

MASTER'S THESIS 2023

# Managing Micro Frontends Across Multiple Tech Stacks - Sharing, Finding & Publishing

Andrej Simeunovic, Uros Tripunovic

Elektroteknik  
Datateknik

ISSN 1650-2884

LU-CS-EX: 2023-25

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY





EXAMENSARBETE  
Datavetenskap

LU-CS-EX: 2023-25

**Managing Micro Frontends Across  
Multiple Tech Stacks - Sharing, Finding &  
Publishing**

Hantering av Micro Frontends över flera  
tekniska plattformar - Delning, Sökning  
och Publicering

Andrej Simeunovic, Uros Tripunovic



---

# Managing Micro Frontends Across Multiple Tech Stacks - Sharing, Finding & Publishing

---

Andrej Simeunovic  
andrej.simeunovic98@gmail.com

Uros Tripunovic  
uros\_1997@hotmail.com

June 21, 2023

Master's thesis work carried out at  
the Department of Computer Science, Lund University.

Supervisors: Lars Bendix, [lars.bendix@cs.lth.se](mailto:lars.bendix@cs.lth.se)  
David Nilsson, [david.nilsson2@ingka.ikea.com](mailto:david.nilsson2@ingka.ikea.com)

Examiner: Per Andersson, [per.andersson@cs.lth.se](mailto:per.andersson@cs.lth.se)



## Abstract

IKEA is a large enterprise composed of multiple teams with diverse tech stacks. These teams are interested in exploring the possibility of sharing frontend features to avoid duplicating existing functionality. The company aims to introduce the concept of Micro Frontends (MFEs) to reduce code duplication, minimize communication overhead, and increase code reusability while enabling autonomous development. However, the introduction of MFEs alongside the current tech stacks poses the challenge of aligning technologies across teams at IKEA. While the teams at IKEA don't currently face challenges in finding and publishing MFEs due to being in the early phase of working with them, it's crucial to address this issue in advance. Doing so will further minimize code duplication, reduce communication overhead, and increase code reusability.

Following a series of experiments inspired by literature studies and interviews, two methods were discovered for the teams at IKEA to share MFEs without needing to change their current tech stack. These methods involve either wrapping framework components as Web Components or exposing a framework's render method within a function and injecting them into the frontend application. Each method requires a different level of effort when integrating them into a tech stack. In terms of finding and publishing MFEs, IKEA already has an existing platform. As part of our evaluation of IKEA's existing platform, we compared it to our requirements specification, which was derived from our interview with developers at IKEA. To enhance the ease of finding MFEs, we suggested improvements to their search functionality by displaying the teams involved in creating each MFE.

**Keywords:** Micro frontends, Frontend development, Reusability, Sharing, Multi-Framework, Multi-Technology





# Acknowledgements

---

We would like to express our gratitude to the Ingka Group Digital for their help, support, and hospitality, which made us feel like we were part of the family. We would also like to extend a special thanks to Mazen, who showed tremendous patience in answering our repetitive questions and helped us gain a deeper understanding of the subject matter, which contributed significantly to the success of our work. Last but not least, we are immensely grateful to our supervisor at LTH Lars Bendix for his continuous support and feedback, as well as for always asking us the critical question of "why".



# Contents

---

<b>1</b>	<b>Introduction</b>	<b>7</b>
<b>2</b>	<b>Background</b>	<b>9</b>
2.1	Context . . . . .	9
2.1.1	IKEA . . . . .	9
2.1.2	Initial interview . . . . .	10
2.1.3	Research questions . . . . .	11
2.2	Methodology . . . . .	11
2.2.1	RQ1 . . . . .	12
2.2.2	RQ2 . . . . .	14
2.3	Theoretical fundamentals . . . . .	15
2.3.1	React . . . . .	15
2.3.2	Micro frontend . . . . .	16
2.3.3	Module federation . . . . .	16
2.3.4	Web Components . . . . .	17
<b>3</b>	<b>Investigating issues and solutions of sharing MFEs</b>	<b>19</b>
3.1	Initial investigation . . . . .	19
3.2	More in-depth investigation . . . . .	21
3.3	Experiment and results . . . . .	22
3.3.1	Web Components . . . . .	23
3.3.2	Cross bundler MFEs . . . . .	25
3.3.3	React-ive ways of sharing MFEs . . . . .	27
3.3.4	Working with the cloud . . . . .	27
<b>4</b>	<b>How to find and publish MFEs</b>	<b>29</b>
4.1	Our ideal requirements . . . . .	29
4.2	Comparing ideal requirements to IKEA's dev portal . . . . .	31
4.3	Improving IKEA's dev portal . . . . .	33
4.3.1	Scenario . . . . .	33

4.3.2	Improved design . . . . .	35
<b>5</b>	<b>Discussion and Related Work</b>	<b>39</b>
5.1	Reflection on work process . . . . .	39
5.2	Threats to validity . . . . .	40
5.3	Generalizability . . . . .	41
5.4	Related work . . . . .	42
5.4.1	Micro frontend architecture for cross framework reusability in practice . . . . .	43
5.4.2	Implementing Micro Frontends Using Signal-based Web Components	44
5.4.3	Experiences on a Frameworkless Micro-Frontend Architecture in a Small Organization . . . . .	45
5.4.4	Evaluating Micro Frontend Approaches for Code Reusability . . . . .	46
5.5	Future work . . . . .	47
<b>6</b>	<b>Conclusion</b>	<b>49</b>
	<b>References</b>	<b>51</b>
	<b>Appendix A Initial investigation</b>	<b>57</b>
	<b>Appendix B Elicit design requirements</b>	<b>59</b>
	<b>Appendix C Requirement specification</b>	<b>61</b>
	<b>Appendix D Proposed designs</b>	<b>63</b>

# Chapter 1

## Introduction

---

IKEA is a large company that consists of multiple independent teams. These different teams at IKEA work under different contexts and where implementations of similar features can occur. This leads to code duplication where time and money could be better spent elsewhere. The company feels that the development is inefficient and wants to mitigate this problem by dividing the teams into only focusing on developing their own product for their frontend application in their own context. By introducing MFEs, features created by different teams will be plug-and-play ready and independent of their development context and thus introduce code reusability and a more efficient development process.

MFE is an architectural approach that breaks down frontend web applications into smaller, independent modules, each responsible for a specific feature or function, following the principles of Microservice [23]. Modules can be updated independently without affecting the entire application, making it easier to scale and maintain the application [22].

Currently, there is no standard way of sharing MFEs without forcing a team to align with another and thus breaking autonomous development. This introduces friction between teams where they are not willing to change their current technology stack and current ways of working. If different teams work with different frameworks or different versions of the same framework, such as React, importing MFEs needs to fulfill the exporter's requirements, meaning that the importer needs to align with the exporter's technology stack to make it work as intended. This current way of work goes against the principle of autonomous development, where introducing MFEs should be used in a way where you can just plug and play without adding extra requirements and changing one's current workflow. Apart from sharing MFEs, there is no efficient way of finding and publishing MFEs. There is a huge communication overhead when sharing MFEs, and if a team doesn't take it upon themselves to broadcast their MFEs it's nigh impossible to know of its existence.

The problems described above formed the following research questions:

- **RQ1** - What are the technical challenges of sharing micro frontends in IKEA's context?
- **RQ2** - How would a team know that an MFE with a desired API already exists?

These research questions will act as well-defined tasks that we aim to explore in this thesis. The overall goal of **RQ1** is to present different options of how to share MFEs without changing their current ways of working as well as their technology stack. The goals of **RQ2** are twofold: first, to help people at IKEA find and publish their MFEs, which will decrease communication overhead; and second, to create a platform where they can share other services, in addition to MFEs.

This thesis will be divided into two phases, one for **RQ1** and the other for **RQ2**. With regards to **RQ1**, the first is to get an understanding of the challenges IKEA has when working with MFEs. And a literature study will be performed in order to understand the concept of MFEs along with interviewing the people at IKEA and studying previous work and theories on how to tackle the challenges people at IKEA are facing through a more in-depth literature study after the interviews. **RQ1** will be addressed by conducting experiments to identify best practices for working with MFEs, resulting in one or more methods for sharing these components. The second phase focuses on **RQ2** and our main research method will be interviewing the developers at IKEA which will result in a requirements specification with the functionality needed to find MFEs with a desired API. The proof of concept will help the developers at IKEA access MFEs and publish them. The proof of concept will also be evaluated in order to improve the user experience.

The following chapter will introduce the relevant theory that will be used throughout the thesis along with the motivations and context of how this thesis was performed. Thereafter, the following two chapters will look into different technical possibilities of sharing MFEs and then follow by a design of a proof of concept for finding and publishing MFEs. Then it proceeds to a discussion and lastly a conclusion to the research questions and our recommendations.

# Chapter 2

## Background

---

The purpose of this chapter is to present the issues IKEA has working with the architectural concept that is MFEs. And in this chapter, we will formulate these challenges into research questions that will present a road map as to how these research questions will be explored and answered. This will help the reader understand the identified initiating problem at IKEA and how we will proceed with answering them. First, the context of the company will be presented, and what challenges they currently have with MFEs to establish relevant research questions, that will act as well-defined tasks to solving the initiating problem. The following section will present our methodology as well as motivations as to why we believe these methods are a suitable approach to solving the initiating problem. Lastly, the relevant theory will conclude the chapter by presenting the theoretical foundation that will help the reader understand this thesis.

### 2.1 Context

To be able to get a grasp on what challenges IKEA has when working with MFEs, it's important to first understand the company's context. The context lays the foundation for what direction to tackle the initiating problem. This section will begin by describing the context in which IKEA is at the moment. Then the initiating problem when working with MFEs will be presented based on the initial interviews being conducted. The section ends by presenting the research questions and the underlying motivation of why they were chosen.

#### 2.1.1 IKEA

IKEA is one of the world's most recognizable retail brands. The vision of "creating a better everyday life for many people" [3] has always driven the company to new heights of success. To be able to stay relevant to the fast-paced evolution of technology in today's world, the

company has entered a new era of the company's history by going through a digital transformation. This led to a business transformation where they are exploring new ways of offering the customers their products/offers and new ways of business operations. The company went on with the trend of introducing E-commerce and when the pandemic arrived, they had to triple down on their online store because 75% of their stores had to temporarily close [28] and more people turned their attention to the internet.

IKEA has more than 1000 developers working in Sweden to constantly develop their online store and improve their decision-making with the help of data and analytics. The developers are divided into teams, and each team is responsible for a certain feature on the website. The teams are autonomous and there is no alignment between them. This means that each team works in its own context, developing the feature in its own environment using preferred developer tools and frameworks that fit them best. In terms of the size of the teams, IKEA uses Amazon's pizza rule as inspiration, where two whole family pizzas will be able to feed 2-12 workers. With this rule, IKEA can contain small teams that can work together without overshadowing each other, but this comes at a price where these teams are autonomous and loosely coupled with other teams or not at all. One of the key benefits of having smaller teams is less architectural/organizational friction by following Conway's Law [12] and the two-pizza rule.

### 2.1.2 Initial interview

The purpose of the initial interview is to understand why the people at IKEA see MFEs as a solution to their problems, what problems are they currently facing, and what is it they would like to do in the future but can't today. From the interviews conducted, it became apparent that IKEA has only recently started working with MFE and has not been successful in trying to share these MFEs with the other teams at IKEA.

The developers we interviewed believed that working with MFEs will help alleviate the problems of code duplication. Code duplication also known as the double maintenance problem [10] arises when multiple copies of the same software exist meaning all of them need to be maintained equally. Maintaining multiple versions of the same software is close to impossible since it only requires one of the multiple copies of the software to be forgotten about. Developers also believe that working with MFEs will result in less communication overhead and fast response times when issues are identified in their applications. What developers didn't expect when working with MFEs is that working with remote teams or other departments requires a huge communication overhead and coordination. Different frameworks and different versions of the same frameworks present an issue where departments are forced to change their technology or give into code duplication.

There is also the issue of not knowing that an MFE exists. The interviewees described their broadcast of MFEs as cumbersome. The developers with an MFE ready at hand would go out of their way to other developers from other departments and ask if they are interested in it.

It was also mentioned that being dependent on a single framework can lead to issues in the future. When working with code libraries the risk of discontinued support is imminent. To alleviate a complete code rewrite further down the line working with multiple framework libraries is believed to help mitigate this issue.

To summarize, MFEs are a relatively new concept that IKEA is trying to implement in



their current context, developers have no reliable way of transporting MFEs without having to cause friction between different departments, and there is also no platform for finding and publishing MFEs, and being too dependent on a framework library can cost a lot of money further down the line because of code rewrites.

### 2.1.3 Research questions

Having well-defined research questions will provide the reader with our research framework for maintaining focus on exploring the initiating problem at IKEA. These tasks will ensure that the research stays on track with the relevant problems at hand and will help the reader understand the scope and boundaries established for this study. From the initial interviews, it was mentioned that there is no reliable way of sharing, finding, and publishing MFEs without great sacrifice. Therefore, the purpose of this thesis is to investigate what causes these sacrifices and how they could be mitigated. With that in mind, the first research question will cover the topic that relates to the technical challenges of sharing MFEs between multiple teams. It is also of interest to us to see how these challenges vary in different IKEA domains, and if these trade-offs that come along with sharing MFEs are worth it. The research questions have been formulated in the following way:

- **RQ1** - What are the technical challenges of sharing MFEs in IKEA's context?
- **RQ1.1** - What are the trade-offs for sharing MFEs seamlessly?

Once the challenges have been identified and explored the next step would be to try and identify how these MFEs can be found and published. We would like to investigate what IKEA needs to prepare so that developers are able to find and publish MFEs. With this research question, we would like to explore their options for finding services today and how that can be applied to MFEs. The research questions have been formulated in the following way:

- **RQ2** - How would a team know that an MFE with a desired API already exists?
- **RQ2.1** - Can a proof of concept be created showing that an MFE with a desired API already exists?

## 2.2 Methodology

Now that the initiating problem has been introduced and the research questions have been formulated, this section will discuss our research methods of use along with motivations as to how these will answer the research questions. This will help the reader understand our thought process and how we've decided to conduct our work when exploring the research questions. This section will start off by discussing how **RQ1** will be conducted and finished off with how **RQ2** is also to be conducted.

## 2.2.1 RQ1

To be able to deal with **RQ1** and **RQ1.1** in a proper way, data gathering is required. There are multiple ways of gathering data and the different methods for gathering them will be discussed.

To understand **RQ1**, an initial literature study was conducted in order to get an overall understanding of what MFEs are. We saw it fit to first understand the subject matter before conducting interviews with developers at IKEA. With an extended vocabulary, we believed that a more meaningful discussion could be accomplished when conducting interviews.

