Technische Hochschule Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Bachelorarbeit

A Study on Micro Frontend Architecture and Its Impact

vorgelegt an der Technischen Hochschule Würzburg-Schweinfurt in der Fakultät Informatik und Wirtschaftsinformatik zum Abschluss eines Studiums im Studiengang Informatik

Tan Phat Nguyen

Eingereicht am: 13. September 2024

Erstsprüfer: Prof. Dr. Peter Braun
Zweitprüfer: Prof. Dr. Tristan Wimmer

Abstract

This study explores the feasibility and impact of adopting micro frontend architecture for the Deutsche Klassenlotterie Berlin project, an online lottery platform operated by LOTTO Berlin. The platform's existing monolithic architecture, developed in 2014, has become outdated, presenting significant challenges in terms of flexibility, maintainability, and scalability. In response, MULTA MEDIO, the development company, aims to modernize the system by transitioning to a micro frontend architecture. Through a comprehensive experimental approach, this study evaluates how the adoption of micro frontends can mitigate the limitations of the current architecture. The research spans eight chapters, covering the architectural transition, its practical implementation, and an analysis of the benefits and drawbacks. The findings offer valuable insights to support the decision-making process for this architectural shift.

Zusammenfassung

Diese Studie untersucht die Machbarkeit und die Auswirkungen der Einführung einer Mikro-Frontend-Architektur für das Projekt Deutsche Klassenlotterie Berlin, einer von LOTTO Berlin betriebenen Online-Lotterieplattform. Die bestehende monolithische Architektur der Plattform, die im Jahr 2014 entwickelt wurde, ist veraltet und stellt erhebliche Herausforderungen in Bezug auf Flexibilität, Wartbarkeit und Skalierbarkeit dar. MULTA MEDIO, das Entwicklungsunternehmen, möchte daher das System durch die Umstellung auf eine Mikro Frontend Architektur modernisieren. Durch einen umfassenden experimentellen Ansatz wird in dieser Studie untersucht, wie der Einsatz von Mikro Frontends die Einschränkungen der derzeitigen Architektur abmildern kann. Die Untersuchung erstreckt sich über acht Kapitel, die sich mit der Umstellung der Architektur, ihrer praktischen Umsetzung und einer Analyse der Vor- und Nachteile befassen. Die Ergebnisse bieten wertvolle Einblicke, um den Entscheidungsprozess für diesen Architekturwechsel zu unterstützen.

Contents

Glossary

***CD* – Continous Deployment** 34, 65, 66

***CDN* – Content Delivery Network** 19, 20, 27

***CI* – Continous Integration** 34, 65, 66

***CORS* – Cross-origin resource sharing** 55

***DDD* – Domain-Driven Design** 7, 8, 9, 11, 34, 35

***DKLB* – Deutsche Klassenlotterie Berlin** 2, 3, 12, 34, 57, 59, 70, 71

***ESI* – Edge-Side Includes** 27, 28

***HTML* – HyperText Markup Language** 1, 19, 29, 30

***SEO* – Search Engine Optimization** 1, 19, 21, 35

***SPA* – Single-Page Application** 1, 4, 17, 35, 64, 65, 66

***SSI* – Server-Side Includes** 26, 27, 28

List of Figures

List of Listings

1. Introduction

In the early days of the World Wide Web, the frontend was relatively simple. The first website, published in 1991 by British scientist Tim Berners-Lee while working at The European Organization for Nuclear Research, was a basic, static webpage created using HyperText Markup Language (HTML) [2]. These early static websites were hosted on servers with content that remained unchanged unless manually updated by a webmaster. The level of interactivity was minimal, primarily focused on displaying information in a straightforward, text-based manner. As the web evolved, the complexity and expectations of web applications grew, leading to more sophisticated and interactive frontend designs.

Today, various modern methodologies have been developed to meet the diverse needs of different projects. One of the most notable approaches is Single-Page Application (SPA), which offers a highly interactive user experience characterized by dynamic content loading and smooth navigations. Despite these benefits, SPA often falls short in terms of performance compared to traditional server-side rendering, particularly concerning speed and Search Engine Optimization (SEO). A hybrid approach that effectively combines the strengths of both methodologies is universal rendering. This approach starts by rendering the application on the server, which results in faster load times and improved SEO. Following this initial load, the client-side manages subsequent interactions, thereby maintaining the high level of interactivity characteristic of SPAs.

Despite these advancements, as the functionality of a frontend application grows, it becomes a complex and large structure. This growth leads to challenges in scaling and maintaining large-scale applications, especially when multiple teams are involved in development. This issue stands in contrast to the backend, which has increasingly adopted the microservices architecture over the past decade. In this architectural style, the backend is divided into smaller, independently deployable services, enhancing flexibility, scalability, and ease of maintenance.

The frontend of a web application functions as the presentation layer that users initially see and interact with, including everything users experience visually and navigate through such as layouts, buttons, images, and forms. A well-designed frontend not only ensures that the application is visually appealing but also guarantees responsiveness, reliability, and simplicity. Because of its direct effects on

user experience and satisfaction, frontend development has become a critical and complex task that companies must approach with thorough attention.

1.1. Motivation

The motivation behind this study arises from the challenges faced by the Deutsche Klassenlotterie Berlin (DKLB) website, a platform for online lottery services, allowing users to participate in various lottery games and access related information provided by LOTTO Berlin [3]. This web application, originally developed in 2014, is based on an outdated monolithic architecture. Although this architectural design was effective and sufficient at that time, it has since fallen behind modern standards, leading to a series of critical issues that demand consideration.

One of the primary issues is the lack of flexibility. Since all components of the system are tightly interconnected, updating or modifying any individual part requires redeployment of the entire application. This process is time-consuming and introduces the risk of potential downtime, which negatively affects user experience and business operations.

Maintainability is another major drawback. As this legacy system ages, maintaining it has become increasingly complex and error-prone. Extensive testing and coordination are required to ensure changes do not cause new problems elsewhere in the application. Furthermore, the reliance on old technologies and a large codebase makes it challenging to attract skilled developers and poses difficulties for new junior developers, who may struggle to explore and understand the codebase. These factors collectively slow down development cycles and delay the release of new features.

Additionally, the current frontend architecture of the DKLB project is misaligned with agile development practices. The tight coupling of components prevents parallel development efforts and continuous integration, as teams are unable to work independently on different parts of the application. This misalignment further underscores the urgent need for a modern architectural approach that can address the limitations of the current system.

1.2. Objective

In response to the challenges faced by the current application, MULTA MEDIO, the company responsible for its development and maintenance, has decided to undertake a complete rewrite of the application. A central concern in this process is the selection of an appropriate frontend architecture. With the growing interest in micro frontend architecture and its benefits, as demonstrated by many large corporations, this study aims to explore the feasibility and potential advantages of adopting this approach for the DKLB application. The results of this investigation are intended to provide in-depth insights and a thorough analysis, which inform the decision-making process regarding the architectural transition.

This study is guided by the following primary research questions (RQs):

- RQ1: How does adopting micro frontend architecture specifically affect the flexibility, maintainability, scalability, and performance of a web application?
- RQ2: Can the micro frontend approach effectively mitigate the specific challenges and limitations inherent in the current monolithic architecture of the DKLB project?

To address these questions, a comprehensive experiment, including multiple stages from planning to deployment, will be conducted. This experiment will serve as a proof of concept, designed to replicate a real-world project scenario to evaluate the impact of the proposed micro frontend architecture on the identified aspects.

1.3. Structure

This study on micro frontend architecture is organized into eight chapters, each building progressively to provide a thorough exploration of the topic. The first chapter, which concludes here, introduces the study and defines its objectives. The second chapter examines the monolithic architecture and the shift towards microservices in the backend, illustrating how this evolution has influenced trends in frontend development. It also introduces the concept of Domain-Driven Design, particularly in relation to micro frontends. With this foundational knowledge in place, chapter three presents a review of the current state of the art, exploring the motivations for adopting micro frontend architecture through both academic research and practical case studies.

In chapter four, a decision framework is used to guide the selection of the most suitable approaches for implementing micro frontends, which are explored in greater detail in chapter five. Chapter six presents an in-depth experiment, demonstrating the multiple stages of implementing micro frontend architecture within the context of the DKLB project. The evaluation of this experiment, including comparisons with the monolithic SPA architecture, and a discussion of the advantages, disadvantages, and limitations of the study, is covered in chapter seven. Finally, chapter eight summarizes the findings and offers suggestions for future research directions.

2. Background

This chapter establishes the foundational concepts by first exploring monolithic architecture, the traditional approach to software design. The discussion then moves to the rise of microservices architecture, a significant paradigm shift in backend development that has significantly influenced modern software design. This exploration is extended to the frontend, where similar principles have been adopted. Finally, the chapter introduces the concept of Domain-Driven Design, with a particular focus on its application within the micro frontend architecture.

2.1. Monolithic Architecture

Sam Newman defines a monolith as a deployment unit in which all system functionalities are deployed together. The most common forms of monolithic architectures are the single-process monolith, the modular monolith, and the distributed monolith [4].

The single-process monolith represents the most traditional form of monolithic architecture, where all code and functionalities are encapsulated within a single process. In contrast, a modular monolith introduces a level of separation within this single process by dividing it into distinct modules. These modules can be developed independently, thus enabling parallel development efforts. However, they must ultimately be integrated for deployment, which retains the monolithic nature of the application. The distributed monolith, while involving multiple services that can be developed and distributed independently, still requires these services to be deployed together as a unified whole. This need for simultaneous deployment ties the architecture back to monolithic principles, despite the distributed nature of the services.

James Lewis further underscores the simplicity of monolithic architectures [1]. Because everything runs in a single process, the development, testing, and deployment processes are more straightforward. Developers can run the entire application on their local machines, allowing for thorough testing and ensuring seamless functioning of all components. Additionally, the deployment process itself is simplified, as the entire application is packaged and deployed as a single unit.

However, as applications grow in size and complexity, the limitations of a monolithic architecture become more clear. Even small changes require the entire application to be rebuilt and redeployed, which introduces inefficiencies. The close coupling of components not only slows down the development process but also complicates the updating of individual modules. Over time, maintaining a well-organized and clean codebase becomes increasingly challenging, leading to a structure that is difficult to manage. This growing complexity negatively impacts the system's reliability and maintainability.

Moreover, scaling a monolithic application poses significant challenges. Because the entire application exists as a single unit, scaling necessitates replicating the entire system, even when only a small part of it is needed to scale (illustrated in Figure 1). This approach results in resource inefficiencies and increased costs. In contrast, transitioning to a microservices architecture provides a more flexible and scalable solution for developing complex applications, enabling the scaling of individual components rather than the entire system.

Monolithic architecture can still be a practical and viable option, particularly for smaller organizations or those that do not encounter significant scalability challenges [1], [4].

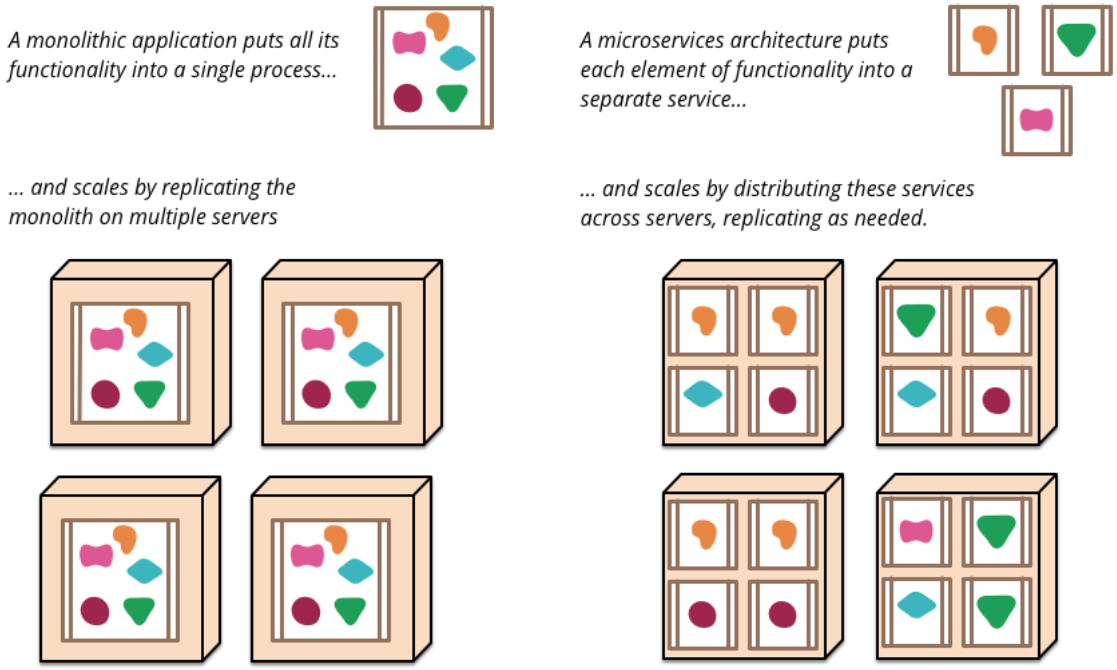

Figure 1: Comparison between the scalability in monolithic and microservices architecture [1].

2.2. Microservices Architecture

Microservices architecture is an advanced design approach where independent services are developed around specific business requirements. Each microservice is responsible for a particular functionality, enabling the creation of complex systems through the integration of modular components [4].

From an external viewpoint, microservices operate as black boxes, delivering business functionality via network endpoints, such as REST APIs. The internal workings of these services are minimally exposed, ensuring that upstream consumers, whether other microservices or external programs, remain unaffected as long as the interfaces remain consistent.

The concept of microservices has been an important subject of research and development for over a decade. Sam Newman identified five key principles that are crucial for understanding the effectiveness of this architectural approach. These principles include independent deployability, designing services around business domains, each service owning its state, and ensuring that the architecture aligns with the organizational structure.

Independent deployability is a core principle of microservices architecture, allowing developers to modify, deploy, and release changes to a microservice without affecting others. Achieving this requires that microservices be loosely coupled, guaranteeing that updates to one service do not necessitate changes in others. This is accomplished by establishing explicit, well-defined, and stable contracts between services.

Building on the idea of independent deployability, modeling microservices around business domains rather than technical layers can further enhance these benefits. Domain-Driven Design (DDD) provides a framework for structuring services to mirror real-world domains, thereby simplifying the introduction of new features and functionalities. When services are aligned with business domains, changes usually involve fewer services, which reduces the complexity and cost of updates. The principles of DDD encourage designing systems that remain flexible and adaptable as business needs change over time.

