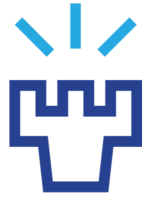

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Jonada Ferracaku

THE STATE OF MICRO FRONTENDS: CHALLENGES OF APPLYING AND ADOPTING CLIENT-SIDE MICROSERVICES

Master's Thesis
Degree Programme in Computer Science and Engineering
June 2021

Ferracaku J. (2021) The State of Micro Frontends: Challenges of Applying and Adopting Client-Side Microservices. University of Oulu, Degree Programme in Computer Science and Engineering, 57 p.

ABSTRACT

Software development had been around for centuries and the need to provide the right techniques for delivering it, it's indispensable as the market is growing fast. Many companies have adapted microservices as a transition from monolithic backend applications. While the microservice approach solved various issues present in backend applications, frontend architecture is still composed as a monolithic application. Issues arise as the frontend application increases in scale and become difficult to maintain.

This study aims to give an introduction to micro frontend architecture as a technique of solving different issues present in frontend development. It will analyze how the micro frontend approach is perceived by developers with different levels of experience compared to a more traditional approach of developing monolithic frontend application, single page application.

To provide answers for the question, we compared the performance of 6 developers with different levels of experience that architectured and implemented a simple frontend application using single page application and micro frontend. The results showed that developers' experience mattered while developing single page application as the approach was perceived as easy. On the micro frontend counterpart, having previous experience with setting up micro frontend applications had a major impact on the perceived difficulty. Although the perceived difficulty of micro frontend remains higher compared to single page application, developers shared the same consent for using the micro frontend approach for large-scale applications.

Keywords: Software Development, Microservices, Monolithic application, Single Page Application, Micro Frontend

TABLE OF CONTENTS

ABSTRACT
TABLE OF CONTENTS
FOREWORD
LIST OF ABBREVIATIONS AND SYMBOLS
1. INTRODUCTION ... 7
2. RELATED WORK ... 8
  2.1. Transitions to Modern Development Trends ... 8
    2.1.1. Monolith Architecture ... 9
    2.1.2. Microservice Architecture ... 9
  2.2. Beyond Microservices ... 10
    2.2.1. Serverless Architecture ... 10
    2.2.2. BFF - Backend for Frontends ... 11
  2.3. History of Frontend Frameworks ... 12
    2.3.1. Chronology of Base Frontend Technologies ... 12
    2.3.2. Transition to Single Page Application ... 13
  2.4. Single Page Applications ... 14
    2.4.1. Modern JavaScript Frameworks ... 14
    2.4.2. Frontend Architectural Patterns ... 17
    2.4.3. Challenges of Single Page Applications ... 19
  2.5. The Emergence of Micro Frontends ... 19
    2.5.1. Use of Microservices in Frontend ... 19
    2.5.2. Frontend Design Architecture Guidelines for Micro Frontend ... 20
  2.6. Micro Frontend Frameworks and Architecture ... 21
  2.7. Micro Frontends in Commercial Services ... 26
    2.7.1. Micro Frontends at Zalando ... 27
    2.7.2. Micro Frontends at Upwork ... 28
    2.7.3. Micro Frontends at HelloFresh ... 29
3. EXPERIMENT DESIGN AND SETUP ... 31
  3.1. Case Study: Micro Frontend Evaluation ... 31
  3.2. Subjects ... 34
  3.3. Methodology ... 35
  3.4. Procedure ... 35
  3.5. Survey Data Collection ... 36
4. RESULTS AND DISCUSSION ... 38
  4.1. Experiment Contributors Overview ... 38
  4.2. Implementation Time ... 38
    4.2.1. SPA ... 39
    4.2.2. Micro Frontend ... 41
    4.2.3. Comparison ... 42
  4.3. Ease of Development ... 44
    4.3.1. How Difficult Are SPA Vs Micro Frontends ... 45
  4.4. The Tool for the Job ... 46
  4.5. Implementation Details ... 47

5. CONCLUSIONS 50
5.1. Summary of Results 50
5.2. Limitations 51
5.3. Future Work 51
6. REFERENCES 53

FOREWORD

The idea of this thesis was born from my discussion with my college Marko Heikkilä about the future of frontend development. The basis of the discussion was if micro frontends were the future and are those ready to be a standard for projects.

I want to thank my supervisors. Aku Visuri for following me in all the steps of writing the thesis and giving constructive criticism continuously. Denzil Ferreira for giving me an initial structure and idea of how to conduct and shape the thesis. Finally I want to thank my college Marko Heikkilä for supporting me throughout the thesis with ideas and checking in with my progress.

Oulu, June 9th, 2021

Jonada Ferracaku

LIST OF ABBREVIATIONS AND SYMBOLS

SPA	Single Page Application
UI	User Interface
UX	User Experience
FAAS	Function As A Service
BFF	Backend For Frontend
AWS	Amazon Web Services
API	Application Programming Interface
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
AJAX	Asynchronous JavaScript and XML
MVC	Model View Controller
MVVM	Model-View-ViewModel
DOM	Document Object Model
REST	Representational state transfer
HTTP	Hypertext Transfer Protocol
CLI	Command Line Interface
SEO	Search Engine Optimization
MVP	Minimum Viable Product
JSON	JavaScript Object Notation

1. INTRODUCTION

Nowadays, in the world of web development, the technique of creating a micro frontend is spreading rapidly. It is an innovative concept in the world of frontend development since the introduction of Single Page Application (SPA). Micro frontend is an architectural style for frontend applications that aims to change the way frontend projects are developed. Micro frontend helps to solve challenges in the frontend architecture by splitting it into smaller pieces. The concept is similar to microservice architecture, which is already present throughout backend development.

Microservice has emerged from the world of domain-driven design, continuous delivery, and scalable systems. While decomposing the backend into microservices is a well-known approach [1, 2] for achieving flexibility in development and operation, most frontend solutions are still running as a monolithic application [2].

The objective of the thesis is to provide an introduction and evaluation of micro frontend development approach and a comparison between two development approaches used to develop a frontend application: single page application and micro frontend in terms of how easy it is to get started depending on the developer experience, extensibility, scalability, application performance, UI/UX experience, team productivity, and technology independence.

This aims to investigate the following research question:

What is the difference in perception of micro frontend architecture for experienced and inexperienced frontend developers?

The contribution of this thesis is to show that the micro frontend approach might be the right one and then it constructs a path for the experiment and the discussion which focuses if this perception is shared by people who work in the industry and how the difference in development experience changes the perception.

This paper is structured in the following manner: Chapter 2 contains background and related work information about the history of backend and frontend development and how they share the same ideology through time.

Chapter 3 provides information about the proposed experiment design and setup and how it was chronologically planned and implemented.

Chapter 4 analyzes the results of the proposed experiment and discusses them in relation to the existing research.

The last chapter summarizes the findings from the result and discussion chapter, its limitations, and proposes possible improvements.

2. RELATED WORK

The related work section aims to show the history of both frontend and backend development. By focusing on the history of each it underlines how they share similar ideologies through time.

Section 2.1 and 2.2 describe the transition from monolithic backend development to the more modern approach of microservice architecture. By showing the advantages that it has brought to backend development and the issues that it aimed to solve in the first place it gives a background as to how frontend development is suffering from similar issues.

The frontend development trends are presented in section 2.3 to show that a part of the issues that happened in the backend development through history are present as well in the frontend. To underline this it will show the different nature of frontend development and why it has started to suffer from monolithic architecture.

To show the current state of frontend development, section 2.4 gives an introduction and description of the current most popular frameworks on frontend development and architectural patterns used in applications.

Section 2.5 introduces in-depth how micro frontend emerged and what are possible solutions to alleviate the problems present in frontend development.

Section 2.6 describes different frameworks and architectures that are used today to implement micro frontend.

Section 2.7 also walks through also how different companies use micro frontend solutions and how they fit with the current development. This paper shows why micro frontends might be the answer to microservices in frontend in the latter part.

2.1. Transitions to Modern Development Trends

This section gives a history of the development trends in software development. Through this, it shows how the emergence of self-contained services development in the backend. This relates to the same needs appearing eventually on the frontend development.

One way of conceptualizing software throughout the years has been by splitting it into frontend, backend, and storage layers. The frontend is considered a user interface through which they interact with the application. The backend consists of the logic done beyond the user interface and usually hidden away from the user and more to be used by machines. Storage is where the data that is associated with the application lays.

Throughout history, the way these pieces have been developed has varied. Initially, monolithic architecture was a main way of development, with all these layers developed in one single application and ran in a single machine. With the increasing popularity of personal computers and the internet, the backend and the database development started to be separated into a singular server layer and the frontend moved in the frontend layer. The backend evolved into the microservices architecture which is today one of the most popular architectures and new trends such as serverless or FAAS (Function As A Service) have appeared in recent years with the explosion of cloud

computing. An overview of the transition from monolith application to microservice is shown in Figure 1 adapted from [3].

