct user interface components, are capable of independent development, deployment, and scaling, and interact through APIs or events. (See Figure 7.)

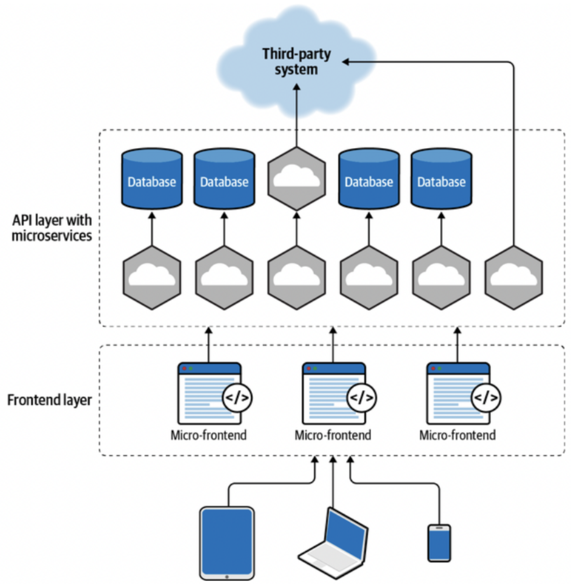


Figure 7: A high level diagram showing the collaboration between Microservices and Micro-frontends within application architecture. (Mezzalira, 2021)

Although both microservices and micro-frontends enable autonomous development, deployment, and scaling, they diverge in focus and domain. Microservices primarily address backend concerns, managing business logic and functionality, whereas micro-frontends concentrate on frontend aspects, governing user interface components. Furthermore, microservices are typically the realm of backend developers, whereas frontend developers usually handle micro-frontends.

On the other hand, Mezzalira (2021) and Newman (2019, p. 28) mention the same microservices principles that can be applied to micro-frontends.
The following list provides the shared principles that can be applied for both architectures:

- Modelled Around Business Domains: "Modelling around business domains is a key principle brought up by domain-driven design (DDD)" (Mezzalira, 2021). It is important to organise software around business domains, utilising shared language and clear boundaries to enhance understanding and teamwork.
- Cultural of Automation: A strong culture of automation is established to support micro-frontends, ensuring smooth integration and rapid feedback loops.
- Hide Implementation Details: Implementation details are hidden and contracts are enforced to enable effective communication between different parts of the application without disrupting other teams.
- Decentralised Governance: Teams are permitted to make domain-specific decisions within established guardrails, promoting expertise and sharing the best practices.
- Deploy Independently: Independent deployment of micro-frontends is enabled, thereby avoiding delays caused by external dependencies and tailoring technical solutions to specific challenges.

- Isolate Failure: Potential network failures or errors are addressed by providing fallback content or the hiding of specific sections of the application, thereby enhancing the user experience.
- Highly Observable: Frontend observability is prioritised through the utilisation of tools such as Sentry and LogRocket to quickly diagnose and resolve issues, aligning with microservices principles.

  (Mezzalira, 2021; Newman 2019, p. 28.)

Figure 8 below providing an overview of the shared principles between microservices and micro-frontends architectures.

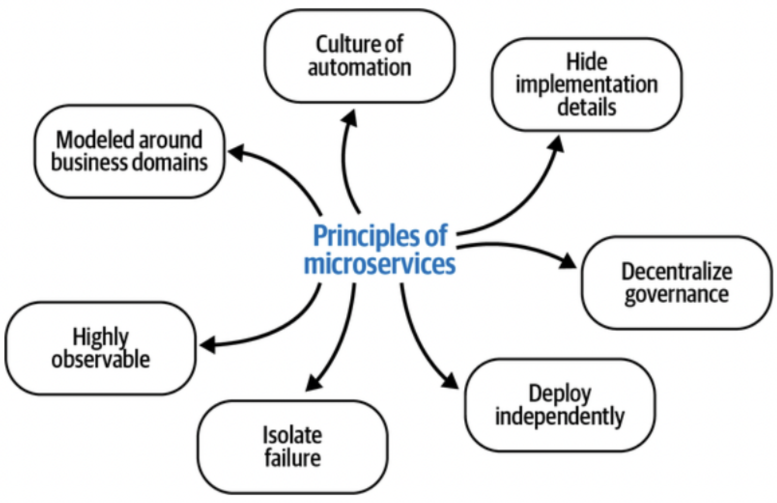


Figure 8: Summary of Micro-frontends Principles. (Mezzalira, 2021)

3.5.4 Micro-frontends Frameworks

What is Single-spa and what are the pros and cons of using it? Single-spa is a JavaScript framework for building micro-frontends. It allows developers to build applications using multiple frameworks, such as React, Angular, Vue.js, and more, all in a single page application (SPA) (Angular, 2024; Vue.js, 2024). Single-spa provides a set of utilities and APIs that enable developers to create a modular architecture for their applications, where each module is a self-contained micro-frontend that can be developed, tested, and deployed independently. (Single-spa, 2024.)

According to Mezzalira (2021), there are some advantages and disadvantages that can be listed as follows:

Advantages:

- Enables building micro-frontends with multiple frameworks within a single page application.
- Provides a set of utilities and APIs for creating a modular architecture for applications.
- Each module is a self-contained micro-frontend that can be developed, tested, and deployed independently.
- Allows team/squad independence, no release trains, code freezes, merge conflicts, and long-lived feature branches.
- Helps avoid QA frustration with testing environments changing/not changing.

(Single-spa, 2024.)

Disadvantages:

- Requires investment in automation, observability, and monitoring to manage a distributed system effectively.
- It causes a risk of incorrectly defining microservices boundary, resulting in strong coupling between services and mandatory deployments.
- It may result in a complex system that becomes hard to extend over time, leading to a “big ball of mud” architecture.
- Is time consuming, and since this topic is quite new, it requires a lot of effort to find the resources.

(Single-spa, 2024.)

3.5.5 In-browser vs Build-time Modules

In the micro-frontend’s world, there are two approaches to get the application on the browser, In-browser JavaScript modules versus build-time modules. In an in-browser module the browser handles imports and exports directly, without the build tool getting in the way. On the flip side, build-time modules come from the node_modules, get compiled before reaching the browser, and that is the usual drill.

Now, here's the interesting part. If some dependencies want to be kept untouched during the build process, they can still come straight from the browser. It is possible to use Webpack externals or rollup externals. (Single-spa, 2024.)

3.5.5.1 SystemJS as In-browser Technique

As it appears in the Single-spa library recommendation, SystemJS is the best choice if the decision is within browser modules (Single-spa, 2024).

SystemJS is a tool that mimics import maps and in-browser modules. However, it is not a full-on polyfill for import maps because of some JS limitations in handling those bare import specifiers to URLs.

Since SystemJS is more like a polyfill, it needs to be compiled in the applications into a specific format called System.register, not the usual ESM format. After it is compiled, it will fully emulate in-browser modules in environments that might not be working in the best way with modules or import maps.

To make the code work with System.register, the Webpack is going to be configured to the target system as the output.library target or set the rollup's format to the system. This is almost equal to giving the code a different language to chat with the browser.

However, some popular technologies like Angular.js do not have System.register versions of their libraries. Nevertheless, these technologies can be obtained from the esm-bundle project. Although. SystemJS can also load them using some methods like global loading or the AMD and named-exports extras. (Single-spa, 2024.)

3.5.5.2 Module Federation as Build-time Technique

Module Federation is a very useful tool used by the Webpack (Webpack, 2024) to share modules when building websites. Each little section of the site (micro-frontends) bundles up all its modules and dependencies, even the modules are shared with other sections. Each shared module or dependency is envisioned to have multiple copies, with one designated for each section.

When opening the site in the browser, it grabs the first copy of the shared modules or dependencies. The other sections do not need to download it again; they just reuse what's already there. It needs to be brought to the playground once only.

It is good to keep in mind that Module Federation is quite new (as of now), and it needs Webpack version 5 or higher. It is still a learning and growing process.

Is it possible to mix both Single-spa and MFE? The answer is that Single-spa helps to organise how different parts of the site are connected. Module-Federation, on the other hand, makes it run faster and work well.

When Module Federation is used, a decision needs to be made on how to load the micro-frontends. The Single-spa experts suggest using SystemJS and import maps. It is like a special way of bringing in those sections. A global variables and <script> elements are also another available option to load the micro-frontends.

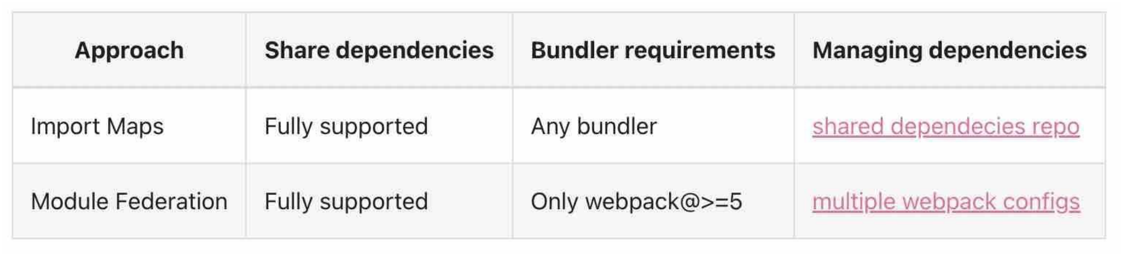
When it comes to sharing outside third-party dependencies, it is recommended to pick either import maps or module federation in Single-spa. It is not recommended to mix them up. The single spa team prefers to work with import maps, but they work with module federation as well. (Single-spa, 2024.)

3.5.5.3 Shared Dependencies

To keep the web application fast, it is recommended to load big libraries like React, Vue, and Angular only once. Although small libraries can make upgrades faster with duplication in each part of the site, it is still not recommended to make everything shared. For the big libraries such as React.js and Moment.js, it is recommended to consider sharing. There are two ways to share:

1. In-browser modules with import maps (SystemJS)
2. Build-time modules with Module-Federation

It is possible to pick one or both. Right now, the Single-spa team suggests sticking to import maps, but module federation works too. More details about comparison can be seen in Figure 9 below. (Single-spa, 2024.)



Approach	Share dependencies	Bundler requirements	Managing dependencies
Import Maps	Fully supported	Any bundler	shared dependecies repo
Module Federation	Fully supported	Only webpack@>=5	multiple webpack configs

Figure 9: Simple compression of the two approaches. (Single-spa, 2024)

The real challenge in micro-frontends is to break down a big web application into smaller, more manageable pieces. It is like having a favourite pizza but sliced into bite-sized portions – easier to handle and share.

Micro-Frontends are all about making the web development life simpler. Instead of dealing with one massive monolith, it can be broken down into smaller, self-contained parts. Each part of the website could be used independently, to function without relying too much on the other parts. On the other hand, breaking down a monolith application can be a complicated process.

There are a few methods by which micro-frontends can be implemented, with the selection of the appropriate tool being crucial for the successful execution. However, a decision needs to be made first to choose which of the two approaches is more fit to the goal.

3.5.6.1 Build-Time module Integration

With Build-Time Integration, the necessary resources for each part of the website are prepared during the building process. The process is efficient, organised, resulting in a seamlessly integrated web application. Achieve this goal requires the employment of a bundler. Various options, such as Gulp, Rollup, and Webpack bundler, are available. (Single-spa, 2024.)

Manfred Steyer introduced a new tool called “angular-architects/module-federation” (Steyer, 2020) to address micro-frontends issues. This tool works well only with Webpack 5 or higher. The tool is easy to use and gives the ability and the flexibility to choose which component to expose. The tool provides a perfect solution for the Angular.js legacy application. It simplifies integration, especially with a straightforward example provided by Manfred Steyer. The provided example explains how to get multi-repo applications (remotes) working together in one app-shell (host) without the need for Single-spa library and its complicity. Instead using Angular-CLI. (Steyer, 2020.)

3.5.6.2 In-Browser Module Integration

Imagine the web application as a canvas, and each micro-frontend is a stroke of paint. Import Maps are like a browser rulebook that helps connect the dots between what is called an 'import specifier' (basically, what module to load) and a URL. However, it should be noted that Import Maps are not universally supported by all browsers. To bridge the gap, SystemJS is brought in, serving as a supporting mechanism to ensure the loading of import maps in the browser.

With the power of SystemJS, the browser takes on the role of an artist, putting everything together in real-time. It is like a masterpiece unfolding as the users navigate through the site. Import and export are resolved within the browser. Some of the dependences are left to the browser, which should take care of building them.

SystemJS serves as a type of polyfill rather than the primary solution. When preparing applications, opting for the System.regester format instead of the usual ESM format is preferable. This approach will mimic in-browser modules, particuarly in scenarios where modules or import maps might not be the optimal choice yet.

To ensure smooth functionality of the System.register format mentioned earlier, adjustments need to be made to Webpack by configuring the output in 'system' mode or setting up rollup in 'system' format. This ensures that the intended polyfill approach mimics in-browser modules effectively, particularly in scenarios where modules or import maps might not be the optimal choice yet.

3.6 Testing Strategy in MFE Architectures

Testing MFE applications can be a challenging process. There are several testing strategies for testing MFE applications. Ensuring robust testing is crucial for maintaining code quality and seamless user experiences. Here are the key strategies:

1. **Unit Testing for Micro-frontends Components.**
   First, writing unit tests for each individual component within the micro-frontend. These tests focus on the smallest units of functionality, ensuring that each component behaves correctly in isolation.

2. **Integration Testing for Communication Verification.**
   After unit tests, integration testing and verify communication between different micro-frontends is the next steps. That will ensure the micro-frontends can collaborate effectively and the micro-frontends data can be exchanged as expected.

3. **End-to-End (E2E) Testing for Simulating User Interactions.**
   E2E testing simulates real user interactions with the application because it covers scenarios where multiple micro-frontends work together. E2E is also

used to validate the entire flow, from user input to backend communication. Using E2E testing gives the possibility to do cross-browser checks which is essential to ensure compatibility across different browsers.

4. **Performance Testing and Isolating Failures.**
   Testing and assessing the performance of the micro-frontends is needed to Identify the bottlenecks, slow-loading components, or resource-intensive processes. It is an important testing strategy to isolate failures and optimize performance to enhance user experience.

5. **Testing API Contracts for Consistency.**
   MFEs often communicate with APIs so testing API contracts to ensure consistency and prevent unexpected changes is a very important process to ensure application consistency. Contract testing helps maintain stability and compatibility.

6. **Addressing Cross-Component Interaction Challenges.**
   Micro-frontends involve multiple smaller units working together (LambdaTest, 2023). The collaboration between the micro-frontends poses some challenges such as data sharing, UI integration, and synchronisation.

To overcome the challenges, strategies such as using mock data, UI testing tools, wait mechanisms, and conducting API verification are recommended. Moreover, parallel test execution, realistic environments, and seamless integration into development pipelines emerge as an essential factors.

3.7 Module Federation and Webpack 5

Module Federation was introduced in Webpack 5. It is the newest straightforward solution that allows loading code from a compiled, deployed, and separated application. Module Federation is the best approach so far for implementing shell-based architecture. (Jackson, 2023.)

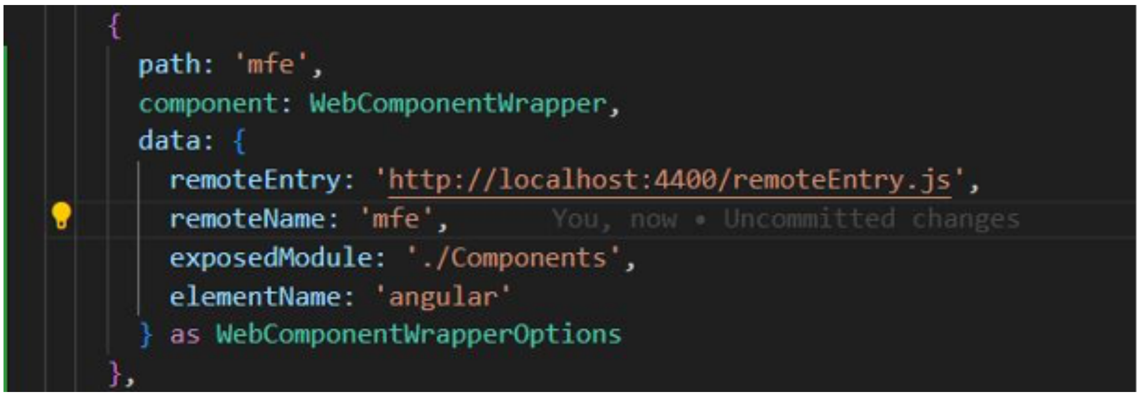
Module Federation is an advanced feature in Webpack, enables JavaScript applications to dynamically load code from other applications. (Jackson, 2023.)