Another key concept is that each microservice should maintain ownership of its state, interacting with other services only when necessary to obtain data. This separation allows services to manage what data is shared and what remains hidden. By maintaining clear boundaries between internal implementation details and external contracts, this approach minimizes backward-incompatible changes and reduces the potential ripple effects of updates across the system.

Furthermore, aligning the architecture with the organizational structure is essential for the success of microservices. While traditional architectures often mirror the communication patterns within an organization [5], a microservices architecture benefits from structuring teams around business domains. This promotes end-to-end ownership of specific functionalities. With this alignment, teams take full responsibility for their services, from development to production, leading to faster and more efficient development processes.

2.3. Micro Frontend Architecture

The concept of micro frontends was first introduced in the ThoughtWorks Technology Radar at the end of 2016, demonstrating how the benefits achieved from microservices in backend development can also be applied to the frontend [6].

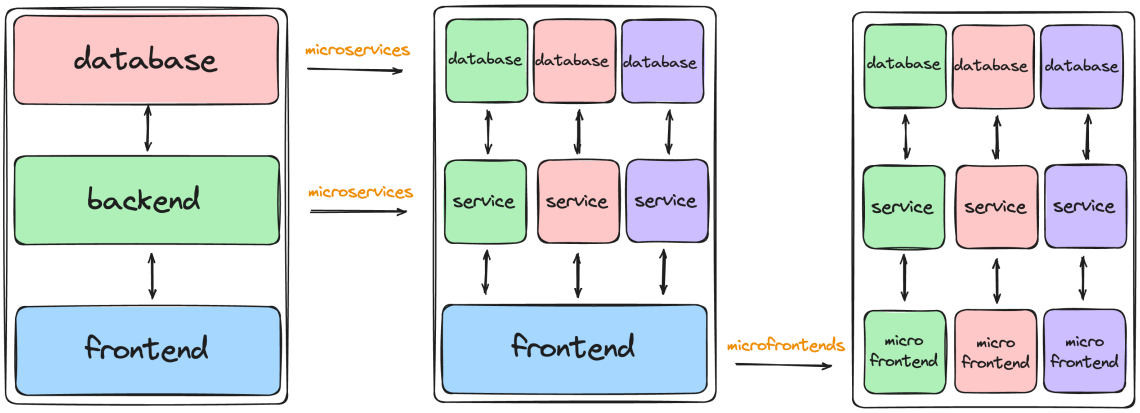

Figure 2: A monolithic application is transformed into one composed of microservices and micro frontends.

The five key concepts of microservices, as they relate to micro frontends, will also be explored in this discussion. An online shop with different sections, each developed by separate teams, will serve as a practical example to provide greater clarity.

Independent deployability allows each section of the frontend to be developed, tested, and deployed independently of other sections. For example, if the team managing the product detail section needs to introduce a new feature or resolve a bug, they can deploy their changes without affecting the checkout or product recommendation sections.

Micro frontends should also mirror the structure of business domains. Building on concepts of DDD, each micro frontend corresponds to a specific business function,

such as a shopping cart, product catalog, or user authentication. This alignment ensures that changes to business logic are contained within a single micro frontend, simplifying updates and making the system more adaptable to evolving business requirements.

Managing its own state within a micro frontend offers several benefits. It helps maintain loose coupling between different parts of the application. For instance, the shopping cart micro frontend exclusively manages what customers add to their cart, while the product catalog micro frontend controls the display of products. This approach simplifies data flow and makes debugging and testing more straightforward, as each micro frontend has clear boundaries for managing its data.

The final key concept is that teams in a micro frontend architecture are organized around business domains, with each team taking full responsibility for the development of their respective micro frontend. This organizational structure minimizes the need for inter-team coordination and enhances the speed of decision-making processes without waiting for input or approval from other teams.

2.4. Domain-Driven Design

Domain-Driven Design (DDD) focuses on the division of a large system into smaller, more manageable services. Introduced by Eric Evans in 2003, DDD provides a structured approach to software development that closely aligns with business goals [7]. Although DDD introduces a wide range of concepts, this summary will concentrate on a few key elements: domain, subdomain, bounded context, and context mapping. These concepts are illustrated through the example of an online shopping application, as shown in Figure 3. The application is structured around four main domains, each containing its subdomains and two bounded contexts.

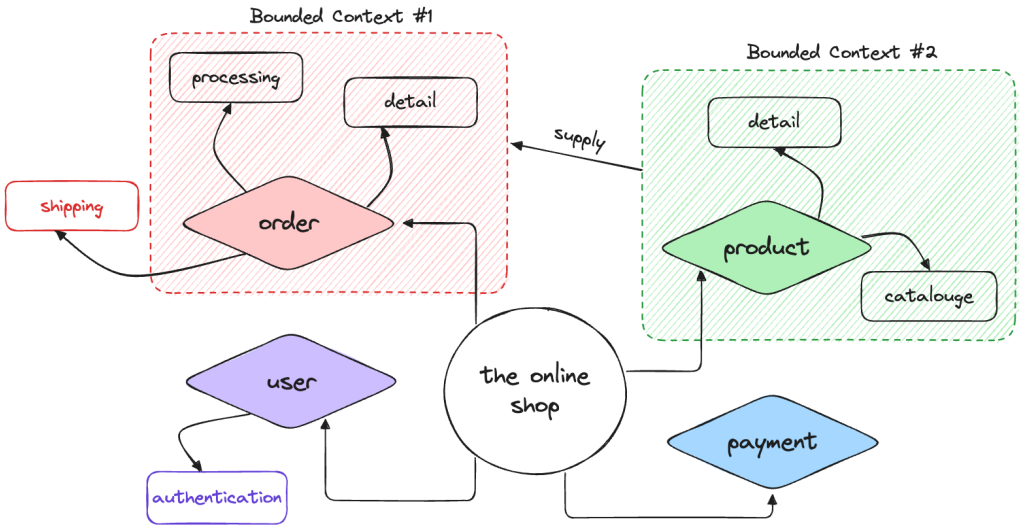

Figure 3: Domain-Driven Design structure of an online shopping application.

The domain represents the specific area of knowledge or activity that a software application is designed to address, including all relevant business logic and rules. In the given example, the domains are order, product, user, and payment.

A subdomain is a smaller, specific part within a domain, categorized into three types: core subdomain, supporting subdomain, and generic subdomain. The core subdomain, which is supported by necessary supporting domains, represents the most important part of the business, delivering its unique value and therefore demanding the highest level of attention and custom development. The generic subdomain, on the other hand, is applicable across multiple domains. In the given example, the product detail subdomain supports order processing, which functions as the core subdomain. Additionally, the user authentication subdomain serves as a generic component, designed to be universally used across the application to identify customers.

A bounded context is a defined boundary that separates subdomains within a specific portion of the domain. Within each bounded context, a unique ubiquitous language is used, which may differ from the languages employed in other contexts. In the example, there are two bounded contexts, both containing a subdomain named “detail,” but with distinct purposes: one is used for displaying order details, while the other is for displaying product details. Despite these distinctions, relationships can exist between bounded contexts, such as when the order domain requires information from the product domain to process an order. These relationships and interactions between different bounded contexts are referred to as context mapping.

Understanding these concepts is essential for applying DDD to micro frontends. Further details on how these concepts are implemented in micro frontends will be discussed in Section 4.

3. State of the Art Review

This chapter is dedicated to identifying instances where micro frontends have been implemented and evaluating the outcomes of these applications. It is organized into two sections: academic literature and industry case studies. The academic literature section examines research that investigates the application and effectiveness of micro frontends across various contexts. The industry case studies section, on the other hand, provides detailed insights into real-world examples, highlighting the practical benefits and challenges encountered by companies that have adopted micro frontend architecture.

3.1. Academic Literature

In their paper, Wang et al. [8] discuss the challenges faced by a monolithic single-page application within an educational management system, specifically issues such as complex logic, high coupling, and difficulties with incremental updates, which are similar to the problems observed at DKLB project. To address these challenges, they propose adopting a micro frontend architecture for East China Normal University's graduate information system, aiming to enhance agile development, service separation, and efficient incremental upgrades. The outcomes of this approach are positive. However, the authors also emphasize the importance of bounded contexts in the design of each micro frontend, which contributes to an improved user experience by maintaining clear functional boundaries.

Perlin et al. [9] explore the application of microservices principles to frontend development by introducing a micro frontend architecture that utilizes multiple frontend frameworks to achieve technological independence and modularity. This approach is validated through a prototype implemented on the Animal Health Defense Platform of Rio Grande do Sul, demonstrating the successful integration of components developed with various frontend frameworks. By employing this strategy, seamless integration and component sharing are achieved. While the case study underscores significant benefits in terms of modularity and flexibility, it also highlights challenges related to interface consistency and operational complexity.

Männistö et al. [10] examine the transformation of a monolithic user interface into a micro frontend solution at Visma, focusing on the work of a small development

team. The primary motivations behind this transition were to enhance customer-specific configurability and reduce operational costs. During the process, the team observed significant improvements in scalability and deployment efficiency. These benefits were realized despite the team's limited size, highlighting the feasibility and advantages of adopting micro frontend architecture even within smaller organizational settings. The study's findings suggest that small teams can confidently pursue the development of frontend applications using a micro frontend approach.

In the field of the Internet of Things (IoT), Mena et al. [11] introduce a progressive web application that utilizes microservices and micro frontends to integrate geospatial data and IoT information. The micro frontend approach allows for the independent development and deployment of UI components, fostering modularity and scalability. This architecture enables the frontend to dynamically adapt to various user contexts, such as location and device type, ensuring a seamless user experience.

Similarly, in the mobile domain, Capdepon et al. [12] present a model-driven engineering approach for migrating monolithic mobile applications to a micro frontend architecture. This method involves re-architecting applications into smaller, more manageable units, achieving comparable benefits in terms of modularity and scalability.

Further advancing this concept, Shakil et al. and Simoes et al. propose a modular architecture based on micro frontend principles, specifically designed to address the unique requirements of industrial applications. Their studies report positive outcomes from this architectural transition, demonstrating the effectiveness of micro frontends in enhancing the composability and adaptability of industrial systems [13], [14].

Lastly, a thorough literature review conducted by Severi Peltonen, Luca Mezzalira, and Davide Taibi, as presented in their 2021 study [15], examines findings from 173 diverse sources, including blogs, articles, conference papers, and journal publications. Their review highlights the advantages of micro frontend architecture in frontend development, such as enhancing team autonomy, accelerating the delivery of new features, and supporting diverse technology stacks. These benefits contribute to easier onboarding processes and improved fault isolation, making micro frontends a valuable strategy for managing large and complex frontend applications. However, the review also brings several challenges, including increased payload size, potential code duplication, complexity in monitoring, and difficulties in maintaining a consistent user experience.

3.2. Industry Case Studies

OTTO, a leading German online retailer, restructured its monolithic system to improve extensibility and maintainability. By adopting a distributed, vertical micro-architecture, now recognized as micro frontend architecture, OTTO successfully divided its application into smaller, independent units aligned with specific business domains. This transformation enhanced system adaptability, development efficiency, and overall performance [16]. Similarly, Galeria Kaufhof, another German retailer, adopted this approach in their "Jump" project, achieving comparable success in improving their system's performance and flexibility [17].

The German bookstore chain Thalia successfully transitioned from a monolithic system to micro frontends, particularly within their eReader-Shop team. For two years, this shift led to a more cohesive team, faster and higher-quality feature rollouts, and greater independence of services. The frequency of deployments increased significantly, with 49 deployments of new services occurring over two months, in contrast to five deployments under the previous system. This approach has enhanced the team's efficiency, transparency, and adaptability, thereby strengthening Thalia's system and streamlining their development processes [18].

Although there are no official documents confirming that SAP, the German software giant, uses micro frontends internally, they have developed a JavaScript framework called Luigi. This framework leverages iframes for implementing micro frontends and offers a comprehensive range of APIs and configuration options to facilitate smooth architectural transformations of applications. Luigi is designed to help developers integrate micro frontends seamlessly into existing systems, providing a flexible and robust solution for modernizing application architectures [19].

Zalando, another leading German online retailer, transitioned to micro frontends through their Mosaic project, breaking down the front end into smaller, isolated components. This shift improved scalability, enabled independent deployments, and empowered team autonomy. However, it initially caused user experience inconsistencies and increased infrastructure complexity. To resolve these issues, Zalando developed the Interface Framework to centralize rendering and ensure platform-wide consistency, refining their micro frontends approach [20]. Similarly, both HelloFresh and IKEA have adopted comparable strategies, underscoring the benefits of micro frontends in the retail sector [21], [22].

4. Decision Framework

This chapter will use the micro frontends decision framework introduced by Luca Mezzalira in his book "Building Micro Frontends". This comprehensive framework focuses on four key areas: definition, composition, routing, and communication [23]. These areas are important and must be thoroughly considered sequentially and decided upon before starting to build a micro frontends system.

To provide a more rounded perspective and deepen the understanding of this framework, this chapter also uses insights from two other important works in the field. "The Art of Micro Frontends" by Florian Rappl [24] offers valuable additional viewpoints on best practices and advanced techniques. Similarly, Michael Geers's "Micro Frontends in Action" [25] provides practical examples and case studies that illustrate the application of the micro frontends framework in real-world scenarios.

4.1. Split Strategies

There are two primary strategies to split a frontend application: horizontally or vertically. In both approaches, a micro frontend is defined to represent a complete subdomain, which can be categorized as core, supporting, or generic, as introduced in Section 2.4. A horizontal split implies a one-to-many relationship between views and micro frontends, where a single view is made up of multiple micro frontends, each responsible for a specific feature within that view. In contrast, a vertical split establishes a many-to-one relationship, where one micro frontend is responsible for managing the functionality of several views.

4.1.1. Horizontal-Split

In this approach, many micro frontends coexist within the same view, each responsible for a specific section of the page. For instance, in a product detail view (as illustrated in Figure 4), three micro frontends managed by the product team, the recommendation team, and the checkout team collaborate to present a unified user interface within a single page.