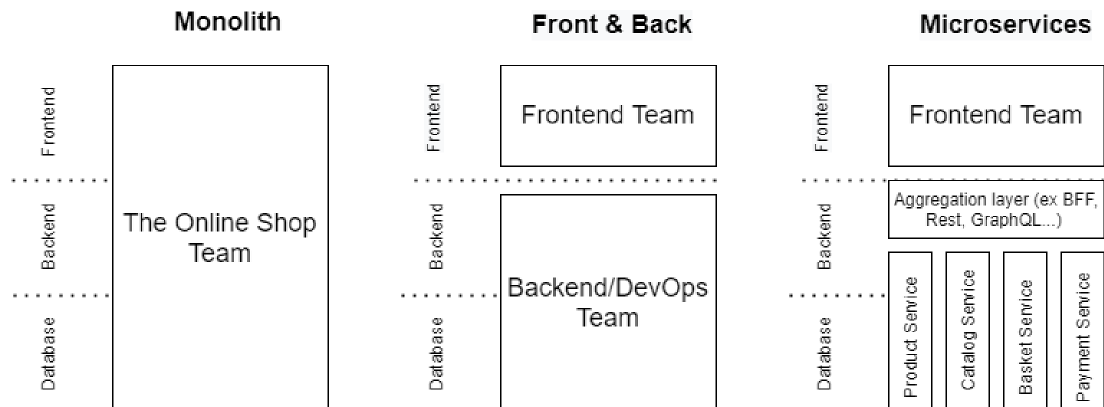

Figure 1. Transition of Backend monolith to microservice while frontend stayed monolith

2.1.1. Monolith Architecture

In the early days, the existence of personal computing was still new and the applications were developed as a single bundle to run into a single machine. All the layers of development were fitted into a single application known as a monolith. These applications were placed into a single repository and the code was organized in such a way that folder naming would imply the layer to which the code belonged. Anything changed in the software would require the whole application compilation to change and the code-breaking risk to unrelated issues was high due to the coupled architecture[4].

With the introduction of personal computing and the popularity increase of the internet, it became evident that splitting your code to fit in the frontend and backend layer was a more fitting idea. This allowed for the separate development of the backend and frontend. It also made it easier for people with different skill sets to work on each layer. This yielded benefits such as better deployment management, more focused work, and better decoupling.

On the other hand, as the approach was to have thin clients and thick server layers, hence a lot of the complexity was moved into the backend layer. There was a need for many applications to split the backend layer into smaller self-contained pieces, usually with their own database. These pieces would be responsible for a single responsibility and usually tied to business functionality. These self-contained pieces are what we call microservices.

2.1.2. Microservice Architecture

Microservices brought a clearer scope and responsibility for each service. In comparison to monolithic systems the deployments are even smaller, developers can be split into smaller teams and can use the technology that they think fits the

best and allowed the services to be decoupled. Frontend applications use different microservices based on their need and different microservices communicate with each other. The microservices allow for scaling with short deployment times and independence from other services.

Transitioning Into Microservice

The monolithic architecture was used by big companies like Netflix and Amazon. Monolithic applications can be deployed, scaled, and maintained independently. Over years, the monolithic application can become large, complex, and difficult to maintain [4, 5].

On the other hand, there is an increased tendency by many organizations to move existing applications to microservice and move away from monolithic architecture. This is also pushed by the cloud architecture and the way it utilizes the microservices to do automatic scaling, high availability and redundancy, easier infrastructure management, and a way to do continuous integration and development. This paradigm has changed the way we design, build, deployment and maintain business applications.[4, 6]

Microservice architecture alleviated a lot of the issues that were present in the backend development prior to it. The same issues are becoming more relevant in frontend as we tend to build better and richer user interfaces. This has brought a need for a similar approach in frontend development. The approach is known as *micro frontends*.

2.2. Beyond Microservices

Cloud computing has become the main part of how companies develop their products. This is reflected also in the changes that have been made to the development of the backend. There are new trends such as serverless architecture where the backend is organized in functions that run usually on a cloud service. Furthermore, there are also patterns like backend for frontend (BFF) which different companies use for consistent UI development by leveraging the backend development. Backend for frontend pattern instructs that a single backend service should be tied to small frontend counterparts and handle all the aggregations there.

2.2.1. Serverless Architecture

Serverless architecture is used for building applications that leverage cloud computing to split the backend into small pieces. These pieces are usually functions that run on a defined endpoint or organized into a single one. These functions run in containers and are ephemeral, which improves the scalability. They are usually used through services known as API gateways, which serve as a routing service. AWS (Amazon Web Services) Lambda is an example of a serverless architecture implementation. It uses FAAS model of cloud computing to achieve the architecture. Linux containers are key enablers of this architecture due to their lightweight images, and fast startup times.

The name is appropriated by the concept of not having a full-blown server and is kind of a black box for the developers. The user deploys the functions without needing to care about any specific information about how the server is run or for virtual machines. [7, 8, 9, 10]

The architecture of FAAS is shown in Figure 2 adapted from [9].

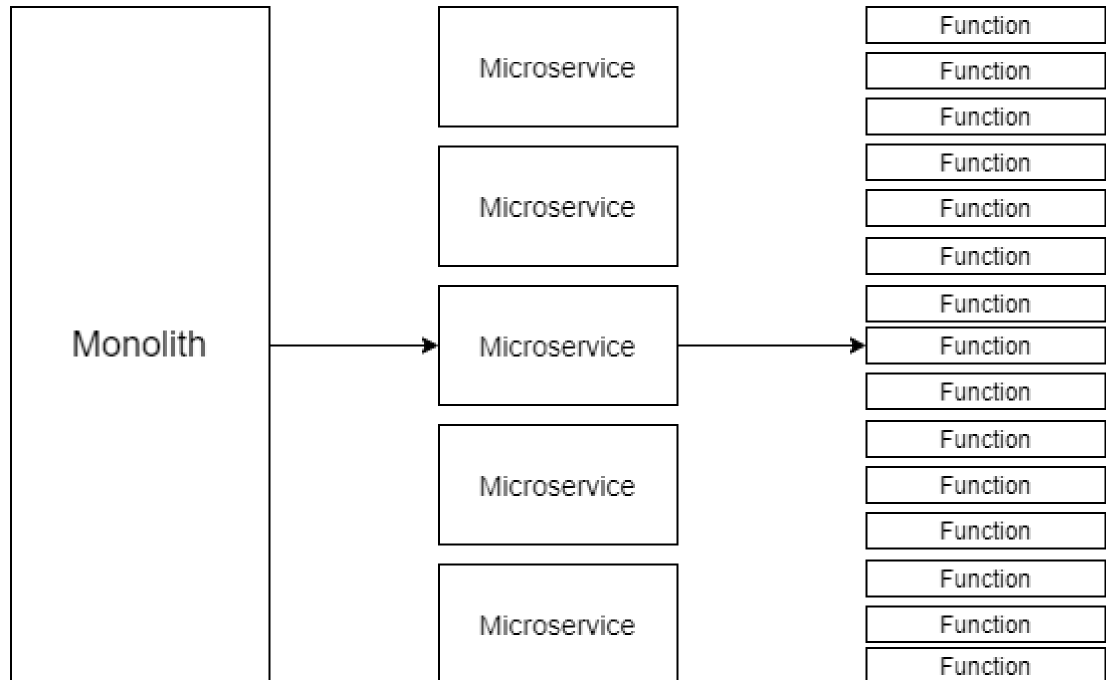

Figure 2. Function as a service serverless architecture

2.2.2. BFF - Backend for Frontends

BFF pattern goes one step beyond the normal microservice architecture. It aggregates multiple microservices to form a single backend. The backend should correspond to a single frontend. They can also be seen as an API (Application Programming Interface) Gateway as they are responsible for multiple requests from different services. This alleviates the need to aggregate the calls in frontend development and offers a unique interface. The backend for frontend translates the requests from the frontend into other requests. The pattern can also be used to develop multiple pieces for complex domains and relate each domain to a single backend. This way it would drive the frontend architecture to fit as well into this concept. This pattern may not be required in all possible scenarios and it may not be worth it the complexity of configurations for applications that are quite efficient with microservices. But it allows a scalable pattern in which the complexity for frontend facing solutions.[11, 12, 13]

BFF design pattern is shown in Figure 3 adapted from [11].

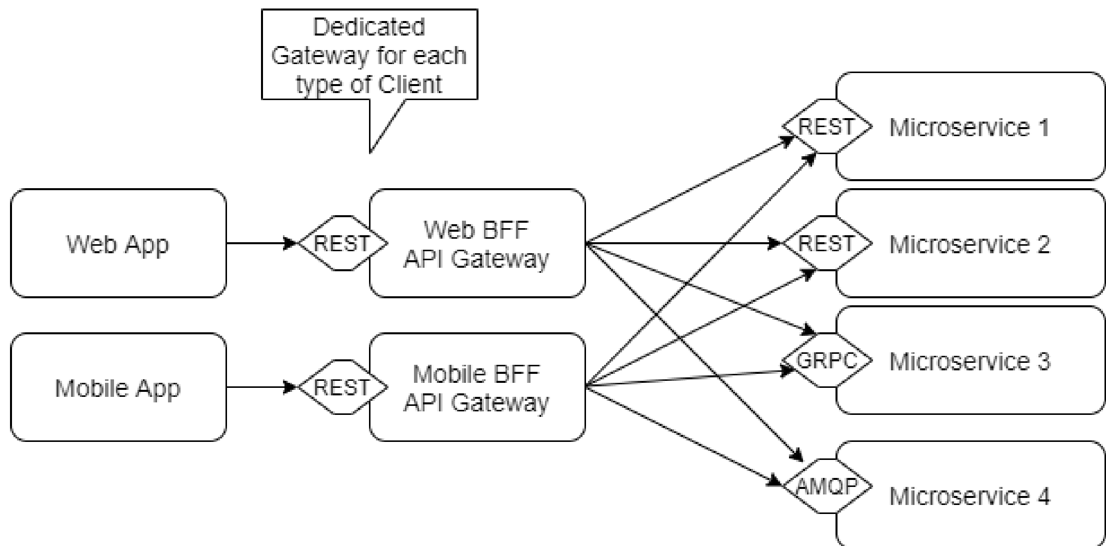

Figure 3. Backend For Frontend design pattern

2.3. History of Frontend Frameworks

The world of frontend development has evolved rapidly in the past few years [14]. Javascript has advanced the most among existing programming languages. The era of writing simple logic on the website by using unstructured code and plugins has been replaced by building a completely functional Single Page Application. Nowadays, a novice developer can build a fully-fledged application in less time compared to 20 years ago.