The key advantages of using Webpack 5 and Module Federation can be highlighted as the following points (Steyer, 2020):

- Module Federation enables referencing parts of other applications dynamically, even if they are not known at compile time.
- Module Federation is particularly useful for MFEs that are compiled separately.
- One of the key advantages is the ability for individual program parts to share libraries.
- Preventing duplication in individual bundles.
- Module Federation allows to share libraries between the host and the remotes MFEs.
- Module Federation comes with several strategies for dealing with version mismatches.

3.7.1 Types of MFE In Module-Federation

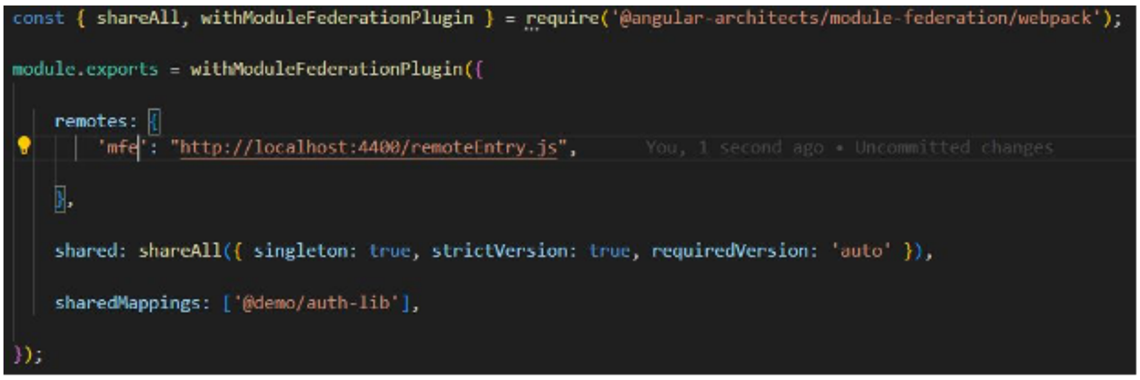
There are two types of applications in Module-Federation, *host* and *remote*. The host application in this case is app-shell. It is hosting all the MFEs by using the router to lazy load every MFE. Listing 5 shows how @angular-architect-module-federation-tools is warping one MFE and how it is given back the MFE as web component with its options:



```
{
  path: 'mfe',
  component: WebComponentWrapper,
  data: {
    remoteEntry: 'http://localhost:4400/remoteEntry.js',
    remoteName: 'mfe',        You, now • Uncommitted changes
    exposedModule: './Components',
    elementName: 'angular'
  } as WebComponentWrapperOptions
},
```

Listing 5: Routing management of MFE in the host application. (Steyer, 2020)

The paths in the *webpack.config.js needs to be configured* to start with the MFE that are directing to another project, as seen in Listing 6.



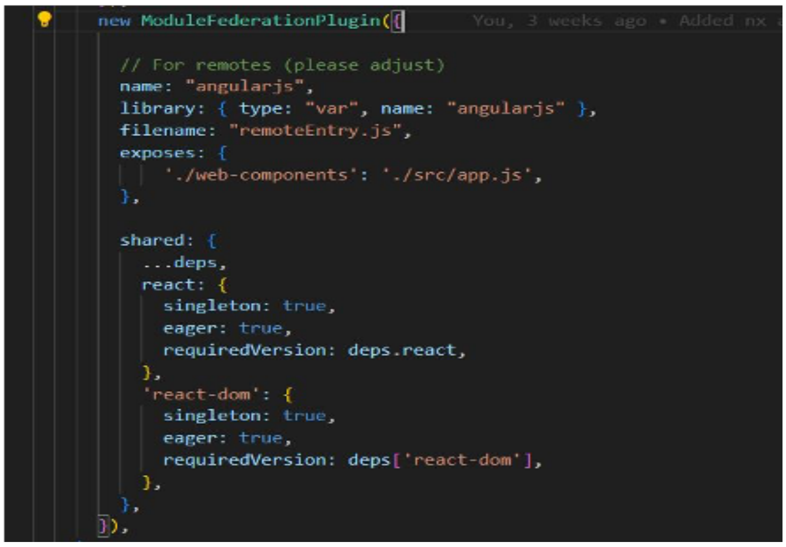
```
const { shareAll, withModuleFederationPlugin } = require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

  remotes: {
    'mfe': "http://localhost:4400/remoteEntry.js",       You, 1 second ago • Uncommitted changes

  },

  shared: shareAll({ singleton: true, strictVersion: true, requiredVersion: 'auto' }),

  sharedMappings: ['@demo/auth-lib'],

});
```

Listing 6: Webpack config of the host application. (Steyer, 2020)

In Listing 6 above, the remotes map the path **MFE** to a totally separated compiled micro-frontend. This lightweight file is generated when building the remote by the Webpack as seen in Listing 7 (Steyer, 2020).

During the runtime the Webpack is taking care of loading the generated lightweight file to get all the information required to interact with the defined MFE by using the Module Federation plugin, as seen in Listing 7 (Steyer, 2020).

The other type of application in Model-Federation is the remote. Every MFE in our project is presenting a remote (Steyer, 2020).



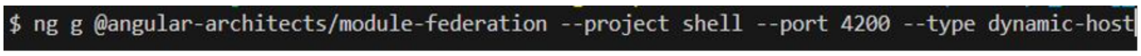
```
new ModuleFederationPlugin({        You, 3 weeks ago • Added nx

  // For remotes (please adjust)
  name: "angularjs",
  library: { type: "var", name: "angularjs" },
  filename: "remoteEntry.js",
  exposes: {
      './web-components': './src/app.js',
  },

  shared: {
    ...deps,
    react: {
      singleton: true,
      eager: true,
      requiredVersion: deps.react,
    },
    'react-dom': {
      singleton: true,
      eager: true,
      requiredVersion: deps['react-dom'],
    },
  },
}),
```

Listing 7: Module Federation plugin configuration of an MFE application. (Steyer, 2020)

In production, a more dynamic approach is needed rather than using the specified URL. A specified URL was chosen because it is more convenient for development.

3.7.2 Generating a Manifest for Dynamic Module Federation with Angular

With Angular architects and Module Federation, a dynamic host can be generated to take the key data representing the MFE from a JSON file at runtime. The following is the command needed for installing it:

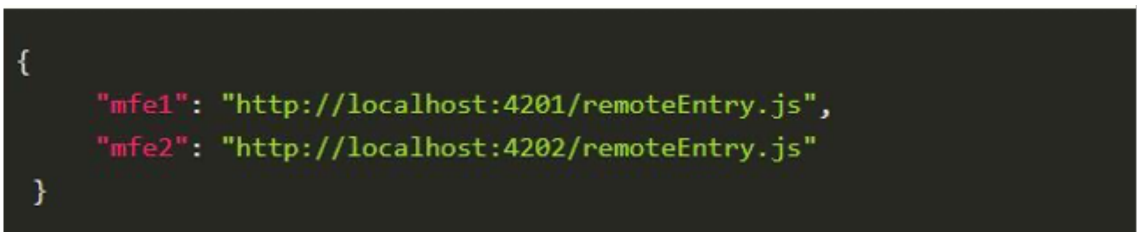


```
$ ng g @angular-architects/module-federation --project shell --port 4200 --type dynamic-host
```

This command generates a skeleton of three files:

- a Webpack configuration
- the manifest
- some code in the main.ts file for loading the manifest

After the skeleton being generated, the manifest file will be location in the shell application. It contains the paths for the MFEs, as seen in the Listing 8.



```
{
    "mfe1": "http://localhost:4201/remoteEntry.js",
    "mfe2": "http://localhost:4202/remoteEntry.js"
 }
```

Listing 8: An example of manifest file structure for MFE application. (Steyer, 2020)

The focus of this thesis leans towards running it in the local environment; thus, the details of the deployment and manifesting are not covered. However, more details are elaborated in the Steyer’s blog under the title “Dynamic Module Federation” (Steyer, 2020).

3.8 Using Nx for Monorepo Management

Nx is a robust open-source build system designed to streamline the management of monorepo projects, and it is particularly beneficial for micro-frontends architectures. It offers an array of tools and methodologies geared towards augmenting developer efficiency, optimising Continuous Integration (CI) performance, and upholding code quality standards.

Nx provides many great benefits. Here are some of the core features:

1. **Efficient Task Execution:** Nx executes tasks concurrently while intelligently sequencing them based on their interdependencies. This approach enhances build speed and overall project efficiency.

2. **Local and Remote Caching:** Nx incorporates both local and remote caching mechanisms to mitigate unnecessary task re-execution. Caching intermediate artifacts, it significantly reduces development cycle times, thereby maximising productivity.

3. **Automated Dependency Management:** Leveraging Nx plugins empowers developers with advanced functionalities such as automated code generation and dependency management. These tools streamline the process of upgrading codebases and dependencies, ensuring projects stay current with the latest standards and libraries.

4. **Customizability and Extensibility:** Nx offers extensive customisation options, allowing developers to tailor the build system to suit specific project requirements. Whether creating bespoke plugins or fine-tuning existing features, Nx provides flexibility to adapt to diverse development environments. Additionally, developers have the option to contribute their custom plugins to the community, fostering collaboration and knowledge sharing.

In addition, Nx serves as a great solution for monorepo management, offering a smoother development experience characterised by accelerated task execution, efficient caching mechanisms, automated dependency management, and extensive customisation capabilities. (Nx, 2024.)

3.9 Comparison between Single-spa and Module Federation

Both approaches offer valuable options for micro-frontends (MFE) implementation, but the choice between them depends on the specific requirements and nature of the project. The following Table provides a comprehensive comparison of these approaches, aiding in the selection of the most suitable option for the project.

Table 1: A comparison between Single-spa and Module-Federation. (Tiwari, 2023; Mezzalira, 2021; Single-spa, 2024)

Feature	Single-spa	Module-Federation

Flexibility	- Supports different JavaScript frameworks. - Allows mixing and matching of micro-frontends with different frameworks.	- Native Webpack support. - Leverages Webpack 5 or later.
Framework agnostic	- Supports multiple frameworks within the same application. - Allows independent development of micro-frontends with chosen frameworks.	- Simplified integration with Webpack. - Reduction in custom code for setup.
Dynamic loading	- Supports lazy loading for efficient resource use. - Dynamically loads micro-frontends as needed, reducing initial load time.	- Enables dynamic loading for on-the-fly updates. - Facilitates dynamic loading with less manual configuration.
Custom configuration	- Requires custom configuration for managing dependencies and routing. - Custom API for orchestrating communication and data sharing.	- Provides a clear structure for module sharing and consumption. - Granular control over shared dependencies.
Community support	- Active community with ongoing development and support.	- Growing community with increasing adoption.
Scalability	- Suited for various frameworks and mixed environments.	- Well-suited for large-scale applications and enterprises.

Complexity	- May require more custom configuration.	- Streamlined integration with Webpack.
Learning Curve	- Potential steeper learning curve due to flexibility.	- Aligns closely with standard Webpack practices.

After looking at Table 1 above, there are still a few things to consider before choosing an approach to help make the best suitable choice for the project. Table 2 below is a simple summarisation of some key considerations. (Tiwari, 2023; Mezzalira, 2021; Single-spa, 2024.)

Table 2: A summarise of some key considerations to help choosing approch for implementing micro-frontends.

Consideration	Single-spa	Module-Federation
Use Case	- Flexibility in supporting various frameworks within the same application. - Allows mixing and matching of micro-frontends with different frameworks.	- Native Webpack solution, especially beneficial for those heavily invested in Webpack.
Community and ecosystem	- Established a community with diverse plugins and extensions.	- Benefits from the broader Webpack ecosystem as it is part of Webpack.
Complexity	- May require more custom configuration for managing dependencies and routing.	- Provides a more streamlined solution, being integrated with Webpack.
Learning curve	- Potential steeper learning curve due to flexibility and customisation options.	- Aligns more closely with standard Webpack practices, potentially reducing the learning curve.

In summary, choosing between Single-spa and Module Federation for implementing micro-frontends involves careful consideration of the various factors. While Single-spa offers flexibility in supporting different frameworks within the same application and has an established community, Module Federation provides a native Webpack solution and benefits from the broader Webpack ecosystem. However, Single-spa may require more custom configuration and could have a steeper learning curve, whereas Module Federation offers a more streamlined solution and aligns closely with standard Webpack practices. Ultimately, the choice depends on the specific project requirements and the team's familiarity with the respective technologies. (Mezzalira, 2021; Steyer, 2020; Single-spa, 2024.)

3.10 Art of Choosing: Why Choose Module-Federation?

Choosing the approach depends on the project, the team, and the vibe going for it. Each method has its benefits and downsides, and it is about finding the approach matching the development philosophy.

It is good to remember that MFEs are all about flexibility and making the life of a developer smoother. Either choice works but choosing the right approach depends on which one is more suitable and fit to the need. The in-browser approach with SystemJS produces several issues and the support for Angular.js together with Webpack 5 is not quite enough at least for this case now. On the other hand, Module Federation is much simpler with a live example showing exactly how it works with Angular.js and React.js both in one framework. It is also providing the flexibility to avoid breaking down the whole project and exposing one component only or exposing the whole application as a web-component to the host application. This sounds ideal to what is working for the specified project as it is going to be discuss further next.

4 Designing the Micro-frontends

The aim of this chapter is to expand the theoretical part of the study and elaborate on microservices design.

4.1 Existing Solution

The web application Train-CM, which is developed using Angular.js, acts like a parent for two other web applications using the iframe approach. This application navigation bar is essential for seamless navigation across different views within the application. The navigation bar governs the content displayed within the application's main section.

Within the user interface, one of the distinct views, namely the Dashboard, represents an autonomous application built with Angular.js often referred to as a Single Page Application (SPA). The other is the Track Condition tab which takes the user to the outer application which is built with Python through a link the iframe uses. They operate independently, each with its backend infrastructure.

Presently, when a user clicks on the Track Condition tab, a separate login is required. This is due to the absence of shared authentication mechanisms between these two views, as they function as distinct applications.

The navigation bar is equipped with a logo and a user account button. Clicking on the user account button expands its functionality, allowing users to access account settings and log out from the application.

The other application called the Track-CM, which is built with React.js, is an independent application that has a navigation bar to switch between tabs. There are three tabs and a logo in the navigation bar; each tab represents a page. The application frontend authentication service is disabled for now and the application does not have a bundler like Webpack for example, but it is working and fully functional.

4.2 Integration Plan

The aim of this chapter is to describe integrating a legacy project which was built using the *Angular.js* framework into *micro-frontends architecture (MFE)*. Once it is integrated, different technologies or frameworks, for example React.js going to be used with the legacy code to develop more features. The Nx monorepo management library is going to be used to simplify the process and get all the features this library provides.

The scope of this thesis focuses on implementing micro-frontends architecture at a view level as EKE-Electronics company requires. Specifically, the project aims to demonstrate the integration of two pages from different applications onto a single page. This approach demonstrates how both the Angular.js and React frameworks can seamlessly collaborate within the micro-frontends architecture. The main idea is to display the Detections page from the Track-CM application implemented with React.js and the Dashboard page from the Frontend application implemented with Angular.js in one view. Hence, the original web applications are two separate monolith applications with multiple functions. In addition, an app-shell, which behaves as a container, to host all the MFE (*remotes*) is part of the implementation plan. The app-shell will contain a navigation bar where every web application is accessible to the user on a simple tab. The user can switch between tabs to see their content independently.