Methods for gathering data aren't always obvious, and the question of does one want to gather qualitative data or quantitative data arises. Due to the nature of **RQ1** being an open-ended and exploratory research question, gathering qualitative data helps us get an in-depth understanding of the underlying causes of trouble when sharing MFEs. With semi-structured interviews, qualitative data can be gathered [27]. Letting the interviewee talk freely can lead us down a path of interest from our predetermined questions and topics. Structured interviews and questionnaires could've been optional for gathering data but we were still not sure what the underlying issues were with MFEs so more open-ended questions were easier to work with.

After expanding our general knowledge about MFEs from our initial literature study and semi-structured interviews, we will be able to easily navigate and better understand what information is considered important to answer **RQ1** and what information could be filtered out. The second literature study will lay the foundation and inspire us on what factors are important to investigate when conducting an experiment to measure the developer experience and explore the challenges of sharing MFEs.

Lastly, an experiment was conducted with the purpose of getting a hands-on experience with the different aspects of sharing and working with MFEs identified in the second literature study. The experiments were conducted because we wanted to see if we could identify any issues when sharing MFEs in regards to **RQ1.1**. The experiment was a form of validation to investigate our theory of combining different MFE concepts.

### Literature study 1

The first thing we did was to search for peer-reviewed papers about MFEs on Google Scholar with the keyword "micro frontend". The search engine prompted multiple papers which mostly were about MFEs in general. We read the papers which broaden our knowledge about the MFE concept and the important technology related to it. New keywords started to pop up, keywords such as "Module Federation" and "Microservices". With the added keywords, we were able to conduct a different kind of search request and thus receive other kinds of peer-reviewed papers which became more relevant to the initiating problem which IKEA had. We also found books that included a lot of great information but it was still challenging for us to scope which chapters of the books were considered important for our research. To conclude, even though we absorbed a lot of information from the articles and the books, it felt still very blur on what kind of information was considered of highest importance for **RQ1** and **RQ1.1**. LUBSearch wasn't an option since the search engine couldn't find any papers about MFEs.

## Interview

The interview was divided into three parts, the challenges of working with MFEs, importing/exporting them, and lastly wishful thinking when working with them. The first part of the interview aims to find out how knowledgeable the interviewee is with MFEs and what difficulties they've had working with them. The follow-up questions were mostly asked from the six honest men's perspectives [20] to let the interviewee talk about the technical- or political aspects of working with MFEs. The second part goes into what challenges they face when importing and exporting MFEs. Same as the first part these questions aim to let the interviewee talk about the technical- or political aspects that come with sharing and using MFEs. Lastly, wishful thinking is where we let the interviewee tell us about their ideal way of importing and exporting MFEs. The aim of wishful thinking is to find out what the interviewees can't do now but would like to be able to do in a year.

Interviews were conducted by one reading the predetermined questions and the other documenting the answers, the one documenting can interject with questions that came up during the writing process. The interviews were also recorded if the interviewee allowed for it which in turn gave us the opportunity for going back and analyzing the interview. After each interview, we would ask the interviewee for feedback on how the questions could be improved, and if there are any suggestions for who to interview next. The interview guide is found in Appendix A.

## Literature study 2

This second literature study focused on researching potential practices and solutions for working with MFEs. Our keywords from literature study one were still in use, they were also combined with new keywords identified from our interview analysis. The combination of old and new keywords was used to find new literature that presents the issues of working with MFEs as well as possible solutions. Once we started literature study two we noticed that peer-reviewed papers and articles going back more than two years were outdated. The world of frontend development is constantly changing, either by introducing new technologies or reintroducing old ones. The larger focus was on reading blogs where senior developers put their experiences with MFEs. The blogs were compared with the sources found in the initial literature study to see if our research and their experiences were built on the same foundation of knowledge. This was also a way for us to think about the blogs in a critical way.

## Experiment

There were two perspectives to consider during the experiment, the importer- and the exporter perspective discovered from literature study two. When working from the exporter's perspective, we took notes when a difficulty or an error occurred and wrote down the necessary steps to overcome this difficulty with the purpose of these steps acting as a guideline for working with MFEs. The same procedures were conducted from the importer's perspective. The experiment ended with investigating the performance by measuring it with the help of the Google Chrome dev tool. The performance that was measured was how long it took for the webpage to render the MFEs.

## 2.2.2 RQ2

Currently, there is no way to find and publish MFEs at IKEA without communication. To explore the options of **RQ2**, the theory of conceptual design [27] will be applied. This concept helps generate multiple design alternatives by choosing the best option based on criteria such as feasibility, functionality, and user experience. Apart from that, the conceptual design allows for proactive problem-solving when identifying potential challenges and limitations when trying to implement a solution.

Before designing a proof of concept interviews were conducted with developers to get an understanding of how they currently work with accessing services and their hardships. There are multiple options for gathering data for **RQ2**, this could be done with structured interviews, questionnaires, or semi-structured interviews. All of these options would result in the same information. With questionnaires, we could reach many more people at IKEA but weren't sure how fast the response time would be. Structured interviews would lead to quantitative data that would be useful for the design phase, but this would leave out the insights of developers to potentially uncover novel ideas and assist with the design phase, rather than including our own preconceptions. So we opted for using semi-structured interviews as our primary source of data gathering. Different teams at IKEA were interviewed because each team works under different conditions and their mind maps will differ this will help us cover different difficulties under different contexts.

Next in line was to conduct a conceptual design. The objective was to present a lo-fi prototype [27] for the people at IKEA based on the data gathered from the interviews on developers' desired workflow for discovering MFEs. Letting developers test the prototype can help identify issues and opportunities for improvement with a focus on functionality and user experience instead of how visually appealing the prototype is. With a less visually appealing prototype people tend to be more honest with their feedback because they are aware that not much thought was put into the prototype.

The proof of concept will be the hi-fi prototype [27]. The prototype will incorporate the feedback received during the lo-fi presentation and will act as a second iteration of the design phase. The hi-fi prototype will act as a quick and dirty solution that shows the people at IKEA a polished and professional representation of the product. Its functionality will not only be useful for finding and publishing MFEs but also for other services and will answer **RQ2.1**.

### Interview

Before being able to design a lo-fi prototype, interviews were conducted to get developers thought processes for accessing APIs or other services. The interview started off by asking how the developers go about their way of accessing a service. The interviews for answering **RQ2** didn't have predetermined topics like in **RQ1**, the focus of these interviews was to get the developer's mind map of how they access services. The follow-up questions would then be asked from the six honest men's perspectives to let the interviewee talk freely. The interviews were conducted in the same manner as **RQ1** and the interview guide can be found in Appendix B.

## Design phase

When the data was gathered from the interviews, we read the interview notes and tried to pick the most important information from each interview. The key takeaways from the interview notes were observed and we tried to connect the dots from these notes to be able to find if there was a common problem that could be mitigated, regardless of what team and context you were working on. The refined data laid the foundation for creating a visual representation of the current workflow in the form of a graph and thus discovering more clearly the flaws in the workflow. A lo-fi prototype was created that assessed these flaws. The prototype was intended to be shown to the people at IKEA to get feedback on the user experience when searching for an MFE and thus being a setup for getting a step closer to answering RQ2.1.

## Proof of concept

The proof of concept will act as a quick and dirty solution to the improvements identified from the lo-fi presentation and will be presented as a dashboard powered by React. This dashboard will support the functionality to help developers find and publish MFEs.

# 2.3 Theoretical fundamentals

The purpose of this section is to present the theory that will be used as a foundation for this thesis. This section is for the reader unfamiliar with the concepts that will be mentioned and used throughout this thesis. If one is unsure about the theory presented, this section will also act as a source for further reading if one wishes to do so. The section will start off by presenting how React works and how it affects the DOM. After React, an introduction to the software architecture that is MFEs will be presented, and then what Module Federation is. Lastly, an introduction to Web Components and how it comes into play with the rest of the theory.

## 2.3.1 React

React [6] is an open-source JavaScript framework and library created by Facebook. It is used to build user interfaces and develop web applications more efficiently with much less code than only using vanilla JavaScript.

Usually, when you request a webpage you type it in the URL in the web browser. This request will be sent by the browser(client) to the server, which will respond with the corresponding HTML file and eventually, the browser will render the page. Each time you click on a link on the webpage, the same client-server process will be conducted to get the new page. Before rendering the page, the browser will create a tree-like structure from the HTML file. This structure is called the Document Object Model, also known as DOM, which JavaScript manually manipulates each time the data changes. Every time the DOM gets manipulated, the whole page reloads again.

React creates something that is called the Virtual DOM, which basically is a copy of the original DOM. Whenever there is a change in data state, the Virtual DOM will be updated. When it gets updated, React will check the difference between the Virtual DOM before the

update and the Virtual DOM after the update to see what elements have changed and only update that element on the original DOM. This minimizes the number of DOM operations when re-rendering the UI.

React diminishes the constant back-and-forth communication between the client and server and thus solves the problem of not constantly having to communicate with the server and reloading the whole page each time you request a new resource or page. This is accomplished by React taking the approach of building a single-page application (SPA). A single-page application loads a single HTML file only on the first request and uses JavaScript to update the specific portion of the page that has been manipulated on the client side. This way of building a web page leads to better performance and a more dynamic user experience.

### 2.3.2 Micro frontend

In order to fully understand the research questions it is necessary to present the theory of MFE [22] and its importance as to why huge companies like IKEA aim to use it. Before being able to explain what MFEs are, an explanation of Microservices will first be presented.

Microservices [23] solve the many issues that come along when working with monolithic code bases. Working with a monolithic code base in large companies only brings negatives, they halt the production of new features, there is a higher chance of breaking APIs, and good luck to the developer that has to find a bug in the tens of thousands of lines of code that exists [22], so "Microservices are independently releasable services that are modeled around a business domain. A service encapsulates functionality and makes it accessible to other services via networks" [23]. Each team of the independent department that creates a service has its own development stack, resulting in complete autonomy. Splitting up a monolithic codebase into multiple smaller ones is called microservices.

The principles that microservices are built on are applied to MFEs as well. Microservices and MFEs aim to solve the same thing and that is to improve the software architecture in the long run by having complete autonomy between developers [22]. Although they are built on the same principles the use of technology is not the same.

The analogy of a house comes to mind when explaining frontend development and backend development. The frontend only has one front door to enter and exit, while the backend has multiple back doors as well as emergency exits if need be. Due to only one front door, the browser is only able to run one version of a framework at a time. There are ways to circumvent but this requires multiple people to go through the front door at the same time [31]. Due to browser restrictions, the use of multiple frameworks or different versions is heavily discouraged [22]. Forcing different versions of a framework can create unintentional bugs and multiple uses of frameworks will increase the bundle size of an application making it slower [31][22].

### 2.3.3 Module federation

Module federation [5] is a JavaScript architecture that is used as a native plug-in in bundlers such as Webpack. It allows developers to share JavaScript code and dependencies both synchronously and asynchronously during runtime between different code bases. The dynamic loading of JavaScript chunks and dependencies during runtime reduces the appearance of

code duplication in the application, where missing dependencies will be downloaded by the host.

An application that uses Module Federation is composed of two parts, which are the host and the remote. The host represents the container that contains one or more MFEs or libraries being loaded. The remote represents the exposed MFE or libraries that are being loaded inside the host during runtime, where one or more objects can be used by the host when it is lazy loaded into an application.

Sharing external libraries across multiple MFEs without the fear of producing clashes during runtime is one important feature that Module Federation has introduced. This feature gives the developers the option of specifying which libraries are shared between the MFEs and thus only loading one version of the library for all micro frontends that use it.

## 2.3.4 Web Components

Web Components [8] are a set of three distinct technologies that allow developers to create reusable code with their functionality encapsulated from the rest of the code base. The three APIs that fall under the umbrella that is Web Components are: Custom Elements, Shadow DOM, and HTML Templates [17], the only one that is of interest for this thesis is the Custom Elements API. Where Custom Elements is "a set of JavaScript APIs that allow you to define custom elements and their behavior, which can then be used as desired in your user interface." [8].

All major frameworks are capable of creating Web Components some better than others. Where React isn't one of the better ones. Data or props passed down to Web Components will be converted to a string, if developers were to pass rich data like objects or arrays the developer would need to use JSON functions like parse and stringify when handling data. React also implements its own synthetic event system meaning "it cannot listen for DOM events coming from Custom Elements without a workaround" [1]. Using Custom Elements with React is cumbersome but there will be a future version of React that will include more support for Custom Elements [6]. Although working with Web Components could be cumbersome it is also stated in a survey that Web Components were one of the most used solutions for building MFEs where they play an imperative role for either sharing or encapsulating MFEs [22].

The encapsulation or wrapping of MFEs as Web Components brings two advantages which are Custom Events that allow for communication and abstracting differences between frameworks or framework versions [31]. Web Components allow MFEs created by different frameworks to communicate with one another, this isn't only limited to different frameworks but also the same framework using a different version.





# Chapter 3

## Investigating issues and solutions of sharing MFEs

---

Sharing MFEs is considered to be challenging when working in larger companies such as IKEA. Many teams are involved in developing different features and each codebase consists of different technologies, which introduces compatibility issues for frontend applications. Therefore, the purpose of this chapter is to present what types of problems exist when working with MFEs in the IKEA context. The section begins its analysis by introducing an initial investigation, then a more in-depth investigation, and concluded with experimentation along with their results. This chapter will focus on the technical demands for sharing MFEs while the following chapter will focus on the challenges of finding and publishing MFEs.

### 3.1 Initial investigation

This section will describe our initial discoveries about the problems when working with MFEs. The initial discoveries of MFEs were to help us get more acquainted with the subject and build up a strong vocabulary for upcoming interviews. The result from the discoveries will be used as a foundation for the upcoming section for motivating as to why we chose a certain path when doing a more in-depth investigation about sharing MFEs.

Sharing code between MFEs owned by different teams in a distributed system can be challenging. It requires coordination to ensure that system agility and evolution are not compromised. When you want to introduce MFE architecture in a company, it's important to think about multiple aspects. One important aspect is how you choose to compose MFEs. There are three ways how you can compose MFEs. These three alternatives are client-side composition, server-side composition, and edge-side composition. The challenge when composing MFEs on all three alternatives is to handle state management [22].

There are two types of components that an MFE consist of, which either is a static component or a dynamic component. These components can be integrated during run time or

during build time. If you choose to work with client-side rendering, then it will render the components during run-time. This allows for a more responsive and interactive user experience. It is preferable to work with more interactive and dynamic components with client-side rendering because it allows for greater flexibility and adaptability to business needs. You can also integrate the components during build time by using the npm packages, which can be more efficient but it may not be as responsive to new npm package updates. Also, it's less flexible because you need to build and deploy your whole application every time you change your MFE [22]. According to an interview with a person at IKEA that works with edge-side composition said; "If you were to develop a dynamic application, which is based on personal content, meaning it's non-interactive and requires caching at the Edge but also high performance, then the edge-side composition is preferable". Server-side rendering is preferable to use when you have a static page that consists of static components due to its superior performance. This is because the components get pre-rendered on the server and thus eliminate the need for the browser to render them on the client side.