This section will walk through how the modern frontend frameworks developed over the years to provide the level of expertise present today in the world of frontend development.

2.3.1. Chronology of Base Frontend Technologies

Frontend applications consist of a combination between HTML (HyperText Markup Language), CSS (Cascading Style Sheets), and JavaScript. Their history dated 20 years back. [15]

1991 - HTML specifications were made public by Tim Berners-Lee. It consisted of 18 tags, and it supported only text.
1994 - CSS become present in October by Håkon Wium Lie.
1995 - Mocha, a new browser scripting language was created by Brendan Eich in only 10 days. Later it got renamed to **LiveScript**. Some months later it got renamed to **JavaScript**, the naming that still has today.
1996 - HTML 4.0 become public. Its specification included also information about supporting CSS. Internet Explorer 3 was the first browser to support CSS.

2.3.2. Transition to Single Page Application

Web applications started as multi-page applications until the first single page application framework entered the market. Multi-page applications work by making multiple requests between the client and server and each request comes from the server causing the page to refresh every time. This was an inefficient process that placed a lot of additional loads on the server making the bandwidth a factor to limit the performance. [16]

At this time, developers were in need of having a technology that would update the view without the actual refresh of the page. On the other side, JavaScript had browser incompatibilities which made the development much more complicated compared to nowadays. Developers had to make sure that the JavaScript code runs as it was expected and supported in different browsers. Developers had the be aware also about strange behaviors especially if their background was from strongly-typed languages.

1996 - AJAX (Asynchronous JavaScript and XML) technology was released. It was a small reflection that showed how single page applications would look in the future. It represents a group of technologies to build a web application that communicates with the server in the background without getting in the way of the state of the page. AJAX had several drawbacks. If the user's browser didn't support JavaScript or XMLHttpRequest, or this functionality was disabled, the user didn't have the chance to get an advantage to leverage the benefits of AJAX.

2006 - JQuery was released as one of the earliest libraries that made working with JavaScript easier. Although it was not a single page application, it brought innovation into frontend development [17]. One of the advantages of the library was that developers didn't have to worry that much about browser compatibility. It included several functions to make a website interactive. While JQuery was very popular among developers and helped to solve browser compatibles, it lacked functionalities to handle data sharing between HTML views.

2009 - Backbone.js was released offering a lightweight client-side framework which can build single page application easily. But it had several drawbacks, to mention the repetitive and huge amount of code.

2010 - AngularJS came into market. It was the true version of single page application. It became very popular since the launch as a JavaScript MVC (Model View Controller) and MVVM (Model View-View Model) framework. The framework gave solution to common issues that JQuery lacked to provide like: *two-way data binding*, *dependency injection*, *routing package* and more. The release of AngularJs brought the beginning of a new era of single page application. But although it solved a lot of issues, developers encountered framework complexity as the project increased in size. At a certain point, the Angular Js team unable to find their way around the issues abandoned the framework completely.

2013 - React was introduced at a JavaScript conference in the US as a game-changer library by Jordan Walke (software engineer at Facebook). It was the first library that encapsulated *Virual DOM (Document Object Model)*, *one-way data flow* and *Flux pattern*. We will talk later about Flux pattern in paragraph 2.4.2

2014 - Vue.js was created. It is a lightweight framework that picked a middle ground between the flexibility of React and the assertiveness of Angular.

The next section will talk more in detail about Single Page Applications.

2.4. Single Page Applications

Single page applications are used to build rich user interfaces in web development. They are web applications that load the main resources once when a web page is requested and then asynchronously load pieces of information upon request. Single page application was introduced as an alternative to multiple page application which are applications which create the HTML on the backend server and then send it through HTTP to browsers. This alleviated the issues of backend and frontend coupling. Single Page Applications are usually combined with REST (REpresentational State Transfer) or other backend API technologies that allow them to load data synchronously in lightweight formats. This method is used widely in frontend development in recent years. While the backend development has trended into the splitting of the services into multiple independent pieces the frontend trended into developing monolithic Single Page Applications. Single Page Applications inherit the issues that monolithic applications have. As the business requirements increase so do the functionalities of the Single Page Application. This leads to difficulties in maintainability. [18]

Single page applications introduce different issues that are common to monolithic applications. The issues that hinder the frontend development when choosing Single Page Applications are listed below.

- **Maintainability**. Single page applications have a monolithic approach to development. This leads to difficulty in splitting teams and maintainability issues increase with team size.
- **Technology independence**. Teams are bound to a singular technology choice. The decision needs to be done early and then it locks in the team.
- **Deployments**. This is a traditional issue with monolithic applications. Since there is a singular application it will need to be deployed as a whole whenever a change is made. This issue relates to deployment times and versioning.

2.4.1. Modern JavaScript Frameworks

In this section, we will talk more in detail about single page application frameworks. It will highlight the core features of their architecture and how they are used to develop frontend applications. Based on their popularity [19], nowadays there are 4 main frameworks for Single Page Application: Angular[20], React [21], Vue [22] and Ember

JS[23]. Section 4 will underline which of the frameworks mentioned here was most preferred or popular among developers that will participate in the experiment.

Angular

Angular allows us to build applications across all platforms. It is an open-source platform that uses TypeScript supported by Google. Angular implements core and optional functionality as a set of TypeScript libraries that we can import. The overall architecture of an Angular application is composed of NgModules, Components, and Services. Angular is a cross-platform framework which means that is being used to build progressive web applications, native and desktop applications. To ease the work of developers, Angular provides Angular CLI, a command-line tool used to generate a new project, components, modules, directives, etc. Angular provides tools to test the application and its documentation provides a clear path for unit and integration testing.

React

React is a popular JavaScript framework created by Facebook. It has gained popularity shortly after it has been presented to the public. While other frameworks focus on pushing MVC and MVVM concepts, React focuses on isolating view rendering from the model representation. React introduced the new architecture to the JavaScript frontend ecosystem which is Flux. React is a declarative framework making the code more readable and easier to debug. React uses virtual DOM by creating a representation of the user interface and refreshes the view only if any change is applied. It is a component-based framework and each component is written using a markup syntax called JSX that resembles HTML.

Vue

Vue is a simple, lightweight, and efficient JavaScript framework. It is called a progressive framework which means that it adapts to the needs of the developer. It is often considered the new JQuery, because you can easily start using it by incorporating a script tag into the project. Vue also makes use of virtual DOM like React and data binding like Angular. It provides an additional feature, CSS transitions, and animations. This can be visible in the application when a user adds new HTML elements, updates or removes elements from the DOM. Vue is a template-based framework that binds the DOM with Vue.js instance data. The templates are rendered then as virtual DOM functions. While compared to other frameworks, Vue provides a simpler framework in terms of design and API.

Ember.js

Ember Js is another client-side JavaScript framework used to create single page applications introduced to the market in 2015. It has a small learning curve, it is easy to use but lately, it is losing its popularity. It is not backed up by a large developer community compared to React and Angular. Ember supports two-way data binding which is beneficial especially when creating complex user interfaces. Ember is the main framework for popular websites like Linkedin, Netflix, Apple, etc. Like other

frameworks, Ember supports the creation of reusable components, provides instance initializers, routes to manage URL within the application, and templates to update the model of the component if any change is applied. Ember also provides an inspector tool to debug Ember applications

According to Stack Overflow Trends, [24], we can see how these 4 technologies have trended since 2008 based on the use of their tags. In Figure 4 we can see an obvious increase in popularity for React and a much lower trend for Ember.js.

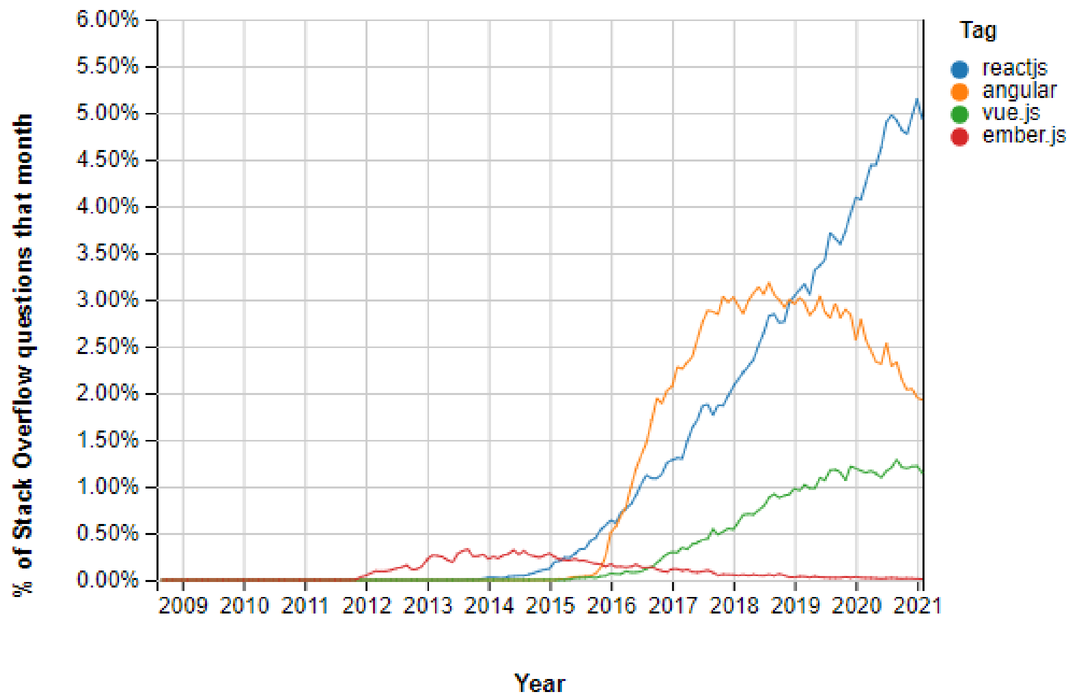

Figure 4. Popularity of four most used JavaScript frameworks

While comparing the popularity of 4 frontend frameworks, Table 1 is from [25] and contains core functionalities used for development and their incorporation within the framework.