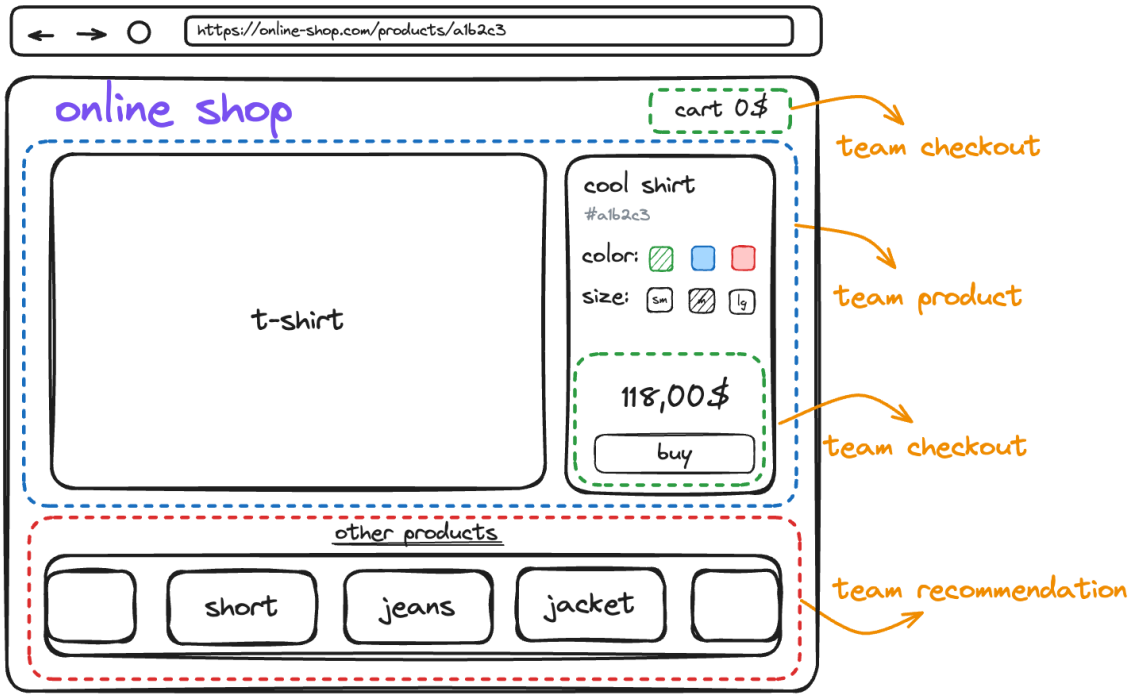

Figure 4: A product detail view composed of three micro frontends.

Horizontal-split architecture is often chosen when there is a requirement for reusable micro frontends across different pages within the application. By supporting granular modularization, this method not only enhances reusability and flexibility but also promotes more efficient resource usage. Additionally, this approach proves particularly beneficial in scenarios where multiple teams are working simultaneously on the application, as it allows for the easy division of subdomains into smaller, manageable units.

Since each micro frontend functions as an isolated component of the user interface, any issues that arise within one section do not affect the entire application. This separation mitigates single points of failure, enhances overall stability, and facilitates more seamless updates and maintenance.

Despite its benefits, implementing a horizontal-split architecture presents considerable challenges. It demands strong governance, regular reviews, and effective communication among teams to ensure that each micro frontend maintains its proper boundaries. Additionally, the number of micro frontends within the same view needs to be limited. Over-engineering can lead to an excessive number of small micro frontends, blurring the line between a micro frontend (a business subdomain) and a component (a technical solution for reusability). This can lead to increased overhead, ultimately outweighing the potential advantages.

4.1.2. Vertical-Split

The vertical-split approach divides an application into multiple slices, with each team managing a specific subset within these slices. For instance, as illustrated in Figure 5, the recommendation team is responsible for the products overview page, which includes displaying new and recommended products to customers. While the product team handles the product detail view, including all specific information about a product.

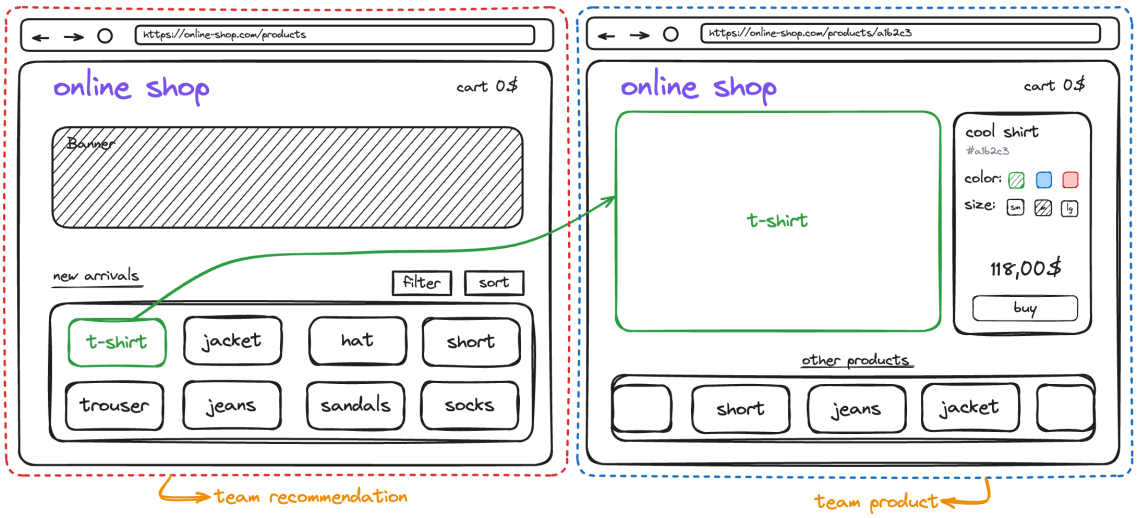

Figure 5: User workflow between two micro frontends.

A key component of this architecture is the application shell, which is responsible for loading and unloading micro frontends. It is the first to be downloaded when the application is accessed and remains a persistent part of the application. The shell typically includes essential elements that appear on every page of the application, such as the navigation bar, footer, or general logic such as a user authentication system.

The vertical-split architecture is particularly beneficial for projects that require a consistent user interface across different sections. By loading the shell first, users experience a more responsive application, as the core interface elements become immediately available while other parts continue to load in the background. Additionally, each team is responsible for managing the complete user experience for their assigned subdomain, leading to a cohesive and unified interface throughout the application.

Moreover, for teams experienced in developing single-page applications, transitioning to a vertical-split micro frontend architecture is relatively straightforward. The tools, best practices, and design patterns from SPA development are directly

applicable, allowing teams to leverage their existing expertise. This continuity reduces the need for extensive retraining and accelerates the implementation process.

Vertical-splits also bring notable challenges that require careful attention. A primary concern is the potential for redundant efforts and code duplication. Without careful management and coordination, individual micro frontends within a vertical-split architecture might independently develop their versions of common functions, such as data fetching or error handling. This redundancy not only increases the maintenance workload but also results in a larger, more complicated codebase. Managing these redundant components can become a complex and time-consuming task, reducing the efficiency that vertical-split architecture is intended to achieve.

Moreover, since a micro frontend represents multiple slices within the application, its failure could potentially impact a large number of users. For instance, in the online shopping application mentioned earlier, if the product detail page encounters an issue or becomes unavailable, customers will be unable to view products and make purchases. This could result in significant sales losses and a negative user experience.

4.2. Composition and Routing

There are three primary composition types: server-side composition, edge-side composition, and client-side composition. The strategy used to divide the application impacts how different parts of the frontend are assembled into a view. Vertical-split approach supports only client-side composition, whereas horizontal-split approach can accommodate all three types. These composition methods correspond to three routing approaches, determined by where the routing logic is executed: server-side routing, edge-side routing, and client-side routing. Before exploring the details of each composition and routing mechanism, the difference between build-time integration and runtime integration will first be discussed.

4.2.1. Build-time vs. Runtime Integration

Build-time integration refers to the process where micro frontends are integrated during the build phase. In this approach, individual micro frontends are combined into a single deployable bundle before the application is served to the user. While this method allows for independence during the development stage, it results in

coupling at the release stage. Consequently, even a minor change in a single micro frontend may necessitate recompiling the entire application, creating a modular monolith architecture that contradicts the fundamental principles of micro frontend architecture.

In contrast, runtime integration involves assembling the micro frontends dynamically when the application is loaded or accessed by the user. This approach helps avoid issues such as lock-step deployment, thereby maintaining the independence of each micro frontend. Given these considerations, runtime integration is generally preferred over build-time integration to prevent unnecessary coupling.

4.2.2. Server-Side

The server-side approach is a web development technique where the server manages routing logic and assembles different parts of an application before delivering a completed page to the client. This method is particularly effective when combined with a horizontal-split architecture as this combination offers greater flexibility in how different sections are composed.

This approach brings several advantages. By delivering a fully-rendered HTML document, the browser can display the page quickly, reducing the computational load. This is especially beneficial for users with lower-powered devices, ensuring a faster and smoother content appearance and significantly enhancing the overall user experience. Additionally, the server-side approach is advantageous for SEO purposes. Search engines can easily crawl and index complete pages, leading to better rankings for websites using server-side composition.

However, during high-traffic periods, the server must handle multiple requests and assemble pages on the fly, which can strain server resources. This can result in slower navigation between pages or, in extreme cases, cause the server to freeze or crash. While a Content Delivery Network (CDN) can help mitigate some of this load by serving cached pages, there is a risk of delivering outdated or non-personalized data if the application relies heavily on dynamic content.

Additionally, managing composition and routing on the server side can introduce latency, particularly when the server needs to fetch data from multiple sources before assembling the final page. This added delay can negatively impact the user experience. Furthermore, server-side integration lacks technical isolation within the browser, which can lead to issues such as CSS class name conflicts or global variable collisions in JavaScript.

4.2.3. Edge-Side

As an alternative to the server-side method, the edge-side composition and routing occur at the edge of the network, typically within a CDN. This approach enables micro frontends to be dynamically assembled as close to the user as possible.

Edge networks can handle high traffic more efficiently than central servers by distributing the load across multiple locations. When implemented correctly, this process can be more lightweight than server-side integration, reducing latency and delivering content more quickly to the end user.

However, one of the main challenges with edge-side rendering is ensuring consistent updates across all edge nodes. Inconsistencies in synchronization can result in users in different locations, such as Germany and Vietnam, seeing different versions of the content if updates are not properly coordinated.

It is also important to note that the use cases for edge-side composition are limited and often used along with other patterns, such as server-side composition. While this method is particularly effective for applications requiring rapid delivery and rendering times, its implementation can be complex. The edge-side composition demands advanced infrastructure and expertise to manage effectively, which is why relatively few developers choose to implement micro frontends using this approach, as indicated by a survey on micro frontends conducted in late 2023 [26].

4.2.4. Client-Side

Client-side composition approach can be used in both vertical-split and horizontal-split architectures, where the application view is assembled in the browser and routing logic is also handled on the client-side. This approach is particularly useful for building applications that require significant interactivity and dynamic content.

Since components are loaded and updated in real-time, users can interact with the application more fluidly, leading to a more engaging experience. Additionally, offloading the composition task to the client reduces the processing burden on the server. This allows the server to handle more requests or focus on other critical tasks, potentially improving overall system performance.

By handling routing in the browser, once the initial application is loaded, all subsequent navigations are managed by the client. This results in faster transitions between views, as there is no need to reload the entire page from the server. Ad-

ditionally, this routing strategy supports complex routing structures such as flat routes and nested routes.

- Flat Routing: This is a straightforward approach where each route corresponds directly to a specific view without any hierarchy (Figure 6, left). This creates a one-to-one relationship between the URL and the view, making flat routing simpler to implement, manage, and understand.
- Nested Routing: This allows for more complex route structures by enabling routes to be nested within other routes. A parent route can have one or more child routes, allowing for hierarchical navigation within the application (Figure 6, right). While nested routing offers greater flexibility, it also adds complexity to the routing logic and requires more careful management of route states.

However, there are challenges associated with client-side composition. Initial load times can be slower because the browser needs to fetch and render multiple components, resulting in a delay before the user sees the fully assembled interface. Additionally, since the composition task is handled by the client, it can consume more resources on the user's device, which could be an issue for users with less powerful hardware or slower internet connections.

Moreover, when SEO is a major concern, alternative composition methods should be considered. While Googlebot is capable of crawling and indexing client-side rendered pages, not all search engine bots can execute JavaScript. This limitation means that relying on client-side rendering could result in incomplete indexing, potentially reducing the site's visibility across different search engines [27].

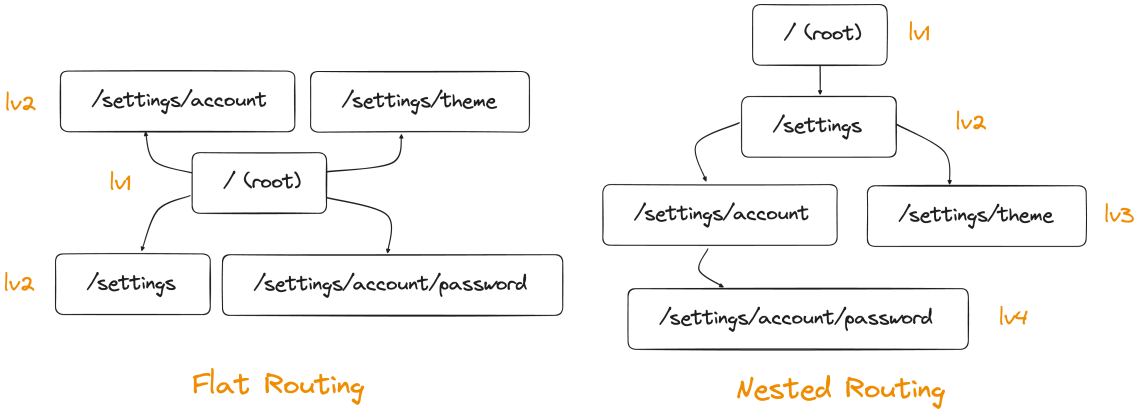

Figure 6: Flat and nested routing strategies in client-side routing.

4.2.5. Universal

It is possible to combine client-side and server-side composition to create a more advanced and efficient architecture. In this hybrid approach, the initial view of the application is composed on the server side, which improves performance and reduces the time it takes for the user to first see the web page. Once the initial content is delivered, interactive components are rendered on the client side, enabling dynamic user interactions. This method leverages the strengths of both server-side and client-side composition, ensuring faster initial load times while preserving the engaging, interactive experience that client-side rendering offers. However, implementing this approach is complex due to the need for rendering on both the server and client sides, and it is typically only feasible within a horizontal-split architecture.