When it comes to sharing MFEs, there are different methods that can accomplish that. The first one is Single SPA [7], which is one of the most popular open-source frameworks for MFEs. The problem with using this framework when sharing MFEs, according to the people at IKEA, is that it's not scalable. The motivation behind this is that there is a risk of being too dependable on an open-source framework that may lead to heavy rewrites on codebases if the framework suddenly stops being supported. Another technology that can be used for sharing MFEs is Module Federation. The problem with Module Federation is that it doesn't work when trying to include MFEs which consists of different version of a framework. Also, if one were to go with the multi-framework approach, it's doable but the application will take a performance hit because all libraries from each MFE will have to be loaded before the browser can render the page [31]. Lastly, working with Web Components can also be used when working with MFEs. The problem with Web Components has already been introduced in the section 2.3.4.

Although Module Federation came as a new feature with Webpack 5, it can be used by other bundlers. There are multiple JavaScript bundlers but the most popular ones are Webpack, Roll-up and the rising star Vite [11]. The friction caused by teams at IKEA is the threat of having to change one's tech stack and during some of the interviews at IKEA, Webpack and Roll-Up were the two bundlers in use. We couldn't find any information on how to use Module Federation between two different bundlers other than support for Vite and Webpack exist [26]. Due to time constraints, we decided not to look further into this.

To conclude our initial investigation, we have decided to dig deeper into the following bullets:

- Components: Dynamic/Static components (non-interactive) vs interactive components
- Feedback: Run- vs Build time vs SSR/Edge
- Sharing: Module Federation + Web Components

The first bullet refers to what kind of components are created at IKEA. To better understand the complexity of sharing MFEs, we investigated the types of components that IKEA would like to share, as the level of complexity can vary. The second bullet is to investigate how the different integration types affect how you obtain feedback, which influences the level of communication overhead. The final bullet is to investigate how these two MFE implementations can help different teams at IKEA keep their autonomy.

## 3.2 More in-depth investigation

The purpose of this section is to narrow our scope even further for what we believe is of the highest importance when sharing MFEs at IKEA without causing any friction, by presenting the arguments as to why we have chosen some of the contents in each bullet from the bullet list in the previous section. Some of the contents, which won't be the focus of this thesis, will also be presented and discussed to strengthen our arguments as to why we choose to exclude them.

It wasn't clear as to what "Components" the people at IKEA would like to share. From the interviews we gathered, both non-interactive and interactive components are created and are to be shared. When working with MFEs, there are three ways to compose a frontend application. These are client-side, server-side, or edge-side each with its own problems already stated in the previous section. Working with interactive components on the client side fits the needs of the team at IKEA we're working with where performance isn't the main goal but rather minimizing code duplication and increasing code reusability without causing friction. MFEs at the `IKEA.com` domain have a larger focus on performance where server-/edge-side composed frontend applications are more suitable approaches. A reasonable question would be "Why not combine all of these compositions" but there is close to no information about combining the three. Server-side rendering can only provide non-interactive components and as of today, there is no standard to send a state to the server [6]. Edge-side composition together with Fastly [2] has been a large focus at IKEA recently and can provide non-interactive components mentioned by one of the people we've interviewed at IKEA but the edge-side composition has the same state management issues as server-side composition. With there not being enough information about Fastly and Server side rendering, we've decided to exclude them from this thesis. The focus will be on client-side composition together with interactive components.

The aim for the people at IKEA is to decrease the communication overhead by improving their "Feedback" loop for new releases. There are two ways to integrate code into the application, which are run- and build time. One of the teams at IKEA doesn't want to work with build-time integration because it leads to massive communication overhead, and if they were to miss one version of an npm package the risk of that build breaking the application is high. With run-time integration, the communication overhead is eliminated. If the latest version of an MFE breaks the application, it's one Slack message away, and the element of surprise after build-time is gone. Although run-time integration leads to less communication overhead, it brings forward another problem of increased load size to the browser. It was mentioned in section 2.3.3 the shared object in Module Federation has to be removed in order for two different versions of React to run at the same time. With the shared object removed, the browser will load in each library separately even if they use the same version of React [31]. On the other hand, when working with build-time integration, the npm package will install the missing libraries for the host application and thus have less load size [4] than run-time integration. The general consensus of using MFEs with Module Federation is to use the same framework and version since it's nigh impossible to test if a range of framework versions are compatible with one another [31]. In this thesis, we want to explore the multi-framework approach when working with MFEs, which is considered to be inadvisable [22], but the team where we performed this thesis at aren't interested in performance so this is a suitable approach.

Even if the people at IKEA work with run time or build time integration, the challenge of "Sharing" MFEs while maintaining complete autonomy doesn't change. The issues of running multiple versions of a framework as mentioned earlier can lead to unintended behaviors. One way of solving this issue is as stated earlier, to test a range of versions to see if they're compatible but that isn't a scalable approach. The people at IKEA can agree to run a single version of their desired framework but they don't want to do that because that could lead to massive code rewrites, as stated in an interview with the people at IKEA. Upgrading to a newer version or downgrading to a lower version of React may require changes to the existing codebase to ensure compatibility. Depending on the size and complexity of the codebase, this involves quite a bit of work. With code changes, there is a risk of introducing new bugs or issues that need to be identified and fixed. Using the same version of a framework requires an initial investment in code rewrites and a huge communication overhead to ensure consistency of framework version across all teams at IKEA. There are positives to using the same version of a framework. The people at IKEA would ensure consistency, where all developers are working under the same conditions and the application performance is the same throughout. Running the same version of a framework ensures that all components and dependencies are compatible with each other, reducing the likelihood of bugs. The developers at IKEA would also spend less time dealing with compatibility issues and more time focusing on building new features and improving the application. More work is required to ensure that a multi-framework approach works, where different technologies need to be combined. The people at IKEA think that the cost of the rewrite outweighs the multi-framework approach because they don't want to cause friction by changing the existing team's technology stack. Single-SPA was mentioned briefly as a solution, where the application shell is independent and can run multiple frameworks at the same time [7]. One can build their own version of a Single-SPA but the uncertainty of something better than their own implementation appearing can lead to wasted time and money and is therefore not an option IKEA is interested in doing.

To summarize, the focus of this thesis will be to share interactive components through run-time integration. The people at IKEA don't want to run the same version of a framework or rely on third-party multi-framework solutions. To ensure complete autonomy for the teams at IKEA, Web Components will be used to share MFEs with a multi-framework approach because this will ensure that the existing tech stack won't change.

### 3.3 Experiment and results

The purpose of this section is to present our findings from the experiment being conducted when sharing MFEs via Module Federation + Web Components and address the underlying issues with different React versions running at the same time. Rather than just reading the literature and presenting what is required for something to work in theory, we wanted to try out the theory and present our hardships of working with it. Developers who are looking into the same theory will most likely encounter the same issues presented in this section.

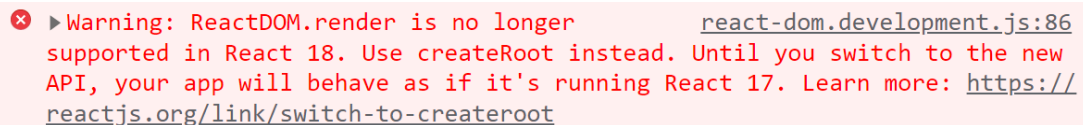
We decided to create our own MFEs from scratch because the team at IKEA is using mono repo with Nx [25]. A mono repo is a single git repository that holds multiple libraries and applications. Nx provides tools and features which streamline the development of large-scale applications. Nx has its own Module Federation configuration which would require us to study Nx before being able to set up our own project. So instead we opted to use an npm

package created by Jack Harrington called *create-mf-app*, which is a lightweight project that can be used to set up prototypes quickly [18]. This package helps create a Module Federation-ready application with the Webpack configuration file already complete.

### 3.3.1 Web Components

This section aims to show the results of how the application behaves when working with incompatible React versions. Running different versions of React at the same time can lead to unintended behavior which is hard to debug and in some cases breaks the application. The results of working with Web Components will be highlighted in two parts: state management, and developer experience. State management will focus on issues of trying to share states between different versions of React wrapped as Web Components and developer experience will present the requirements for sharing MFEs.

Our application shell in this case was a React component that used React version 17. Our remote MFE, which used React version 18, was a React component, wrapped inside a Web Component. The shared object in the Module Federation plugin was removed on both MFEs so that we don't force the highest compatible version of React to run at the application, which in this case is React version 18. Without wrapping the remote application as a Web Component, the browser issued an incompatibility warning, see figure 3.1. The browser did not have any complaints or warnings when running incompatible versions of React and thus the application worked as intended. It took the application shell approximately 600ms to load and render its contents to the browser. This performance hit was because the shared object was removed and had to load in all libraries from each MFE application.

A screenshot of a browser warning message. The message is displayed in a light red background with a red 'x' icon on the left. The text of the warning is: 'Warning: ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot'. The file path 'react-dom.development.js:86' is visible on the right side of the warning.

**Figure 3.1:** Running a remote React 18 on a React 17 application shell

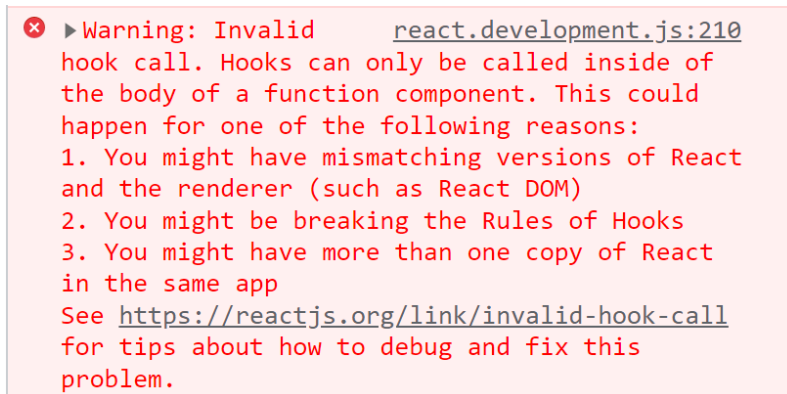
### State management

Although Web Components solve the issue of running multiple versions of React at the same time, this now presents an issue of how to share data and state between MFEs. This section will present our attempts and findings on how to share states across MFEs.

At the beginning of the experiment with sharing data between React components and Web Components, we tried to share data in the same way as one would normally do when working with React. When we tried to send props from a React component(parent) to a Web Component(child), we noticed that the instance of the prop sent by the parent didn't update correctly to the child. The first thing we noticed was that all attributes that were sent through a Web Component were a string. Apart from that, the biggest reason and the initiating problem as to why the prop is not updating correctly is because of how the wrapped component is created. When we wrap our React component inside a Web Component, we create a React root and render it inside its lifecycle method *connectedCallback*. This method gets called only once when it gets mounted into the application shell DOM. This means when

the prop gets updated, the React component inside the Web Components will not be re-rendered with the updated data. We also tried to update the prop using the lifecycle methods *attributeChangedCallback* and *observedAttributes*, which gets called every time the attribute gets changed but the result was the same where React didn't allow us to create a new root.

The next thing we tried to share state was by trying out the Reacts Context API, Zustand, and Redux. The result in all of these three ways was the same, the browser complained that the hook rule was broken because all the state manager libraries that were used were based on using React hooks, see figure 3.2.

A screenshot of a browser warning message. The message is displayed in a light red background with a red 'x' icon in a circle at the top left. The text is as follows: 'Warning: Invalid hook call. Hooks can only be called inside of the body of a function component. This could happen for one of the following reasons: 1. You might have mismatching versions of React and the renderer (such as React DOM) 2. You might be breaking the Rules of Hooks 3. You might have more than one copy of React in the same app See https://reactjs.org/link/invalid-hook-call for tips about how to debug and fix this problem.' The file path 'react.development.js:210' is highlighted in blue.

```
✖ ▶ Warning: Invalid hook call react.development.js:210
hook call. Hooks can only be called inside of
the body of a function component. This could
happen for one of the following reasons:
1. You might have mismatching versions of React
and the renderer (such as React DOM)
2. You might be breaking the Rules of Hooks
3. You might have more than one copy of React
in the same app
See https://reactjs.org/link/invalid-hook-call
for tips about how to debug and fix this
problem.
```

Figure 3.2: React hook rules were broken

To circumvent the problem of props drilling between a React component and a Web Component, we tried to share data using Custom Events and PubSub. Both methods are based on the same principle and use the window object in the browser to share data as a global state. This way of sharing worked as intended, the parent updates the prop and dispatches the event together with the prop to the window object. Every listener or subscriber to the event will get the signal that the data has changed and thus re-render the component with the help of the *useEffect* hook.

## Developer Experience

When it comes to the developer experience of working with Web Components and state management, it adds extra effort and boilerplate code to be able to make it work seamlessly.

It doesn't require much effort to wrap a React component inside a Web Component. Trying to use Web Components' own life cycle methods requires significantly more effort and adds more boilerplate code which sours the developer experience.

When implementing state management using Custom Events and PubSub, the efforts and the size of the code do vary even though their principles are similar. When working with Custom Events, the parent components need to create a Custom Event for every state change, an example could be to increment or decrement a counter. The child component(s) needs to add an event listener for each Custom Event created in the *useEffect* hook, which adds a lot of additional lines of code. The developer experience is not the best because it loses the benefits of working with React and instead leans more towards working with vanilla JavaScript. When working with PubSub however, the developer experience is better in the way that you do not need to add much additional code in either the parent component or the child component. Also, working with PubSub makes it feel like you still use the benefits of working with React.

In terms of the size of the workload between the importer and exporter, we found out that it's much heavier for the latter. The exporter has the responsibility to create the wrapped component, expose it, and prepare the data for state management so it works as intended with PubSub or Custom Events. The importer only needs to import the MFE and also prepare the application to listen when the shared state changes so it gets re-rendered properly.

### 3.3.2 Cross bundler MFEs

JavaScript bundlers are used to divide code into smaller chunks which are then loaded into the browser to improve browser performance [16]. Bundlers use Module Federation to transport these chunks between applications, and the purpose of this section is to present our findings of working with different bundlers' support for Module Federation. These findings will help the teams at IKEA avoid changing their tech stack when working with Module Federation. We mentioned in the previous section that the two bundlers in use at IKEA are Webpack and Roll-up, but we decided to try Webpack + Vite because Vite adopts Roll-ups plugin API and infrastructure [33]. And because Vite is built upon Roll-up and their syntax is similar, we made the assumption that if Vite can work with Webpack, so can Roll-up.

The purpose of this experiment is to show that using Module Federation with different bundlers work. There is literature out there mentioning that Module Federation between different bundlers works but we wanted to experience the challenges firsthand and see for ourselves what it was like to work with it.