Every MFE will be hosted by the app-shell which is built with Angular 14. The app-shell will manage the routing between the MFEs with some simple style for the app-shell itself. The implementation and integration process will be documented with code snippets.

As a side note, the global MFE style and authentication is a complex topic on its own and out of the scope of this thesis. The main aim is to demonstrate the migration process as a root map for future development under the umbrella of MFE architecture.

4.3 Characteristics of Original Applications before Migration

In this project, two monolith applications will be migrated. The first one is the original frontend *Angular.js* application that has been developed and maintained for several years by multiple teams, and more specifically the VR profile, which consists of two views and multiple functionalities. This application is built to allow the user to monitor the condition of the trains.

The application contains multiple pages and components built with *Angular.js*, which are discussed in the next section.

4.3.1 Login Page

The Login Page is the first page the user will see when the application starts. It has a button for directing the user to an authentication service page where they must provide a username and a password. The authentication service uses the *Load Balancer* provided by *Amazon* and held by the backend service. Once the user has entered their username and password, they will be directed to the Dashboard Page as shown in Figure 10.

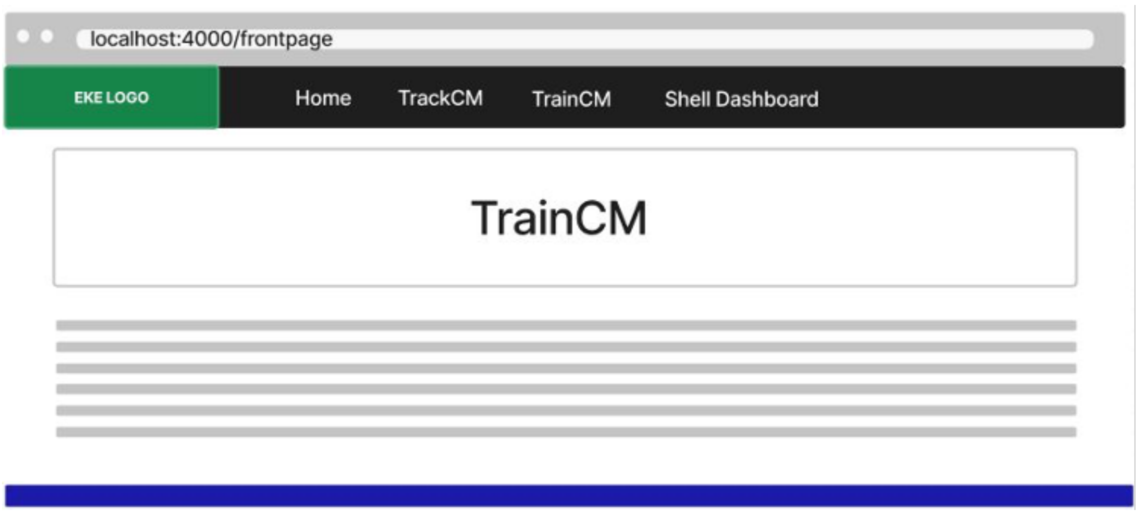


Figure 10: Frontend angular.js application Login Page. (Figma, 2024)

4.3.2 Dashboard Page

After the authentication, the user will be redirected by the service to the Dashboard Page. The page has a navigation bar where the user can see two tabs. The first tab is the initial tab, which is called Dashboard and contains two widgets with multiple functionalities. The second tab, named Track-Condition, will direct the user to a new sign-in page through the iframe approach, allowing access to an external application developed with Python.

In this thesis, the Dashboard view from this application going to be taken and migrated to the MFE architecture. See Figure 11 below for an overview of the Dashboard Page design of the Angular.js application.

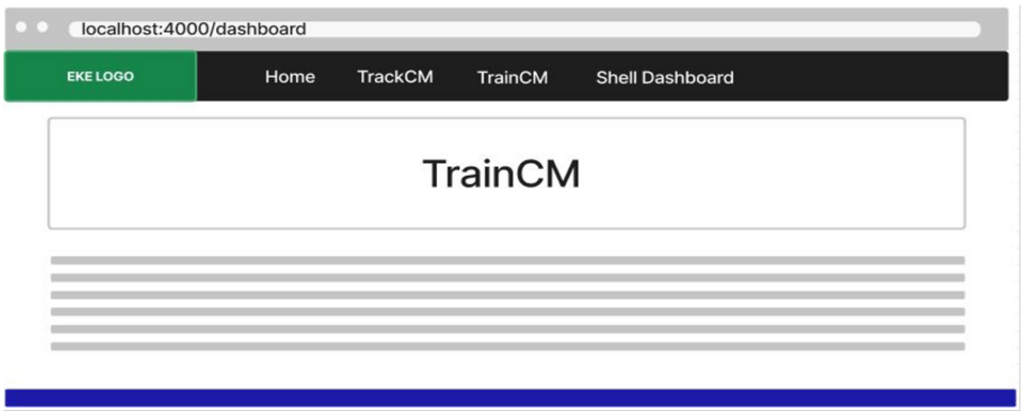


Figure 11: Frontend Angular.js application Dashboard Page. (Figma, 2024)

The second one is the application called Track-CM, which is used for monitoring the condition of the train tracks. It is implemented with React.js and contains a navigation bar with three tabs, and each one represents a separate page. One page only going to be taken, which is the Track Condition Detections Page out of the three to integrate it into the MFE architecture. The authentication service for this app is temporarily disabled so the user will be directed to the dashboard as the initial page when the application is up and running. The Track Condition Detections Page consists of multiple widgets such as map, chart, and table so the user can see the information and the location of the issue in the track. The goal is to take the Detections page and turn it into an MFE application. To have a clearer view of how the TrackCM application currently looks, see Figure 12 below.

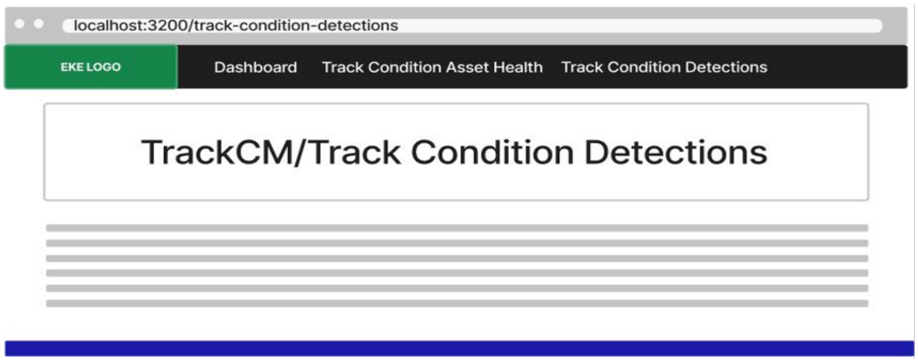

localhost:3200/track-condition-detections
EKE LOGO
Dashboard
Track Condition Asset Health
Track Condition Detections
TrackCM/Track Condition Detections

Figure 12: The T-C Detections page in the TrackCM application. (Figma, 2024)

As a side note, the local development going to be the main environment since deployment can be quite a complex topic, so it is outside the scope of this thesis.

4.4 Micro-frontends Application Requirements

One of the most recent working solutions with MFE is the Module Federation architecture, as mentioned by Manfred Steyer in his article, where a complete example is provided for hosting multi frameworks within one application shell (Steyer, 2020). To use this solution, a couple of requirements had to be fulfilled. The most important one is Webpack 5 bundler, without which Module Federation cannot be used.

In this MFE project scenario, the project is divided into three applications, assuming three different teams would work separately.

- Team Shell will work on the app-shell (the host) where it will host the other applications (the remotes) and implement the navigation bar.
- Team TrainCM will work on the Frontend Angular.js legacy code, which consists of the Dashboard view, and turn it into an MFE remote type application.

- Team TrackCM will work on the React.js application, which consists of the Detection view and integrating Webpack 5 into it and make the necessary changes to turn it into an/the MFE remote type application.

4.5 MFE Application Overlook Architecture

First, the monolith application TrainCM, developed with Angular.js and already integrated into Webpack 5 bundler, is discussed. The objective in this case is to modify the Webpack configuration file and implement the necessary changes to transform the application into a MFE application, utilising the Module Federation architecture for this purpose.

The second monolith application Track-CM, which is built with React.js, has no bundler. In this case, integration of the Webpack 5 bundler is required first and then turn it to an MFE application using the same architecture.

In a perfect scenario, every page should be separated into an MFE handled by one team with all its functionalities, but since there are not many resources available now, one team will take care of it all.

In this implementation, every monolith application going to be turned into MFE individually after finishing the shell app first. The shell app is the host for all the remotes (MFEs). In addition, it is expected that each remotely integrated MFE application functions independently from the host application as it did previously. Figure 13 shows an overlooked architectural diagram of the MFEs applications.

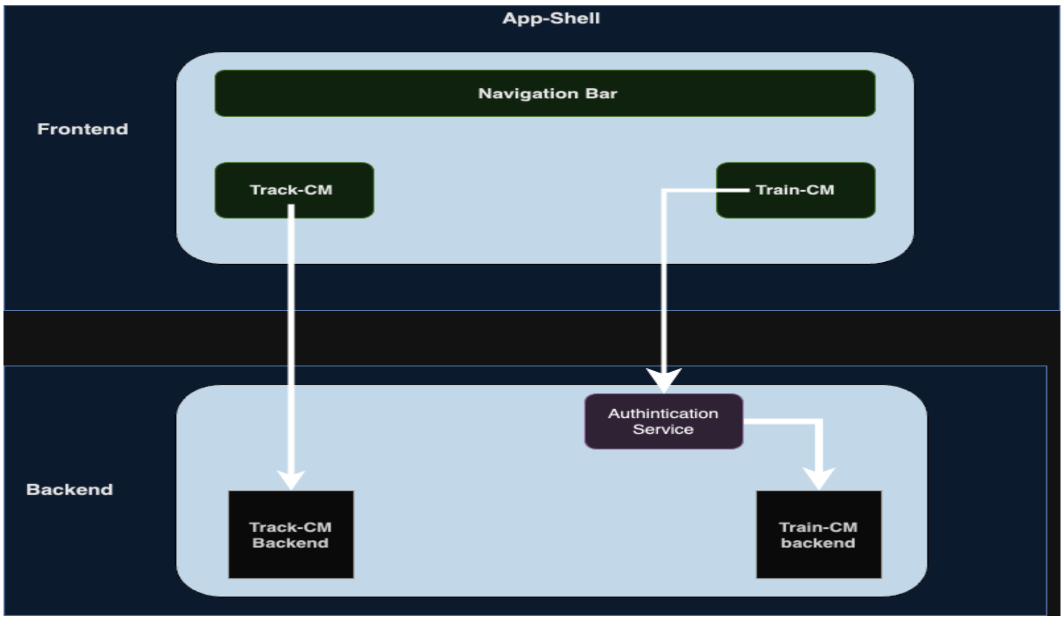


Figure 13: A diagram showing the expected architectural overlook of the MFEs applications.

4.6 Testing Strategy

Testing MFE applications can be a complicated process and time consuming To streamline testing efforts, focus was given on one strategy initially.

4.6.1 End-to-End Testing

End-to-end testing evaluates the entire application workflow by simulating real-world user scenarios and replicating live data. This ensures that all components of the application function together as expected, validating overall system performance and reliability. (LambdaTest, 2023.)

In other words, E2E testing mimics how the software operates in real life, running common user scenarios and identifying any errors or malfunctions throughout the entire application flow. It provides insights into how the application functions from the end user's perspective, ensuring that it delivers the expected output as a unified entity.

4.6.2 Why Choose E2E Testing for MFEs?

To gain insight into the importance of E2E testing for MFEs, the following reasons are considered:

- Holistic View: E2E testing provides a complete view of the application.
- User Journey Validation: It validates the entire user journey, including interactions between micro-frontends.
- Early Detection: E2E testing allows for the early identification of integration issues, UI glitches, and unexpected behaviour during the development process.
- Seamless Experience: E2E testing ensures that the micro-frontends work seamlessly together, providing a consistent experience for users across different components and browsers.

4.6.3 Choosing E2E Testing Tool and Technology

Given the complexity of modern software systems with multiple components (UI, API layers, databases, third-party integrations), and comprehensive E2E testing, it is crucial to validate the behaviour of the entire system from an end user's perspective. For that, Python and Selenium library for testing going to be the choice. The reasons are described below:

Why Python:

- Readability and Expressiveness: Python's clean syntax and readability make it an excellent choice for writing test scripts. It allows testers to focus on logic rather than boilerplate code.
- Rich Ecosystem: Python boasts a vast ecosystem of libraries and frameworks, making it versatile for various tasks. In this case, Selenium integrates seamlessly with Python.
- Cross-Platform Compatibility: Python runs on multiple platforms, ensuring consistent test execution across different environments.

- Community Support: Python has an active community, which means abundant resources, tutorials, and support.

Why Selenium:

- Cross-Browser Testing: Selenium allows testing the MFE across different browsers (Chrome, Firefox, Edge, etc.). This is crucial because users access web applications using various browsers.
- Interaction with Web Elements: Selenium provides methods to interact with web elements (such as clicking buttons and filling forms). This is essential for simulating user actions.
- Headless Testing: Selenium supports headless browser testing, which is useful for running tests without a visible browser window.
- Integration with Other Tools: Selenium can be integrated with other tools like TestNG, JUnit, and Cucumber for test reporting and management.
- Widely Adopted: Selenium is widely used in the industry, ensuring a wealth of documentation and community knowledge.
  (Katalon, 2018; Testim, 2021; Barak, 2018.)

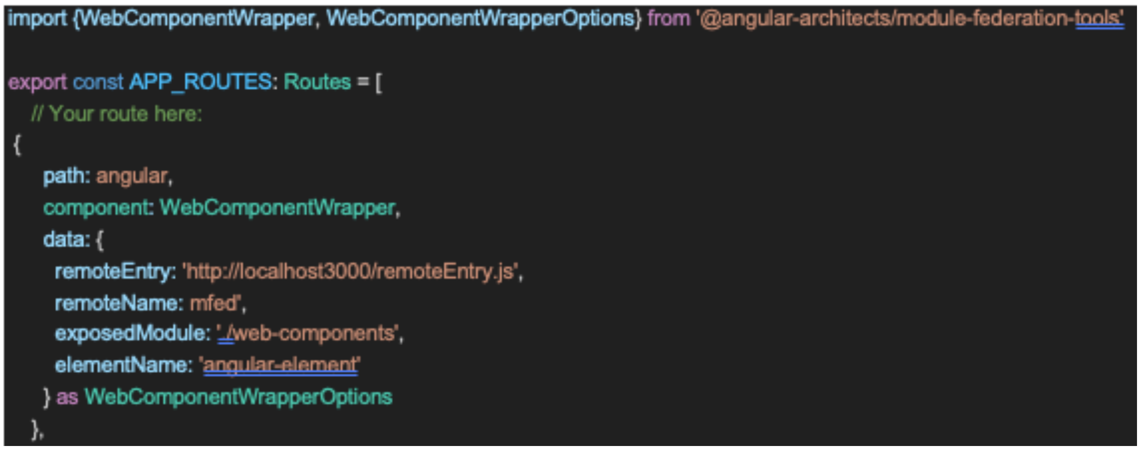
5 Implementation

Implementing multiple MFEs can be quite a complex one. To avoid this, the implementation process is going to be divided into three applications. The main application is the app-shell (the host) with two MFEs (the remotes).

5.1 Implementation of App-Shell and Navigation Bar (EKE host)

A shell is a host application for all the remotes where all will be sharing the same navbar or as Steyer said, “[t]he **host** is a JavaScript application that **loads a remote when needed**” (Steyer, 2020). To define the app shell, a module resource going to be written so that it is possible to share. In this case, it should render a header (navbar) and pass content through to its body.

The app-shell uses the lazy route to load every MFE when needed. It uses two main methods provided by calling angular-architects/module-federation-tools for that purpose, as it seen in Listing 9.