4.3. Communication

Micro frontends are designed to be independent, modular, and loosely coupled, ideally operating without the need for communication with each other. However, in practice, especially in horizontal-split applications, multiple micro frontends within a single view often need to interact to ensure the correct actions are displayed when a user interacts with the page or to exchange data. This requirement for communication can increase coupling and risk undermining the principle of independent deployment if not implemented carefully. Below are common strategies for enabling communication between micro frontends, each with its specific use cases and considerations:

4.3.1. Query Parameters

Using query parameters to pass state through the URL is an effective method for enabling communication between micro frontends. This approach preserves the independence of each micro frontend by limiting their interaction to the URL. The visibility of state in query strings allows users to bookmark or share the page, enhancing the overall user experience. However, this method is most appropriate for handling simple, serializable data types. Overusing query parameters can lead to excessively long and unwieldy URLs. Therefore, this technique should be re-

served for simple and transient state sharing between micro frontends, keeping URLs concise and meaningful to avoid unnecessary clutter.

For example, as shown in Figure 5, when navigating from the product catalog page at `/products` to the product detail page with the URL `/products?id=a1b2c3`, the product team can extract the `id` from the URL and use it to fetch the corresponding product information.

4.3.2. Web Storage & Cookies

Web storage and cookies both store data in the browser, allowing any micro frontend to access and share data efficiently. There are two types of saved data: temporary and persistent.

Temporary data includes session storage and session cookies, which are accessible only for the duration of the browser session and are deleted when the browser is closed. Persistent data, in contrast, includes local storage and persistent cookies. Data stored in local storage remains available across sessions and browser tabs and persists until it is explicitly deleted by the user or the application. Persistent cookies, on the other hand, stay on the user's device for a predefined period, such as one hour or one week, before automatically expiring [28], [29].

However, personalized or sensitive data, such as information from an authentication micro frontend, stored in web storage or cookies must be encrypted to ensure security. Careful management is also required to prevent conflicts and ensure data integrity. By using web storage and cookies effectively, micro frontends can maintain their independence while sharing necessary information, thereby enhancing the overall functionality and user experience of the application.

For example, as illustrated in the figure below, two micro frontends can communicate through cookies during a login process. Once the user successfully signs in, the authentication token is encrypted and stored in a cookie. The micro frontend by the team user can then retrieve and decrypt this token to verify the user's authentication status and display the appropriate user data.

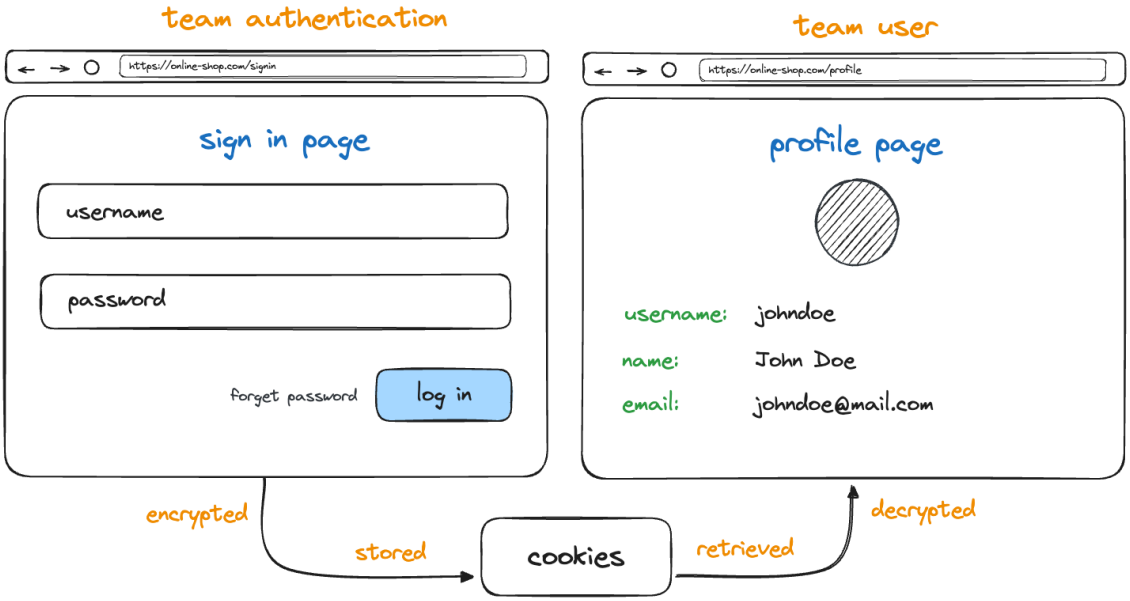

Figure 7: Two micro frontends communicate through cookies during a login process.

4.3.3. Event Bus

Another strategy is the event bus, which enables communication between different micro frontends without requiring direct awareness of each other. The event bus operates through two primary mechanisms: event emission and event subscription.

During the event subscription phase, a micro frontend subscribes to specific events on the event bus. The event bus then notifies the subscribing micro frontend whenever another micro frontend emits one of the subscribed events to the bus. For example, in Figure 4, when a user selects a different shirt color, the product team emits an event to notify the checkout team that the shirt configuration has changed. As a result, the checkout team will update the price accordingly.

4.3.4. Global State Manager

A global state manager becomes especially useful when client-side composition is used. Most frontend frameworks offer state manager libraries, such as Redux[1] for React.js or Pinia[2] for Vue.js. Similar to communication methods like web storage

[1]https://redux.js.org/
[2]https://pinia.vuejs.org/

or an event bus, these tools enable components within a view to communicate more easily and effectively. By centralizing the state, a global state manager allows components to access and update shared states reactively and seamlessly, simplifying the data flow and ensuring a more consistent user experience. However, it relies on web storage to persist data, as the global state is reset upon reloading the page.

5. Micro Frontend Implementation Approaches

Building on the Decision Framework outlined in the previous chapter, this section presents several commonly adopted approaches for implementing micro frontends. The effectiveness and challenges of these approaches often depend on the composition strategy they are paired with. As a result, general benefits and drawbacks will not be discussed here, except in cases where a specific approach has distinct characteristics.

For example, approaches involving server-side and edge-side composition will be demonstrated within the context of a horizontal-split architecture, while those using client-side composition will be examined through the perspective of a vertical-split architecture.

5.1. Server-Side Includes

Server-Side Includes (SSI) is a server-side scripting language often used in a server-side composition approach, where web pages are constructed on the server by fetching content from various micro frontends before delivering the final page to the user. SSI accomplishes this by providing a set of specific directives within an HTML file, which the server processes to execute commands such as setting variables, printing the current date and time, or including common elements from other files, like headers or footers, within the page ([30]). This capability makes SSI particularly useful for maintaining consistency across multiple pages of a website.

However, SSI's utility is generally limited to simpler tasks, as it lacks the flexibility and power required for more complex website architectures. While SSI is effective at including static components across multiple pages, it is not designed to support dynamic interactions between components within a single page. Since page composition occurs on the server side, any communication between different micro frontends within the view must be routed through the server, typically using REST APIs or similar server-side communication methods. Consequently, SSI is better suited for basic page assembly tasks rather than for scenarios that

demand complex, interactive user interfaces or real-time communication between components.

```
<!-- http://header.mfe/index.html -->
<html>
  <header>Header</header>
</html>
```

```
<html>
  <body>
     <!--#include virtual="http://header.mfe/index.html" -->
     <!--#include virtual="http://catalogue.mfe/index.html" -->
     <!--#include virtual="http://footer.mfe/index.html -->
  </body>
</html>
```

Listing 1: An example of using Server Side Includes.

Additionally, several frameworks are specifically designed to implement micro frontend architectures in combination with server-side composition, such as OpenComponents[3], OneApp from American Express[4], Mosaic from Zalando[5], and Podium[6]. These frameworks offer more robust solutions for developing modular, scalable frontend applications.

5.2. Edge-Side Includes

The primary purpose of Edge-Side Includes (ESI), a markup language, is to enable edge-side composition, which allows web pages to be constructed from fragments directly at the edge of the network. The key difference between ESI and SSI lies in where the page assembly occurs: ESI operates at the network edge, typically within a CDN, whereas SSI performs this function on the server-side [31].

However, implementations can vary significantly, with some CDNs not supporting ESI at all. In environments where ESI is not natively supported, tools like nginx or Varnish can be employed to mimic ESI's functionality by providing similar edge-

[3]https://opencomponents.github.io/
[4]https://github.com/americanexpress/one-app
[5]https://www.mosaic9.org/
[6]https://podium-lib.io/

side processing capabilities. These tools can intercept requests and dynamically assemble content at the edge. Furthermore, ESI shares some of the same disadvantages as SSI, such as being more suitable for simple static websites and offering limited communication capabilities between components.

```
<!-- http://cdn.mfe/header.html -->
<html>
  <header>Header</header>
</html>
```

```
<html>
  <body>
     <esi:include src="http://cdn.mfe/header.html" />
     <esi:include src="http://cdn.mfe/catalogue.html" />
     <esi:include src="http://cdn.mfe/footer.html" />
  </body>
</html>
```

Listing 2: An example of using Edge Side Includes.

5.3. iframe

An iframe is an inline frame embedded within a webpage that allows the loading of a separate HTML document from different sources. It offers one of the highest levels of isolation within a browser, as it maintains its own context and resources independently from the parent document [32]. Because of this strong isolation, communication between iframes often relies on the `postMessage` method [33]. Additionally, iframes are advantageous due to their ease of implementation, making them a common and intuitive choice when considering micro frontend architectures.

Despite the strong isolation benefits provided by iframes, their performance is often criticized by the community for being suboptimal and CPU-intensive, particularly on websites that use multiple iframes. This performance issue, combined with the difficulty of making iframes easily indexable by search engine crawlers, limits their suitability primarily to desktop or intranet applications, as demonstrated by Spotify's use of iframes in their desktop apps [34]. Additionally, accessibility concerns arise with iframes. While they can visually integrate seamlessly into a

web application, they essentially represent separate small pages within a single view, which can pose significant challenges for accessibility tools like screen readers. These tools must navigate multiple documents, hierarchical information, and varying navigation states within a single page, complicating the user experience for individuals with disabilities.

This method is a type of client-side composition. As explained in Section 4, this composition strategy starts with the browser downloading a shell application, which manages the loading and unloading of various micro frontends. As illustrated in the figure below, the shell application determines the appropriate HTML file path based on the current URL and assigns it as the source of the iframe element.

```
<!-- http://home.mfe/index.html -->
<html>
  <body>
    <h1>Home</h1>
    <!-- other elements -->
  </body>
</html>
```

```
<html>
  <body>
    <iframe src="" />

    <script>
      const routes = {
        '/':         'http://home.mfe/index.html',
        '/product':  'http://product.mfe/index.html',
      }
      const src = routes[window.location.pathname]
      const iframe = document.querySelector('iframe')
      iframe.src = src
    </script>
  </body>
</html>
```

Listing 3: An example of using iframe.

5.4. Web Components

Web components are a collection of web platform APIs that enable developers to create reusable and encapsulated custom elements. These components are based on three key specifications: Custom Elements, Shadow DOM, and HTML Templates [35].

- Custom Elements: This set of JavaScript APIs allow developers to define their own HTML elements with custom behaviors. Once defined, these elements can be used just like standard HTML tags.
- Shadow DOM: Another set of JavaScript APIs provides encapsulation by creating a hidden context, a shadow DOM, that includes the internal structure, styles, and behavior of the component. This encapsulation ensures that the component is isolated from the rest of the main DOM, preventing style and script conflicts.
- HTML Templates: This feature allows developers to define reusable HTML fragments that are not rendered during the initial page load. These templates can be reused as needed throughout the application.

While web components provide substantial benefits, they also present certain challenges. The concept of web components has been around for some time, however, full support is only available in modern browsers. To maintain compatibility with older browsers, developers often need to rely on polyfills[7]. Additionally, the use of custom elements and the shadow DOM within web components differs from traditional frontend development practices, which may introduce a learning curve for developers who are not yet familiar with these concepts.

Web components are primarily intended for client-side composition, where they are rendered and executed within the browser. However, they can also be integrated with server-side composition by having the server load other parts of the HTML, while the web components are executed after the page has been loaded, allowing for a hybrid composition strategy.

[7]https://remysharp.com/2010/10/08/what-is-a-polyfill

```
// http://home.mfe/index.js
class HomeApp extends HTMLElement  {
  constructor(){
    const shadowRoot = this.attachShadow({ mode: 'open' })
    const heading = document.createElement('h1')
    heading.textContent = 'Home'
    shadowRoot.appendChild(heading)
  }
}
customElements.define('home-app', HomeApp)
```

```
<html>
  <head>
    <script src="http://home.mfe/index.js"></script>
    <script src="http://product.mfe/index.js"></script>
  </head>
  <body>
    <div id="root">
      <home-app /> <!-- <h1>Home</h1> -->
    </div>

    <script>
      const routes = {
        '/':         'home-app',
        '/product':  'product-app',
      }
      const root = document.getElementById('root')
      const elementName = routes[window.location.pathname]
      const element = document.createElement(elementName)
      root.appendChild(element)
    </script>
  </body>
</html>
```

Listing 4: An example of using Web Components.

5.5. Module Federation

Module Federation, introduced in Webpack 5, is a feature of this popular JavaScript bundler that enables different parts of an application to be treated as separate modules. These modules can be shared and used by other parts of the application at runtime [36], [37]. There are two types of modules:

- Exposed Module: Also referred to as a remote application, this is a module that is made available for other applications to consume. It can change its behavior at runtime and is typically defined to provide resources such as a component library or utility functions to other parts of the application.
- Consuming Module: Known as the host application, this module can utilize exposed modules without needing to bundle them directly into its codebase. As a result, if the exposed module is updated, the consuming application automatically integrates the latest version.

Module Federation is an approach that can be seamlessly integrated with both vertical and horizontal splitting strategies, as well as with client-side or server-side composition. In a survey on micro frontends conducted in late 2023 [26], Module Federation appeared as the most adopted approach, highlighting its effectiveness as a solution in modern web development.

Moreover, enabling code sharing across different parts of an application, significantly reduces duplication and decreases the overall size of the application bundle compared to iframe or Web Components. For instance, if multiple micro frontends rely on the same library, they can all access a single shared instance rather than bundling it separately in each module.

However, Module Federation introduces certain complexities, particularly in managing the versions of shared modules across different applications. This process can be complex and requires careful configuration, especially in environments with multiple modules or complex dependency structures. The challenge is further expanded when dealing with commonly used modules that are widely consumed by other parts of the application. These modules must be cautiously managed and monitored to avoid becoming a single point of failure, as any changes to them can have widespread effects across the entire application ecosystem.

```vue
<!-- home/App.vue -->
<template>
  <h1>Home</h1>
</template>
```

```
// home/webpack.config.js
export default defineConfig({
  plugin: [
     new ModuleFederationPlugin({
        name: 'remote',
        exposes: { './App': './src/App.vue' },
        shared: ['vue'],
      }),
  ]
})
```

```vue
<!--  host/src/App.vue -->
<script setup>
import App from 'remote/App'
</script>

<template>
  <App /> <!-- <h1>Home</h1> -->
</template>
```

Listing 5: An example of using Module Federation.