We started with Vite as the host and Webpack as the remote. Vite couldn't access the resources from Webpack which resulted in an error, see figure 3.3. For Vite, to be able to access Webpack resources the following configuration was added to Webpack:

```
headers: {
  "Access-Control-Allow-Origin": "*",
  "Access-Control-Allow-Methods":
    "GET, POST, PUT, DELETE, PATCH, OPTIONS",
  "Access-Control-Allow-Headers":
    "X-Requested-With, content-type, Authorization",
},
```

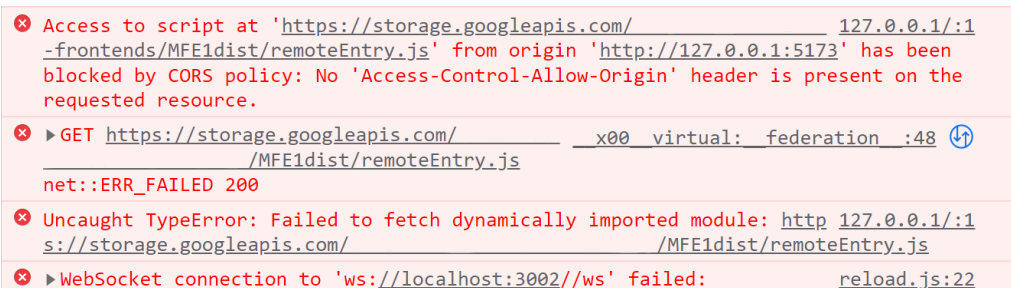


Figure 3.3: Vite can't access the resources from Webpack

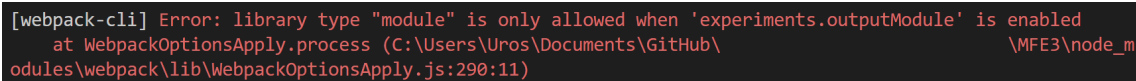
This allows Vite to access the resources from Webpack but it still didn't work because Vite had trouble understanding what library type was used by Webpack. At the time of getting this error, we didn't know that Webpack uses the CommonJS module system while Vite uses

the ES Module system. Vite doesn't have support for other systems than ES module [16] while Webpack does [35]. This required Webpack to add the following configuration:

```
library: { type: "module" },
```

This resulted in another error in the IDE, see figure 3.4. The following configuration was required to be added:

```
experiments: {  
  outputModule: true,  
},
```



```
[webpack-cli] Error: library type "module" is only allowed when 'experiments.outputModule' is enabled  
at WebpackOptionsApply.process (C:\Users\Uros\Documents\GitHub\MFE3\node_modules\webpack\lib\WebpackOptionsApply.js:290:11)
```

**Figure 3.4:** Webpack error with the use of library type module

The experiments tag is a feature added by Webpack to encourage users to try out new features [34].

With these three configurations added to Webpack, Vite is able to federate MFES. Vite requires no changes to its configuration as a host other than the standard of adding the remote files.

In the following experiment, we aim to see if the reverse works as well, where the host application uses Webpack and the remote Vite. Webpack is required to have the added changes except for the CORS configuration. Vite is not able to provide a remote entry file while in development mode. Vite is required to build the MFE application and then serve it. The following configurations need to be added by Vite:

```
build: {  
  modulePreload: false,  
  target: "esnext",  
  minify: false,  
  cssCodeSplit: false,  
},
```

The tag inside the build object that is of importance is the "target" tag, which specifies what ECMAScript version the code should be compiled to [32]. After building the Vite application and serving it, Module Federation didn't work, and on top of that, there was no error message in the browser or the IDE. We mentioned earlier that Webpack uses the CommonJS module system and Vite uses the ES module system. This required us to re-compile the Webpack application from CommonJS modules to ECMAScript modules. After converting CommonJS modules to ES modules, Module Federation between the applications work. One thing to note is that the development process with Vite as a remote was tedious. Every time the Vite MFE is updated, it required a new build to register new changes at the host's end.

To conclude our experiments with Module federation between two bundlers, the underlying issue is that different bundlers have different module systems. As long as there is support for their module systems, Module Federation between different bundlers will work.



### 3.3.3 React-ive ways of sharing MFEs

After our experimentation with sharing MFEs, we found Module Federation's official GitHub page. The GitHub page has examples of sharing components between different versions of React components without wrapping them as Web Components. We wanted to try out these examples and see how they compare to our findings when working with Web Components and if these examples are of value to IKEA. The two GitHub repositories in question are dubbed "Isolation" [15] and "16-18" [14].

It's possible to share MFEs without wrapping them as Web Components. These are React-specific solutions and mesh well with what the team at IKEA wants to do with MFEs. Both of the solutions are able to pass props as one would normally do when working with React. Their differences are that "16-18" has a more natural way of passing props while "Isolation" requires one to pass props inside a `useEffect` hook. This would require the developers to work with the dependency array to ensure that the page re-renders correctly. When it comes to the "16-18" solution, it uses the deprecated React class library to be able to render and update the remote React MFE. We tried to update the deprecated class library to React's standard of function components, but this gave the same error as seen in figure 3.2. This leads us to believe that working with a global state manager isn't possible here either.

The "Isolation" solution has more readable code and is akin to the developer's experience with Web Components in the sense that the importer only needs to import one file. With the "16-18" solution the exporter exposes the remote while the importer needs to prepare more on their end to accept the remote application.

### 3.3.4 Working with the cloud

IKEA's primary way of storing code is via the cloud and from one of the interviews, they mentioned that they want to store their MFEs there. This section will present our findings of what is required for them to work with MFEs in the cloud.

When working with Module Federation using the cloud, each MFE needs to be built and then uploaded. When working with the cloud, the built application is uploaded and subsequent chunks have to change their visibility from *private* to *public*. To be able to access the MFE from the cloud and display it, the remote entry file needs to have access to the subsequent js files. We didn't try to work with restricting access to an MFE, so anyone with the public URL can access the remote entry file and mount the MFE on their own application. This was all done by hand, uploading the build and changing access to all chunks, a tedious process but MFEs in the cloud are possible. If one updates the MFE, the same tedious process needs to be done again.



# Chapter 4

## How to find and publish MFEs

---

Now that we know what the technical demands for sharing MFEs are, it's now important to raise awareness of how to find and publish MFEs. Even though one is able to share MFEs, not being able to locate them will leave IKEA at the same starting position, where there is code duplication and no code reusability. So by extension of being able to find and publish MFEs, this will result in less code duplication and more code reusability. The initial idea was to create our own proof of concept where developers can find and publish MFEs. But during our research on this subject matter, we found out that IKEA already had an existing dev portal where developers can find and publish code. We felt that it was appropriate for us to change our research method by instead investigating room for improvements when it comes to IKEA's dev portal and comparing it with our ideal dev portal. By doing this, we will present our ideal requirements and compare them with IKEA's dev portal. Rather than creating a new portal with no user base from scratch, we thought it was more appropriate to improve an existing platform for an already established user base. This will result in us presenting an improved version of our ideal requirements and suggestions for improvement of IKEA's dev portal based on our initial requirements. Finally, we will present an improved design of what the dev portal could potentially look like and the intended usage of it when trying to find MFEs.

### 4.1 Our ideal requirements

In order for the people at IKEA to find MFEs, we first need to understand their challenges with accessing services from different teams or departments. Understanding their workflow for accessing services will help us elicit requirements for an ideal way of finding MFEs and how to publish them. This will help the people at IKEA decrease the communication overhead even further and create a one-stop shop where developers can find and publish MFEs among other services, for example, APIs.

The interviews conducted for the second research question discussed the challenges faced

by teams accessing APIs and the communication overhead that results from the process. The primary reason for this is the difficulty in navigating through Confluence [36] websites due to a lack of documentation. One interviewee even stated, "It's like finding a needle in a haystack" when trying to navigate the Confluence page. Some of the interviewees knew of IKEA's already existing dev portal whereas most of them didn't. The people at IKEA heard about the portal in passing and haven't been properly introduced to it and thus haven't looked further into it. As a result, teams rely on communication to access APIs, which can lead to double maintenance and wasted time. To address the communication overhead issues, we suggest the development of a one-stop shop for discovering and publishing MFEs, and other services with clear documentation on ownership, API usage, and requirements for use, which can save time and energy. To conclude the interview analysis, these are the two main problems identified for developers trying to find and publish services at IKEA:

- Communication overhead
- A lack of documentation

Based on a deeper analysis of our interviews, we have created our ideal requirement specification of features that need to be included for finding MFEs:

- The one-stop shop should have a clear and organized structure that makes it easy for users to navigate and find the information they need.
- The one-stop shop should provide detailed documentation for each MFE, including ownership, API usage, and requirements for use.
- The one-stop shop should provide connections with other teams to assist users in finding MFEs that match their desired API or feature. If there are no connections between the published MFE and the search term, then the user can be confident that there are no MFEs available that are related to that team's API or service.
- The one-stop shop should be able to let users upload their own MFEs along with their documentation.
- The one-stop shop should let the user update the MFE.
- The one-stop shop should have a search function that allows users to quickly find specific MFEs and their relations to other teams.
- The one-stop shop should be accessible to all teams within the company, regardless of location or department. Sharing MFEs should be done globally at IKEA and not constrained locally.
- The one-stop shop should have the contact information of the MFE owner.
- The one-stop shop should have a search function that allows users to quickly find specific MFEs based on keywords or other criteria.

The elicited requirements help with minimizing communication overhead and the documentation is only as good as the developer typing it out. We believe that if IKEA's dev portal fulfills our ideal requirements, it will lead to a growth in reputation and thus increase awareness and usage. This will lead to developers finding and publishing their MFEs to the dev portal which in turn removes code duplication and increases code reusability.

## 4.2 Comparing ideal requirements to IKEA's dev portal

As stated earlier, IKEA has an already existing platform for finding and publishing code. Instead of focusing our efforts on creating a new proof of concept, we will evaluate IKEA's dev portal and compare it to our ideal requirements for finding MFEs. Comparing our ideal requirements with the existing dev portal will help us get a deeper understanding of how to improve the platform. The comparison will result in a new improved list of our ideal requirements as well as identify improvement areas for IKEA's dev portal. IKEA's dev portal is built on Backstage, which is an open-source network created by Spotify and it enables teams to deliver code quickly without compromising autonomy [9].

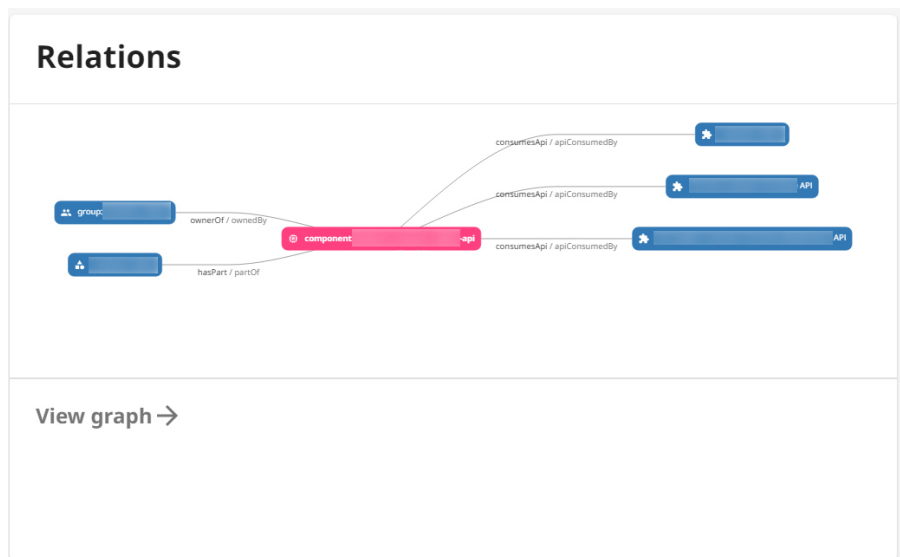
When we compared our ideal requirements with IKEA's dev portal, we attempted to execute each requirement on the portal. We noticed that some requirements were missing while there were features the dev portal has that we considered were good and didn't think about. Our requirement for letting users upload their own MFEs is reliant on the users taking it upon themselves to keep the MFE up to date. IKEA's dev portal has this feature but takes a step further. Once the developer has uploaded their MFEs to the portal, the MFEs will update each time the developer has pushed an update to git. IKEA's dev portal will be in sync with git and thus ensure that the code base is always up to date. Although the MFEs are always up to date, the documentation is still reliant on the developer to change and update it, this is a problem neither we nor IKEA's dev portal can solve. The idea was to create a proof of concept for finding and publishing MFEs, so security was something that completely slipped our minds. IKEA's dev portal has this requirement already in place. To access IKEA's dev portal you need to have proper authentication to IKEA's GitHub repo. The people at IKEA have a team dedicated to improving the dev portal, the portal also provides feedback mechanisms that allow users to suggest improvements and report issues with a feature. This lets the dev portal improve over time with new features giving users a reason to stay as time goes on.

The most critical requirement for finding and publishing MFEs is the search function. When comparing the search function we aimed to implement with Backstage's search function, the difference in complexity is significant. However, both search implementations have a flaw in how developers would go about searching for the unknown. It was mentioned in the interviews that if developers were to search for an MFE, they would go to the team they think the MFE would reside in.

MFEs that are only present in a local scope are typically limited to a specific building or area. If they cannot be found within that scope, it is unlikely that they exist. Attempting to locate them elsewhere would only increase communication overhead and require more time and resources than developing them from scratch. By contrast, developers working within the local scope can benefit from immediate feedback and engage in more detailed one-on-one interactions, which can promote a deeper understanding of how to use the MFE. To reduce the communication overhead involved in finding and publishing MFEs, it may be useful to establish a global Slack channel where developers can announce their MFEs. This would shift the focus away from a local scope and create a global environment where developers can locate and share MFEs without needing physical interaction. While a shared Slack channel may make it easier to find and publish MFEs, there are potential downsides to relying on a

global approach. One issue is the risk of missing important announcements due to inactivity in the channel or the sheer volume of messages. With developers working across different time zones, there is a greater likelihood that important messages will be missed. Despite this drawback, developers can still interact with Slack messages and communicate directly with MFE creators, potentially leading to increased usage.

While a global Slack channel and local scope may have limitations for discovering and publishing MFES, IKEA's dev portal overcomes these challenges by being independent of location and time zones. This means that developers can access the portal at any time without fear of missing important information. However, we should note that the dev portal doesn't have a robust search functionality for team relations. Neither Slack nor a local scope could solve this issue without increasing communication overhead, but we found that team relations are critical when searching for MFES. Therefore, developers at IKEA can search for the team they think has the relevant MFE, and if it does not appear in the search results, it's likely that the MFE doesn't exist. This will help with searching for the unknown. To solve this, we wanted the search function of our own proof of concept to catch the team names used to create the MFE in its documentation. This search function is then completely reliant on the developer keeping the documentation up to date which isn't a realistic approach in hindsight. IKEA's dev portal does the same but has a more complex search functionality that accounts for more than just the contents of a .txt file. The dev portal runs into the same issue of being reliant on developers keeping the documentation up to date. This is where we found something interesting in IKEA's dev portal and that is a relation graph. This graph shows what services and APIs the MFES are composed of, it also shows the owners of these services and APIs, as seen in figure 4.1. This relation graph is only displayed on its own URL page and is not accessible to the search engine. If we were able to extract this relation or add it to the search engine then this would solve the problem of where to look for the unknown.