	Framework			
Core Feature	Angular	React	Ember	Vue
View/Templating	X	X	X	X
Router	X	X	X	X
Form processing	X		X	
Form validation	X			
Http communication	X		X	

Table 1. Core Features of frontend frameworks

2.4.2. Frontend Architectural Patterns

As the frontend application is meant to be changed frequently, there are many ways we can manage it.

MVVM - Model View ViewModel

MVVM is a commonly used pattern when organizing a frontend application. In the context of Single Page Application, the Model is the Application State and the View is the HTML template. The model is referred to as a domain object. It represents the data that the application is working with. While it contains data, it cannot alter or manipulate them. The view is the user interface that the user interacts with. It is the presentation of the data. On the other side, ViewModel is an object that exposes, properties and methods, and important information that is needed to maintain the state of the view, manage the model according to the view's actions, and trigger events in the view. MVVM gets rid of boilerplate code that usually needs to keep both Model and View in sync. This provides a much higher level of abstraction. [26, 27]

The MVVM pattern is shown in Figure 5 adapted from [27].

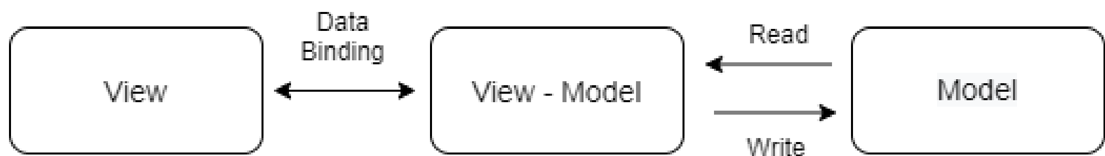

Figure 5. Model View ViewModel pattern

Event Bus, 1-way Binding

Event bus receives messages from components when an event is triggered, for example, a button is clicked and they can subscribe to it for events that they concern. Components will be notified by the event bus whenever a subscription is made to them. This pattern is also known as one-way binding. It is a middleman-like pattern because the component cannot directly read and write to the model. Instead, the component subscribes to part of the model to read, and using the state handler, it emits events to write to the model. A framework that makes use of one-way binding is React. Developers would need to orchestrate the read-write cycle by themselves. The separation of read-write logic using one-way binding is often suitable for small-scale projects where the application state is not complicated. Using this approach for larger applications like e-commerce, it might be difficult to manage and synchronize numerous state handling functions that want to mutate the same application state. [27] An example of how one-way binding looks like in a React component is shown in Figure 6 adapted from [27].

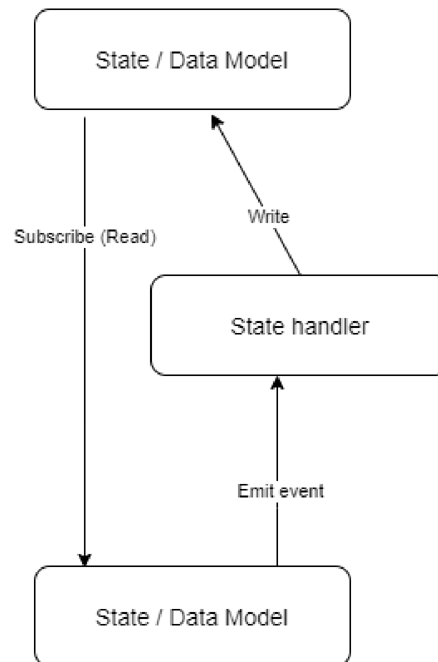

Figure 6. Example of a React component implementing one-way binding

Flux Pattern, Redux

Flux is an architectural pattern that tackled the problem of dealing with application state management for complex applications by providing explicitness of state mutation. A typical data flow of flux pattern is shown in Figure 7 adapted from [28].

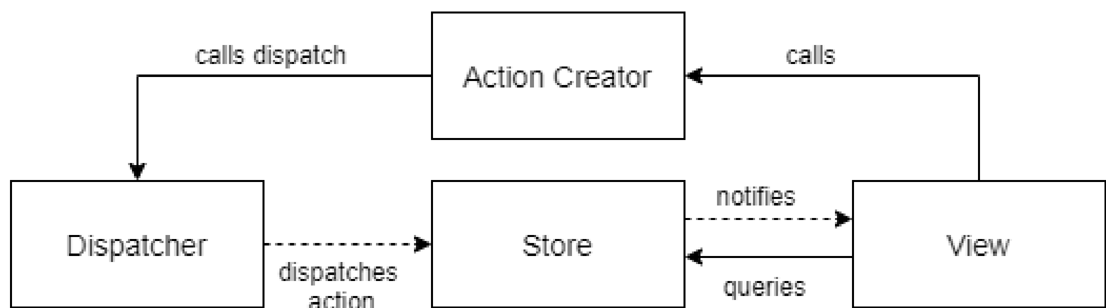

Figure 7. Flux pattern is composed on four segments in a uni-directional way: action, dispatcher, store and view

There is only one dispatcher in the flux pattern. On the other side, there can be multiple actions and stores. Actions are base functionalities of the application. For example, the user creates a post, the user deletes a comment or the user adds a comment. Actions are triggered by the view based on the user interaction and listen to the store. When the store changes, the view changes as well. The state of the application is changed only by the dispatcher which ensures explicitness of state mutation. [29]

2.4.3. Challenges of Single Page Applications

Although the concept of Single Page Application is quite popular for providing better performance, UX (User Experience), and conversations, they are built on traditional monolithic architecture. As a single page applications scale, their maintenance and deployment become slow, with every small change requiring thorough regression testing. This increases the efforts in deployment and leads to a high risk of memory leaks as well [30]. Scalability still tends to be a problem with the front-end community, not only with the growing code base and dependencies but even how development teams are managed [31].

2.5. The Emergence of Micro Frontends

The term Micro Frontend first came up at the end of 2016 at ThoughtWorks Technology Radar [3]. According to [32], micro frontend is an approach to split browser-based code of a web application into its features, and each feature is owned, frontend to backend by a different team. This ensures that every feature is developed, tested, and deployed independently from other features. This approach to development is usually confused with web application development using reusable components, but they are different from each other. A reusable, component-based architecture uses multiple pieces of code (components) and then combines them together to create a monolith frontend that sits on the top of backend services. A micro frontend, on the other end, ensures that each feature is developed, deployed, and tested independently from others [31]. With micro frontend architecture, the codebase is smaller and cohesive and thus is convenient to maintain. Since the features and pages are decoupled and independent teams are working on them, it is comparatively easier (than monoliths) to upgrade, update, and rewrite parts of the application.

Micro frontend design concept is manifested as ‘technology agnostic‘ and team code isolation [3]. Developers can choose to develop using their desired tech stack, build independent programs, set name isolation, and use browser event communication.

2.5.1. Use of Microservices in Frontend

The following information is derived from [33, 34]. Both articles listed the following micro frontend implementation schemes:

Route Distribution

While using route distribution, frontend applications are distributed through different services using routing. It is the easiest and more efficient way to slice modules. To achieve this, a reverse proxy of the HTTP (HyperText Transfer Protocol) server is needed to be implemented or by using routing of a chosen application framework. The representation of this scheme is a group of frontend applications that are clustered together to work as a single application. The main disadvantage of this scheme is loading the data that is present in different modules as the modules might take time to

be loaded. The user might experience blank screens while navigating the app which brings poor user experience. The type of systems that might benefit from this scheme are systems that difficult to be upgraded or not so much effort is spent to properly design the application.

Iframe Embedding

The Iframe is not a new technology in the world of frontend development. It is one of the ways to combine applications of the browser. Each sub-application is embedded in the system into its own iframe. Frontend applications run independently of each other as they create a standalone hosting environment without coordinating dependencies and tools with other applications. One of the main advantages of iframe is that they isolate the run-time environment of components and applications. Each sub-application can be developed using a different frontend framework either Angular, React, Vue, or native JavaScript. Iframes though should come from the same source to provide messaging between them. The disadvantage of the iframe is the bundle size: the same libraries may be sent multiple times as you cannot extract public dependencies at build time. It is also difficult to support multiple nested Iframes.

Web Component

Web Component is a group of different technologies that includes custom elements, shadow DOM, HTML templates, and imports. It gives you the ability to create reusable custom components with encapsulated functionality to use in web applications. The import in web applications is very elegant. At the same, time web components might also be bundled with microservice functionality. Although web components have some disadvantages. They are not fully supported in all browsers. When it comes to system architecture, it becomes slightly more complicated as communication between components becomes particularly challenging.