6. Experiment

In this chapter, a detailed experiment will be conducted, simulating the entire development cycle of a web application using micro frontend architecture across several stages: planning, setup, implementation, build, deployment, testing, as well as Continous Integration (CI) and Continous Deployment (CD). Each stage will be closely monitored to gather comprehensive insights into this architecture. Additionally, a single-page application version of the same application, derived from the micro frontends version, will be implemented to compare performance metrics and evaluate the overall system behavior of both approaches in the next chapter. The primary goal is to provide a deep understanding of the micro frontend approach, highlighting its potential benefits and drawbacks. Furthermore, optimizations for enhancing the developer experience will be discussed after the development cycle.

Note that the code examples presented in this chapter may differ from the actual code. For accurate and precise code, refer to the GitHub repository at [38].

6.1. Planning Stage

In this initial stage of the experiment, the current state of the backend for the Deutsche Klassenlotterie Berlin (DKLB) application will be briefly reviewed. Following this, each step in the decision framework outlined in the Section 4, will be applied. These steps will collectively provide valuable insights for the upcoming setup stage.

6.1.1. Backend

Fortunately, the backend architecture of the DKLB is already organized as microservices, following Domain-Driven Design (DDD) principles. In this context, domains such as user, games, and cart are clearly defined. For instance, the games domain is further divided into multiple core subdomains based on specific games. The `/lotto6aus49/**` path interacts exclusively with microservices dedicated to the Lotto game, while the `/eurojackpot/**` path engages with APIs associated

with the Eurojackpot game. These distinct separations emphasize that each core subdomain focuses on the unique functionalities within the overall application.

However, to ensure the experiment is comprehensive, a mocked server that simulates the backend will be set up to handle requests from the micro frontends. While this server could be built using a web framework in any programming language, a JavaScript-based framework will be used to minimize additional setup efforts and maintain consistency with the frontend technologies.

6.1.2. Horizontal-Split

The principles of DDD offer a strong foundation for implementing a vertical-split strategy for micro frontends. In this approach, each micro frontend aligns with a specific subdomain, such as a particular game, effectively mirroring the microservices architecture established on the backend.

However, to increase the system's flexibility, a horizontal-split strategy will be adopted instead. While the majority of micro frontends will continue to follow a vertical-split, focusing on specific game subdomains, certain functionalities or cross-cutting concerns will be shared across multiple micro frontends. For example, although the homepage and Lotto game are developed in separate micro frontends, the component responsible for displaying Lotto quotes can be exposed and reused within the homepage micro frontend. This adoption of a horizontal-split strategy ensures that the architecture remains adaptable and responsive to diverse requirements.

6.1.3. Module Federation

Module Federation has been chosen for its significant potential in the development process. It supports both horizontal and vertical splitting of the application, as well as client-side and server-side composition, providing the flexibility needed to accommodate future requirements with ease. Furthermore, the use of a single frontend framework across the entire application eliminates the dependency management challenges typically encountered with Module Federation, making it an even more advantageous choice.

In this experiment, client-side composition will be chosen because the potential development teams at MULTA MEDIO are already familiar with SPA development. Additionally, if the application later requires enhanced SEO, faster load

times, or improved performance, transitioning from client-side to server-side composition will be straightforward, given that Module Federation already supports this capability. Further details regarding client-side routing will be discussed during the implementation stage.

For enabling communication between micro frontends, any methods outlined in Section 4.3 will be effective. These methods can be easily opted in or out, without the need for upfront planning.

6.2. Setup Stage

The setup stage will concentrate on selecting the appropriate tools and technologies required for the new frontend architecture. This involves choosing frameworks and libraries, as well as setting up deployment environments, and defining the project structure.

6.2.1. Tools for Development

- Vue.js: It is a progressive JavaScript framework used for building user interfaces. It is highly adaptable, supporting the creation of both simple and complex applications through its reactive and component-based architecture. This design facilitates the development of a modular and scalable frontend [39].
- Tailwind CSS: It is a utility-first CSS framework designed to facilitate the rapid development of custom user interfaces. By utilizing utility classes, it enables developers to style elements efficiently, minimizing the need for extensive custom CSS and preventing CSS class collisions, particularly in the context of micro frontend architecture. This methodology results in cleaner, more maintainable code, aligning well with Vue.js's modular structure [40].
- Vite: It is a modern build tool that significantly enhances the development experience. It provides a fast and efficient setup, offering features like instant server start, hot module replacement, and optimized build processes. This tool integrates seamlessly with Vue.js and Tailwind CSS, improving development speed and efficiency, making it an ideal choice for modern web projects [41]. Also, there is a plugin for Vite that enables the Module Federation feature for Vite [42].
- ElysiaJS: It is a web framework that enables developers to set up routes for handling different HTTP requests, making it ideal for building Restful APIs.

With its robust set of features, ElysiaJS allows for the efficient and maintainable development of applications, effectively mimicking a backend server to serve API endpoints in this experiment [43]. However, since this is primarily for simulation purposes, other web frameworks like Express.js or Fastify could also be used effectively.

6.2.2. Tools for Deployment

- Docker: It is a platform that enables developers to package applications into containers, ensuring consistency across different environments. These containers encapsulate all the necessary components, such as code, runtime, libraries, and settings, making deployment and scaling straightforward and reliable [44].
- Nginx: It is a high-performance web server and reverse proxy server known for its speed, stability, and low resource consumption. Nginx is widely used for serving static content, load balancing, and as a reverse proxy for distributing traffic across multiple servers. It is particularly favored for its ability to handle a large number of concurrent connections efficiently [45].
- GitHub Actions: It is a powerful automation tool integrated with GitHub repositories. It allows developers to create workflows for continuous integration and continuous deployment. With GitHub Actions, the deployment process can be automated, including linting, running tests, building Docker images, and deploying, ensuring a consistent and efficient pipeline from development to production [46].

6.2.3. Tools for Testing

- Vitest: It is a highly efficient testing framework built on top of Vite, designed to facilitate the writing and execution of unit tests. By utilizing Vitest, developers can ensure that each component behaves as intended, helping to maintain the overall reliability of the software [47].
- Playwright: It is an essential tool for end-to-end testing, addressing aspects of application quality that go beyond what unit tests with Vitest can achieve by allowing for comprehensive testing of the entire application, simulating real-world user interactions across different browsers. Playwright helps to identify issues that might only arise when the entire system is in use, making it an

important tool for maintaining the overall quality and stability of a web application [48].

6.2.4. Monorepo Strategy

In software development, there are two repository strategies: monorepo and polyrepo. A monorepo architecture stores code for multiple projects using a single repository. For example, a monorepo repository contains three folders, one for a web app project, one for a mobile app project, and one for a server app project. In contrast, a polyrepo approach uses multiple repositories for each project [49].

In this experiment, a monorepo strategy will be employed. This approach involves storing all micro frontends, a UI library, toolings, and a server application in a single repository, simplifying the setup stage, especially within the scope of this experiment.

Aligning with the monorepo strategy, the project structure is designed to ensure clarity and scalability. The structure is organized into several key directories, each serving a specific purpose. Below is an overview of the project structure:

```
.
├── apps
│   ├── home
│   ├── lotto
│   ├── shell
│   └── ...
│
├── packages
│   ├── ui
│   ├── mfe-config
│   └── ...
│
├── server
│
├── e2e
│
├── tools
│   ├── tailwindcss
│   └── ...
│
└── ...
```

- apps: This directory contains the host, known as shell application (`shell`), which integrates and manages remotes, referred to as micro frontends, such as `home` and `lotto`. Each remote, along with the host, is developed and maintained within its own subdirectory.
- packages: Here stores shared logic and resources, such as the `ui` library or `mfe-config`, an overview configuration file for all applications. These shared packages ensure consistent styling and functionality throughout the project.
- server: This folder houses the server application, which acts as a simulated backend, processing requests from the micro frontends.
- e2e: Here are end-to-end tests for the application, which are an important part of the continuous integration pipeline.

Listing 6: Project structure in the experiment.

- tools: This directory holds based configurations for development dependencies such as Tailwind CSS.

6.2.5. App Configurations

- For host application (`shell`)

In the vite configuration shown in Listing 7, `@originjs/vite-plugin-federation` plugin is used to establish the host application running on port `8000`. This host is configured to have two remote applications, `home_app` and `lotto_app`, which operate on ports `8001` and `8002`, respectively. The configuration also includes the `shared: ['vue']` option, ensuring that the Vue package is shared between the host and the remote applications.

```
// apps/host/vite.config.ts
import federation from '@originjs/vite-plugin-federation'
export default defineConfig({
  plugins: [
    federation({
      remotes: {
        'home_app': 'http://localhost:8001/assets/remoteEntry.js',
        'lotto_app': 'http://localhost:8002/assets/remoteEntry.js'
      },
      shared: ['vue'],
    }),
  ],
  server: { port: 8000 }
})
```

Listing 7: Vite configuration for the host application.

- For remote applications (`home` and `lotto`)

As outlined in the configuration for the host application (Listing 7) the `home_app` is configured to run on port `8001`, while the `lotto_app` is set to run on port `8002`. Both remote applications also use the `@originjs/vite-plugin-federation` plugin to expose their respective `App` components from their source directories (Listing 8). These `App` components can later be imported and displayed, for example, using `import HomeApp from 'home_app/App'`.

```
// apps/home/vite.config.ts
import federation from '@originjs/vite-plugin-federation'
export default defineConfig({
  plugins: [
    federation({
      name: 'home_app',
      exposes: { './App': './src/App.vue' },
      shared: ['vue'],
    }),
  ],
  server: { port: 8001 }
})
```

```
// apps/lotto/vite.config.ts
import federation from '@originjs/vite-plugin-federation'
export default defineConfig({
  plugins: [
    federation({
      name: 'lotto_app',
      exposes: { './App': './src/App.vue' },
      shared: ['vue'],
    }),
  ],
  server: { port: 8002 }
})
```

Listing 8: Vite configuration for the home and lotto applications.

- For UI library

To prevent a single point of failure that could potentially bring down the entire application if the UI library fails, the UI will be bundled within each micro frontend during the compile time, rather than being deployed as a separate micro frontend. This approach ensures that all essential base elements are packaged together, thereby simplifying the management of multiple deployments and minimizing the risk of version inconsistencies. As illustrated in Listing 9, the UI library will be built using the ES module format. Additionally, the Vue package is excluded from the build, as it is already present in both the host and remote applications.

```
// apps/host/vite.config.ts
export default defineConfig({
  build: {
    lib: {
      entry: 'src/index.ts',
      fileName: 'index',
      formats: ['es'],
    },
    rollupOptions: {
      external: ['vue'],
      output: { globals: { vue: 'Vue' } },
    }
  }
})
```

Listing 9: Vite configuration for UI library

6.2.6. Tailwind CSS

To ensure consistent styling across all applications, a base Tailwind CSS configuration has been established, as illustrated at the top of Listing 10. This setup allows Tailwind CSS to scan all Typescript and Vue files to generate the required styles. The `preflight` option, responsible for generating reset CSS rules, is enabled exclusively in the shell application, which helps minimize the amount of CSS that users need to download. Furthermore, a specific set of colors has been defined to maintain a uniform color scheme across the applications. At the bottom of Listing 10 is the `tailwind.config.ts` file for the shell application, which extends this base configuration.

```
// tools/tailwind/index.ts
export default {
  content: ['src/**/*.{vue,ts}'],
  corePlugins: {
    preflight: false,
  },
  theme: {
      colors: {
        primary: '#d22321',
        secondary: '#c5c5c5'
      }
  }
}
```

```
// apps/shell/tailwind.config.ts
import config from '@dklb/tailwind'
export default {
  content: config.content,
  corePlugins: {
    preflight: true,
  },
  presets: [config],
}
```

Listing 10: The default Tailwind CSS configuration, along with its extended version.

6.3. Implementation Stage

Following the selection of the required development tools, this stage focuses on implementing the host application, the two micro frontends (`home` and `lotto`), the UI library, and the server application. Additionally, a routing issue will be identified during this process, and a solution will be developed to address it.

A quick note: Vue Router[8], the official routing library for Vue.js, is used in this experiment to manage routing.

6.3.1. Host Application

The host application should be simple and lightweight, including elements that remain consistent across all pages, such as the navigation menu and footer. For the main content area, the `RouterView` component from Vue Router is utilized as a slot, responsible for loading the appropriate registered component based on the current URL state.

```
<!-- apps/shell/App.vue -->
<template>
  <TheNav />
  <main
    <RouterView />
  </main>
  <TheFooter />
</template>
```

Listing 11: App component of the host application.

After defining the entry component `App.vue`, all necessary routes will be registered as illustrated in Listing 12. The first route is associated with `home` micro frontend for the path `/`, representing the homepage. The second route, mapped to `lotto` micro frontend, corresponds to the path `/lotto6aus49`, where users can participate in lotto games. If no path matches the specified routes, the user is redirected to an error page, which is handled by the `Error.vue` component.

[8]https://router.vuejs.org/

Vue Router supports lazy loading of components using the promise syntax. For example, the syntax `component: () => import('home_app/App')` means that the `App` component of the `home` application is only loaded when the user navigates to the homepage. This optimization reduces the amount of JavaScript that needs to be downloaded initially, improving page load times.

```
// apps/shell/router.ts
const router = createRouter({
  routes: [
    {
      path: '/',
      component: () => import('home_app/App'),
    },
    {
      path: '/lotto6aus49',
      component: () => import('lotto_app/App'),
    },
    {
      path: '/:pathMatch(.*)*',
      component: () => import('./pages/Error.vue'),
    },
  ],
})
```

Listing 12: Router configuration of the host application.

6.3.2. Micro Frontends

The implementation of each micro frontend is straightforward and aligns with the development of a normal single-page application. For instance, the `App.vue` component in the `home` micro frontend might contain a simple heading displaying “Homepage”. When a user navigates to the homepage, Vue Router loads this template into the `App` component of the `shell` application, producing the result shown below.

```vue
<!-- apps/home/App.vue -->
<template>
  <h1>Homepage</h1>
</template>
```

```vue
<!-- apps/shell/App.vue -->
<template>
  <TheNav />
  <main>
    <h1>Homepage</h1>
  </main>
  <TheFooter />
</template>
```

Listing 13: The `App` component of the `shell` application after the `home` is loaded.