**Figure 4.1:** Relation graph shows that blurred component is owned by blurred team

Our ideal requirements weren't complete when comparing it to IKEA's dev portal. Backstage, which the portal is built on, has solutions to problems we didn't think of such as security,

maintenance, and updating published MFEs automatically. The ideal requirements list has been improved and added the following, the complete requirements specifications can be found in Appendix C:

- The one-stop shop should have a secure login system that limits access to authorized users and protects sensitive information.
- The one-stop shop should be integrated with the company's existing dev portal with GitHub to ensure updates are automatic.
- The one-stop shop should have a feedback mechanism that allows users to suggest improvements or report issues with the one-stop shop.

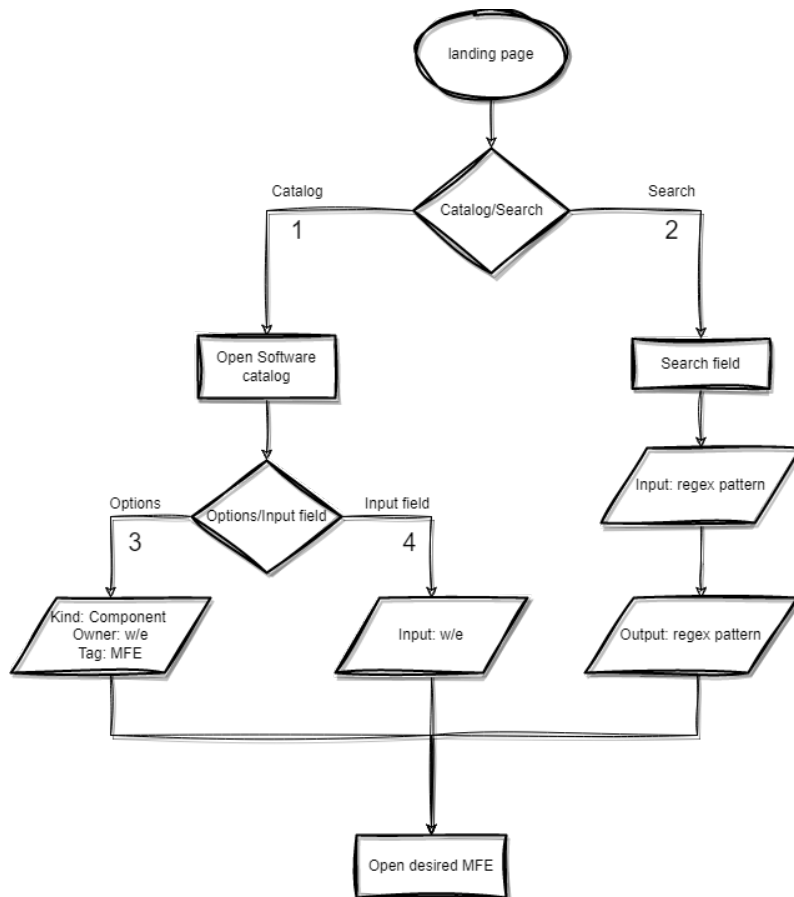
We thought of adding the following requirement to our ideal: "The documentation for each MFE should be regularly updated to reflect any changes to the API or ownership.", this isn't a feasible solution and one we can't overcome. For the graph of the relation to be updated, it requires developers to write what APIs and services were used. After publishing their MFEs, developers only need to update the documentation with any new APIs or services they use. IKEA's dev portal will then update the relationship graph automatically, eliminating the need for developers to republish their MFEs multiple times. This is what we will look into for improving the search and publish functionality of IKEA's dev portal.

## 4.3 Improving IKEA's dev portal

In the previous section, we identified the search and publish requirement of IKEA's dev portal as an important improvement area. Analyzing the portal's current workflow resulted in figure 4.2, which gave us a better oversight of where the improvements could be added to the platform. The underlying intention for understanding the current workflow was to see where the inclusion of relations would benefit the user searching for MFEs. This will simplify the process of finding MFEs for users, even if they're uncertain about what they are looking for. The problems identified from the figure will be used to present user scenarios of how a user should interact with an improved version of IKEA's dev portal. These scenarios will then be presented as design implementation for finding MFEs with feedback from the people at IKEA evaluating the design.

### 4.3.1 Scenario

Currently, there are two ways to find published code in IKEA's dev portal, through its software catalog and its search engine, as can be seen in figure 4.2. Branch 1 refers to the workflow of searching for an MFE using the software catalog and Branch 2 is for searching MFEs using the search engine. These two branches will be presented as user scenarios of how the workflow can be improved, step by step. These scenarios will help us to better understand more precisely where there are issues in the workflow when trying to find MFEs and thus use it as additional input for our improved design.



**Figure 4.2:** Current workflow for accessing services in IKEA's dev portal

### Scenario 1: Searching for API/MFE in search field

Team A is about to start creating a new product, and one of the requirements is using an API from Team C. Pelle from Team A starts by using the search field from the dev portal's landing page, which refers to branch 2 in Figure 4.2. Pelle isn't sure what he wants to search for so he tries his luck with writing "team C API". The output will prompt all of Team C's APIs as well as all the MFEs that have a relation to these APIs.

Now that Pelle has found a huge amount of MFEs to choose from. Pelle is not sure what MFEs to choose from. Pelle is now required to read the documentation to see if any of these MFEs fit his project's needs. Pelle reads up on what is required of him to perform a seamless integration of team C's MFE. Integration with the MFE to team A's project was of no issue but Pelle needed an access token from team C. Luckily for Pelle finding a name to contact from team C was of no issue. Pelle went to team C's group page and found people to contact. He wrote to one of them in Slack and was granted an access token.

### Scenario 2: Searching for API/MFE in the software catalog

Sara from team A is searching for an MFE in the dev portal's software catalog page and she is looking for an MFE from team C. Sara puts in the tag "micro-frontend" in the options field and is shown an array of MFEs, from branch 3 in figure 4.2. Sara is interested in those of team

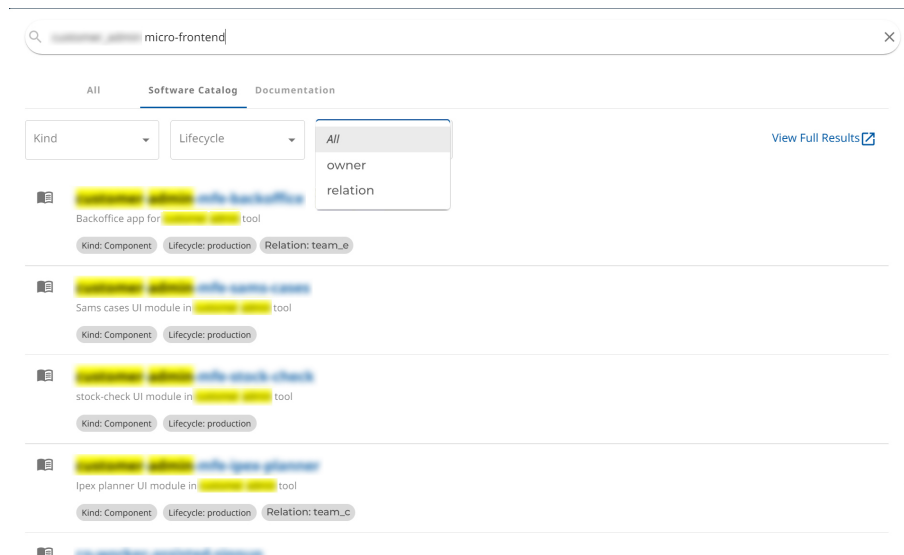


C and with the help of the input field, branch 4 from the same figure, types in “team C” and is only displayed MFEs of where team C is either the owner or has a relation to that MFE.

Sara notices that one of the MFEs she’s interested in is created by team C (owner) but the MFE has a relation to team D. When Sara reads through the documentation she sees that team C used a service from team D to create the MFE. Sara would like an access token for Team C’s MFE all she has to do is contact someone from Team C on Slack the same way Pelle did.

## 4.3.2 Improved design

Now that we have explained the intended workflow and refined the ideal requirement specification on what IKEA’s dev portal needs to have for obtaining the goal of finding MFEs, it’s time to present how the design could look like. Figure 4.3 showcases the first scenario when searching for MFEs in the dev portals search engine. The user searches for a MFE and all MFEs, which don’t have the relations tag, are considered to be the owner of the blurred team name. All MFEs that have a relations tag use services from the team which is prompted in the relations tag. Figure 4.4 showcases the second scenario when searching the software catalog. From our evaluations of IKEA’s dev portal, the second scenario doesn’t have a search engine but a simple filter field. The suggested improvement is to add a relations column where users can use the same logic as with scenario one by typing out their desired team. The rest of our design images can be found in Appendix D.



**Figure 4.3:** User searches for an MFE in the search field

When you enter the MFE documentation, there will be a graph with a visual representation of its relations to other services of how the MFE was composed, see figure 4.5.

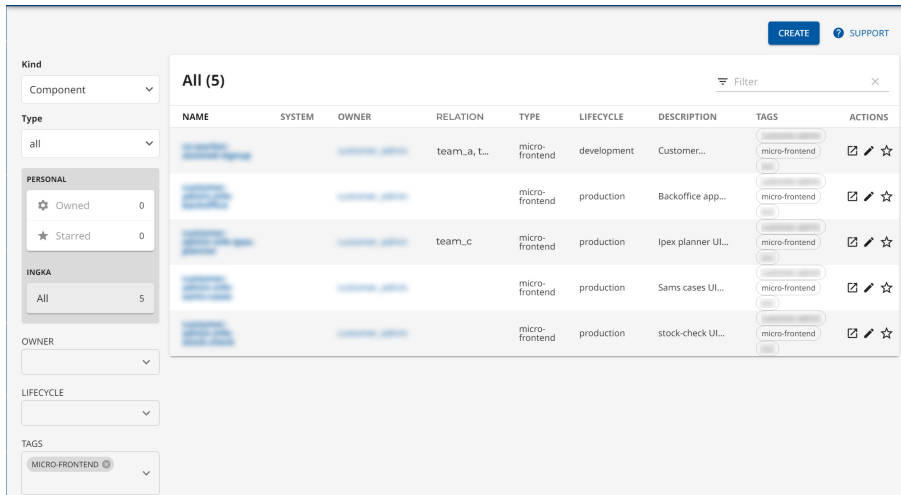


Figure 4.4: User searches for an MFE in the software catalog

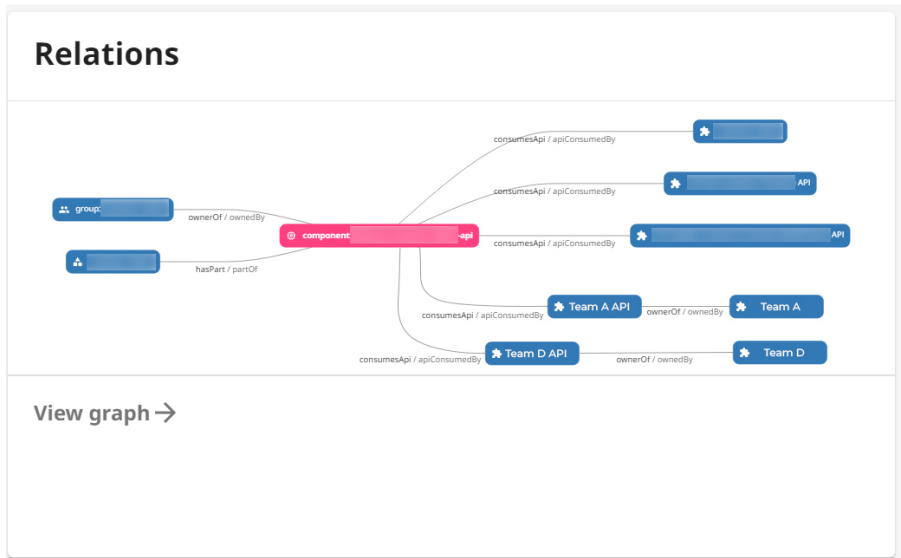


Figure 4.5: Relations graph for the MFE in the middle

## Validation of design

The improved design was presented to the developers that maintain the dev portal with the purpose of finding out if it was feasible to implement the features or not. If it's not possible to include the features presented above, then IKEA should consider changing to another dev portal, where it's possible to accomplish this or create their own dev portal to ensure code reusability. According to the developers, it's possible to implement our suggestions. The maintainers of the dev portal provided us with a JSON file that contained data on all the relationships between various services. However, this file was massive. With the provided JSON file at hand, we soon realized that the implementation of our proposed feature was a complex task and one that couldn't be accomplished in a single afternoon. We recognized that it would require a significant amount of time and resources to fully comprehend the data provided, and then it would require more to improve the search engine.

To conclude this chapter we created our requirements specification and compared it with the dev portal, which resulted in an even more complete specification, see Appendix C. We identified an area for improvement in their search function, where it should support relations to the published MFEs, see Appendix D for the complete design. These relationships will contribute to the search for the unknown by displaying all types of relationships that the MFE is built on, as shown in Figure 4.5.



# Chapter 5

## Discussion and Related Work

---

Before concluding our thesis work, we will in this chapter discuss our results. We will evaluate our research method and discuss what we could've done differently with the current knowledge we have. Thereafter, we will discuss the validity of our results and then discuss to what extent the results can be generalized to other contexts. We will relate our work to previous research and conclude this chapter by presenting future work.

### 5.1 Reflection on work process

In this section, we will reflect on our work process and discuss it by reflecting on our flaws as well as highlighting our strengths. The purpose of this reflection is to identify what we have learned and how we can improve our approach for future projects.

Overall we're satisfied with our work process with the first research question. With the lack of knowledge that we had for this subject matter at the beginning of the work process, we felt that we obtained the right amount of knowledge from the literature studies and the interview guide we created helped us answer our research questions properly. When it comes to our initial literature study, we read a lot of papers to gain more knowledge about the principles of working with MFEs. The initial literature study was there to help us improve our vocabulary about MFEs so that when we interview the people at IKEA we could have more of a dialogue with them about their problems with MFEs. After we interviewed the people at IKEA to understand their initiating problem, we noticed quickly that a lot of information that we thought was important wasn't. An example of unimportant information would be knowing whether IKEA is interested in having one MFE per view or having multiple in the same view [22]. Although we thought that the information wasn't important it helped us make the process of reviewing relevant literature more efficient and effective for a more in-depth literature study. The more in-depth literature study was something we were satisfied with, which helped us scope our exploration even further. When it comes to the experiment, we felt satisfied with the work process and the results that were obtained. It gave us hands-on

experience working with MFEs and not just relying on the theory of working with them.