There are several questions on how to approach micro-frontend development and best practices in doing so. Paragraph 2.5.2 summarizes methods, approaches and, how-tos regarding micro-frontend.

2.5.2. Frontend Design Architecture Guidelines for Micro Frontend

Although the concept of micro frontend seems thrilling to implement into frontend architecture, there are several key decisions that need to take during the early stages of a micro-frontend application. [35]

Single Responsibility

The first thing to analyze when using a micro frontend architecture is the decision to split the application into self-contained and independently developable and deployable smaller apps. This will assure the team to be decoupled, and independent so that one smaller app would not interfere with other smaller apps. This can be achieved by building a domain-specific micro frontend with a single responsibility.

Handle Communication Between Different Components

For micro frontends to work as a single web application, they need a common and consistent way to communicate with each other. Even if they are highly independent, they still need to talk to each other. One of the common approaches is to have an application that works as an integration layer. The app can work as a container to render different micro frontends and also facilitate communication between them.

Consistent Look and Feel

Although the user interface is divided into multiple micro frontends, the users should feel as if they are interacting with a single application. The apps should have a consistent look and feel, and also the ability to make user interface changes easily across multiple apps. For example, the ability to change the font or the primary colors across multiple micro frontends should be possible. This can be done by sharing CSS and assets like images, fonts, icons, etc.

Also sharing the components that use the same user interface across the app should be possible too. This can be achieved by creating a common library of user interface components, which can be shared by micro frontends. Using shared assets and a user interface component library will allow us to make changes easily instead of having to update multiple micro frontends.

2.6. Micro Frontend Frameworks and Architecture

There are different frameworks and architectures that speeds-up micro frontend development. While talking about frameworks and architecture, we need to make sure that we know the difference between the two concepts. It is common that the concept of framework and architecture are intertwined.

According to [36], software architecture is:

> “the highest level concept of a system in its environment. The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces.”

In other words, the software architecture is the system blueprint that serves as an abstraction layer to manage the complexity of the system and establish communication among components [37].

On the other side ISO/IEC/IEEE 42010 [38] defines the concept of architecture framework as:

> "conventions, principles, and practices for the description of architectures established within a specific domain of application and/or community of stakeholders."

In other words, the framework is the implementation and part of the architecture. It is a collection of tools that speed up software development.

The sections below will talk about some of the top frameworks and architectures that are used today in the world of the frontend development.

Bit

Bit [39] is one of the most popular frameworks there is for Micro frontends. It allows developers to create frontends by using independent components. The components are then available for other teams to use. An overview of its architecture is shown in Figure 8 adapted from [39].

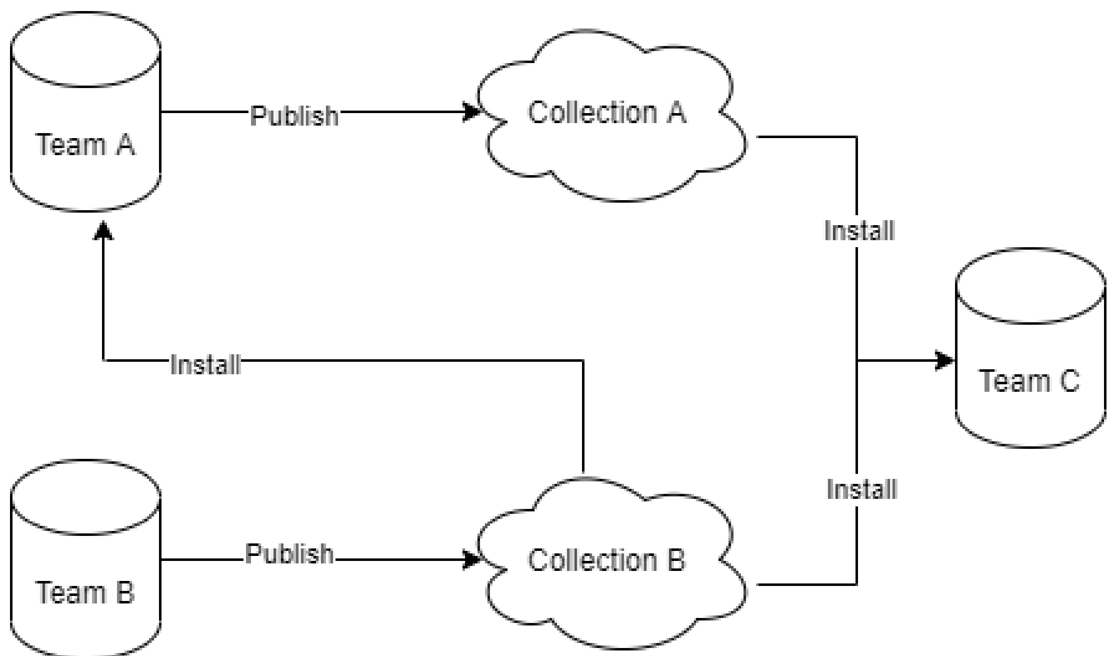

Figure 8. Bit process consists of assuming that each team has its own repository and manages deployment processes of their micro-frontend. Each team publishes their components into a collection of components that can be installed from other teams and integrated into their application. Code repositories are linked to specific collections in bit.dev. This ensures awareness of new versions being used in micro frontend applications. When there is a component change, each application that uses the component can build and deploy the newest version for the application [39].

Module Federation

Module Federation [40] is a JavaScript architecture that allows developers to create independent builds without any codependency. A common use-case for module federation is shown in Figure 9 adapted from [41].

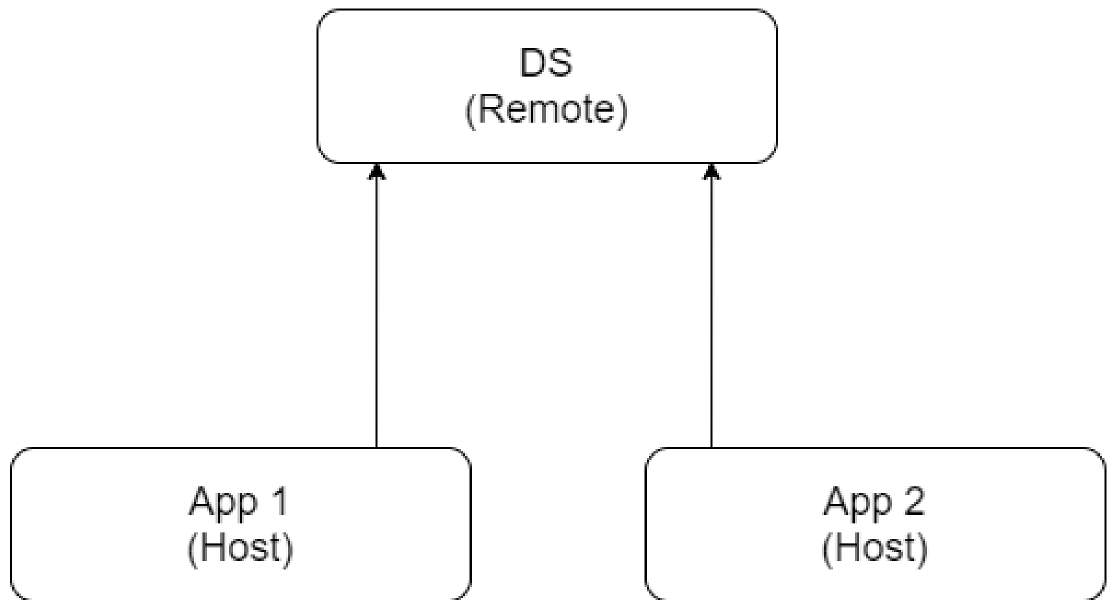

Figure 9. Module Federation Architecture is a Webpack plugin that helps to build micro frontends. It allows the developer to export code into libraries or applications to be reused by other applications. This is done by marking pieces as reusable in the plugin and then you can mark this application or library as remote which would make it usable in the other project. As shown in the figure, App1 and App2 use the application or library that is marked as remote.

Piral

Piral is the first choice to consider when you have to build a portal application [42]. A portal application is a collection of information taken from different sources, bundled into a user interface, and customized for the users based on their preferences [43]. It consists of decoupled modules known as Pilets. The application itself is modular and is expended during application run-time [44].

An overview of the Piral framework is shown in Figure 10 adapted from [44].