6.3.3. UI Library

The UI library must be designed to be minimal, highly extensible, and independent of any specific location within the application, ensuring its effectiveness and usability across various parts of the system.

As shown in Listing 14, a basic `UiButton` component is implemented as a simple HTML button element with predefined Tailwind CSS classes and no context-specific logic. If the home micro frontend requires a customized button, it can create a wrapper around this component to extend its styles, as demonstrated by the `HomeButton` component.

```vue
<!-- packages/ui/UiButton/UiButton.vue -->
<template>
  <button class="inline-flex text-sm uppercase">
    <slot />
  </button>
</template>
```

```vue
<!-- apps/home/components/HomeButton.vue -->
<template>
  <UiButton class="bg-primary text-white">
    <slot />
  </UiButton>

  <!-- will be rendered as below -->
  <button class="inline-flex text-sm uppercase bg-primary text-white">
    <slot />
  </button>
</template>
```

Listing 14: The `UiButton` component and its wrapper `HomeButton`.

6.3.4. Server Application

The server application is configured to listen on port `3000` and only accepts requests originating from port `8000`, where the host application is running. It verifies the request's origin, setting `authorized` to true or false based on whether the origin is `localhost:8000`, and configures CORS to permit only this specific origin. This setup creates a security layer that helps prevent unauthorized requests from third parties, including REST clients like Postman or web browsers, ensuring that only the host application can securely interact with the server application.

```ts
const app = new Elysia()
  .derive(({ request }) => {
    const origin = request.headers.get('origin')
    return { authorized: origin === 'http://localhost:8000' }
  })
  .use(
    cors({
      origin: /http:\/\/localhost:8000/
    }),
  )
  .listen(3000)
```

Listing 15: The configuration for the server application.

6.3.5. Routing Problem

The path `/lotto6aus49` alone is insufficient to fully represent the entire subdomain for the Lotto game, as there is still a need for a page to view the results of previous draws. A proposed solution is to use `/lotto6aus49` as a prefix and then remove the existing route, replacing it with two new routes: one for displaying the play field at `/lotto6aus49/normalschein` and another for querying previous results at `/lotto6aus49/quoten`.

```
// apps/shell/router.ts
const router = createRouter({
  routes: [
    // ...
    {
      path: '/lotto6aus49/normalschein',
      component: () => import('lotto_app/Normalschein'),
    },
    {
      path: '/lotto6aus49/quoten',
      component: () => import('lotto_app/Quoten'),
    },
    // ...
  ],
})
```

Listing 16: Router of the host application with recently added routes.

Additionally, the `lotto` micro frontend must expose its corresponding components for these new routes, ensuring that the correct components are available to be loaded and displayed by the host application.

```
// apps/lotto/vite.config.ts
export default defineConfig({
  plugins: [
    federation({
      name: 'lotto_app',
      exposes: {
        './Normalschein': './src/Normalschein.vue',
        './Quoten': './src/Quoten.vue',
      },
      shared: ['vue']
    }),
  ]
})
```

Listing 17: Vite configuration for lotto micro frontend with more exposed components.

These routes share the prefix `/lotto6aus49`, which suggests that a separate Vue Router instance should ideally be created within the `lotto` micro frontend to manage its nested routes. This approach would allow the host application's router to register only the top-level routes for its remotes, while deeper-level routing would be handled within each micro frontend. However, this approach is not feasible under the current Module Federation setup. In this architecture, only a single instance of Vue is created within the host application, which utilizes the router defined in Listing 12. Consequently, no additional Vue or Vue Router instance exists within the `lotto` micro frontend to manage nested routing independently.

However, if a new route is now required to display instructions for the Lotto game, or if an existing route, such as the one for displaying results, needs to be removed, similar steps must be repeated to achieve the desired outcome. This repetition not only increases the potential for errors but also indicates a poor developer experience. Therefore, a more automated solution is desirable, which would involve creating a mechanism where routes can be dynamically registered and managed without the need for extensive manual input.

6.3.6. Routing Solution

1. Overview configuration

The initial step in this routing solution is to create an overview configuration for all applications. This overview specifies the directory locations, operating ports, names, and prefixes for each application. This configuration is important not only for defining how each micro frontend is served but also for enabling the host application to access information about its remotes. Henceforth, the term "overview configuration" will refer to this specific configuration.

```
// packages/mfe-config/index.js
export default {
  shell: {
    dir: 'shell',
    port: '8000',
  },
  home: {
    dir: 'home',
    port: '8001',
    name: 'home_app',
    prefix: '/',
  },
  lotto: {
    dir: 'lotto',
    port: '8002',
    name: 'lotto_app',
    prefix: '/lotto6aus49',
  },
}
```

Listing 18: Overview configuration for all applications.

2. Automated Components Exposure

Firstly, a list of routes needs to be generated. Drawing inspiration from the file-based routing systems used by popular meta-frameworks like Nuxt.js, a similar approach will be implemented to create this list. Secondly, this list of routes will be converted into a format that the vite plugin can understand. Finally, a wrapper function will encapsulate these processes and return a value that will be passed into the vite configuration for the host application. Additionally, to enhance flex-

ibility, this wrapper function can also accept extended exposes, for cases where components cannot be located at the specified locations, and custom remotes, which enable a horizontal-split method. The minimal code for this implementation is provided below.

```
function wrapper(name, _exposes, _remotes){
  const files = getFiles(name)
  const exposes = getExposes(files, _exposes)
  saveExpsoses()
  const remotes = getRemotes(_remotes)
  return federation({
    name,
    exposes,
    remotes
  })
}
```

Listing 19: A wrapper function is built on top of the vite plugin.

In the context of the `lotto` micro frontend, its folder structure is illustrated in Listing 20, left. After the execution of the wrapper function, a `routes.json` file is temporarily saved on disk and also included in the `exposes` object, which the host application will later access. The content of this file is illustrated in Listing 20, right.

```
.
└── apps
    ├── lotto
    │   │
    │   ├── pages
    │   │   ├── normalschein.vue
    │   │   └── quoten.vue
    │   └── ...
    │
    └── ...
```

```
[
  {
    "path": "/normalschein",
    "component": "Normalschein"
  },
  {
    "path": "/quoten",
    "component": "Quoten"
  }
]
```

Listing 20: The folder structure of the `lotto` micro frontend (left) and the generated `routes.json` file (right).

3. Automated Routes Registration

The final step in this routing solution is the automated registration of routes. With the overview configuration established in the first step and the details about the exposed components of each micro frontend obtained in the second step, the host application's router can now iterate through the overview configuration, reading the corresponding `routes.json` file and then process compiles a flat array of all possible routes within the application.

```
const router = createRouter()
for (const config of mfeConfig){
  const routes = getRoutes(config)
  router.addRoutes(routes)
}
app.use(router)
```

Listing 21: Router configuration with automated routes registration.

From now on, any changes in the directory monitored by the wrapper function will automatically trigger the creation of a router with the correct routes, ensuring the proper routing of the application.

6.4. Build Stage

As illustrated in the dependencies graph below, the UI library must be built before both the host application and the micro frontends. This sequence is necessary because the UI library is not defined as a separate micro frontend but rather as a dependency that is bundled during the build process. Once the UI library is built, the host and remote applications can be built either sequentially or in parallel, as their dependencies are resolved only at runtime. In contrast, the server application, which has no dependencies, can be built in the usual manner without any special considerations.

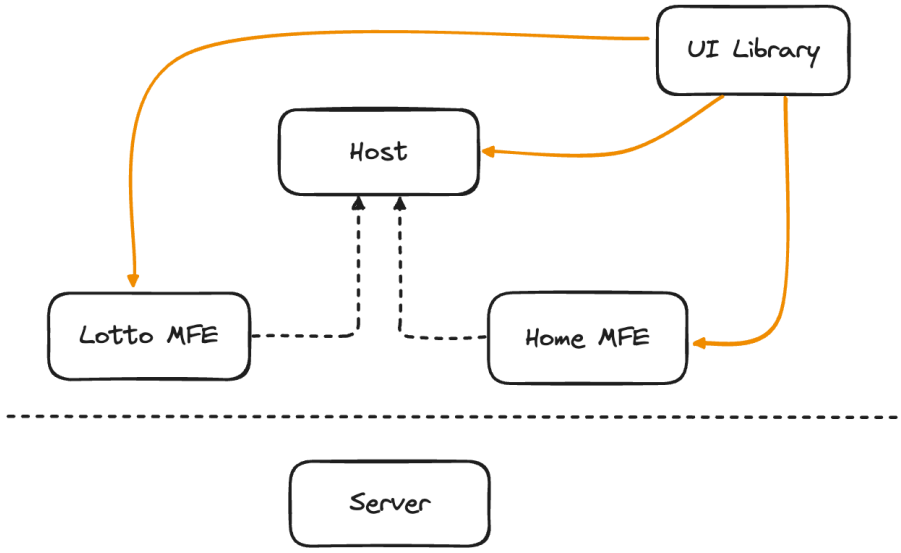

Figure 8: Dependencies graph: Solid arrows indicate build-time dependencies; Dashed arrows indicate runtime dependencies.

6.5. Testing Stage

The testing stage is focused on verifying the functionality and reliability of the application, ensuring that all components operate as expected before deploying to production. In this experiment, two types of testing will be covered: unit testing and end-to-end testing.

6.5.1. Unit Testing

Unit testing focuses on the smallest testable parts of the application, such as individual functions or components. Due to their limited scope and complexity, unit tests typically execute quickly. It is advisable to write unit tests for each component and utility function during the development process, particularly for the UI library. Below are two basic unit tests for the `UiButton` component from the UI library.

```
// packages/ui/UiButton/UiButton.test.ts
const slots = { default: () => 'Click me' }

it('should be rendered as a button', () => {
  const component = mount(UiButton, { slots })
  const button = component.find('button')
  expect(button.exists()).toBe(true)
  expect(button.text()).toBe('Click me')
})

it('should be rendered as a link', () => {
  const component = mount(UiButton, { slots, props: { to: '/about' } })
  const anchor = component.find('a')
  expect(anchor.exists()).toBe(true)
  expect(anchor.text()).toBe('Click me')
  expect(anchor.attributes('href').toBe('/about')
})
```

Listing 22: Two unit tests for the `UiButton` component.

6.5.2. End-to-End Testing

End-to-end (E2E) testing is a comprehensive method for evaluating the entire workflow of an application. Unlike unit tests, which focus on isolating components within a simulated environment, E2E testing replicates user interactions in a production-like setting to ensure that the system meets its requirements and functions as expected. Below is a basic E2E test intended to validate the navigation workflow.

```
// e2e/tests/app.test.ts
test('Navigation', async ({ page }) => {
  await page.goto('http://localhost:8000/')
  await expect(page).toHaveTitle('LOTTO Berlin')

  const playBtn = page.getByRole('link', { name: /Jetzt Spielen/ }))
  await expect(playBtn).toBeVisible()
  await playBtn.click()

  const url = page.url()
  expect(url).toBe('http://localhost:8000/lotto6aus49/normalschein')

  const heading = page.getByRole('heading', { name: /Normalschein/ }))
  await expect(heading).toBeVisible()
})
```

Listing 23: Simple E2E test to test the navigation workflow.

6.6. Deployment Stage

After completing the building and testing stages, this phase shifts its focus to determining the most effective solution for containerizing the application using Docker.

6.6.1. Server Container

The Dockerfile of the server application defines a two-stage build process. In the first stage, the application is built by installing dependencies and compiling the code. The second stage uses the same base image and copies the compiled output from the first stage. This approach helps keep the final Docker image small and efficient by including only the essential files needed to run the application. Finally, the Dockerfile exposes the necessary port and specifies the command to start the server application when the container is launched.

```dockerfile
# dockers/Dockerfile.server
FROM oven/bun:slim AS build
WORKDIR /dklb
COPY ./server ./
RUN bun install && bun run build

FROM oven/bun:slim
COPY --from=build /dklb/dist/index.js ./index.js
EXPOSE 3000
ENTRYPOINT ["bun", "run", "./index.js"]
```

Listing 24: Dockerfile for the server application.

6.6.2. Containers for Micro Frontends

To avoid the problems related to manual management, particularly regarding routing issues during the implementation phase, an automated approach for generating Dockerfiles based on the overview configuration is preferred. This approach involves two steps: first, generating an Nginx configuration file for both the host and remote applications, and second, creating a corresponding Dockerfile for each of these applications.

1. Nginx configurations

Firstly, Cross-origin resource sharing (CORS) headers must be appended to each nginx configuration of remote applications. This step is essential to guarantee that only requests originating from the host application are permitted and also prevent any CORS-related issues. Secondly, the host's nginx configuration is configured to always attempt to load the `/index.html` file, regardless of the URI requested. Without this configuration, the nginx server may return a "Not found" error for requests that do not explicitly point to existing resources.

```
// scripts/docker.ts
const cors = `
add_header 'Access-Control-Allow-Origin' 'http://localhost:8000';
add_header 'Access-Control-Allow-Methods' 'GET';
add_header 'Access-Control-Allow-Headers' 'Content-Type';`

for (const { port, dir } of Object.values(mfeConfig)) {
  const isShell = dir === 'shell'
  const path = `.nginx/${dir}.conf`
  const str = `
  server {
    listen ${port};
    server_name localhost_${dir};
    ${isShell ? '' : cors}
    location / {
      root      /usr/share/nginx/html/${dir};
      index     index.html;
      ${isShell ? 'try_files $uri $uri/ /index.html;' : ''}
  }`
  await write(path, str)
}
```

Listing 25: Generation of `nginx.conf` files based on the overview configuration.

2. Dockerfiles

The process of generating Dockerfiles for each micro frontend can be seamlessly integrated into the same loop that creates the Nginx configuration files. The Dockerfile follows a two-stage approach. In the first stage, the necessary files and dependencies are gathered, followed by the build process for the UI library and the specific micro frontend. The second stage sets up the environment for serving the micro frontend. By removing the default configuration, this stage ensures that only custom configurations and assets are used, and it prepares the built assets to be served by the web server on the defined port. Lastly, the command to start the server application by launching the container is specified.

```
// scripts/docker.ts
const content = `
FROM oven/bun:slim AS build
WORKDIR /dklb
COPY . .
RUN bun install
RUN bun run build:ui
RUN bun run --cwd apps/${dir} build

FROM nginx:alpine
WORKDIR /usr/share/nginx/html
RUN rm -rf * && rm -f /etc/nginx/conf.d/default.conf
COPY .nginx/${dir}.conf /etc/nginx/conf.d
COPY --from=build /dklb/apps/${dir}/dist ${dir}
EXPOSE ${port}
ENTRYPOINT ["nginx", "-g", "daemon off;"]`

await write(`dockers/Dockerfile.${dir}`, content)
```

Listing 26: Generation of the `Dockerfile` files based on the overview configuration.