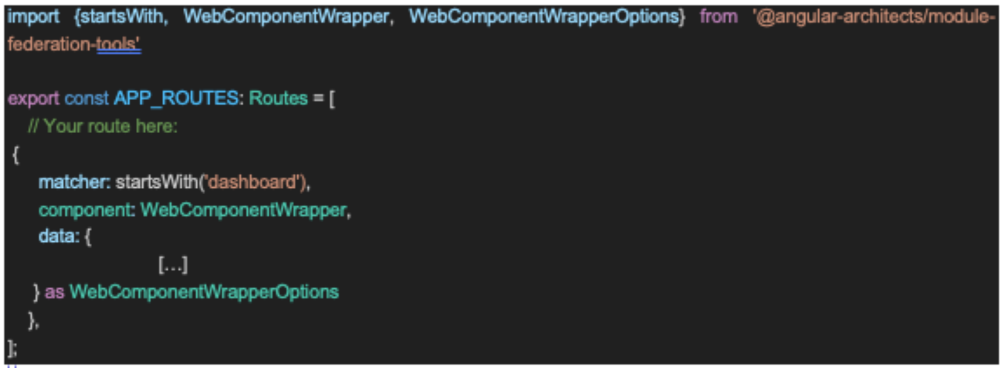
```
import {WebComponentWrapper, WebComponentWrapperOptions} from '@angular-architects/module-federation-tools'

export const APP_ROUTES: Routes = [
   // Your route here:
 {
     path: angular,
     component: WebComponentWrapper,
     data: {
       remoteEntry: 'http://localhost3000/remoteEntry.js',
       remoteName: mfed',
       exposedModule: './web-components',
       elementName: 'angular-element'
     } as WebComponentWrapperOptions
   },
```

Listing 9: The MFE loader script.

Listing 9 above shows the sample of script needed to load the MFE into the app-shell. The scripts use the two methods to give the MFE application a path to be found using the WebComponentWrapper method. The other block contains the remoteEntry, remoteName, exposedModule, and elementName converted using the WebComponentWrapperOptions method as a type.

Another property, called *type,* that is not mentioned in the Listing 9 above, is utilised when the compiled MFE is with Angular version 13 or higher. In this case, the property *type* needs to be set to *module, for example:' type: 'module' '*. The differences arise simply because Module Federation is handled differently in Angular CLI 13 and above.

Another tool provided by the *@angular-architects/module-federation-tools* used to append further segments to the URL is called *startsWith* matcher. It is used when MFE brings its own router by telling the app-shell using a matcher key word in the router (Steyer, 2020), as shown in Listing 10:

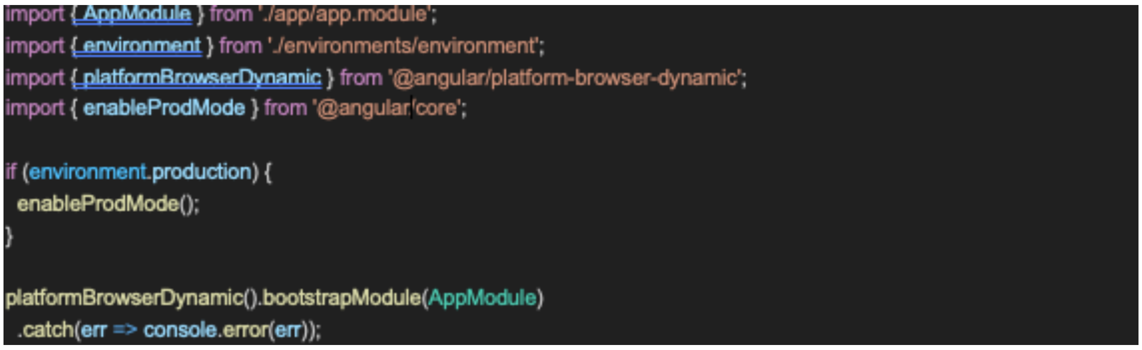


```
import {startsWith, WebComponentWrapper, WebComponentWrapperOptions} from '@angular-architects/module-federation-tools'

export const APP_ROUTES: Routes = [
  // Your route here:
 {
    matcher: startsWith('dashboard'),
    component: WebComponentWrapper,
    data: {
                    [...]
   } as WebComponentWrapperOptions
  },
];
```

Listing 10: The MFE method startWith matcher.

Listing 10 above shows how startWith matcher is used to give a key word so the app-shell can use it to bring its own router.

Bootstrap shell with the bootstrap method is required instead of the angular one, to make several Angular applications work in one browser. Listing 11 below illustrates how it is used.



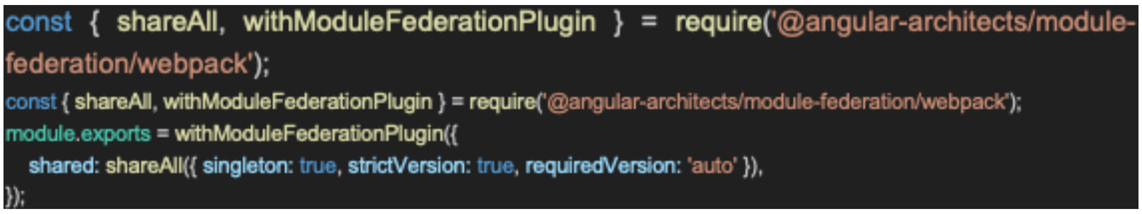
```
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Listing 11: The platformBrowserDynamic to bootstrap the MFE application.

Next, it is necessary to load the MFE into the app-shell is needed and the enable Module Federation using @angular-architects/module-federation plugin:

```
npm i @angular-architects/module-federation -D
```

Then the webpack.config.js file is configured using the withModuleFederationPlugin and shareAll methods, as shown in Listing 12 below:



```
const { shareAll, withModuleFederationPlugin } = require('@angular-architects/module-
federation/webpack');
const { shareAll, withModuleFederationPlugin } = require('@angular-architects/module-federation/webpack');
module.exports = withModuleFederationPlugin({
  shared: shareAll({ singleton: true, strictVersion: true, requiredVersion: 'auto' }),
});
```

Listing 12: The Module Federation method used to configure the Webpack file.

By using the withModuleFederationPlugin to expose the module, the MFE will be able to be loaded into the app-shell. The shareAll method is used to constrain and specify the version required to avoid any versioning conflict.

5.2 Implementation of TrainCM (EKE remote 1)

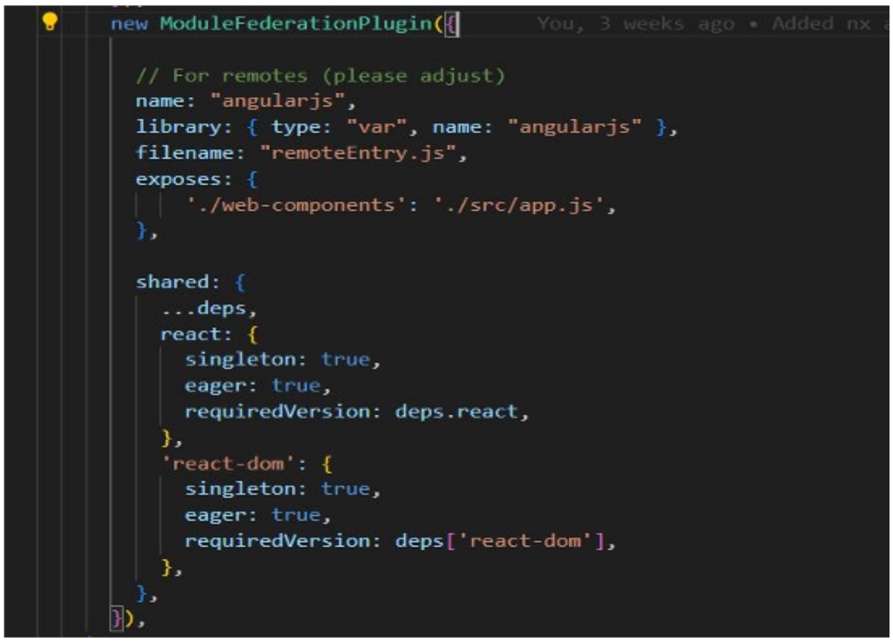
The current TrainCM Angular.js application components are all managed by one main component, the *index.js*. It is the entry component where the user is directed to the right page after checking their authentication. By exposing index.js components using the Model-Federation plugin, the entire application can be turned into an MFE at once, regardless of user authentication.

In this manner, the complexity associated with implementing the authentication service and components separation in TrainCM can be bypassed, as the scope of this thesis is limited to documenting the process of migrating two monolith applications into MFEs (remotes) hosted by another MFE application (host).

Although with the power of Webpack 5 bundler, it is straightforward to migrate the applications to MFE using the Module Federation plugin, but still some steps need to be taken to get it working:

- **Step 1:** Make the necessary modifications to the existing Webpack configuration file to make it work with the Module Federation plugin.
- **Step 2:** Make the necessary changes to the main file *index.js* to convert it to an MFE and wrap it in a Web Component.
- **Step 3:** Make an entry file that will just contain a dynamic import to load the rest of the application. This pattern gives the Module Federation the necessary time for loading the shared dependencies. (Steyer, 2020.)
- **Step 4:** Exposing the Web Component using the Webpack configuration *exposes*.
- **Step 5:** Using the *shared* section inside the plugin will make sure to mix several versions of a framework but also reuse an already loaded one if the version numbers match exactly.

The *requiredVersion* used to point to the installed version that can be found in the *package.json*. The *@angular-architects/module federation* has a helper method called *shared,* which also takes care of this when setting *requiredVersion* to auto or uses an exact version number (without any ^ or ~), as seen in Listing 13.

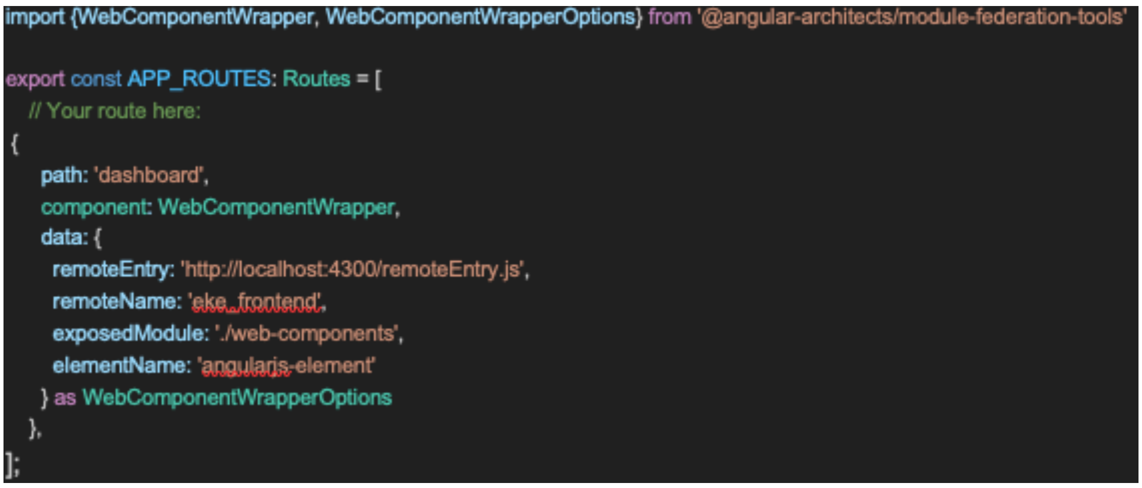


```
new ModuleFederationPlugin({        You, 3 weeks ago • Added nx

  // For remotes (please adjust)
  name: "angularjs",
  library: { type: "var", name: "angularjs" },
  filename: "remoteEntry.js",
  exposes: {
      './web-components': './src/app.js',
  },

  shared: {
    ...deps,
    react: {
      singleton: true,
      eager: true,
      requiredVersion: deps.react,
    },
    'react-dom': {
      singleton: true,
      eager: true,
      requiredVersion: deps['react-dom'],
    },
  },
}),
```

Listing 13: The requiredVersion property as part of the shared property block within the ModuleFederationPlugin.

Above, Listing 13 displays the *requiredVersion* property as part of the shared property block within the *ModuleFederationPlugin.* It points to the installed version specified in the application dependencies. This configuration enables to the applications to function without encountering versioning conflict.

- **Step 6:** load MFE into the shell app (the host) with lazy routing inside the in the *app.routes.ts* file. A better view can be seen in Listing 14:



```
import {WebComponentWrapper, WebComponentWrapperOptions} from '@angular-architects/module-federation-tools'

export const APP_ROUTES: Routes = [
  // Your route here:
{
    path: 'dashboard',
    component: WebComponentWrapper,
    data: {
      remoteEntry: 'http://localhost:4300/remoteEntry.js',
      remoteName: 'eke_frontend',
      exposedModule: './web-components',
      elementName: 'angularjs-element'
    } as WebComponentWrapperOptions
  },
];
```

Listing 14: MFE loader script for the Angular.js application.

In Listing 14 above, the presentation of an MFE wrapped in web component using *WebComponentWrapper* provided by the @angular-architects/module-federation-tools is demonstrated, aiming to achieve several advantages (Steyer, 2020), which are listed below:

- Abstracting differences between frameworks
- Mounting/ Unmounting Web Components is easy.
- Shadow DOM helps with isolating CSS styles.
- Custom Events and Properties allow communicating.

Angular App
Grained web Component
React App

Figure 14: MFE wrapped in Web Component. (Steyer, 2020)

Figure 14 is a simplified overview of how the Module Federation plugin is used to grain the Angular application into the React application as a web component to be mounted.

To load the MFE developed with Angular via Model-Federation it is necessary for it to be wrapped in a web component by using *WebComponentWrapper* and the given key data. In the code snapped above, the application developed with Angular is fetched as a specified URL, which is more convenient for development environments. The values of *remoteName* and *exposedModule* are coming from the MFE Webpack configuration. An HTML element is created by the wrapper component with the name *angular.js-element,* so the web component can use it to be mounted in.

5.3 Implementation of TrackCM (EKE remote 2)

TrackCM currently has three main views, and it is implemented with React.js and has no Webpack bundler. The only view going to be converted to MFE is the Detections view. The implementation for TrackCM is quite like the previous one, with a slightly different setup. To accomplish this transition, a series of steps are outlined below :

- **Step 1:** Integrate Webpack 5 into the TrackCM: To be able to use the Module Federation plugin, the Webpack 5 need to be installed by using the following command:

```
yarn add webpack –dev
```

- **Step 2:** Install the HTML, COPY and CSS plugins:

```
yarn add html-webpack-plugin
yarn add mini-css-extract-plugin
yarn add copy-webpack-plugin
```

- **Step 3:** Install the *webpack-dev-server* to output proxy debug info to the console:

```
yarn add webpack-dev-server
```

- **Step 4:** Make a configuration file in the root level of the TrackCM project:

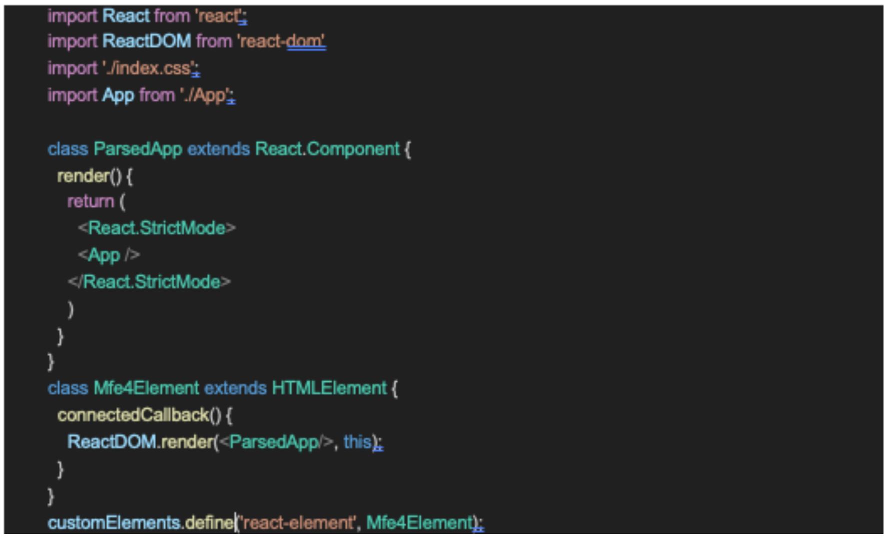
```
track-cm/webpack.config.js
```

- **Step 5:** Implement the webpack.config.js file and define all the necessary rules for loaders.

In the case of the React framework, this is all just about adding ModuleFederationPlugin to the configuration file. It is also important to add an entry file that contains a dynamic import to load the rest of the application:

```
//entry.js
import('./src/index.js');
```

- **Step 6:** Wrap the MFE in a Web Component: to do that with React, it is necessary to change the main file, the index.tsx, to index.js file and hand-warp the application. Figure 15 below gives an overview of how wrapping an MFE is done in a web component.

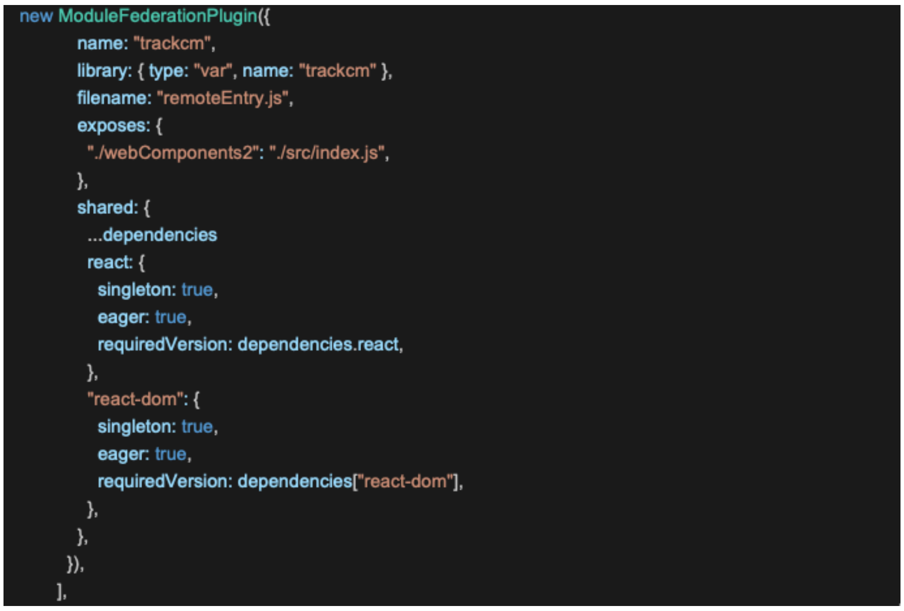


```
import React from 'react';
import ReactDOM from 'react-dom'
import './index.css';
import App from './App';

class ParsedApp extends React.Component {
  render() {
    return (
      <React.StrictMode>
      <App />
    </React.StrictMode>
    )
  }
}
class Mfe4Element extends HTMLElement {
  connectedCallback() {
    ReactDOM.render(<ParsedApp/>, this);
  }
}
customElements.define('react-element', Mfe4Element);
```

Figure 9: Wrapping the MFE application in a web component.

As a result of this script in Figure 15 above, the browser will mount the MFE with every ‘react-element’ tag that occurs in this application.

- **Step 7:** Exposing the wrapped application with the Module Federation plugin will require some configuration just like the previous MFE application. Figure 16 below shows the script needed using the specified versioning from the dependencies.



```
new ModuleFederationPlugin({
        name: "trackcm",
        library: { type: "var", name: "trackcm" },
        filename: "remoteEntry.js",
        exposes: {
          "./webComponents2": "./src/index.js",
        },
        shared: {
          ...dependencies
          react: {
            singleton: true,
            eager: true,
            requiredVersion: dependencies.react,
          },
          "react-dom": {
            singleton: true,
            eager: true,
            requiredVersion: dependencies["react-dom"],
          },
        },
      }),
    ],
```

Figure 16: The MFE ModuleFederationPlugin configuration.

- **Step 8:** Remove all the routes of the views leaving only the route of the Detections view. In another words, omit the routes in the *track-cm/src/Router.tsx* file that will not be utilised by the MFE application.

6 Testing and Results

Testing MFE applications can be a challenging process. There are several testing strategies for testing MFE applications as mentioned in chapter 3. This chapter explains how the E2E testing strategy was applied on the application. The test cases going to be implemented using the robot test library to measure the MFE application based on different concepts such as performance and accessibility. Next, identifying the test scenarios based on user expectations (in the case of this project, the EKE-Electronics company) to ensure the quality and reliability of the application.

6.1 Testing Plan and E2E Testing

The End-to-End (E2E) testing technique allows mirroring the entire application journey from inception to the destination. However, there is a twist: the project is not quite ready for the grand production stage. The testing plan is described as below.

6.1.1 Local Context Focus

The MFE application boasts a straightforward interface, concealing intricate infrastructure behind the scenes. To maintain simplicity, the responsibilities to abstraction classes going to be delegated by using the Factory Design Pattern (Rodriguez, 2020).

6.1.2 Step-by-Step Journey and Test Plan

In this subsection will dive into steps with details of E2E testing process:

6.1.2.1 Identifying Test Scenarios

The first step is to identify the necessary high-level test scenarios to meet the user (in this case EKE-Electronics company) expectations. Here are the possible scenarios for these applications:

1. **User Authentication Flow:** Authentication was not part of this thesis so the main authentication for the Main app-shell is not implemented yet. However, there is an authentication in the TrainCM application to

test that the user will be able to log in into the MFE application after the right credentials are provided.

2. **Navigation between MFEs:** Ensuring smooth transitions between the React.js (TrackCM) and Angular.js (TrainCM) applications.

3. **Dashboard Views:** Validating the Dashboard Content in both MFEs.

   Combined View: Testing the Display of both MFEs together in the Shell-Dashboard.

6.1.2.2 Map Scenario Steps

These steps outline the User Authentication Flow, Navigation between MFEs, and Combined View actions.

1. **User Authentication Flow:**
   a. Users open the home page.
   b. Users locate the login form.
   c. Users Enter valid/invalid credentials (Simulate successful/unsuccessful login attempts).
   d. Verify redirection after successful login.

2. **Navigation between MFEs:**
   a. User clicks the TrackCM tab.
   b. Verify the TrackCM view content.
   c. User clicks the TrainCM tab.
   d. Validate redirection to the authentication system.
   e. Complete the authentication process.
   f. Confirm redirection back to the TrainCM tab.
   g. Verify the TrainCM view content.

3. **Combined View:**

a. User clicks the combined view tab.
b. Ensuring both MFEs are displayed correctly.

6.1.2.3 Manual Testing

Before getting into the automation, manually do the validation assumptions:

a. Navigate through the app as a user would.
b. Verify UI elements, interactions, and transitions.
c. Confirm that the authentication for the TrainCM MFE app works as expected.

6.1.2.4 Automating Tests

Drawing from insights generated during manual testing, the process of developing robust automated test scripts involve the following steps:

- Using E2E testing framework called Selenium to implement test cases (Selenium, 2024).
- Writing the test scripts for each identified scenario.
- Employing the Factory Pattern to manage page objects and interactions (Rodriguez, 2020).

6.1.2.5 Meaningful Assertions

After running the test cases a comparison of the actual outcomes going to be taken with the assertions to verify whether the application behaves as expected, to ensure correctness, and to meet the EKE-Electronics company’s expectations. The following assertions are based on the minimum expectations of the EKE-Electronics company of this application:

a. After logging in successfully to TrainCM, the user is redirected to the TrainCM page.

b. All relevant content is displayed for TrackCM, TrainCM and Shell-Dashboard (combined view) as expected.
c. The transition occurs seamlessly without timing issues when navigating between MFEs.
d. The MFE app's performance is fast and appropriately valued.

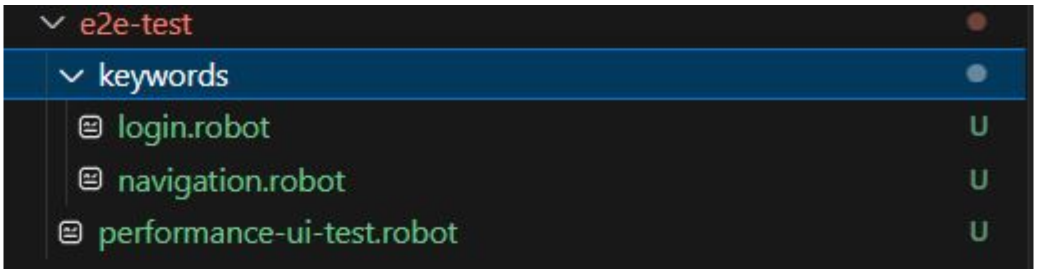
6.2 Implementing E2E Test

To start implementing the E2E test cases there are some principles to be considered. The following subsections providing more details.

6.2.1 Testing Hierarchical Structure and Defining Class Names

When creating E2E tests, some principles should be considered. For example, keeping the E2E test code separate from the application code, the group related tests and maintain logical links between different parts of the test suite.

To start implementing the E2E test classes, a new directory on the top root level of the application need to be made (Orr Sella, 2024). Listing 15 shows the hierarchical overview.



```
e2e-test
  keywords
    login.robot               U
    navigation.robot          U
  performance-ui-test.robot   U
```

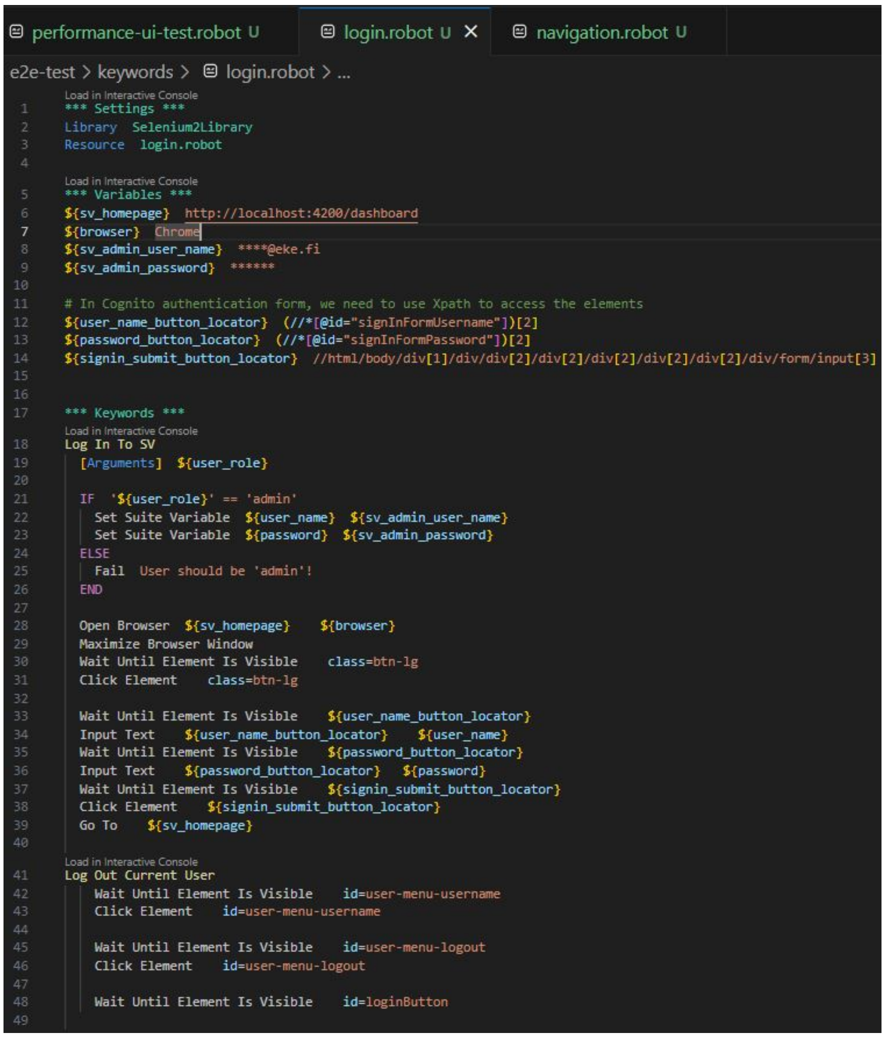
Listing 15: Screenshot of a hierarchical overview of the e2e testing

Listing 15 above shows a creation of the e2e-test directory. It contains the subdirectory named keyword which has the login and navigation classes. The module is organised using the Factory Design Pattern to create specific test instances like login and navigation tests. (Rodriguez, 2020.)

Next, on the top directory e2e-test, the performance-ui class where the test cases are also created. In addition, different scenarios are going to be defined. The performance-ui class behaves as the main class where all test cases are defined and going to be executed.

6.2.2 User Authentication Flow: Implementing Login Test Class

TrainCM uses the Cognito from AWS (AWS, 2024). The test script will be interacting with the Cognito authentication form. Xpath to is used to access elements within the form. The login test instance is created and written in the Python script language and using the Selenium library, the popular test automation robot framework. Listing 16 shows a screenshot of the test script in more detail:


performance-ui-test.robot U
login.robot U
navigation.robot U
e2e-test > keywords > login.robot > ...

```
Load in Interactive Console
*** Settings ***
Library  Selenium2Library
Resource  login.robot

Load in Interactive Console
*** Variables ***
${sv_homepage}  http://localhost:4200/dashboard
${browser}  Chrome
${sv_admin_user_name}  ****@eke.fi
${sv_admin_password}  ******

# In Cognito authentication form, we need to use Xpath to access the elements
${user_name_button_locator}  (//*[@id="signInFormUsername"])[2]
${password_button_locator}  (//*[@id="signInFormPassword"])[2]
${signin_submit_button_locator}  //html/body/div[1]/div/div[2]/div[2]/div[2]/div[2]/div[2]/div/form/input[3]


*** Keywords ***
Load in Interactive Console
Log In To SV
    [Arguments]  ${user_role}

    IF  '${user_role}' == 'admin'
        Set Suite Variable  ${user_name}  ${sv_admin_user_name}
        Set Suite Variable  ${password}  ${sv_admin_password}
    ELSE
        Fail  User should be 'admin'!
    END

    Open Browser  ${sv_homepage}    ${browser}
    Maximize Browser Window
    Wait Until Element Is Visible    class=btn-lg
    Click Element    class=btn-lg

    Wait Until Element Is Visible    ${user_name_button_locator}
    Input Text    ${user_name_button_locator}    ${user_name}
    Wait Until Element Is Visible    ${password_button_locator}
    Input Text    ${password_button_locator}   ${password}
    Wait Until Element Is Visible    ${signin_submit_button_locator}
    Click Element    ${signin_submit_button_locator}
    Go To    ${sv_homepage}

Load in Interactive Console
Log Out Current User
        Wait Until Element Is Visible    id=user-menu-username
        Click Element    id=user-menu-username

        Wait Until Element Is Visible    id=user-menu-logout
        Click Element    id=user-menu-logout