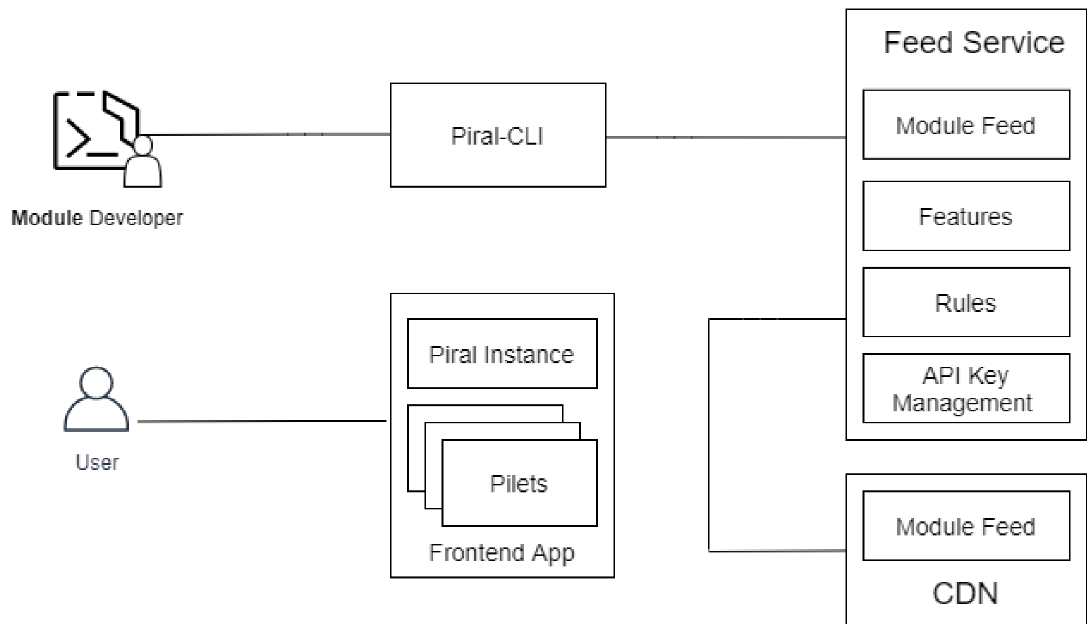

Figure 10. Piral framework is composed of three main parts: user interface application, feed service, and Piral-CLI (Command-Line Interface). The user interface application consists of the application shell called also the *piral instance* and different micro frontends that are being loaded in the piral instance. Micro frontends are called *pilets*. *Feed service* is an improved server-side rendering to serve different micro frontends. *Piral-CLI* it's a tool that comes in handy to developers who use Piral framework. Developers can easily publish their micro frontend application using the command line [44].

Single Spa

Single-spa is a framework that brings together multiple JavaScript Micro frontends in a single frontend application. Applications are combined into one single application regardless of the framework or library. In Figure 11, adapted from [45] is an example how a single-spa application looks like.

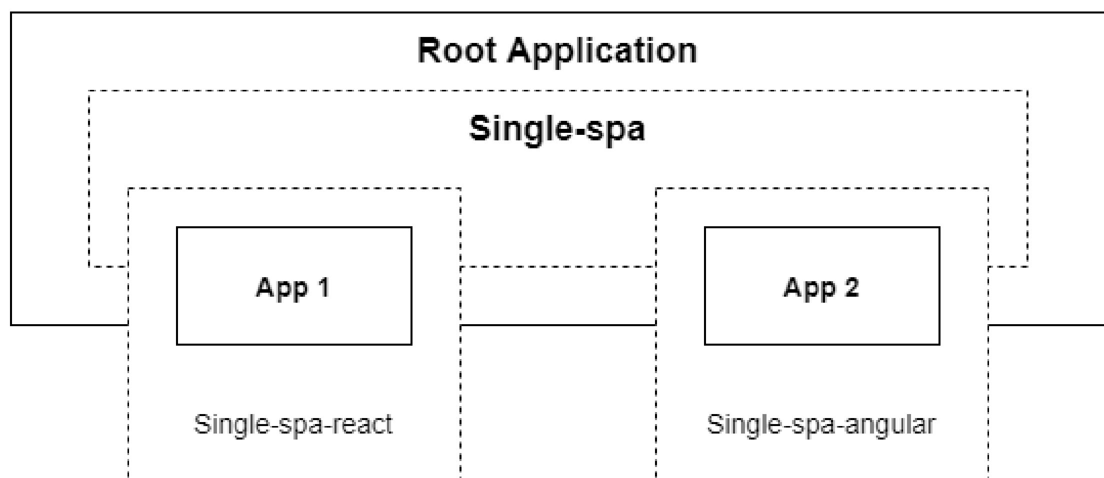

Figure 11. Application using Single-Spa framework

The parts that compose single-spa framework are:

- **SystemJs** is a module loader that loads asynchronously individual application that composes the application.
- **Wrappers** are provided by single-spa to wrap each framework created in the application that is needed for integration and bundling it into a common single-spa.
- **API** is provided by single-spa framework to communicate between the individual application.

Figure 12 adapted from [45] shows an example of how the individual application in a common single-spa application communicate between each other.

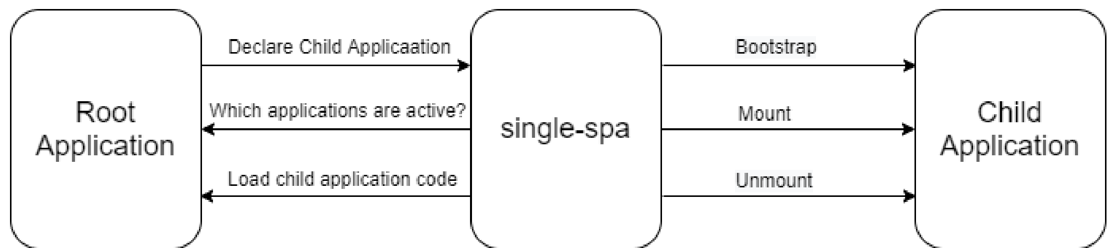

Figure 12. The *root application* registers *single-spa* as the main framework alongside with their SystemJs configurations to correctly load external applications. Each wrapper or each *child application* that composes single-spa must declare public methods: *bootstrap*, *mount* and *unmount*. The framework uses these methods to manually bootstrap the application. Almost every modern single page application framework provides a ready-to-use wrapper for the purpose of simplifying and automating the process of integration. [45]

Mosaic 9

Mosaic 9 is a bundle of services and libraries. The library contains details and specifications that determine how components interact with each other as part of a large microservice architecture application. Mosaic 9 is developed and used by Zalando. An overview of Project Mosaic architecture is shown in Figure 13. Figure 13 is from [46].

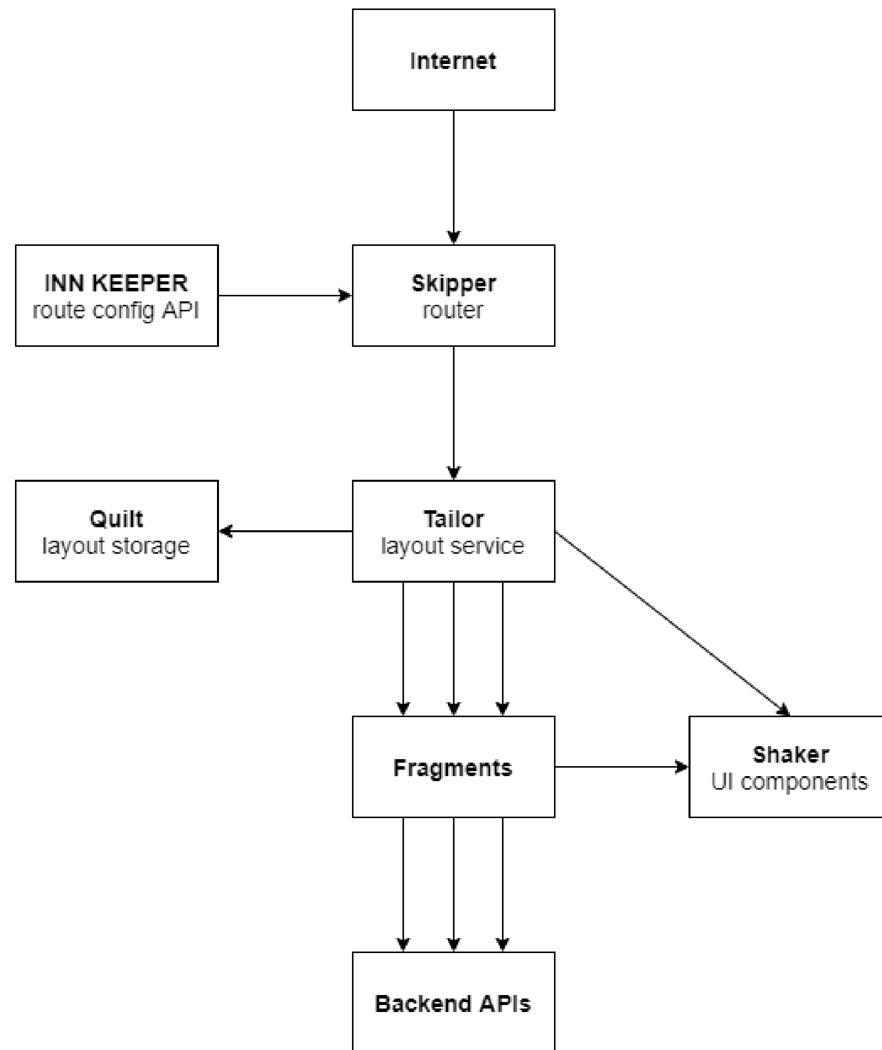

Figure 13. Zalando is using Project Mosaic, a set of services for building large-scale web applications with microservices for the frontend. It uses a Tailor library that can combine the HTML fragments, Skipper a router built on top of HTTP, Shaker for building the user interface components, Quilt for template storage so that they can be fetched, Inkeeper for storing routes across multiple teams, and Tessellate for React server-side rendering. Mosaic is used also by other companies to build their micro frontends.

2.7. Micro Frontends in Commercial Services

The micro frontend concept has been already implemented by some companies. Some of the most prominent examples come from Zalando, Upwork, and HelloFresh. In the different implementations, it is defined also as frontend micro-services. This section will dive into the overview of how the implementations are done. This will help to identify the reasons why the companies decided to use the micro frontends, advantages, disadvantages, and an idea of how one can implement micro frontends. The companies presented in the research question are known to be early adopters and

have vast experience in terms of developers. The information gathered in this section will be used to compare to the results gathered in the implementation phase.