Even though our initial plans with the second research question didn't come to fruition some aspects of it did, the aspect of creating our ideal requirements specification. The process of eliciting requirements was what we had planned from the start, interviewing the people at IKEA about their challenges when finding and publishing code. It turned out that IKEA already had an existing proof of concept. We didn't think to investigate this earlier in the process because our focus was on exploring the first research question. What we could've done differently is ask if IKEA has an already existing dev portal while we conducted the first research question. As a result of this discovery, we had to adapt by comparing our ideal requirements to the existing platform. With an already existing platform in place, we had more time to validate the platform which resulted in us completing our ideal requirements with ideas from the competing platform. By conducting this comparison, we were able to refine our requirements specification, which we are satisfied with. Had we solely focused on our own proof of concept, we may have overlooked certain issues that we were unable to anticipate due to time constraints. The design process didn't deviate from what we initially had planned out and turned out better because we only focused on one specific feature rather than the whole platform. The interview guide we prepared from the first research question helped us out with the second one. The questions asked were good and we felt that they helped us elicit requirements that are needed for a well-functioning dev portal. Compared to the first research question, we felt that the number of interviews was lacking. We didn't get as many interviews as we did with the first research question. This could be attributed to due to us having to change up our research strategy late into the thesis. If we knew earlier about the existing dev portal, we would've prepared better for how to validate the dev portal more thoroughly, created a different interview guide, and eventually interview the people that maintain the dev portal which we didn't do much of.

We have gained more experience in obtaining information related to our research questions, realizing that it is not only about reading everything with a specific keyword but also about relating it to what we aim to explore and solve. And until next time we know that it would have been beneficial to reassure beforehand whether our research questions aren't already explored and solved after defining them.

## 5.2 Threats to validity

In this section, we will discuss the accuracy and credibility of our results in this thesis. We will evaluate the weaknesses of our results and present ways to strengthen them to increase their validity.

We consider our second literature study to have a high degree of validity despite most of the scientific papers, articles, case studies, and thesis works focused on introducing the principle of MFEs. The frontend industry is a fast-evolving one, and the scientific works we found that were only two or more years old would be considered obsolete due to the old technology used, such as Web Components with vanilla JavaScript. Because of this, we had to rely on blog posts, GitHub repos, and YouTube videos to explore the options of how to share MFEs. The sources we used to conduct the in-depth study all used the same source of information as did we, the book from Mezzalira [22], to ensure their and our validity we would cross-reference their work with Mezzalira.

When it comes to the degree of validity of the first research question for sharing MFEs, we think that the result could be strengthened even further if we were to investigate more extensively how the application would behave when having the shared object enabled in Module Federation. During this phase of the experiment, we felt satisfied by trying to break Reacts hook rules. In hindsight, this reasoning is not strong enough to use to avoid using the shared object. Breaking the hook rules by introducing a hook from one version of another isn't enough to argue the shared objects removal. We feel that we could try out more scenarios where the application shell breaks by forcing it to run the highest compatible version of React. By doing this we would strengthen our arguments for removing the shared object.

A threat to the validity of our experiments is that our MFEs don't represent IKEA's context well. As we mentioned earlier we didn't know what kind of components IKEA was creating so the ones we did doesn't reflect the developer experience that well. Due to limited time, we weren't able to apply our results successfully to IKEA's components. When we tried to wrap their React components as Web Components, we couldn't get the Module Federation in Nx to work. And because of time constraints, we didn't have the opportunity to fully investigate the problem of why Module Federation didn't work.

The threat to the validity of React-ive ways of sharing MFEs is that we didn't spend much time testing it thoroughly, because we found these solutions late into the thesis. At first glance, the solutions provide what IKEA is looking for when sharing MFEs but their future is uncertain. We know that Web Components are future-proof but the React-ive solutions require more experimentation.

Another threat to the validity is that we couldn't implement or test our design propositions. We got feedback from the maintainers that it's doable but we can't know unless it's put into practice. We only got feedback from one developer of the dev portal about our design so it might be possible that our proposed design might not be appreciated by others.

## 5.3 Generalizability

The purpose of this section is to discuss how generalizable our findings are and if they can be applied to any context outside of IKEA.

In general, we believe that our results can be applied to any context. We do that because our experiments were based on the theory related to how one can hide framework-specific rendering systems and component lifecycle methods by wrapping them as Web Components. By doing this, any company looking into the multi-framework approach will be able to work with any technology. One thing to note is that React isn't that good with Web Components and thus needs to be complemented with other technologies to be able to communicate flawlessly, and this could be circumvented using Custom Events or PubSub. Companies using other frameworks that are more compatible with Web Components might not run into the issues React has. The use of a global state manager without dependencies is generally a good practice because one doesn't need to be concerned about whether it's compatible with one's current tech stack if the application is using a multi-framework approach. In the future, React will release an update that better supports Web Components [19], which might alleviate the problems we had when working with it, but that remains to be seen in the future.

The principle of exposing a render method as a function, as the "Isolation" solution did with React, could be applied to any framework that has a render method. We noticed that

frameworks that have their specific rendering systems need to expose their render method as a function and the receiving framework will call its lifecycle method of the exposed function and render it on its page, in theory. This principle could be applied to any framework that is not Web Components based or has its own render method. Web Components-based frameworks or the use of Web Components as we've seen with our results only require an import of the component and no need for the use of lifecycle methods to inject the MFE onto the page. The principles of working with Web Components and "Isolation" have the same objective of isolating the MFEs inside the application shell letting them run as it was its stand-alone application. Both Web Components and "Isolation" can be applied to any context. In regards to the "16-18" solution, this is a React-specific context. We don't know if this solution could be applied to any other framework or if this would work with other frameworks like Web Components and "Isolation" because we didn't have enough time to look further into it.

It's important to note that not all bundlers can share modules with one another. This means that our results may not apply to all bundlers, as they each have their own module systems and varying levels of support for them. It's crucial to research beforehand whether your bundler supports integration with others before going into production. Additionally, it's worth mentioning that the Webpack Module Federation, which enables cross-bundle support, is an experimental feature. This means that our findings could become outdated once it leaves the experimental phase. We believe that this experimental feature will go into production because Webpack has support for multiple module systems to enable Module Federation. Some of the results we've presented might be simplified, for example, by re-compiling CommonJS modules to ES modules.

Based on our findings with the cloud, it appears that IKEA or any other company would benefit from improving its pipeline when uploading MFEs. This would help to eliminate the tedious task of building their application each time there is an update, as well as making every chunk public in order to utilize an MFE.

The design that was presented is considered to be a general approach for finding and publishing MFEs. This is because the dev portal that was analyzed is based on an open-source platform called Backstage. This means that any company that is interested in applying our design can use Backstage to build their own version of a dev portal that fulfills our requirement specification. Furthermore, companies that choose to build their own dev portal from scratch can use our requirement specifications as a template. These requirements are not context-specific, making them easily adaptable for use in other companies or contexts.

## 5.4 Related work

During our literature studies, relevant work to our thesis was discovered. The papers gave an overview of MFEs in their current landscape and before three years ago. We will present four related work papers each with its own summary for the reader to understand its contents and a discussion in relation to the results that we've obtained.



### 5.4.1 Micro frontend architecture for cross framework reusability in practice

The thesis by Smet aims to provide an overview of how the concepts of MFEs can be used to minimize workload by reusing code from already existing frontend applications [13]. Since our thesis is aimed at removing code duplication we deemed this thesis to be relevant.

#### Summary

Smet tries to address the problem of how a consultancy firm can save money by reusing code from already existing frontend applications. The paper aims to address the problems at hand for the consultancy firm by presenting the theory of MFEs, Microservices, and Domain Driven Design but not the underlying causes as to why MFEs don't work as well as the other two theories presented. Smet instead focuses on the underlying issue of reusing code between two different framework applications and aims to provide proof of concept showing that code re-usage is possible. The contributions of this paper don't provide any information that hasn't already been covered in previous literature. Smet relies heavily on the already existing theory by Mezzalira [22] where Smet created a proof of concept of the most common implementations for MFEs, which was web components. Although the proof of concept was built upon common implementations this set up a foundation for Smet to measure performance, technological independence, and Search Engine Optimization. Smet also reused the code in their web components for a React and Angular application. The code was technologically independent, the page loading performance varied between 23 seconds to 94 seconds, and the Search Engine Optimization couldn't find the contents inside the proof of concept. With Mezzalira's theory and Smet's proof of concept, the reusability of code was a success. Although the proof of concept was a success, Smets concluded that their single proof of concept wasn't enough data to answer if MFEs are the way going forward.

#### Discussion

This paper didn't pinpoint what the actual crux of the problem was. There is no presentation of what the underlying causes are for why the concepts of Microservices and Domain Driven Design work in theory with MFEs but not in practice. That's why we believe that our literature study and understanding of the crux of MFEs at IKEA can strengthen Smet's claims of why MFEs aren't working as well as one would hope.

Smets could've presented the actual problem the consultancy firm faced and why the firm was looking into MFEs as a solution to their problem. Smets does a good job of showing that code re-usage is possible but not for example why rewriting React to Angular is not. Working with MFEs according to Smet requires that you would need to learn a new framework that doesn't have the features a team is looking for. We feel that Smet relied heavily on Mezzalira's theory when working with MFEs. In our case, we evaluated Mezzalira's theory and tried to combine the strengths and weaknesses of implementations to show that code rewrites aren't a good solution. We present an argument as to why code rewrites are less appealing and in this thesis case eliminated.

## 5.4.2 Implementing Micro Frontends Using Signal-based Web Components

In this paper, various methods for sharing states among MFEs are examined [24], with a focus on the relevant technologies used in this thesis, namely Web Components and React. These technologies are relevant to our work and contribute to our research and findings.

### Summary

The paper investigates different methods for implementing MFEs and discusses the pros and cons when comparing these. MFE and its architectural style involve combining different applications, which are independently created, to stitch them together and thus create a large single-page application. The problem which arises when different teams work on their own artifact independently is maintaining high modularity and passing data seamlessly between MFEs. To overcome this challenge, this paper tries to resolve it by presenting its solution to the problem with the proposed framework.

The contributions of the authors are to present a new type of approach for implementing MFEs and at the same time maintain high modularity but also enable high declarative workflow. The authors accomplish this by using signals in Reactive programming as an API.

The authors substantiate their claims by demonstrating the effectiveness of their framework by expressing Web Components themselves as signals, which were self-developed by the authors themselves as a JavaScript extension, and used for implementing a single-page application. The implementation of the self-developed signals was then compared to the implementation when using the Web components API. Lastly, they also compared it with the implementation when using the React framework. The single-page application implemented was a shopping site that consisted of different components, where each component was presented as an MFE.

The conclusion is that the implementation of MFEs using signal gives a declarative data flow. When using the Web Component API, it becomes harder to understand the behavior of the application due to the overuse of callback functions being made. When it comes to React, the team division becomes more complex, which is not a big trouble when working with signals

### Discussion

The paper discusses an interesting perspective of the problems when only using Web Components for handling state management between multiple MFEs, which we didn't notice because our application didn't include multiple MFEs that shared multiple states between each other. It has to do with the difficulty of understanding the behavior of the application due to the heavy use of callback functions which makes the data flow between the components hard to follow. This could be used as a strengthening argument as to why the developer experience is bad when trying to share states using Custom Events or Web Components' own lifecycle methods. Signals eliminate the callback functions by replacing all custom events with signals making the code more readable and the data flow becomes easier to follow.

The author's motivation for the signal approach being better than using React seems to have some interesting insights. The first comparison they bring up as to why to use signals is

because props and states must belong to a component and the issue of passing them if there is no parent-child relation, whereas signals don't. This is one of the underlying issues as to why we choose to use the window object for global state management to share states between Web Components and React components. Another motivation, which in our opinion seems vague, is that React re-renders asynchronously when states change meaning that there is a risk that the framework doesn't re-render the component when the state updates. This problem hasn't been noticed when working with our experiment and it has not been brought up by IKEA as a problem. Rather than debating which approach is superior, we could have explored how signals could be combined with React and Web Components for more seamless state management while leveraging the benefits of React for developer experience. Since most companies use a framework to develop applications rather than vanilla JavaScript, such an investigation could have provided valuable insights.

### 5.4.3 Experiences on a Frameworkless Micro-Frontend Architecture in a Small Organization

The paper introduces MFEs to a small company where Web Components are used as an MFE implementation [21]. Since our thesis looked into the usage of Web Components to solve the issues of working with different frameworks we deemed this thesis to be relevant.

#### Summary

The authors aim to investigate and provide more data for the niche market which is MFEs, which currently lacks scientific papers, thesis works, and case studies. Their paper provides a case study of their implementation of MFEs at Visma, detailing their experiences and technical solutions for implementing MFEs. Visma's motivation for adopting MFEs isn't due to scalability problems with their product but rather to create a plug-and-play architecture for their customers with minimal code duplication.

The authors proposed that the developers at Visma introduce Web Components together with LitElement to compose their MFEs. They also used an evaluation framework to measure the implementation of MFEs at Visma.

To substantiate their claims, the authors evaluated Visma's product after completely re-writing it to Web Components. They used an evaluation framework to evaluate the rewritten product with input from developers, resulting in a table of two columns: concerns and relevance. The concerns included reasons for working with MFEs, their benefits, and issues, which were then represented by their relevance in the form of High, Medium, and Low.

The authors concluded that introducing Web Components was successful at Visma as the plug-and-play MFEs resulted in better development, deployment, modifiability, and cost reduction, as per the evaluation framework. Additionally, the authors noted that there is no need to introduce JavaScript frameworks, which could bloat the browser with unnecessary code and hinder its performance.

#### Discussion

The authors did not provide any clear reasoning for their decision not to use frameworks, stating that they had evaluated them and ignored them due to performance. This reasoning

is weak, as there are performance-focused frameworks that don't bloat the browser with code and hinder its performance. In contrast to the authors' use of Web components, we opted to use Web Components to enable a multi-framework approach. Our results showed that this approach does have a significant impact on performance, but the authors did not present any performance results to support their decision. To strengthen their claims about performance, it would have been helpful if the authors had compared vanilla JavaScript with a performance-focused framework and evaluated the developer experience. If the performance-focused framework is found to be preferable, the issue of managing multiple versions of the framework arises. In this regard, our results suggest that this issue can be addressed with either the "isolation" or wrapping a framework component as a Web Component. The authors mentioned the possibility of wrapping components as Web Components but didn't provide clear reasoning for their decision not to use this approach. Our results indicate that working with Web Components can be challenging if they are not well-supported, and may require workarounds. Thus, our findings could help strengthen the authors' claims and provide additional support for their decision.