6.6.3. Docker Compose

The final step in this deployment section involves using docker-compose, a convenient tool for simplifying the process by allowing to define and run multiple docker containers.

The server application's service is first defined, specifying the location of its Dockerfile and the port it will run on. Following this, the overview configuration is looped through again to generate service definitions for each micro frontend. These definitions include the service name, relevant build settings, and necessary port mappings. The script also ensures that each micro frontend service waits for the server to start, maintaining the correct initialization sequence. Once all configurations are defined, they are written into a `docker-compose.yml` file.

This Docker-based strategy enables the DKLB application to be easily deployed on any machine with Docker installed, streamlining the deployment process and ensuring consistency across different environments.

```
const contents = [
`services:
    server:
      build:
        context: .
        dockerfile: dockers/Dockerfile.server
      ports:
        - '3000:3000'`,
]

for (const { port, dir } of Object.values(mfeConfig)) {
  const content = `
  ${dir}:
    build:
      context: .
      dockerfile: dockers/Dockerfile.${dir}
    ports:
      - '${port}:${port}'
    depends_on:
      - server`
  contents.push(content)
}

await write('docker-compose.yml', contents.join('\n'))
```

Listing 27: Generation of `docker-compose.yml` file based on the overview configuration.

6.6.4. Other Containerization Approach

In the containerization approach implemented above, each micro frontend is deployed in its container. This design, while flexible, results in increased memory usage, as the memory requirements scale with the number of micro frontend containers. An alternative is to run the host application and all micro frontends only within a single container. As illustrated in Figure 9, the multi-container approach requires around 27MB of memory, whereas the single-container approach needs only about 9MB for the entire frontend. This reduction in memory usage can be advantageous in resource-constrained environments.

However, the single-container approach has trade-offs. Redeploying a micro frontend in this setup can be more cumbersome, as developers must apply the necessary changes, rebuild the micro frontend, and push the built assets to the correct directory in the container, responsible for that micro frontend. In some scenarios, this might even require taking down the entire container, leading to downtime for the whole application. On the other hand, with the multi-container approach, individual containers can be stopped and restarted independently, allowing updates to specific micro frontends without disrupting the entire system. This independence reduces the operational burden on developers and can minimize application downtime.

Several deployment strategies are available that can optimize the deployment process and effectively address the issues previously mentioned. One such strategy is blue-green deployment, which involves the use of two identical production environments, referred to as blue and green. The blue environment handles live traffic, while the green environment remains idle or is used for staging new releases. When a new version is ready, it is deployed to the green environment. After comprehensive testing, traffic is switched to the green environment, allowing for seamless updates. Should any issues arise, traffic can be reverted to the blue environment. This approach ensures minimal downtime during deployments, enhances reliability, and offers quick rollback capabilities [50].

In this experiment, both single and multi-container approaches are suitable. However, if the DKLB project later decides to adopt a micro frontend architecture and must select one, it will be essential to carefully weigh the importance of memory efficiency against the flexibility and ease of maintenance.

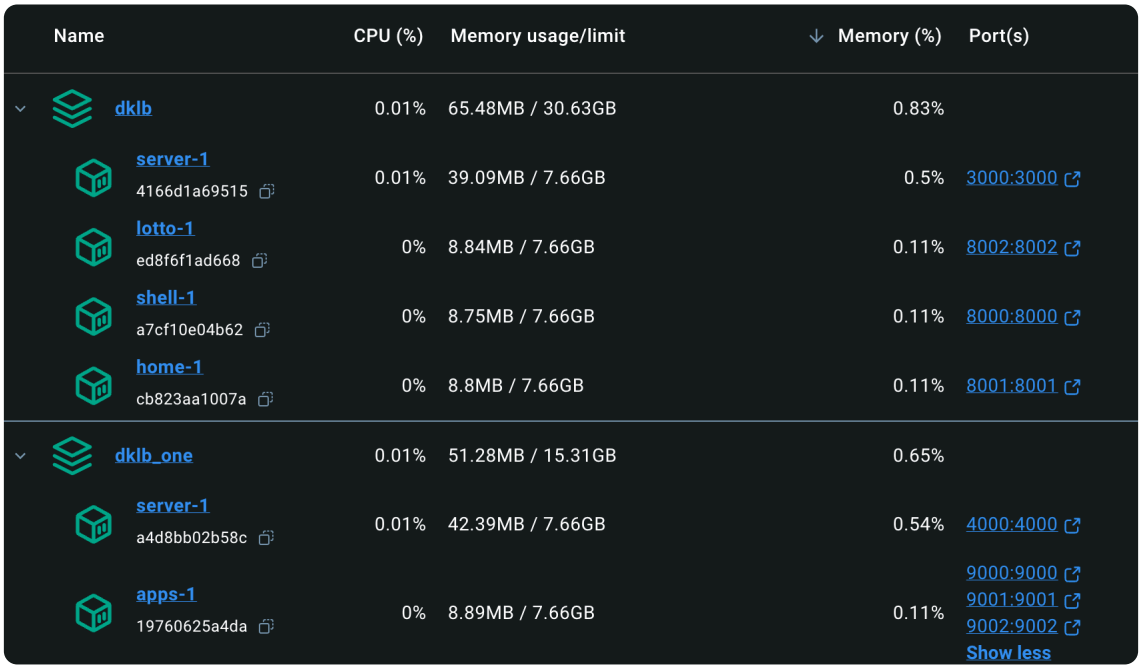

Figure 9: Memory usage comparison: multi-container vs. single-container approach.

6.7. CI/CD Stage

In this final stage of the experiment, the process for automating the integration of code changes from all applications and the UI library into the main branch is executed. This automation involves running linting, building, and testing tasks within each application. These steps enable the early detection of issues, ensure compatibility between new code and the existing codebase, and maintain a high standard of code quality. To provide a concrete example, the following section presents a code snippet that defines a pipeline for end-to-end testing, which is triggered when a pull request is made to the main branch. For clarity, the specific commands for each step have been omitted.

```yaml
# .github/workflows/e2e.yml
name: e2e
on:
  pull_request:
    branches:
      - main
jobs:
  e2e:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repo
      - name: Setup Bun
      - name: Install Dependencies
      - name: Setup Docker
      - name: Build Images
      - name: Run Containers
      - name: Install Playwright
      - name: Wait for Containers
      - name: Run E2E Tests
      - uses: Upload Report
      - name: Stop Containers
```

Listing 28: Pipeline configuration for end-to-end testing.

After completing the code integration, the focus shifts to automated deployment. Digital Ocean has been selected as the provider for the virtual private server (VPS). Significant effort has been invested in generating the necessary Docker configuration files during the deployment stage. The following code snippet demonstrates the action file used to manage the deployment process. This workflow is triggered whenever new code is pushed to the main branch and contains three primary steps: configuring SSH keys, installing the command-line interface for Digital Ocean, and establishing access to the VPS. Once access is secured, a sequence of commands is executed, including pulling the latest code via Git, bringing down existing Docker containers, rebuilding them, and bringing them back up.

```yaml
# .github/workflows/e2e.yml
# ...
steps:
  - name: Set up SSH
    run: |
      mkdir -p ~/.ssh
      echo "${{ secrets.SSH_PRIVATE_KEY }}" > ~/.ssh/id_rsa

  - name: Install doctl
    uses: digitalocean/action-doctl@v2
    with:
      token: ${{ secrets.DIGITAL_OCEAN_TOKEN }}

  - name: Redeploy on Droplet
    run: |
      doctl compute ssh ${{ env.DROPLET_ID }} --ssh-command "
        cd /root/DKLB
        git pull origin main
        bun run prepare:docker
        docker compose down
        docker compose up --build -d
      "
```

Listing 29: Pipeline configuration for deployment.

6.8. Developer Workflow Optimization

To enhance the developer workflow, a scaffold script is implemented to streamline the creation of new micro frontend applications. Initially, a `.template` directory is established to store the templates for the micro frontend application and the pipeline configuration file. Following this, a lightweight command line interface (CLI) is implemented to prompt the developer for the location and prefix of the micro frontend. The corresponding Dockerfile is then generated, and necessary updates are made to the `docker-compose.yml` file. Finally, the script asks whether to install dependencies or perform the build process.

```
.templates
├── app
│   ├── src
│   ├── vite.config.ts
│   └── ...
└── workflows
    ├── ci.yml
    └── ...
```

Listing 30: `.template` directory's structure.

```
DKLB git:(main) nr create:app
$ bun run scripts/create_app.ts
Create a new micro-frontend app.

Enter folder name:
eurojackpot

Enter prefix:
/eurojackpot

.$ bun run scripts/docker.ts


Install dependencies?
No

Build?
No

Done.

DKLB git:(main) ✗
```

Figure 10: The scaffold CLI for the creation of new micro frontend applications.

7. Evaluation

In this chapter, the results of the experiment are evaluated and compared with those of a monolithic architecture using a single-page application (SPA). The decision to use the monolithic SPA approach for comparison stems from its close alignment with the experiment's solution, as both rely on client-side composition and routing. Additionally, MULTA MEDIO is currently working on another rewrite project for a lottery platform using the same monolithic SPA version, which already offers several advantages. This consistency provides developers within the organization with a unified perspective. It is also worth noting that the SPA version is essentially a simplified adaptation of the micro frontends version.

Following this comparison, the four key aspects affected by adopting micro frontend architecture, as they relate to the first research question, are discussed.

The source code for both the micro frontends and monolithic versions is available in the repositories [38] and [51].

7.1. Comparison of Development Cycle

The table below outlines the key differences in the development cycle between the micro frontend and monolithic SPA approaches.

	Micro frontends	**Monolithic SPA**
Setup Stage	Although the project structure of micro frontends is organized to ensure a clear overview, it remains complex due to the presence of numerous directories. All remotes, the host application, and the UI library must be properly configured to ensure seamless integration.	The directories `apps`, `packages`, and `tools` mentioned in Listing 6 are redundant. Instead, a single `app` directory is used to store the entire frontend. Both dependency management and configurations are simplified, as the application utilizes a single `package.json` file for all dependencies and a single

	Micro frontends	**Monolithic SPA**
	Managing dependencies between micro frontends introduces additional complexities.	`vite.config.ts` file for all configurations.
Implementation Stage	As discussed in Section 6.3, Module Federation with client-side composition offers a development experience similar to that of the SPA approach, leading to comparable implementations for both the host and remote applications in each method. However, the routing challenges and the integration of the UI library encountered in the micro frontend version are significantly easier to manage in the SPA version.	
Build Stage	The UI library must be built first before the host and remote applications can be successfully bundled. The process of building the server application remains identical in both approaches.	No special considerations are necessary, as the entire frontend can be built in a single process.
Testing Stage	Unit testing and end-to-end testing are the same for both approaches. Unit testing occurs at the component level, while end-to-end testing primarily simulates user interactions in a real browser environment. Both types of testing focus on verifying functionality and user workflows, rather than on how components are composed into the view.	
Deployment Stage	Deployment with micro frontends is more complex, as it necessitates the creation of key configuration files depending on the number of applications involved. However, the more effort invested during this stage, the less work will be required in the CI/CD process.	The configuration files can be written once and require minimal modifications thereafter, as there will consistently be two applications running in parallel: the frontend and the server application.

	Micro frontends	Monolithic SPA
CI Stage	The number of configuration files for CI increases with the number of micro frontends. However, only the pipeline responsible for a specific micro frontend will be triggered when changes are made to that micro frontend.	The CI steps are mostly identical in both approaches. But in SPA approach, having a single configuration file for CI results in a shorter pipeline runtime. However, as any modification in any part of the application will trigger the pipeline for the entire application.
CD Stage	The CD pipeline is identical for both approaches, with each configured to run after code changes are merged into the main branch, triggering redeployment on the virtual private server.	

Overall, the SPA approach is simpler, with fewer directories, configurations, and a single build process. In contrast, micro frontends add complexity in setup, build, and deployment, requiring more configuration.

7.2. Impact on Flexibility, Maintainability, Scalability, and Performance

Based on the results of the experiment, this section will discuss how these four aspects of a web application are impacted by adopting micro frontend architecture.

7.2.1. Flexibility

The `home` and `lotto` micro frontends offer the flexibility to use different dependencies during development. For example, the `home` micro frontend can use `zod` for schema validation, while the `lotto` micro frontend can utilize `valibot`. Furthermore, if a new frontend framework is chosen in the future to replace Vue.js, it can be applied incrementally in the `home` application, while the `lotto` application continues using Vue.js, ensuring that the overall functionality of the application remains unaffected during the transition.

Additionally, new features or patches can be quickly applied at runtime without disrupting the other. If, during an update, the updated micro frontend becomes temporarily unavailable, the host application will detect the issue and navigate the user to an error page, providing clear and appropriate information about the disruption.

However, this flexibility introduces complexity in maintaining unified functionality across micro frontends, as different libraries may not behave consistently. Additionally, if the choice of a UI library is not carefully planned from the planning stage, the application may suffer from inconsistent styling. These drawbacks can result in a poor user experience.

7.2.2. Maintainability

The frontend is divided into `home` and `lotto` modules, with each module located in its directory. This modular structure simplifies the management and maintenance of the overall system by allowing developers to focus on specific micro frontends without needing to understand the entire application. This approach makes onboarding new developers more efficient, as they can work on individual components without having to grasp the full scope from the beginning. It also reduces the likelihood of introducing unintentional bugs or inconsistencies when making changes.

This separation, though beneficial, can result in redundancy in certain areas. For instance, shared logic, such as the fetch function used to retrieve gaming history quotes, might be duplicated across multiple micro frontends, which violates the DRY (Don't Repeat Yourself) principle. Additionally, maintaining consistency becomes more challenging, as refactoring or updating shared logic may not always be synchronized across all micro frontends. This can lead to a potential lack of similarity and increased maintenance efforts over time.

7.2.3. Scalability

If the `lotto` module experiences a surge in traffic, it can be scaled independently, optimizing resource usage by ensuring that only the necessary parts of the system receive additional resources. The team responsible for this micro frontend can tailor the scaling strategy based on its specific workload or user interaction patterns, allowing for a more efficient and responsive system.

Independent scaling can introduce increased infrastructure overhead. Each micro frontend may require its own hosting and monitoring, which adds complexity and raises operational costs. Additionally, common backend services, such as databases, may become bottlenecks if not properly optimized to handle the demands of independently scaled micro frontends. This can result in performance issues that impact the entire application, despite the modularity of the frontend components.