        Wait Until Element Is Visible    id=loginButton
```

Listing 16: Screenshot of the User Authentication – Login testing class.

6.2.3 Navigation between MFEs: Implementing the Test Class

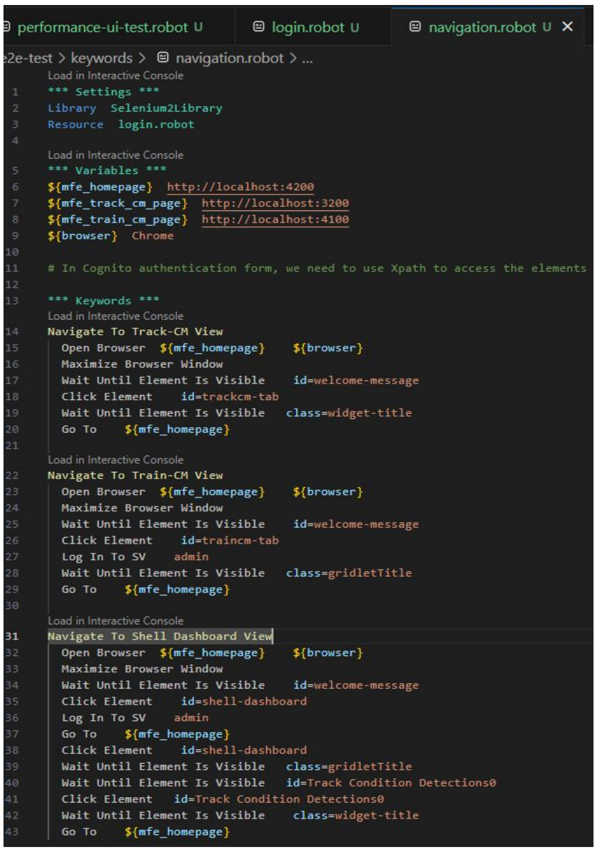
To implement the navigation test class, the script is going to be written using the same robot framework. The script will contain as the previous Login test class some variables, and keywords that can be reused across the application test class. These variables are:

- ${mfe_homepage}: “http://loaclhost:4200”
- ${mfe_track_cm_page}: “http://localhost:3200”
- ${mfe_train_cm_page}: “http://localhost:4100”
- ${browser}: Chrome

Several keywords will be reused to navigate between MFEs:

- Navigate To Track-CM View: Navigates to the TrackCM MFE page.
- Navigate To Train-CM View: Navigates to the TrainCM MFE page.
- Navigate To Shell-Dashboard View: Navigates to the combined view that displays both MFEs together at the same time.

Listing 17 below is showing as screenshot of the implementation in the final stage of the keyword navigation class.

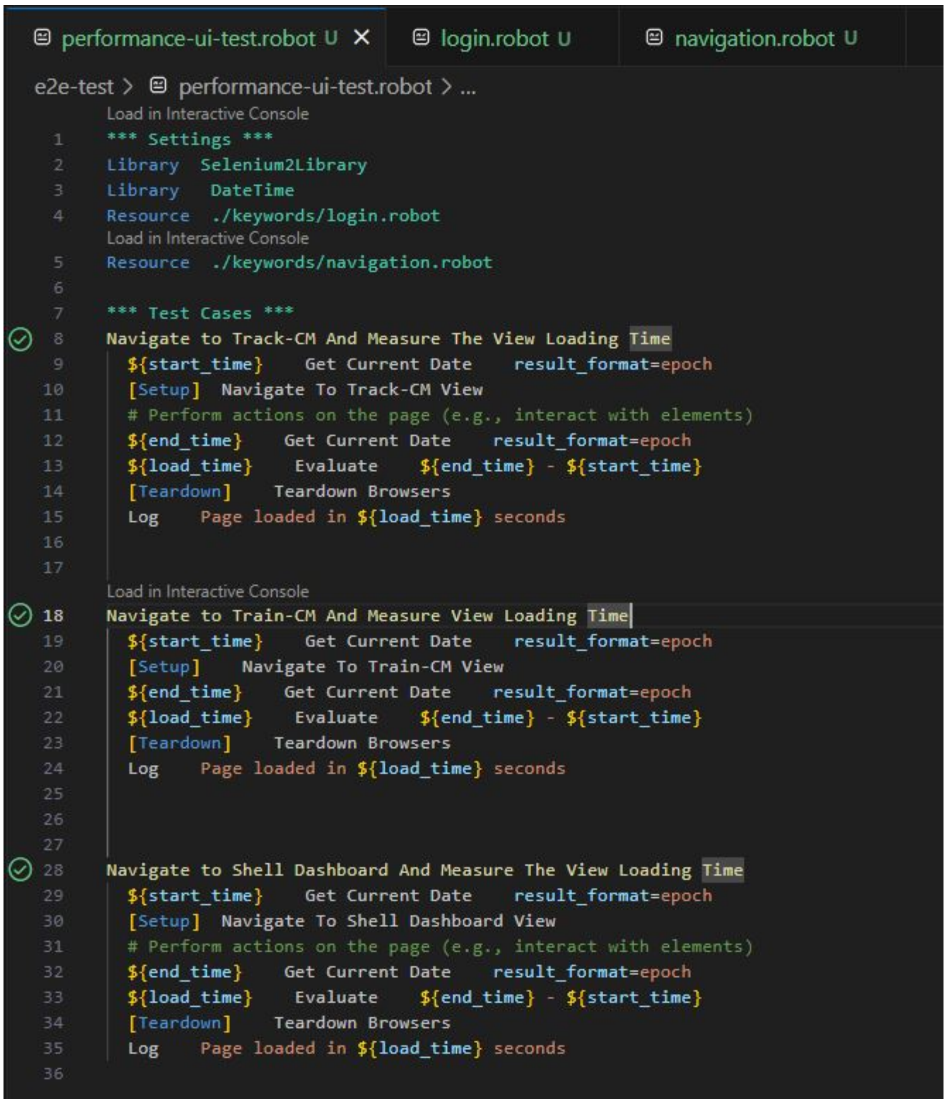

performance-ui-test.robot U
login.robot U
navigation.robot U
e2e-test > keywords > navigation.robot > ...

```
Load in Interactive Console
*** Settings ***
Library  Selenium2Library
Resource  login.robot

Load in Interactive Console
*** Variables ***
${mfe_homepage}  http://localhost:4200
${mfe_track_cm_page}  http://localhost:3200
${mfe_train_cm_page}  http://localhost:4100
${browser}  Chrome

# In Cognito authentication form, we need to use Xpath to access the elements

*** Keywords ***
Load in Interactive Console
Navigate To Track-CM View
  Open Browser  ${mfe_homepage}    ${browser}
  Maximize Browser Window
  Wait Until Element Is Visible    id=welcome-message
  Click Element    id=trackcm-tab
  Wait Until Element Is Visible   class=widget-title
  Go To    ${mfe_homepage}

Load in Interactive Console
Navigate To Train-CM View
  Open Browser  ${mfe_homepage}    ${browser}
  Maximize Browser Window
  Wait Until Element Is Visible    id=welcome-message
  Click Element    id=traincm-tab
  Log In To SV    admin
  Wait Until Element Is Visible   class=gridletTitle
  Go To    ${mfe_homepage}

Load in Interactive Console
Navigate To Shell Dashboard View
  Open Browser  ${mfe_homepage}    ${browser}
  Maximize Browser Window
  Wait Until Element Is Visible    id=welcome-message
  Click Element    id=shell-dashboard
  Log In To SV    admin
  Go To    ${mfe_homepage}
  Click Element    id=shell-dashboard
  Wait Until Element Is Visible   class=gridletTitle
  Wait Until Element Is Visible   id=Track Condition Detections0
  Click Element   id=Track Condition Detections0
  Wait Until Element Is Visible    class=widget-title
  Go To    ${mfe_homepage}
```

Listing 17: Screenshot of the navigating test class to test navigation between MFEs.

6.2.4 Main Testing Class: Performance-UI Testing Class

In the main test class, the keywords test classes going to be reused to avoid redundant, for example, login and navigation keywords. These keywords classes going to be imported as a resource at the beginning of the settings. This way, the keywords can be reused across all the test cases. As Listing 18 shows, a view of the final test script.


performance-ui-test.robot U ×
login.robot U
navigation.robot U
e2e-test > performance-ui-test.robot > ...

```
Load in Interactive Console
*** Settings ***
Library   Selenium2Library
Library    DateTime
Resource   ./keywords/login.robot
Load in Interactive Console
Resource   ./keywords/navigation.robot

*** Test Cases ***
Navigate to Track-CM And Measure The View Loading Time
  ${start_time}    Get Current Date    result_format=epoch
  [Setup]  Navigate To Track-CM View
  # Perform actions on the page (e.g., interact with elements)
  ${end_time}    Get Current Date    result_format=epoch
  ${load_time}    Evaluate    ${end_time} - ${start_time}
  [Teardown]    Teardown Browsers
  Log    Page loaded in ${load_time} seconds


Load in Interactive Console
Navigate to Train-CM And Measure View Loading Time
  ${start_time}    Get Current Date    result_format=epoch
  [Setup]    Navigate To Train-CM View
  ${end_time}    Get Current Date    result_format=epoch
  ${load_time}    Evaluate    ${end_time} - ${start_time}
  [Teardown]    Teardown Browsers
  Log    Page loaded in ${load_time} seconds



Navigate to Shell Dashboard And Measure The View Loading Time
  ${start_time}    Get Current Date    result_format=epoch
  [Setup]  Navigate To Shell Dashboard View
  # Perform actions on the page (e.g., interact with elements)
  ${end_time}    Get Current Date    result_format=epoch
  ${load_time}    Evaluate    ${end_time} - ${start_time}
  [Teardown]    Teardown Browsers
  Log    Page loaded in ${load_time} seconds
```

Listing 18: Screenshot of the main test class – the Performance-ui testing class.

In the performance-ui.robot class, the UI testing, and three test cases going to be running and visible to do the following:

- Navigating to Track-CM and measuring the view loading time.
- Navigating to Train-CM and measuring the view loading time.
- Navigating to shell Dashboard and measuring the view loading time.

Each test case involves steps such as:

- Getting the current date
- Navigating to specific views
- Performing actions on the page
- Calculating end time
- Evaluating load time
- Logging the results

The purpose of the test class is to check the following:

- Performance testing, specifically measuring how long it takes for different views to load within the application. The measure result is printed using the Log.
- Accessibility by navigating between the tabs and making sure the content is displayed.

6.3 Manual Testing: Running the MFE Applications

The aim of this section is to test the application manually to observe its behaviour and subsequently compare it with the result of the E2E testing after running the test cases. The test will be done by the developer team, and the result is demonstrated to the project PO for approval.

6.3.1 Starting the MFE Application

To test the MFEs manually, the MFEs need to be started one by one. The order of starting them manually is simply by opening a new terminal and executing the order as shown in Table 3:

Table 3: Order of commands for running the MFE application.

App name	Description	Command
TrainCM	- Open a new terminal - Run the application	*cd apps/frontend-legacy* *npm start*
TrackCM	- Open a new terminal - Start the application	*cd apps/track-cm* *yarn start*
Shell	- Open a new terminal	*nx serve shell*

6.3.2 Manual Testing

Following this testing plan, the manual testing should be done for validation assumptions before getting into the automation.

The assumptions were to navigate through the application as a user would, verify UI elements, interactions, and transitions. One more assumption is to confirm that the authentication for the TrainCM MFE application works as expected. Figma design is used to replace the real UI screenshots due to the privacy policies of the EKE-Electronics company.

The result of manual testing is an MFE application that contains multiple MFEs applications with different frameworks respective to the versions.

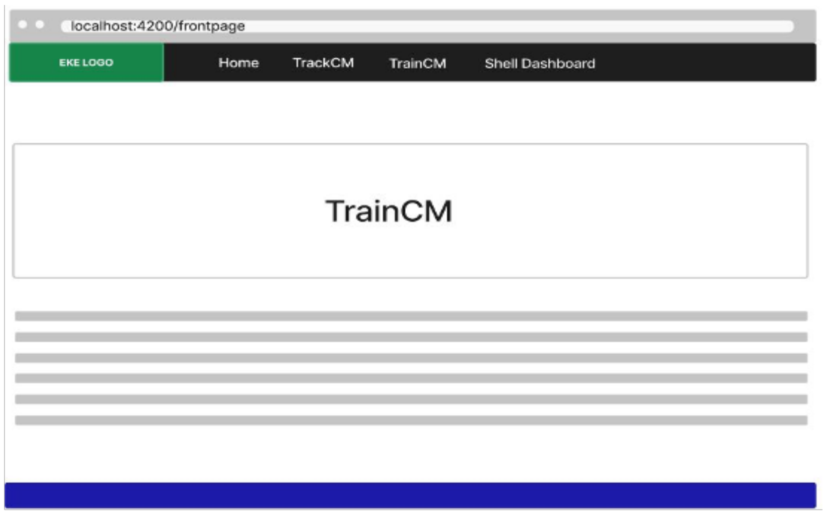


Figure 17: Screenshot of the TrainCM displayed on the shell-port 4200 after successful login. (Figma, 2024)

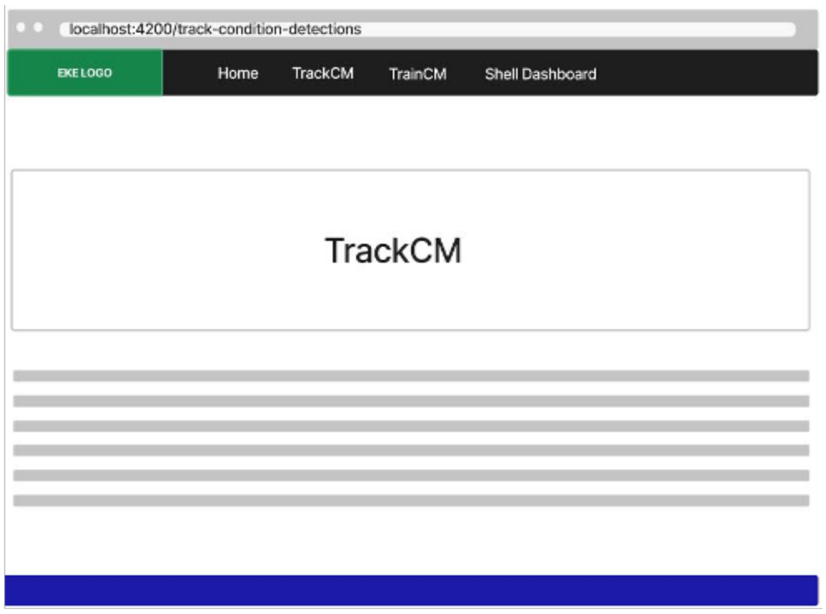


Figure 18: Screenshot of the TrackCM displayed on the shell-port 4200. (Figma, 2024)

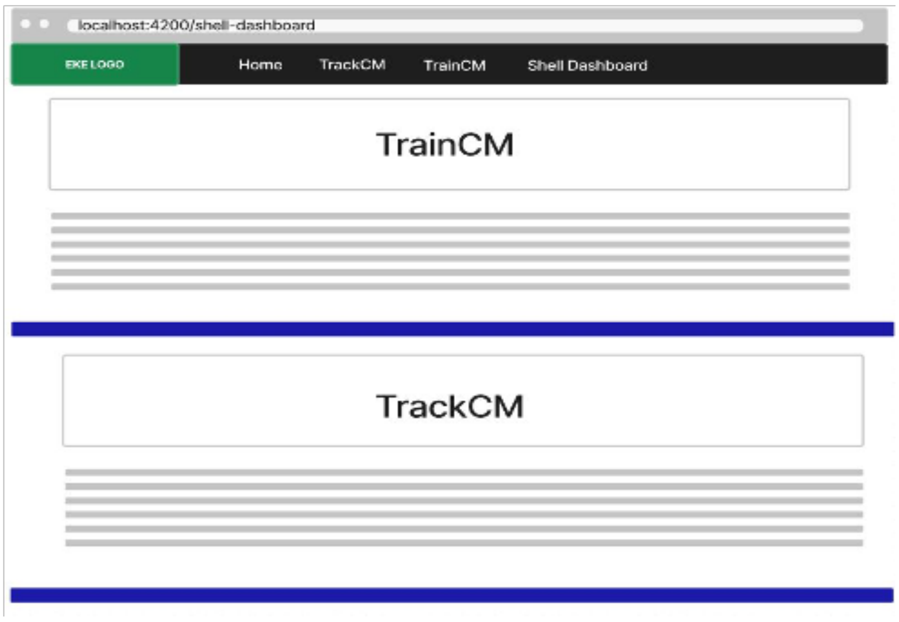
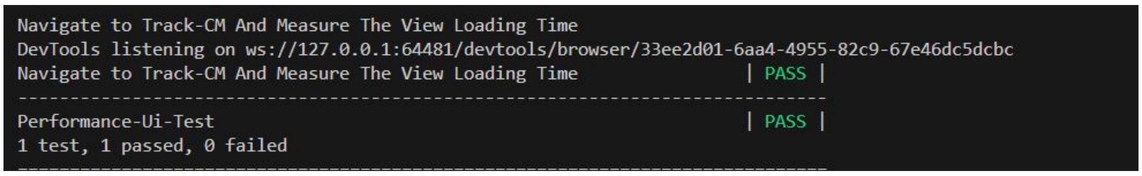


Figure 19: Screenshot of both the TrackCM & TrainCM displayed on the shell-port 4200. (Figma, 2024)

Figures 17, 18, and 19 above are screenshots representing the MFE application in the local environment with all its views (MFEs) running successfully. This manual test was conducted by running the application locally by the developer team. The result shows that each tab contains a functioning standalone application. Finally, Figure 19 above shows the Shell Dashboard tab, displaying content represents of all applications running within a single view.

6.4 Running Automate Test Cases

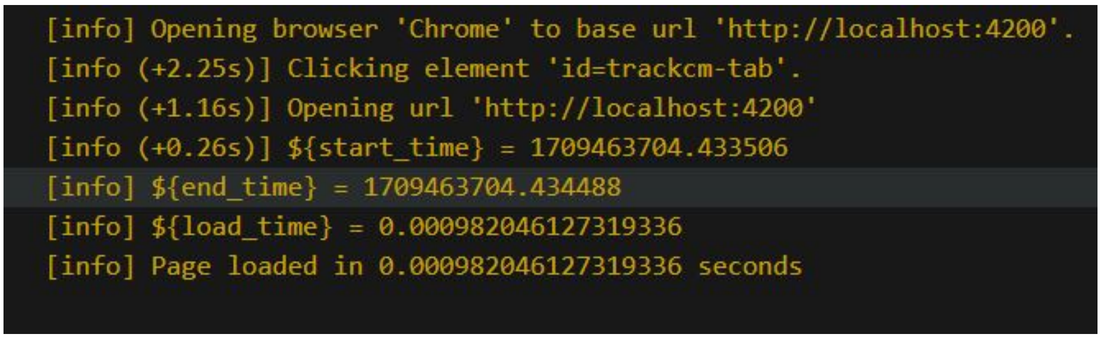
To execute the unit tests, automated tests can be run either vie terminal or through the user interface (Perälä, 2020). Simply by moving the pointer inside the *performance-ui* class to the specified test case, a menu appears which contains the option *run-test*. Clicking on the *run-test* option would execute the test case. To execute all the test cases, simply clicking on the *run suits* option from the start of the class would run all the test cases. For example, if one test case is run with PASS showing in green colour, it means the test has passed successfully as shown in Listing 19.