#### **5.4.4 Evaluating Micro Frontend Approaches for Code Reusability**

The paper aims to provide an overview of how the concepts of MFEs can be implemented to create scalable single-page applications with MFE-ready components [29]. Since our thesis is aimed at removing code duplication and increasing code reusability we deemed this thesis to be relevant.

##### **Summary**

The research problem that this paper attempts to address is how to achieve code reusability and improve scalability in modern web applications using MFE architecture. The paper investigates the possibility of using an MFE architecture to achieve code reusability and evaluates the possible MFE approaches found in the literature. It also provides a technical solution that demonstrates how MFE architecture can be used to better organize the code of an existing single-page frontend application and how the code can be reused in a new environment. By addressing this research problem, the paper aims to provide valuable insights for developers and organizations looking to optimize their frontend code structure.

The authors substantiate their claims by conducting a thorough literature review of existing MFE architectures and evaluating them based on defined qualitative attributes such as performance, modularity, testability, developer experience, simplicity, and scalability. The technical solution consists of two phases. The first phase is about how to decompose the existing SPA into MFEs, where two types of decompositions are compared. The first one is when virtually splitting the independent parts of the SPA using Web Components, meaning that the fragments will only be simulated as independent applications and the SPA and its original structure will remain the same. The second way of decomposing was to split the domains into independent applications which are built, tested, and deployed independently. The paper includes two practical use cases for building a container application and integrating the MFE introduced in the decomposition phase, with the first case integrating with Web Component. It explains how a Web Component can act as a shell to encapsulate an entire

frontend application. In the second case, Module Federation is used to integrate MFEs. This particular use case displays an alternative strategy that manages the integration logic at the configuration level.

The conclusion is that both technical implementations are beneficial in terms of modularity, scalability, and reusability. Both implementations are highly modular because the MFEs are integrated on the client side at run time. Simultaneously, the container application retrieves the MFEs as static JavaScript files, which makes both solutions highly scalable. Overall, this paper provides valuable insights for developers and organizations looking to optimize their frontend code structure using MFE architecture.

## Discussion

The authors employed the same approach as we did in wrapping framework components as Web Components. However, their reasoning for doing so was based solely on Mezzalana's recommendation in his book on MFE implementations [22], which we consider to be a weak justification. Our own results and motivations would have strengthened the authors' claims. Furthermore, although the authors experimented with MFE implementation using Module Federation, they failed to acknowledge the potential issues that can arise from running multiple versions of the same framework. Interestingly, their interviewees also cited communication overhead as a challenge with Web Components when used in conjunction with npm package modules. We addressed these issues by combining Web Components and Module Federation in our thesis, which not only reduced communication overhead but also addressed the issue of multiple framework versions. Although our methodologies for working with MFEs aimed to share them across teams without changing their tech stack, we did not consider how routing would work in a real application. As a result, our implementations were lacking in this regard.

## 5.5 Future work

During this thesis, we discovered ideas that we decided to put on the back burner due to limited time. In this section, we will discuss these ideas in the hope of them becoming future research topics.

In section 3.2, we decided to not investigate the composition methods server-side and edge-side with the reasons being that there simply isn't enough information at the time when we conducted the initial and more in-depth investigation. One of the interviewees at IKEA mentioned that using server-side and edge-side composition would greatly enhance the performance but this is only possible for non-interactive components. When reading new articles, blog posts, and watching YouTube videos, they're starting to introduce state management in server components. It would've been interesting to further investigate if one could combine multiple composition methods to share MFEs. But as it stands today there isn't enough information on how to deal with server-side components. Unfortunately, there is no information about this for edge-side composition.

From the article which was mentioned in the previous section about sharing state management using signals, it would be interesting to conduct further research on combining their proposed methodology of handling state management together with a framework, such

as React. Our way of handling state management is by relying on the window object as a global variable, which excludes mobile app users. A signal is a function based on vanilla JavaScript that does not rely on the window object while at the same time could introduce a better developer experience. More and more frameworks are starting to embrace signals [30] as an alternative way for handling state management between MFEs. There was another way of sharing state that we didn't have enough time to try out and that was server-side state management. It would've been interesting to see how this could affect the developer experience.

Our focus for this thesis was to solely work with React and it would've been interesting to further investigate the outcome of the results with the "Isolation" solution in a multi-framework approach. One aspect we didn't consider in this thesis was the URL routing of MFEs. This together with multiple frameworks could be added as an interesting future research.

We identified that the current pipeline with the team at IKEA we're currently stationed at doesn't support the publication of MFEs to the cloud. This is an area of improvement that will be of interest to IKEA once they decided to work with MFEs more intensely.

What remains to be seen is the implementation of our design proposition. The relations tag should be further investigated to see if it's of value for the people at IKEA or companies looking to implement a similar feature.

# Chapter 6

## Conclusion

---

This thesis investigates the possibility of sharing, finding, and publishing MFEs at IKEA to achieve code reusability, minimize code duplication, eliminate communication overhead as well as eliminate the need for a change in a team's current tech stack.

After conducting extensive research through literature reviews, interviews, and experiments, as a part of **RQ 1.1** we have identified two methods for sharing MFEs that do not require teams at IKEA to alter their current tech stack or worry about which version of the framework is being used. The first possible approach is to use Web Components for wrapping framework components, such as React, and handle state management by either using Custom Events or PubSub. The other option is "Isolation", which exposes the render method as a function and thus eliminates the need to use Web Components. In our opinion, we believe the "Isolation" solution to be the best one for IKEA even though we haven't tested it thoroughly because it doesn't come along with extra workarounds as it does with Web Components.

At the time of starting this thesis, as a part of **RQ 2.1** the people at IKEA didn't have the problem of finding and publishing MFEs because they are in the early phase of working with them. We decided to jump ahead and prepare IKEA for how to best find and publish MFEs after presenting our technical solutions for sharing them. The initial idea was to create our own dev portal proof of concept, but IKEA has one already in place. This dev portal has the potential of being utilized for finding and publishing MFEs. We believe that our design proposition can boost the potential of finding and publishing MFEs even higher.

Simply solving the technical challenges of sharing MFEs is not enough for IKEA to reap the benefits of minimizing code duplication, reducing communication overhead, and increasing code reusability. It is also essential to have a means for finding and publishing MFEs as this would naturally lead to IKEA reaping the benefits even more.





# References

---

- [1] Custom elements everywhere. <https://custom-elements-everywhere.com/#react>. Accessed: 2023-04-03.
- [2] Fastly. <https://www.fastly.com/>. Accessed: 2023-04-20.
- [3] Ikea. <https://about.ikea.com/en>. Accessed: 2023-04-03.
- [4] Installing dev dependencies with npm: Beginners' guide. <https://www.knowledgehut.com/blog/web-development/npm-install-dev-dependencies>. Accessed: 2023-04-20.
- [5] Module federation. <https://webpack.js.org/concepts/module-federation/>. Accessed: 2023-04-03.
- [6] React, the library for web and native user interfaces. <https://react.dev/>. Accessed: 2023-04-03.
- [7] single-spa. <https://single-spa.js.org/>. Accessed: 2023-04-20.
- [8] Web components. [https://developer.mozilla.org/en-US/docs/Web/Web\\_Components](https://developer.mozilla.org/en-US/docs/Web/Web_Components). Accessed: 2023-04-03.
- [9] What is backstage? <https://backstage.io/docs/overview/what-is-backstage/>. [Online; accessed 28-April-2023].
- [10] Wayne A Babich. *Software configuration management: coordination for team productivity*. Addison-Wesley Longman Publishing Co., Inc., 1986.
- [11] \_CODE. 10 best javascript bundler libraries. <https://openbase.com/categories/js/best-javascript-bundler-libraries?vs=%40babel%2Fcore%2Cwebpack%2Cesbuild>. Accessed: 2023-04-20.
- [12] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

- [13] Tim De Smet. Micro frontend architecture for cross framework reusability in practice. *Universiteits Biblio Theek Gent*, 2020.
- [14] Module Federation. different-react-versions-16-18. <https://github.com/module-federation/module-federation-examples/tree/master/different-react-versions-16-18>. Accessed: 2023-04-21.
- [15] Module Federation. different-react-versions-isolated. <https://github.com/module-federation/module-federation-examples/tree/master/different-react-versions-isolated>. Accessed: 2023-04-21.
- [16] Maitray Gadhavi. Understanding a comparison: Webpack vs vitejs. <https://radixweb.com/blog/webpack-vs-vitejs-comparison>. Accessed: 2023-04-21.
- [17] Michael Geers. *Micro frontends in action*. Simon and Schuster, 2020.
- [18] Jack Harrington. create-mf-app. <https://www.npmjs.com/package/create-mf-app/v/1.0.14>. Accessed: 2023-04-21.
- [19] Marcus Hellberg. Exploring react's new web component support. <https://dev.to/marcushellberg/exploring-reacts-newly-added-web-component-support-19i7>. Lasted Updated: 2021-12-13 Accessed: 2023-05-09.
- [20] Rudyard Kipling. I keep six honest serving men. *Just so stories*, 1902.
- [21] Jouni Männistö, Antti-Pekka Tuovinen, and Mikko Raatikainen. Experiences on a frameworkless micro-frontend architecture in a small organization. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pages 61–67. IEEE, 2023.
- [22] Luca Mezzalana. *Building Micro-Frontends*. " O'Reilly Media, Inc.", 2021.
- [23] Sam Newman. *Building microservices*. " O'Reilly Media, Inc.", 2021.
- [24] Yuma Nishizu and Tetsuo Kamina. Implementing micro frontends using signal-based web components. *Journal of Information Processing*, 30:505–512, 2022.
- [25] Nx. Why monorepos. <https://nx.dev/more-concepts/why-monorepos>. Accessed: 2023-04-24.
- [26] OriginJS. vite-plugin-federation. <https://github.com/originjs/vite-plugin-federation>. Accessed: 2023-04-20.
- [27] Yvonne Rogers, Helen Sharp, and Jennifer Preece. *Interaction design: beyond human-computer interaction*. John Wiley & Sons, 2023.
- [28] Thomas Stackpole. Inside ikea's digital transformation. <https://hbr.org/2021/06/inside-ikeas-digital-transformation>, 2021.

- [29] Emilija Stefanovska and Vladimir Trajkovik. Evaluating micro frontend approaches for code reusability. In *ICT Innovations 2022. Reshaping the Future Towards a New Normal: 14th International Conference, ICT Innovations 2022, Skopje, Macedonia, September 29–October 1, 2022, Proceedings*, pages 93–106. Springer, 2023.
- [30] Manfred Steyer. Signals in angular: The future of change detection. <https://www.angulararchitects.io/aktuelles/angular-signals/>. Lasted Updated: 2023-02-03 Accessed: 2023-05-09.
- [31] Manfred Steyer. *Enterprise Angular: Micro Frontends and Moduliths with Angular*. Leanpub, 2022.
- [32] Vite. Build options. <https://vitejs.dev/config/build-options.html>. Accessed: 2023-04-21.
- [33] Vite. Why vite. <https://vitejs.dev/guide/why.html#why-not-bundle-with-esbuild>. Accessed: 2023-04-21.
- [34] Webpack. Experiments. <https://webpack.js.org/configuration/experiments/#experiments>. Accessed: 2023-04-21.
- [35] Webpack. Modules. <https://webpack.js.org/concepts/modules/>. Accessed: 2023-04-21.
- [36] Wikipedia contributors. Confluence (software) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Confluence\\_\(software\)&oldid=1149971333](https://en.wikipedia.org/w/index.php?title=Confluence_(software)&oldid=1149971333), 2023. [Online; accessed 27-April-2023].



# Appendices



# Appendix A

## Initial investigation

---

The most appropriate approach to interviewing depends on the purpose of the interview, and the questions to be addressed.

### Challenges

For this question, we aim to get a little understanding of why creating MFEs is hard as well as get some insight into what architecture they use to compose an application. **Don't forget to use "What, Why, When, How, Where, and Who".**

- What are the challenges of creating MFEs for your team?
  - Follow-up questions that could be regarded to the MFE architecture if the interviewee becomes technical.
  - **Six honest:** What are the goals, why is x architecture used, when do you use an MFE and when not i.e develop your own, who is developing and testing and how do you integrate them?

### Import/export

Here we would like to know what the problems are for sharing and accessing components, they could be technical, or communicational. **Don't forget to use "What, Why, When, How, Where, and Who".**

- Why complex is it to share/import a component with/from another team?
  - Why is it complex?
  - See where this question leads, improvise the questions but also look at the original interview guide for follow-up questions.
- How do you currently share MFEs internally?
  - What problems and difficulties do you face and why are they an obstacle?
- Have you tried sharing MFEs externally as you do internally?
  - What problems and difficulties do you face and why are they an obstacle?
- Are there any MFEs you are aware of that are using a different framework?
  - Is it a different version or a different framework entirely?
- How much work is needed to support the integration from another team?
  - Anarchy (you follow under a leader or framework) or Autonomy (Having other teams accustom to you).
  - Are there issues with requirements from different teams?
- Is there a reason why you choose not to update to the latest version of react?
- **Six honest:** What are the benefits of sharing, why is it important and how will it benefit IKEA, when is it appropriate to share and when is it not, where will it be hosted and maintained, who would be responsible for the compatibility and integration of the MFE, and how will the MFE sharing be coordinated.

### Wishful thinking

These are the questions that they would like to have the answer to in a year or something that they can not do now but will make your life easier.

- What is your ideal way of importing components?
- What is your ideal way of exporting components?
- If MFEs are the solution, then what is the problem?



# Appendix B

## Elicit design requirements

---

Purpose of this is to get the information flow from developers on how they would access an API. Aimed at developers

- How do you currently access APIs? If you could take us step by step.
  - Is there a platform/dashboard where you can find existing APIs?
    - What information does this platform/dashboard contain? (IKEA's dev portal vs confluence page)
  - When do you want to access an inhouse/outside API?
  - Where do you draw the line of using external APIs?
- What are the challenges when accessing APIs?
  - How do you overcome these challenges?

Now that we have a general understanding of how information flow is we will now try and ask how this can be applied to when accessing MFEs.

- How would you go about finding the MFE? By its creator or the API in use?
- What are your ideal ways of accessing MFEs?

# Appendix C

## Requirement specification

---

This is the list of our complete ideal requirements specification.