7.2.4. Performance

To evaluate performance, the open-source tool Sitespeed.io is used to analyze website speed based on performance best practices [52]. The table below compares the micro frontends and SPA versions, with metrics gathered from the homepage of the application using the Chrome browser over five iterations. The results are color-coded: blue for informational data, green for passing, yellow for warnings, and red for poor performance.

	Micro frontends	Monolithic SPA
First Contentful Paint	60 ms	44 ms
Fully Loaded	80 ms	58 ms
Page Load Time	7 ms	16 ms
Largest Contentful Paint	185 ms	168 ms
Total Requests	26	15
JavaScript Requests	15	6
CSS Requests	3	1
HTML Transfer Size	563 B	458 B
JavaScript Transfer Size	299.6 KB	143.3 KB
CSS Transfer Size	24.2 KB	16.8 KB
Total Transfer Size	333.3 KB	167.8 KB

Table 1: Comparison between the micro frontends and monolithic SPA versions.

An important metric to consider is the JavaScript Transfer Size, which accounts for approximately 85-90% of the Total Transfer Size. In the micro frontends implementation, this transfer size is nearly double that of the SPA version, leading to longer page load times. The primary reason for this increase is the requirement for the host application to fetch the `remoteEntry.js` files from its remote modules. These entry files play a crucial role in Module Federation, containing essential information about the remote module, such as its name and the components it exposes [37]. This additional overhead will slow down the initial load, as the host must retrieve and process these files to properly display the micro frontends and manage their interactions.

7.3. Limitations

Due to the limited scope of the experiment, the implemented application is relatively small, making it difficult to fully examine the advantages of micro frontend architecture for larger, more complex applications. Additionally, the experiment focused on a single implementation approach, leaving several key aspects unexplored. For instance, the potential benefits of using native Web Components instead of Module Federation, as well as the impact of integrating Module Federation with server-side composition, were not examined. These alternatives could offer valuable insights into how micro frontends might perform in different scenarios.

8. Summary

In this concluding chapter, the insights from Section 3 and the key findings from the evaluation chapters are synthesized to address the two primary research questions of this study. Subsequently, a plan for future research is proposed to further explore and enhance the feasibility of adopting a micro frontend architecture for the DKLB project.

8.1. Conclusion

A table outlining the advantages and disadvantages across the four aspects of flexibility, maintainability, scalability, and performance will be presented first.

	Advantages	**Disadvantages**
Flexibility	Independent development and deployment of frontend components, reducing downtime and enabling parallel work.	Introduces added complexity, particularly in maintaining consistent functionality and ensuring a seamless user experience across the different micro frontends.
Maintainability	Smaller, modular codebases improve maintainability, making it easier to manage, and onboard new developers.	Managing multiple repositories or codebases can lead to fragmented maintenance efforts and potential duplication.
Scalability	Enables independent scaling of specific frontend components, optimizing resource usage.	Potential for increased infrastructure overhead, as each micro frontend may require separate hosting and monitoring.

Performance	More efficient loading, with the possibility to load only necessary parts of the application, improving user experience.	Initial setup may be complex, with potential performance challenges around integration and communication between micro frontends.

The table above indicates that a micro frontend architecture can effectively address the challenges and limitations of the current monolithic system in the DKLB project. This approach introduces greater flexibility in development and deployment, while also improving maintainability and scalability for individual parts of the frontend. These characteristics align well with agile methodologies, promoting iterative development and enabling faster delivery. Moreover, as highlighted in the research by Männistö et al., even small teams can leverage the benefits of this architecture provided [10].

However, adopting a micro frontend architecture introduces additional complexity in management and monitoring, particularly in ensuring smooth integration of components and a consistent user experience. Additionally, the decentralized nature of this approach requires further optimization to maintain adequate performance levels.

In conclusion, micro frontend architecture presents a promising solution for large-scale web applications, delivering notable advantages while also introducing certain challenges. The decision to implement this approach should be driven by the project's specific requirements, carefully considering whether the additional complexities are justified by the benefits. While this study has examined crucial aspects of the web application development process, further in-depth analysis is necessary to fully assess and optimize its potential for the DKLB project.

8.2. Future Research

One necessary optimization is bundle analysis, which aims to reduce duplicate code by ensuring proper sharing of library code across JavaScript chunks, as these redundancies can negatively impact performance. To address this issue, gaining deeper knowledge of the Vite plugin could enable more precise intervention in its configuration.

Alternatively, Rspack could be considered as a replacement bundler, given its official support for Module Federation. This provides a key advantage over Vite, which currently depends on a third-party plugin that is no longer actively maintained. Furthermore, the case for adopting Rspack is strengthened by its collaboration with the creator of Module Federation on the forthcoming release of Module Federation 2.0. This update promises new features, expanded use cases, and improved performance, making it a more robust solution.

Additionally, improved error-handling mechanisms should be implemented. For example, when a horizontal micro frontend encounters an error, redirecting the user to an error page disrupts the experience, as only a portion of the view may be affected. A more refined approach would involve updating the overview configuration at runtime to control which micro frontends are displayed or hidden, allowing for greater flexibility and enabling more dynamic solutions.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorgelegte Bachelorarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wötliche und sinngema¨ße Zitate wurden als solche gekennzeichnet.

Würzburg, am 13. September 2024

Zustimmung zur Plagiatsüberprüfung

Hiermit willige ich ein, dass zum Zwecke der Überprüfung auf Plagiate meine vorgelegte Arbeit in digitaler Form an PlagScan (www.plagscan.com) übermittelt und diese vorübergehend (max. 5 Jahre) in der von PlagScan geführten Datenbank gespeichert wird sowie persönliche Daten, die Teil dieser Arbeit sind, dort hinterlegt werden. Die Einwilligung ist freiwillig. Ohne diese Einwilligung kann unter Entfernung aller persönlichen Angaben und Wahrung der urheberrechtlichen Vorgaben die Plagiatsüberprüfung nicht verhindert werden. Die Einwilligung zur Speicherung und Verwendung der persönlichen Daten kann jederzeit durch Erklärung gegenüber der Fakultät widerrufen werden.

Würzburg, am 13. September 2024

Bibliography

[1] L. James and F. Matin, “Microservices.” Accessed: Jun. 13, 2024. [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] “The Birth of the Web | CERN.” Accessed: Jun. 13, 2024. [Online]. Available: https://home.cern/science/computing/birth-web

[3] M. M., “LOTTO Berlin.” Accessed: Jun. 13, 2024. [Online]. Available: https://www.lotto-berlin.de/www.lotto-berlin.de

[4] S. Newman, *Building Microservices, 2nd Edition*, 2nd ed. O'Reilly Media, Inc., 2021.

[5] M. E. Conway, “HOW DO COMMITTEES INVENT?,” 1968.

[6] “Micro Frontends | Technology Radar.” Accessed: Jun. 02, 2024. [Online]. Available: https://www.thoughtworks.com/radar/techniques/micro-frontends

[7] E. Eric, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[8] D. Wang *et al.*, “A Novel Application of Educational Management Information System Based on Micro Frontends,” *Procedia Computer Science*, vol. 176, pp. 1567–1576, 2020, doi: 10.1016/j.procs.2020.09.168.

[9] R. Perlin, D. Ebling, V. Maran, G. Descovi, and A. Machado, “An Approach to Follow Microservices Principles in Frontend,” in *2023 IEEE 17th International Conference on Application of Information and Communication Technologies (AICT)*, Baku, Azerbaijan: IEEE, Oct. 2023, pp. 1–6. doi: 10.1109/AICT59525.2023.10313208.

[10] J. Männistö, A.-P. Tuovinen, and M. Raatikainen, “Experiences on a Frameworkless Micro-Frontend Architecture in a Small Organization,” in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, L'Aquila, Italy: IEEE, Mar. 2023. doi: 10.1109/ICSA-C57050.2023.00025.

[11] M. Mena, A. Corral, L. Iribarne, and J. Criado, “A Progressive Web Application Based on Microservices Combining Geospatial Data and the In-

ternet of Things," *IEEE Access*, vol. 7, pp. 1–14, 2019, doi: 10.1109/ACCESS.2019.2932196.

[12] Q. Capdepon, N. Hlad, A.-d. Seriai, and M. Derras, "Migration Process from Monolithic to Micro Frontend Architecture in Mobile Applications."

[13] M. Shakil and A. Zoitl, "Towards a Modular Architecture for Industrial HMIs," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vienna, Austria: IEEE, Sep. 2020, pp. 1267–1270. doi: 10.1109/ETFA46521.2020.9212011.

[14] B. SimõEs, M. D. P. Carretero, J. MartíNez Santiago, S. MuñOz Segovia, and N. Alcain, "TwinArk: A Unified Framework for Digital Twins Based on Micro-frontends, Micro-Services, and Web 3D," in *The 28th International ACM Conference on 3D Web Technology*, San Sebastian Spain: ACM, Oct. 2023, pp. 1–10. doi: 10.1145/3611314.3615915.

[15] S. Peltonen, L. Mezzalira, and D. Taibi, "Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review," *Information and Software Technology*, vol. 136, p. 106571–106572, Aug. 2021, doi: 10.1016/j.infsof.2021.106571.

[16] "On Monoliths and Microservices | OTTO Tech | Blog." Accessed: Jun. 21, 2024. [Online]. Available: https://www.otto.de/jobs/en/technology/techblog/blogpost/on-monoliths-and-microservices_2015-09-30.php

[17] i. gmbh, "inoio gmbh | Jump - ein Technologie-Sprung bei Galeria Kaufhof." Accessed: Jun. 22, 2024. [Online]. Available: https://inoio.de/blog/2014/09/20/technologie-sprung-bei-galeria-kaufhof/

[18] "TECHNOLOGY." Feb. 2019. Accessed: Jun. 21, 2024. [Online]. Available: https://tech.thalia.de/another-one-bites-the-dust-wie-ein-monolith-kontrolliert-gesprengt-wird-teil-i/

[19] "Luigi - The Enterprise-Ready Micro Frontend Framework." Accessed: Jun. 21, 2024. [Online]. Available: https://luigi-project.io/

[20] "Zalando." Dec. 2018. Accessed: Jun. 21, 2024. [Online]. Available: https://engineering.zalando.com/posts/2018/12/front-end-micro-services.html

[21] P. Senders, "Front-End Microservices at HelloFresh." Accessed: Aug. 24, 2024. [Online]. Available: https://engineering.hellofresh.com/front-end-microservices-at-hellofresh-23978a611b87

[22] S. Jan, "Experiences Using Micro Frontends at IKEA." Accessed: Jun. 11, 2024. [Online]. Available: https://www.infoq.com/news/2018/08/experiences-micro-frontends/

[23] L. Mezzalira, *Building Micro-Frontends, 2nd Edition*, 2nd ed.

[24] F. Rappl, *The Art of Micro Frontends*. Packt, 2021.

[25] M. Geers, *Micro Frontends in Action*. 2020.

[26] M. Steyer, "Consequences of Micro Frontends: Survey Results." Accessed: Aug. 24, 2024. [Online]. Available: https://www.angulararchitects.io/en/blog/consequences-of-micro-frontends-survey-results/

[27] "Understand JavaScript SEO Basics | Google Search Central | Documentation." Accessed: Aug. 24, 2024. [Online]. Available: https://developers.google.com/search/docs/crawling-indexing/javascript/javascript-seo-basics

[28] "Web Storage API - Web APIs | MDN." Accessed: Aug. 24, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

[29] "Using HTTP Cookies - HTTP | MDN." Accessed: Aug. 24, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

[30] "Introduction to Server Side Includes - Apache HTTP Server Version 2.4." Accessed: Aug. 09, 2024. [Online]. Available: https://httpd.apache.org/docs/current/howto/ssi.html

[31] "ESI Document." [Online]. Available: https://www.akamai.com/site/zh/documents/technical-publication/akamai-esi-developers-guide-technical-publication.pdf

[32] "<iframe>: The Inline Frame Element - HTML: HyperText Markup Language | MDN". Accessed: Aug. 09, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

[33] "Window: postMessage() Method - Web APIs | MDN." Accessed: Aug. 09, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

[34] S. Engineering, "Building the Future of Our Desktop Apps." Accessed: Aug. 09, 2024. [Online]. Available: https://engineering.atspotify.com/2021/04/building-the-future-of-our-desktop-apps/

[35] “Web Components - Web APIs | MDN.” Accessed: Aug. 10, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_components

[36] “Webpack.” Accessed: Jun. 13, 2024. [Online]. Available: https://webpack.js.org/

[37] “Module Federation.” Accessed: Jun. 13, 2024. [Online]. Available: https://module-federation.io/

[38] T. P. Nguyen, “DKLB.” Accessed: Aug. 17, 2024. [Online]. Available: https://github.com/chubetho/DKLB

[39] “VueJS.” Accessed: Jun. 13, 2024. [Online]. Available: https://vuejs.org/

[40] “Tailwind CSS.” Accessed: Jul. 07, 2024. [Online]. Available: https://tailwindcss.com/

[41] “Vite.” Accessed: Jun. 13, 2024. [Online]. Available: https://vitejs.dev/

[42] “Originjs/Vite-Plugin-Federation.” Accessed: Jun. 13, 2024. [Online]. Available: https://github.com/originjs/vite-plugin-federation

[43] “Elysia.” Accessed: Jul. 14, 2024. [Online]. Available: https://elysiajs.com/

[44] “Docker.” Accessed: Jul. 14, 2024. [Online]. Available: https://www.docker.com/

[45] “Nginx.” Accessed: Aug. 17, 2024. [Online]. Available: https://nginx.org/en/

[46] “GitHub Actions.” Accessed: Jul. 14, 2024. [Online]. Available: https://docs.github.com/en/actions

[47] “Vitest.” Accessed: Jul. 14, 2024. [Online]. Available: https://vitest.dev/

[48] “Playwright.” Accessed: Jul. 14, 2024. [Online]. Available: https://playwright.dev/

[49] J. P. Henderson, “Monorepo-vs-Polyrepo.” Accessed: Aug. 13, 2024. [Online]. Available: https://github.com/joelparkerhenderson/monorepo-vs-polyrepo

[50] M. Fowler, “Blue Green Deployment.” Accessed: Jul. 18, 2024. [Online]. Available: https://martinfowler.com/bliki/BlueGreenDeployment.html

[51] T. P. Nguyen, “DKLB_Monolith.” Accessed: Aug. 17, 2024. [Online]. Available: https://github.com/chubetho/DKLB_Monolith

[52] “Site Speed IO.” Accessed: Aug. 22, 2024. [Online]. Available: https://www.sitespeed.io/