```
Navigate to Track-CM And Measure The View Loading Time
DevTools listening on ws://127.0.0.1:64481/devtools/browser/33ee2d01-6aa4-4955-82c9-67e46dc5dcbc
Navigate to Track-CM And Measure The View Loading Time                   | PASS |
------------------------------------------------------------------------------
Performance-Ui-Test                                                      | PASS |
1 test, 1 passed, 0 failed
```

Listing 19: Screenshot of the results of running the Track-CM automated test case.

Another important thing this test does is measuring the view loading page using the Log terminal as shown in Listing 20 below.

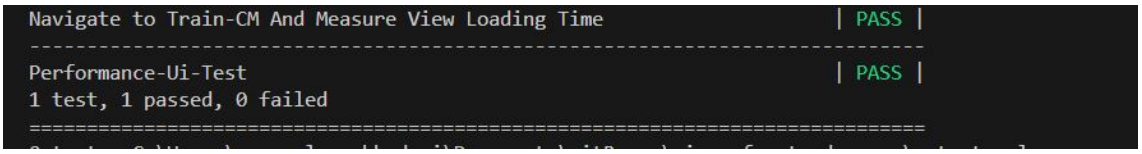


```
[info] Opening browser 'Chrome' to base url 'http://localhost:4200'.
[info (+2.25s)] Clicking element 'id=trackcm-tab'.
[info (+1.16s)] Opening url 'http://localhost:4200'
[info (+0.26s)] ${start_time} = 1709463704.433506
[info] ${end_time} = 1709463704.434488
[info] ${load_time} = 0.000982046127319336
[info] Page loaded in 0.000982046127319336 seconds
```

Listing 20: Screenshot of the Track-CM MFE application loading speed measurement.

Performance in MFE using Module Federation going to be measured to comparing to the iframe approach later.

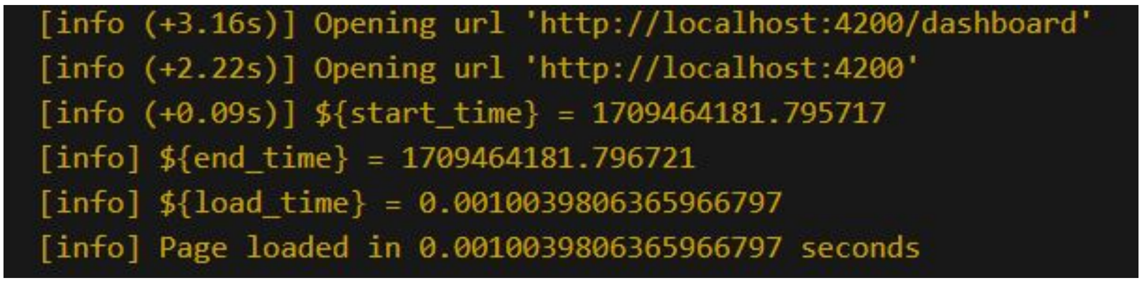
The same measurements and approach going to be applied for the Train-CM:



```
Navigate to Train-CM And Measure View Loading Time                       | PASS |
------------------------------------------------------------------------------
Performance-Ui-Test                                                      | PASS |
1 test, 1 passed, 0 failed
==============================================================================
```

Listing 21: Screenshot of the results of the Train-CM automated test case.

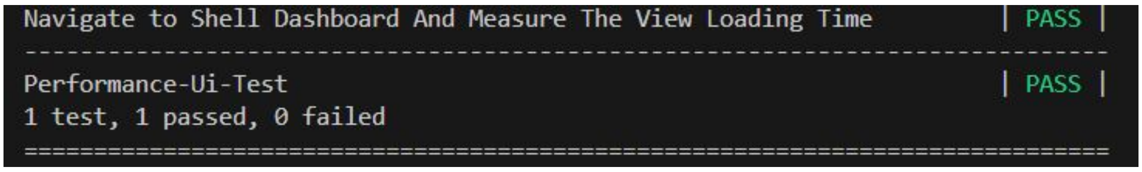
The log terminal in Listing 22 below shows the measuring of loading the Train-CM as well:



```
[info (+3.16s)] Opening url 'http://localhost:4200/dashboard'
[info (+2.22s)] Opening url 'http://localhost:4200'
[info (+0.09s)] ${start_time} = 1709464181.795717
[info] ${end_time} = 1709464181.796721
[info] ${load_time} = 0.0010039806365966797
[info] Page loaded in 0.0010039806365966797 seconds
```

Listing 22: Train-CM MFE application loading measurement in the log.

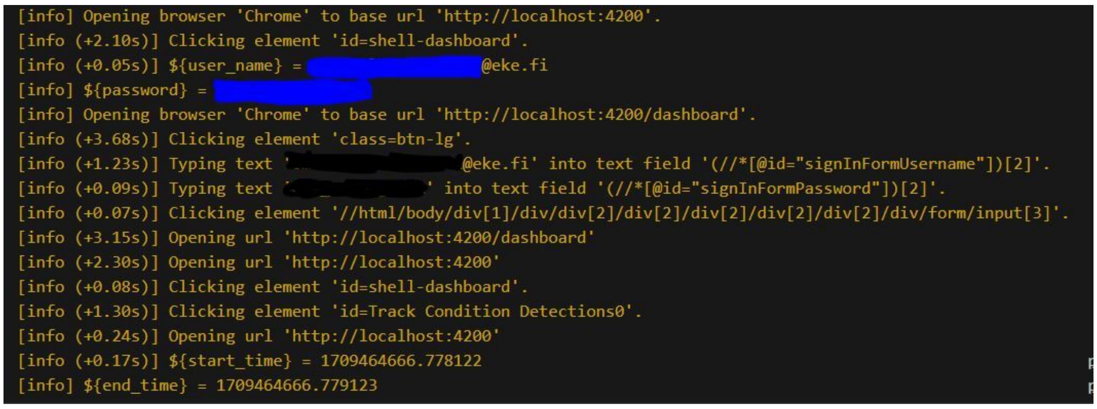
The last test case involves running the testing of the Shell-Dashboard to display the combined MFEs in single page (see Listing 23):



```
Navigate to Shell Dashboard And Measure The View Loading Time         | PASS |
------------------------------------------------------------------------------
Performance-Ui-Test                                                   | PASS |
1 test, 1 passed, 0 failed
==============================================================================
```

Listing 23: Screenshot showing the results of running the Shell-Dashboard automated test case.

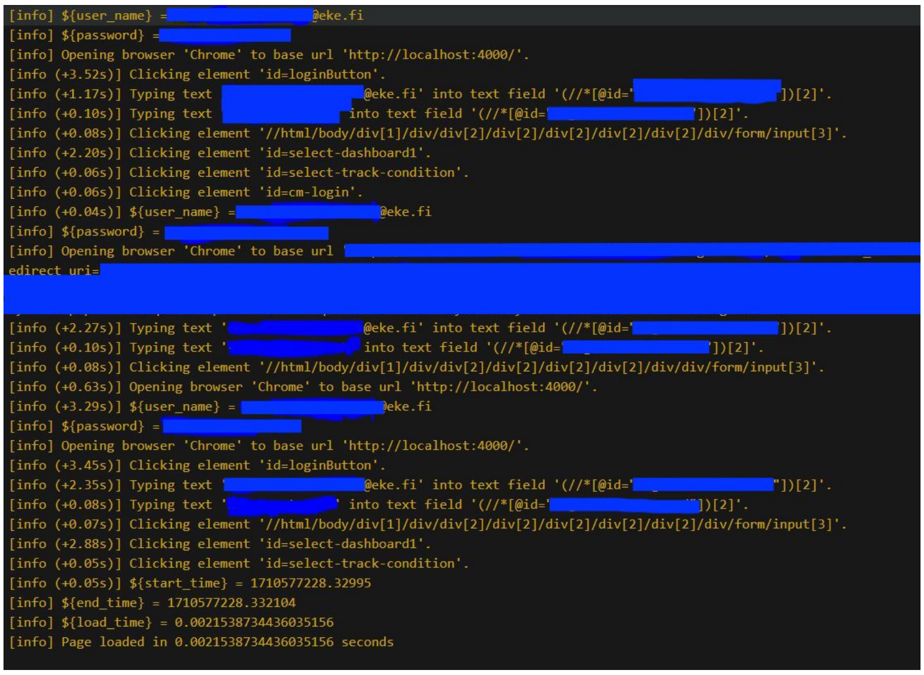
The loading page measurement results of the dashboard are displayed as well in the terminal:



```
[info] Opening browser 'Chrome' to base url 'http://localhost:4200'.
[info (+2.10s)] Clicking element 'id=shell-dashboard'.
[info (+0.05s)] ${user_name} = [redacted]@eke.fi
[info] ${password} = [redacted]
[info] Opening browser 'Chrome' to base url 'http://localhost:4200/dashboard'.
[info (+3.68s)] Clicking element 'class=btn-lg'.
[info (+1.23s)] Typing text '[redacted]@eke.fi' into text field '(//*[@id="signInFormUsername"])[2]'.
[info (+0.09s)] Typing text '[redacted]' into text field '(//*[@id="signInFormPassword"])[2]'.
[info (+0.07s)] Clicking element '//html/body/div[1]/div/div[2]/div[2]/div[2]/div[2]/div[2]/div/form/input[3]'.
[info (+3.15s)] Opening url 'http://localhost:4200/dashboard'
[info (+2.30s)] Opening url 'http://localhost:4200'
[info (+0.08s)] Clicking element 'id=shell-dashboard'.
[info (+1.30s)] Clicking element 'id=Track Condition Detections0'.
[info (+0.24s)] Opening url 'http://localhost:4200'
[info (+0.17s)] ${start_time} = 1709464666.778122
[info] ${end_time} = 1709464666.779123
```

Listing 24: Screenshot of the Train-CM MFE application loading speed measurement in the log.

The original TrainCM application, implemented with iframe approach, is also tested using the same measurements applied for MFE performance test to compare the end results. To achieve that, the same robot test class is applied on the legacy code as shown earlier in Listing 18. The loading page measurement result is shown in the log just like in Listing 25 below:



```
[info] ${user_name} =[redacted]@eke.fi
[info] ${password} =[redacted]
[info] Opening browser 'Chrome' to base url 'http://localhost:4000/'.
[info (+3.52s)] Clicking element 'id=loginButton'.
[info (+1.17s)] Typing text [redacted]@eke.fi' into text field '(//*[@id='[redacted]])[2]'.
[info (+0.10s)] Typing text [redacted]into text field '(//*[@id=[redacted]'])[2]'.
[info (+0.08s)] Clicking element '//html/body/div[1]/div/div[2]/div[2]/div[2]/div[2]/div[2]/div/form/input[3]'.
[info (+2.20s)] Clicking element 'id=select-dashboard1'.
[info (+0.06s)] Clicking element 'id=select-track-condition'.
[info (+0.06s)] Clicking element 'id=cm-login'.
[info (+0.04s)] ${user_name} =[redacted]@eke.fi
[info] ${password} = [redacted]
[info] Opening browser 'Chrome' to base url [redacted]
edirect_uri=[redacted]
[info (+2.27s)] Typing text '[redacted]@eke.fi' into text field '(//*[@id='[redacted]])[2]'.
[info (+0.10s)] Typing text '[redacted] into text field '(//*[@id=[redacted]'])[2]'.
[info (+0.08s)] Clicking element '//html/body/div[1]/div/div[2]/div[2]/div[2]/div[2]/div/div/form/input[3]'.
[info (+0.63s)] Opening browser 'Chrome' to base url 'http://localhost:4000/'.
[info (+3.29s)] ${user_name} = [redacted]eke.fi
[info] ${password} = [redacted]
[info] Opening browser 'Chrome' to base url 'http://localhost:4000/'.
[info (+3.45s)] Clicking element 'id=loginButton'.
[info (+2.35s)] Typing text '[redacted]@eke.fi' into text field '(//*[@id='[redacted]"])[2]'.
[info (+0.08s)] Typing text '[redacted]' into text field '(//*[@id='[redacted]"])[2]'.
[info (+0.07s)] Clicking element '//html/body/div[1]/div/div[2]/div[2]/div[2]/div[2]/div[2]/div/form/input[3]'.
[info (+2.88s)] Clicking element 'id=select-dashboard1'.
[info (+0.05s)] Clicking element 'id=select-track-condition'.
[info (+0.05s)] ${start_time} = 1710577228.32995
[info] ${end_time} = 1710577228.332104
[info] ${load_time} = 0.0021538734436035156
[info] Page loaded in 0.0021538734436035156 seconds
```

Listing 25: Screenshot of the legacy application with iframe approach loading performance measurement displayed in a log.

Listing 25 above shows the performance measurement and the long journey for the user to reach the Track-CM content.

6.5 Automated Test Results

This section discusses the test results to see whether they match the assertions defined earlier in the planning. Whether the application was verified and was behaving as expected.

- **Assertion:** After logging in successfully to TrainCM, the user is redirected to the TrainCM page.

Test Result: The user was successfully authenticated. However, in practice the user is directed to another port. The issue behind this failure is that Cognito is configured to redirect the user to port:4000 upon successful authentication. Unfortunately, adjusting this behaviour requires significant effort from the Dev-Ops team on the AWS cloud side, which falls outside the scope of this thesis. To address this, an alternative “Go To” keyword is provided to ensure redirection to the correct page.

- **Assertion:** All relevant content is displayed for the TrackCM, TrainCM and Shell-Dashboard (combined view) as expected.

  Test Result: The contents of all pages in each MFE are displayed successfully. However, there is a slight issue with the styles when the contents of the shell-Dashboard are displayed. It is worth noting that this style issue falls also outside the scope of this thesis.
- **Assertion**: The transition occurs seamlessly without timing issues when navigating between MFEs.

  Test Result: When the user navigates between MFE applications after he has been authenticated successfully, everything appears to be working as expected without any timing issue.
- **Assertion**: The MFE application’s performance is fast and appropriately valued.

  Test Result: The MFE application appears to be very fast with zero performance issues and aligns with other people’s research (Medium, 2024; Smirnov, 2019). The performance improvement is based on the observation and using the automated test results for loading speed compared to the iframe loading speed time.

Table 5 below, showing the findings of performance ratio comparison results for iframe and Module Federation approaches:

Table 4 : Findings of performance ratio comparison

Performance metrices name	iframe T1	Federated T2	Findings (T1/T2) * 100

Page loading time Track-CM	0.0021538734436035156s	0.000982046127319336s	=215% (Federated faster)
Page loading time Train-CM	0.000999275207519531s	0.0009989738464355469s	=100% (Federated faster)

In Table 3, the findings are analysed using the ratio T1/T2 multiplied by 100 to calculate the percentage. When users navigate to the Track-CM using the iframe approach, they encounter an additional window containing the login page, requiring reauthorisation. Only after successful authorisation does the Track-CM content appear. This process is cumbersome and time-consuming unlike Module Federation. Users are authenticated once to access all application contents seamlessly. The results demonstrate that federated modules outperform iframes. This approach enhances performance by dynamically loading modules with lazy loading, significantly reducing the initial application load time.

7 Discussion

In this section, the results of the project going to be discussed and see whether the findings are aligned with other similar studies. Furthermore, in this discussion the research questions that was raised at the beginning going to be answered.

7.1 Micro-frontends in Practice

The implementation of micro-frontends using Module Federation in the EKE-Electronics company’s web application presents a practical case study. The seamless integration of Angular.js and React.js applications under a single MFE app-shell offers a glance into the potential advantages and challenges of adopting this architectural approach.

The implementation results of MFEs using Module Federation in the project show several advantages that align with other similar studies mentioned in this thesis as well as other studies (Victor, 2021). It allows for the sharing of framework versions, streamlining development and ensuring consistency. Additionally, it enables the hosting of multiple MFEs by the app shell, facilitating the seamless integration of different technologies within a single view frame as is shown in the previous section. However, it also introduces complexities and larger bundle sizes, raising considerations about the trade-offs involved in using this technique that also aligns with other studies (Steyer, 2022).

7.2 Answering Research Questions

To address the research questions from the **Research Questions and Methodology** section earlier, the following answers are provided based on the results of this thesis.

Q1) Main Motivations for Adopting Micro-frontends Architectures:

- **Flexibility and Autonomy:** Practitioners and industrial companies are motivated to adopt micro-frontends architectures due to the flexibility and autonomy they provide in development, deployment, and scaling. The results of this project show that two separate technologies can be implemented in one main MFE application. In addition, providing the space for different expertise skills in the development team to participate in the implementation process gives better developer experience. Furthermore, with monorepo it is easier to deploy the MFE applications individually.
- **Scalability:** The results of this project demonstrate the ability to scale different parts of a web application independently. That is a key motivation for adopting micro-frontends.
- **Ownership and Independence:** micro-frontends enable single-team ownership of specific components, fostering a sense of responsibility and independence in development teams and give better development experience as shown in this project.

- **Improved Collaboration:** The architecture encourages collaboration among teams by allowing them to work on separate parts of the application without strong interdependencies.

Q2) Key Benefits of Micro-frontends in Web Application Development:

- **Simplified Development:** Breaking down a monolithic application into smaller, manageable parts simplifies the development process and facilitates component sharing. The thesis results demonstrated the integration of SPA applications into the micro-frontends architecture. Furthermore, the applications could be broken down into smaller manageable pieces, each doing a specific function.
- **Autonomous Evolution:** The project results showing that each micro-frontend can evolve independently, enabling teams to make changes and updates without affecting other parts of the application.
- **Efficient Deployment:** Micro-frontends allow for independent deployment of components, reducing the risk of system-wide failures and enabling faster release cycles. The deployment part has been discussed, although it was not implemented since it is out of the scope of this thesis.
- **Improved Maintainability:** With a focus on distinct business domains and autonomous codebases, micro-frontends make it easier to maintain and update specific functionalities without impacting the entire application. This applies when that functionality has been implemented as an individual MFE application. In the final MFE application, two SPA applications have been transformed into MFEs (remotes) hosted by another MFE application (host), at the view level.
- **Enhanced Testing Strategies:** Effective testing strategies within the context of micro-frontends architecture ensure the reliability and stability of the application. In this project, a simple comparison was demonstrated using E2E testing to measure the performance of the new approach implemented using MFE with the original iframe approach. Furthermore, a unit test and other testing strategies could improve the stability and the reliability of the application.

Q3) What challenges and issues may arise when implementing an MFE and how do practitioners address these concerns?

The challenges and issues that may arise when implementing micro-frontends architectures include:

- **Complexity of Integration:** Based on the thesis results, integrating multiple micro-frontends into a cohesive application can be challenging, especially when dealing with different technologies, frameworks, and dependencies.
- **Consistent Design and User Experience:** Ensuring a consistent design language and user experience across diverse micro-frontends can be a challenge, as each team may have its own interpretation of design guidelines. For example, the two applications discussed in this thesis, have totally different individual designs from each other.
- **Communication and Data Sharing:** Coordinating communication and data sharing between micro-frontends while maintaining isolation and encapsulation can pose challenges, particularly in scenarios where shared state management is required. This was not discussed in this thesis due to the Train-CM design complexity. However, with the help of monorepo it is possible to break down both MFEs (remotes) into smaller maintained components.
- **Performance Optimisation:** Managing performance across multiple Micro-frontends, especially in terms of loading times, code splitting, and resource optimisation, can be a significant challenge that practitioners need to address. For example, SEO (search engine optimisation) is a challenge that SPA application might face due to its nature.
- **Versioning and Dependency Management:** Handling versioning and dependency management across different micro-frontends to ensure compatibility and consistency can be complex, requiring robust strategies and tools. For example, having multiple dependencies for different applications can increase the bundle size significantly.

Practitioners address these challenges and issues in micro-frontends implementations through various strategies, including:

- **Establishing Clear Communication Channels:** Based on the result of this thesis, effective communication channels are set up between teams working on different MFEs to ensure alignment on design, architecture, and integration aspects. In this thesis, such a problem did not appear because the team is quite small. This issue would appear more clearly with bigger teams. A clear design and communication are essential keys for successful implementation.
- **Implementing Shared Design Systems:** Developing shared design systems and component libraries to maintain consistency in design and user experience across MFEs is a common practice. In this thesis, there were no shared components between MFEs due to the complexity of the Train-CM application.
- **Utilising Micro-Frontends Frameworks:** Leveraging MFE frameworks and tools that provide standardised approaches to integration, communication, and data sharing. In this thesis for example Nx is used in this project.
- **Implementing Performance Optimisation Techniques:** Employing performance optimisation techniques such as lazy loading, code splitting, and caching to enhance the overall performance of the micro-frontends architecture is a recommended practice. For example, Nx was used in this project for cashing. In addition, lazy loading with Module Federation was used as shown earlier in the performance measurement in the testing results. As a result of integrating MFE architecture with Module Federation approach instead of iframe, the page loading time has increased signifyingly by 215% as shown in Table 5. However, until the application is deployed, it is difficult to measure how the increased bundle size of the application can impact the loading time and the performance due the amount of data transferred over the network.

- **Adopting Continuous Integration and Deployment (CI/CD)**: Implementing CI/CD pipelines to automate the testing, deployment, and versioning processes across micro-frontends, ensuring consistency and efficiency in development workflows. The subject was discussed in this thesis, but implementation did not happen because deployment is out of the scope of this thesis.

7.3 Transitioning from Monolith to Micro-frontends

Transitioning from a monolithic architecture to micro-frontends, as seen in the case of the EKE-Electronics company, presents a multifaceted challenge encompassing technical, organisational, and financial aspects. Decoupling the existing monolithic codebase into smaller, self-contained frontend modules requires careful planning to ensure minimal disruption to functionality and user experience.

This process involves restructuring the application into autonomous modules, understanding dependencies, and managing compatibility issues. Additionally, the transition requires organisational alignment, effective communication, and change management strategies to coordinate development and deployment across teams. Incrementally introducing micro-frontends while maintaining existing functionality is crucial to mitigate risks.

Overall, this transition highlights the complexity of the task, requiring a comprehensive approach addressing technical, organisational, and financial considerations for successful and sustainable transformation. The importance of independence in MFE development, as emphasised in earlier studies, facilitates easier testing, deployment, and maintenance, contributing to the robustness and scalability of applications in the long term.

8 Conclusions

8.1 Achievements and Contributions

The final year project investigated the frontend infrastructure transformation of the EKE-Electronics company's web application, particularly the successful integration of disparate technologies such as Angular.js and React.js under a single micro-frontend application (?) shell. The project serves as a noteworthy achievement, demonstrating the feasibility and advantages of transitioning to a microservices architecture.

Furthermore, the identification of challenges and the proposed solutions for implementing micro-frontends, as evidenced in the case of the Condition Monitoring and Train Monitoring applications, adds practical value to the field, offering actionable insights for organisations seeking to adopt micro-frontends in their frontend development endeavours. This has significant implications for enhancing the scalability, maintainability, and user-centricity of their web applications, offering a roadmap for leveraging an MFE to streamline development and improve the overall user experience.

Overall, the achievements and contributions of this project underscore the significance of MFEs as a transformative approach in frontend development, with implications for the advancement of web application design, development, and maintenance.

8.2 Future Directions

Future research in the field of MFE architecture could focus on deployment and its strategies, including exploring different deployment models such as continuous deployment and blue-green deployment. Additionally, after deployment there is a need for comprehensive performance testing and analysis to understand the real-world performance implications of MFE architecture, along with strategies for optimising their performance. Another area for future investigation is the exploration of best practices for managing the development and versioning of micro-frontends in large-scale projects, ensuring consistency and compatibility across micro-frontends developed by different teams.

Furthermore, potential research could focus on the impact of MFE on the overall development workflow and developer experience, including the integration of MFE with modern development tools and workflows. The goal would be to identify strategies for streamlining the development and testing of micro-frontends in a collaborative environment. Overall, future research in the field of MFE aims to address the practical challenges and considerations involved in the real-world implementation and management of MFE, providing actionable insights and best practices for organisations adopting this architectural approach.

8.3 Closing Remarks

In conclusion, the implementation of an (?) MFE using Module Federation represents a significant advancement in frontend development, offering greater flexibility, scalability, and productivity. The case study of the EKE-Electronics company's web application serves as a testament to the feasibility of transitioning from a conventional approach to an MFE architecture. The successful integration of disparate technologies, such as Angular.js and React.js, under a single micro-frontend application shell demonstrates the potential of MFEs (?) to streamline development and enhance the maintainability of complex web applications.

In general, the goal of this project has been achieved by providing a successful implemented solution that can serve as a good start to further development towards adapting the MFE architecture in the EKE-Electronics company.

In closing, the journey towards MFE architecture (?) represents an ongoing evolution in frontend development, with the potential to revolutionise the way web applications are designed, developed, and maintained. By embracing the opportunities for future research and innovation, the industry can continue to harness the benefits of MFE architecture (?), ultimately shaping the future of frontend development and enhancing the overall user experience.

References

AWS (2024). Amazon Cognito. Available at: https://aws.amazon.com/cognito/. (Accessed: 3 May 2024).

Azizyan, S. (2022). Moderation Panel for Virtual Event Platform as a Micro Frontend Module. Bachelor’s thesis. JAMK University of Applied Sciences. Available at: https://www.theseus.fi/bitstream/handle/10024/754390/Opinnaytetyo_Azizyan_Samson.pdf?sequence=2&isAllowed=y. (Accessed: 3 May 2024).

Barak, T. (2018). ‘Part I –E2E Testing and Selenium’, Medium, 1 February. Available at: https://medium.com/the-hitchhikers-guide-to-e2e-testing/part-i-e2e-testing-and-selenium-ef031978ee20. (Accessed: 3 May 2024).

Bui, S. (2021). Micro frontend: Microservice Implementation on Web Development. Bachelor’s thesis. Metropolia University of Applied Sciences. Available at: https://www.theseus.fi/bitstream/handle/10024/511484/Son_Bui.pdf?sequence=2. (Accessed: 3 May 2024).

EKE-Electronics Ltd (2024). Provider of Train Control and Management System. Available at: https://www.eke-electronics.com/company/. (Accessed: 3 May 2024).

Figma (2024). The Collaborative Interface Design tool. [Online]. Available at: https://www.figma.com/. (Accessed: 3 May 2024).

Jackson, Z. (2023). ‘Understanding Module Federation: A Deep Dive’, Medium, 29 August. Available at: https://scriptedalchemy.medium.com/understanding-webpack-module-federation-a-deep-dive-efe5c55bf366. (Accessed: 5 May 2024).

Katalon (2018). ‘Basic ways of Using Selenium WebDriver in Katalon Studio’, Medium, 28 August. Available at: https://medium.com/katalon-studio/basic-ways-of-using-selenium-webdriver-in-katalon-studio-d95b1e89c312. (Accessed: 6 May 2024).

LambdaTest. (2023). ‘Testing Strategies for Micro Frontends’, LambdaTest Blog, 23 August. Available at: https://www.lambdatest.com/blog/micro-frontends-testing-strategies/. (Accessed: 3 May 2024).

Mezzalira, L. (2021). Building Micro-Frontends: Scaling Teams and Projects, Empowering Developers. [Online]. Available at: https://www.oreilly.com/library/view/building-micro-frontends/9781492082989/. (Accessed: 14 January 2024).

Medium (2024). Your Micro Frontend React App 307% Faster. Available at: https://medium.com/@ar.aldhafeeri11/your-micro-frontend-react-app-runs-307-times-faster-ea4e15e00ff8. (Accessed: 10 March 2024).

Newman, S. (2019). Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, 1st ed. [Online]. Available at: O’Reilly Media, Inc. (Accessed: 3 May 2024).

Newman, S. (2020). Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, 2nd ed. [Online]. Available at: O’Reilly Media, Inc. (Accessed: 3 May 2024).

Nx (2024). A powerful open-source build system providing tools and techniques for enhancing developer productivity and help to monorepo management. Available at: https://nx.dev/getting-started/intro. (Accessed: 3 May 2024).

Orr Sella. (2014). Integration and End-to-End Test Configurations in SBT. Available at: https://orrsella.com/2014/09/24/integration-and-end-to-end-test-configurations-in-sbt-for-scala-java-projects/. (Accessed: 3 May 2024).

Perälä, J. (2020). ‘How to run Robot Framework test from command line?’, Dev Community, 26 March. Available at: https://dev.to/juperala/how-to-run-robot-framework-test-from-command-line-5aa. (Accessed: 3 May 2024).

Python (2024). ‘What is Python? Executive Summary’. Available at: https://www.python.org/doc/essays/blurb/. (Accessed: 3 May 2024).

React (2024). The Library for Web and Native User Interfaces. Available at: https://react.dev/. (Accessed: 6 May 2024).

Rodriguez, J. (2020). ‘How to Build an E2E Testing Framework Using Design Patterns’, freeCodeCamp, 9 November. Available at: https://www.freecodecamp.org/news/build-an-e2e-test-framework-with-design-patterns/. (Accessed: 3 May 2024).

Selenium (2024). A Web Testing Library for Robot Framework. Available at: https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html. (Accessed: 9 March 2024).

Single-spa (2024). The recommended setup. [Online]. Available at: https://single-spa.js.org/docs/recommended-setup/#module-federation. (Accessed: 3 May 2024).

Smirnov, A. (2019). ‘3 Reasons You Might Not Want to Use iframes’, OSTraining Blog, 30 April. Available at: https://ostraining.com/blog/. (Accessed: 3 May 2024).

Steyer, M. (2020). The Micro Frontend Revolution: Module Federation with Angular. [Online]. Available at: https://www.angulararchitects.io/en/blog/the-microfrontend-revolution-part-2-module-federation-with-angular/. (Accessed: 3 May 2024).

Steyer, M. (2022). Enterprise Angular: Micro Frontends and Moduliths with Angular. [Online]. Available at: https://www.angulararchitects.io/en/ebooks/micro-frontends-and-moduliths-with-angular/. (Accessed: 14 January 2024).

Steyer, M. (2022). Multi-Framework and -Version Micro Frontends with Module Federation: Your 4 Steps Guide. [Online]. Available at: https://www.angulararch