2.7.1. Micro Frontends at Zalando

Zalando is using micro frontends on different projects. One of the ways mentioned on the company blogs is by using **Fragments**.

Fragments are isolated independent HTML pieces that are used in combination to create pages of a web application. An example of a website using fragments is shown in Figure 14 adapted from [47].

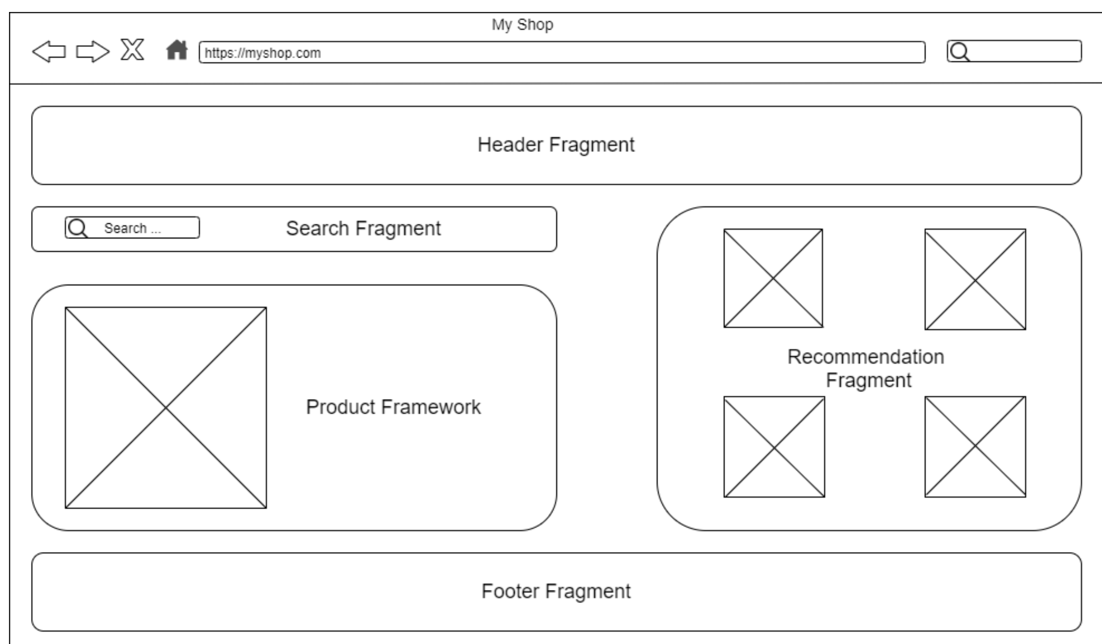

Figure 14. Example of a website using fragments

The implementation gives these benefits:

- ease of deployments with smaller pieces or fragments that are managed separately
- better scalability
- smaller complexity as each piece is smaller and understood better in isolation and the technology stack is independent

The technological stack isolation has its downside as using different user interface libraries may increase the bundle size for the main app. Most of the pieces are usually done using the same library. [47, 46]

The main disadvantage of using Mosaic is one shared with backend systems, inconsistencies in the long run. This is exacerbated by the need for user interfaces to

be uniform and provide a consistent user experience. This disadvantage is mitigated by using boilerplate projects for each micro frontend and sharing the user interface libraries. This will also mitigate the overhead caused by the need of setting up the infrastructure for each micro frontend. Other ways to help with the disadvantage is to use libraries to share reusable components between the different fragments.

Zalando is actively working on replacing the Tailor part of Mosaic with a new framework. The framework will use an API aggregation layer, multiple renderers which will be self-contained code that will have their own data and views, a backend service for combining the different renderes to build pages and an orchestrator for mapping the data to the different renderers called rendering engine.[47, 46]

2.7.2. Micro Frontends at Upwork

Upwork [48] is a freelancing platform that connects clients like business owners, entrepreneurs with freelancers when they need help to get a job done [49]. The domain of projects varies from web and mobile development to SEO (Search Engine Optimization), social media marketing, content writing, graphic design, admin help, and thousands of other projects.

Upwork has made the jump to microservices in the backend and after the transition, they decided to modernize the frontend with the micro frontend approach. The approach was chosen because of the benefits found in the backend with the microservices and trying to replicate those in the frontend. The main reasons are independence between different projects, team expertise focused on smaller pieces, more manageable deployments, and being able to prototype and test out new technologies faster. One of the main issues found in the Upwork case was the difficulty of maintaining the user interface look and feel consistency.

- **Navigation**: The modernization process of the already existing monolith into an architecture composed of micro frontends yielded difficulties with navigation. Upwork had a complex user-based navigation menu generated dynamically. The logic splitting was an issue here and reflects on the point that monolithic frontends suffer from big components with tangled logic. The challenge was solved in few steps. First by leaving the navigation logic in the monolith and exposing it, introducing so a dependency on the monolith. Later they split the logic into a split service that would be used to produce the needed information about the complex logic and another service that would consume the logic and produce the HTML representation.

- **User Interface library**: Another challenge faced by the teams at Upwork was the need for a shared user interface library to provide the look and feel consistency. They used the opportunity to introduce a new look for their application. They created a separate library with the components that would be used by other micro frontends. They also leveraged versioning and each micro

frontend would have a version of the component library. This allowed Upwork to mitigate the moving to newer versions in a controlled manner.

- **Routing**: Lastly they introduced a new routing system that would combine different micro frontends. The challenge was when trying to move from one already stable way of routing to a new one where services would need to register themselves into the routing solution. This was mitigated by an Nginx solution with automation of configuration generation.

As a summary, Upwork benefited and was able to solve common issues that appear when moving into microarchitecture. It is important to know the issues from an already proved solution that walked through the transition and have a success story with a company of this stature satisfied with the architecture.[50, 51]

2.7.3. Micro Frontends at HelloFresh

HelloFresh [52] is a leading global provider of fresh food at home [53]. It offers a subscription service that delivers everything you need to cook. The business headquarter is in Berlin, Germany and it operates in twelve international markets. The US market has the largest market by having 1.48 million active customers. HelloFresh reported 1.3 billion euros in sales in the first nine months of 2019 [54].

HelloFresh started moving into micro frontends in 2016. It is described as frontend microservices. They define it as a separate frontend that serves HTML, Javascript, and CSS at a unique endpoint tied to that frontend. The technology it uses should be freely chosen, the external dependencies should be minimal, it can be deployed separately and it can run independently of other environments.
On the implementation side, HelloFresh uses a similar approach to Upwork in the routing part by having services register to an Nginx server. This allows adding new routes independently. HTTP errors are handled by a split micro fronted which displays the error pages in a consistent manner.

They use three main pieces to compose their frontends:

- **Fragments** are inspired by Zalando approach but are defined in HelloFresh as an entire service serving a page, particles are the inner pieces of the fragment that are loaded synchronously and tag parts that are loaded asynchronously. Fragments are mentioned as a single technology at the time of the written article, a React application with server-side rendering.

- **Particles** are pieces of a page such as a header, footer, etc. These are implemented with a server-side rendering approach as well. The particles are independent and can live also outside a single application. Fragments request the particles and load them to form the page. The tag is usually used for smaller pieces such as asynchronous modals that don't have their own split routes.

- **Tags** in HelloFresh are also usually small react applications which load in windows and are appended on the body.

As a summary, it all ties together by having a request go through the router, which will call the requested fragment, which in turn will be composed of particles. The tags are then only loaded asynchronously depending on the use case and live separately. This is shown also in Figure 15 adapted from [55].

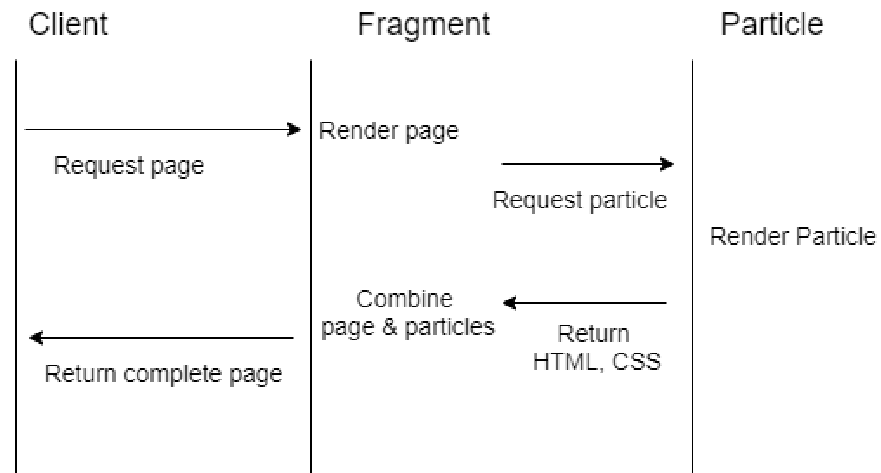

Figure 15. HelloFresh micro frontend infrastructure

Moving to micro frontends slowed down the development speed initially at HelloFresh as it requires setting up new services end-to-end for new functionality. After surpassing the learning curve the development speed picked up by a lot. This was also followed by an increase in the confidence of developers since they spent time to surpass the learning curve steepness. Another topic mentioned is the technology freedom of projects. While the solution allows using split technologies the experience was that it was beneficial sticking to a singular technology but this was by choice and allowed otherwise if needed. Error tracing is mentioned as a good result since the HelloFresh team uses mostly a boilerplate with unified and consistent logging for each fragment. As a summary of the benefits, they were able to move from a big monolithic app to a maintainable one with fewer dependencies, isolated, and much easier to test due to the isolated environment.