- The one-stop shop should have a clear and organized structure that makes it easy for users to navigate and find the information they need.
- The one-stop shop should provide detailed documentation for each MFE, including ownership, API usage, and requirements for use.
- The one-stop shop should provide connections with other teams to assist users in finding MFEs that match their desired API or feature. If there are no connections between the published MFE and the search term, then the user can be confident that there are no MFEs available that are related to that team's API or service.
- The one-stop shop should be able to let users upload their own MFEs along with their documentation.
- The one-stop shop should let the user update the MFE.
- The one-stop shop should have a search function that allows users to quickly find specific MFEs and their relations to other teams.
- The one-stop shop should be accessible to all teams within the company, regardless of location or department. Sharing MFEs should be done globally at IKEA and not constrained locally.
- The one-stop shop should have the contact information of the MFE owner.
- The one-stop shop should have a search function that allows users to quickly find specific MFEs based on keywords or other criteria.
- The one-stop shop should have a secure login system that limits access to authorized users and protects sensitive information.

- The one-stop shop should be integrated with the company's existing dev portal with GitHub to ensure updates are automatic.
- The one-stop shop should have a feedback mechanism that allows users to suggest improvements or report issues with the one-stop shop.

# Appendix D

## Proposed designs

---

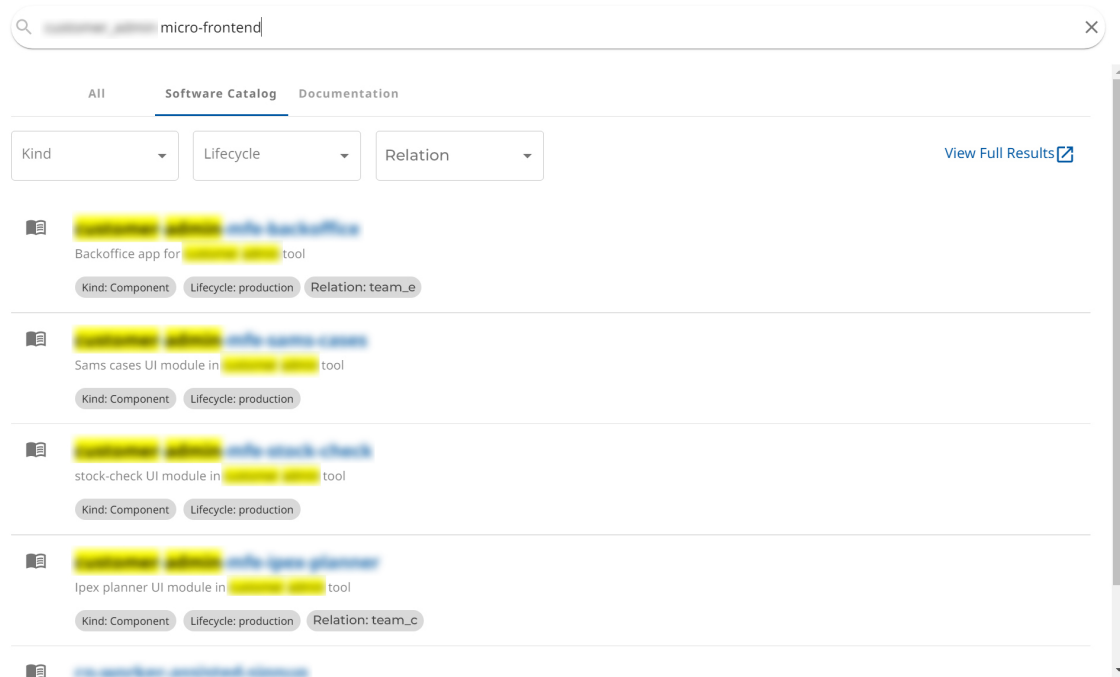


Figure D.1: Blurred team’s published MFEs

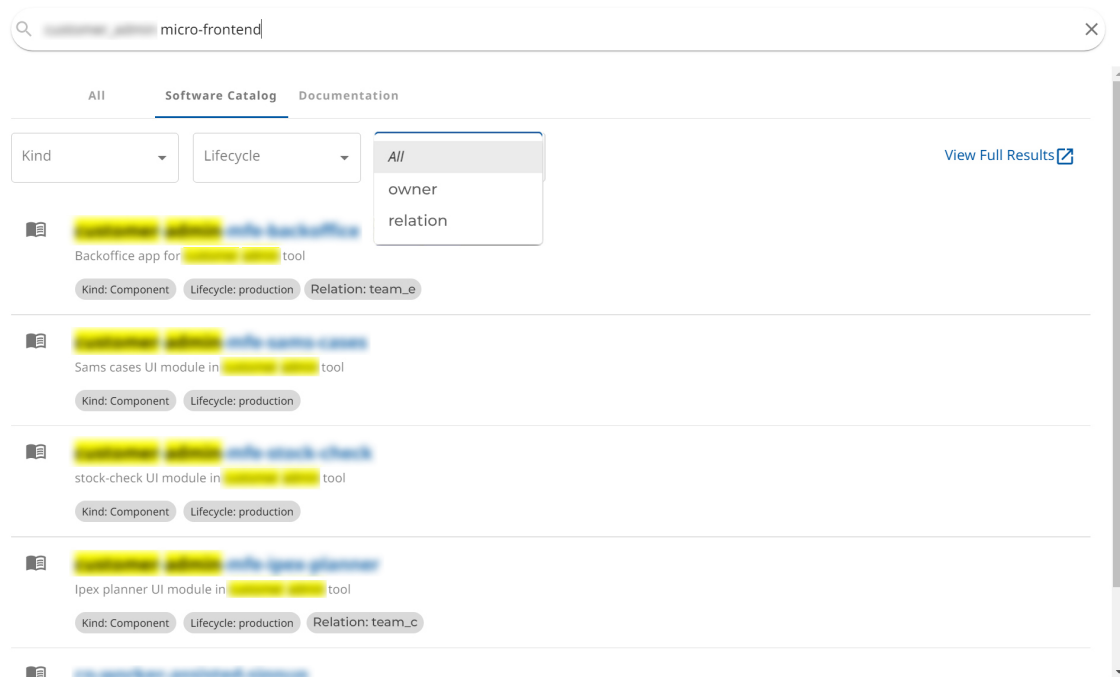


Figure D.2: Search field filter option opened

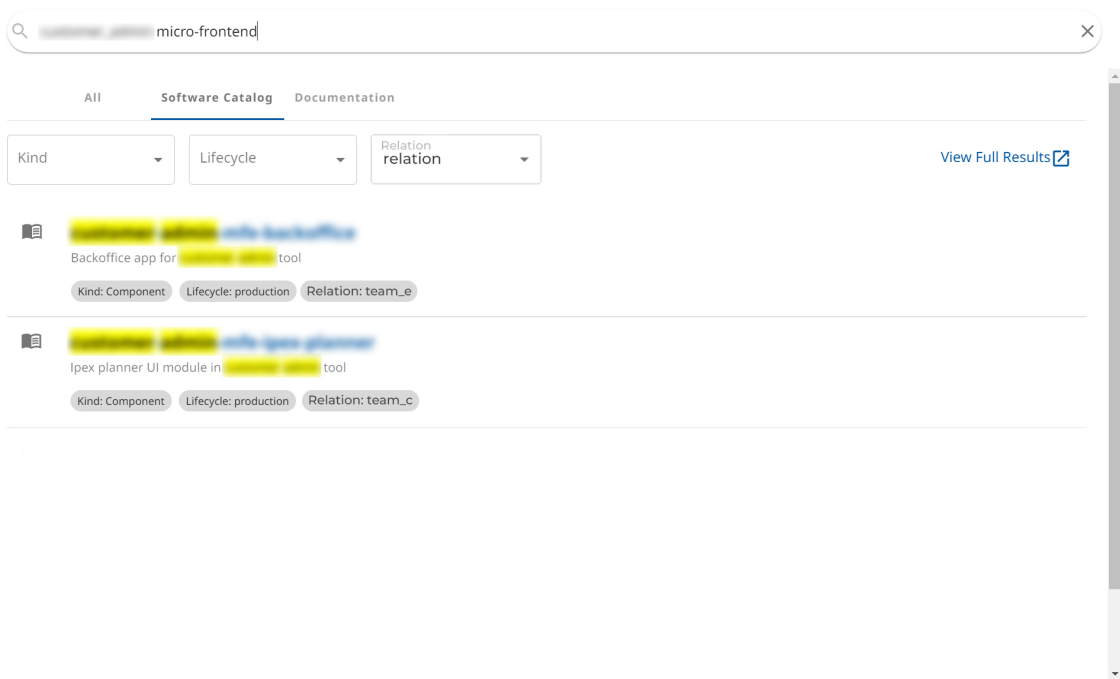


Figure D.3: Relation option filter is picked

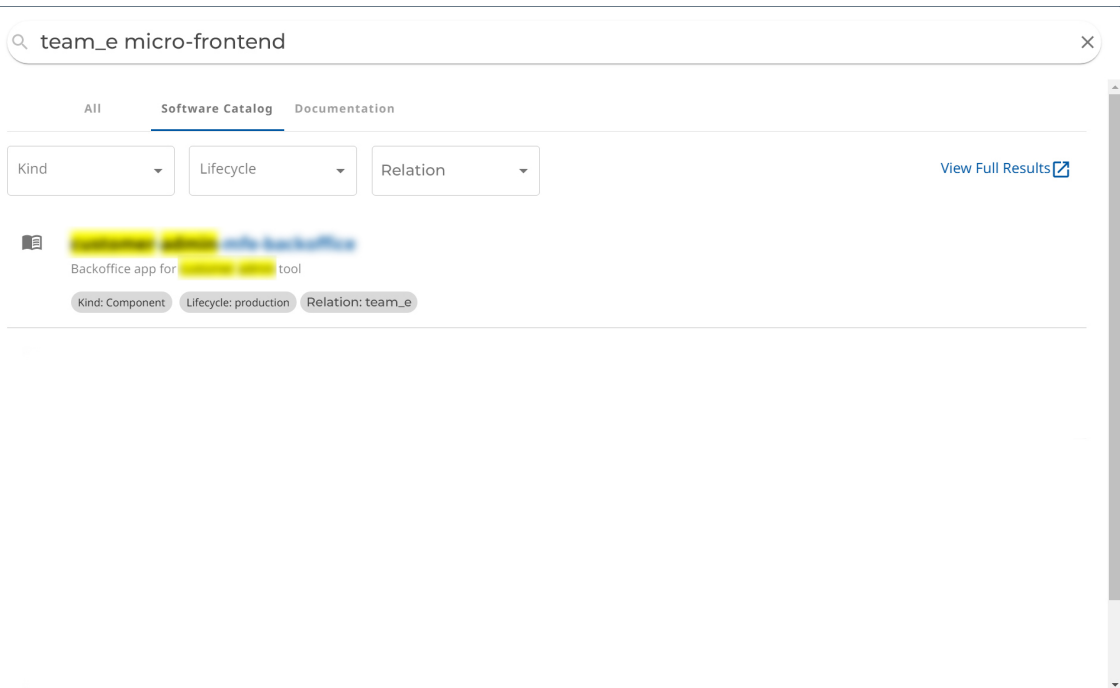


Figure D.4: team\_e API is used to create blurred team MFE

All (5)									
NAME	SYSTEM	OWNER	RELATION	TYPE	LIFECYCLE	DESCRIPTION	TAGS	ACTIONS	
[blurred]	[blurred]	[blurred]	team_a, t...	micro-front	development	Customer...	micro-front	[edit] [star]	
[blurred]	[blurred]	[blurred]		micro-front	production	Backoffice app...	micro-front	[edit] [star]	
[blurred]	[blurred]	[blurred]	team_c	micro-front	production	Ipex planner UI...	micro-front	[edit] [star]	
[blurred]	[blurred]	[blurred]		micro-front	production	Sams cases UL...	micro-front	[edit] [star]	
[blurred]	[blurred]	[blurred]		micro-front	production	stock-check UL...	micro-front	[edit] [star]	

Figure D.5: Software catalog of MFEs

All (5)									
NAME	SYSTEM	OWNER	RELATION	TYPE	LIFECYCLE	DESCRIPTION	TAGS	ACTIONS	
[blurred]	[blurred]	[blurred]	team_c	micro-front	production	Ipex planner UI...	micro-front	[edit] [star]	

Figure D.6: Software catalog filtered to MFEs created with the help of team\_c



**STUDENTER** Andrej Simeunovic, Uros Tripunovic**HANDLEDARE** Lars Bendix (LTH)**EXAMINATOR** Per Andersson (LTH)

# Hantering av Micro Frontends över flera tekniska plattformar - Delning, Sökning och Publicering

---

**POPULÄRVETENSKAPLIG SAMMANFATTNING Andrej Simeunovic, Uros Tripunovic**

---

Detta examensarbete undersöker möjligheten att dela, hitta och publicera Micro Frontends (MFEs) på IKEA för att uppnå återanvändning av kod, minimera kodduplicering, eliminera kommunikations överbelastning samt eliminera behovet av att ändra ett teams nuvarande IT stack.

IKEA är ett stort företag som består av flera oberoende team. Dessa olika team arbetar under olika kontexter där implementering av liknande funktioner kan förekomma. Detta leder till kodduplicering, där tid och pengar kan förbrukas på något annat.

I nuvarande läge finns det inget standardsätt att dela MFE:er utan att tvinga ett team att anpassa sig till ett annat teams IT stack och därmed bryter den autonoma utvecklingen. Detta introducerar friktion mellan team då de inte är villiga att ändra sin nuvarande IT stack och nuvarande sätt att arbeta. Ändringar i IT stacken går emot principen om autonom utveckling, där införandet av MFE:er ska användas på ett sätt där man bara importerar MFE:en utan att lägga till extra krav och ändra på sitt nuvarande arbetsflöde. Förutom att dela MFE:er finns det inget effektivt sätt att hitta och publicera MFE:er. Det finns en enorm kommunikations överbelastning då man delar MFE:er, om ett team inte tar på sig att dela sina MFE:er då är det nästintill omöjligt att veta om dess existens. Problemen som beskrivs skapade följande forskningsfrågor:

- **RQ1** - Vad är utmaningarna med att dela

MFE:er i IKEAs kontext?

- **RQ2** - Hur skulle ett team veta att en MFE med ett önskat API redan existerar?

Efter en följd av experiment inspirerade från våra litteraturstudier och intervjuer, upptäcktes det två metoder för hur teamen på IKEA kan dela MFE:er utan att behöva ändra sin nuvarande IT stack. Dessa metoder involverar att antingen slå in ramverkskomponenter som Web Components eller exponera ett ramverks renderingsmetod i en funktion och injicera den i en frontend applikation. När det gäller att hitta och publicera MFE:er, så har IKEA redan en existerande plattform. Utvärdering av IKEAs plattform jämfördes med vår kravspecifikation, som härleddes från intervjuer på IKEA. För att göra det enklare att hitta och publicera MFE:er, föreslogs en förbättring på deras sök motor som visar de team involverade i att skapa en MFE. För att IKEA skall fullständigt åstadkomma minimering av kod duplicering, kommunikations belastning och öka återanvändning av kod med MFE:er, är det viktigt att kombinera den tekniska aspekten samtidigt som dessa kan hittas och publiceras.