3. EXPERIMENT DESIGN AND SETUP

This chapter walks through the planning of the experiment design and the experimental setup to see how developers perceive and experience micro frontend development and how different level of development experience changes this perception. The goal of this study is to tackle the advantages and disadvantages of using micro frontend comparing to traditional frontend application development like single page application by gathering developers' opinions and first-hand experience on the topic.

This study aims to help to understand whether the micro frontend approach is the solution to the issues that frontend monolithic architecture is facing and how micro frontend helps to solve them.

3.1. Case Study: Micro Frontend Evaluation

The motivation behind this paper is to find out whether the micro frontend development approach is the answer to alleviate the issues present in traditional frontend development by gathering opinions of developers with different levels of experience. For this purpose, we propose a method/approach of micro-frontend development that will evaluate and measure how the approach of using micro frontend is perceived among experienced and inexperienced frontend developers. In order to achieve the best result, developers will be asked to setup a simple frontend architecture of an MVP (Minimum Viable Product) project using 2 different approaches: single page application and micro frontend. By obtaining experience from this experiment or extending their current experience on setting up single page application and micro frontend application, developers will share time measurements of preparing for the experiment, designing the architecture of the application, and implementing the functionalities. This information will be used to see and compare how different developers experienced micro frontend development in comparison to single page application development. Their perceived opinions about the topic through questionnaires will be used to gather quantitative data.

Scenario

You work in a consultancy company. A client has come in. They want to develop a business-to-business online shopping system. They want something that will be developed from the ground up. Your task is to develop a frontend MVP to start with. Then this will be expanded later. There are 2 basic actors: Wholesaler and Customer. The MVP has 4 base functionalities.

1. Customers can make orders to a wholesaler.
2. The wholesaler confirms the sale orders
3. Customers can make subscriptions for their orders.
4. The wholesaler can check who visits their stores.
5. The wholesaler can add and keep track of the products and manage their prices and discounts.

In this scenario, the developers need to implement only the admin dashboard view where the wholesaler can:
1. See the customer orders
2. See the sale orders they confirmed
3. See the subscriptions
4. See the visitors to their store

The setup of the MVP should be mindful for the future of the system. The system may be expanded in the future so that it includes more dashboards. Such functionalities are planned to be such as a dashboard that shows product trends, price history, and revenue. Other functionalities may be added in the future. Although these are out of the scope of this task they should be considered when developing the MVP.

The application interfaces are going to be developed and implemented using 2 different approaches:

- No micro-frontend. Participants will use their own experience and technology to complete the task using single page application approach. The participant should create an API consumer using their chosen frontend technology (Angular, React, Vue, Svelte, etc.). The server instructions are provided in a Github repository shared beforehand. The participant should follow the design which can be seen if he runs locally the project in the repository. The goal is to make it future-proof keeping in mind that more features could be added.

- With micro-frontend. Participants will be asked to use micro frontend approach in order to deliver the task. The server can be run using instructions from the Admin-Dashboard repository. The participant should follow the design which can be seen if he runs locally the project in the repository. The goal is to make it future-proof keeping in mind that more features could be added. The frontend should be split into different micro frontends for each of the views or the participant is free to choose a different approach as seems more reasonable.

GitHub repository

Github repository is a good source to provide virtual storage for projects that other developers can access. The GitHub repository for the experiment was setup before the experiment took place. It provided boilerplate templates for the participants to help them get started with the application setup. The repository was named Admin-dashboard [56] and it also provided instructions on how to setup the server-side of the application. Developers could download the project or clone it.

Wireframes

This section covers the wireframes and the details of the implementation that the participants need in order to execute the experiment. The features that are needed to be implemented are:

1. Display the list of menu items on the left side of the template. It should have an icon and label and the possibility to navigate to different contexts.

2. Default active link is overview and on the right side of the template a widget that contains an overview of visitors, subscribers, sales, and orders card should be shown
3. When clicking each of the cards or link in the sidebar menu, the app should navigate to a different route displaying each widget in more detail in the form of tabular data.

All data will be provided by using a full fake REST API JSON (JavaScript Object Notation) Server [57]. In order to have a more fluent experiment, a basic HTML structure will be provided to experiment participants alongside CSS styles.

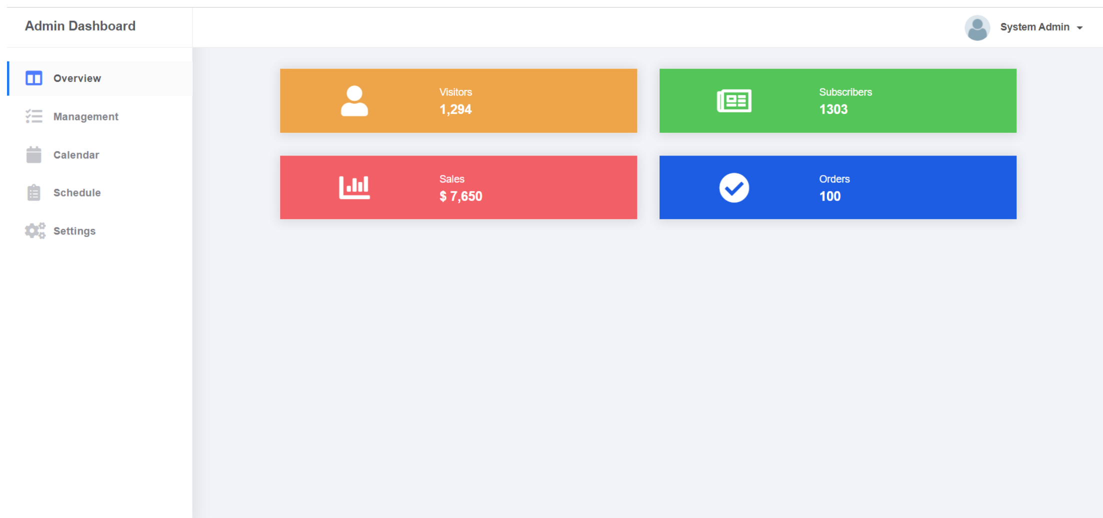

Figure 16. Admin Dashboard main page screenshot

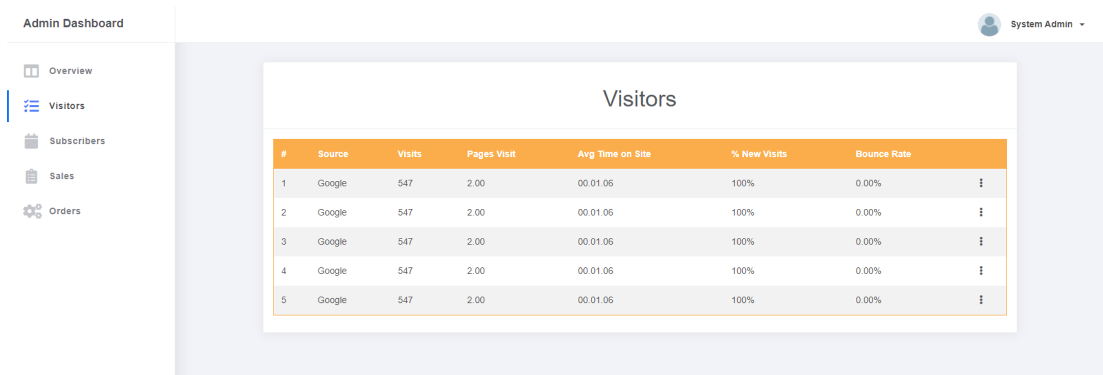

Figure 17. Visitors details page screenshot

Admin Dashboard

System Admin

Overview
Visitors
Subscribers
Sales
Orders

Subscribers

#	Name	Phone	Address	City	State	Zip	Email
1	Mark Otto	(217) 482-1341	319-7778 Erat Street	Rolling Hills	ND	94547	name.surname@gmail.com
2	Mark Otto	(217) 482-1341	319-7778 Erat Street	Rolling Hills	ND	94547	name.surname@gmail.com
3	Mark Otto	(217) 482-1341	319-7778 Erat Street	Rolling Hills	ND	94547	name.surname@gmail.com
4	Mark Otto	(217) 482-1341	319-7778 Erat Street	Rolling Hills	ND	94547	name.surname@gmail.com
5	Mark Otto	(217) 482-1341	319-7778 Erat Street	Rolling Hills	ND	94547	name.surname@gmail.com

Figure 18. Subscribers details page screenshot

Admin Dashboard

System Admin

Overview
Visitors
Subscribers
Sales
Orders

Sales

#	Date	Quarter	Customer	Region	Product	Quantity	Revenue
1	01.02.2013	Q1	Customer 1	West	Product 9	15	$1,530
2	01.02.2013	Q1	Customer 1	West	Product 9	15	$1,530
3	01.02.2013	Q1	Customer 1	West	Product 9	15	$1,530
4	01.02.2013	Q1	Customer 1	West	Product 9	15	$1,530
5	01.02.2013	Q1	Customer 1	West	Product 9	15	$1,530

Figure 19. Sales details page screenshot

Admin Dashboard

System Admin

Overview
Visitors
Subscribers
Sales
Orders

Orders

#	Order Number	Order Date	Product number	Description	Number Ordered	Quoted Price
1	21608	20.10.2015	9	Product description	20	$21.95
2	21608	20.10.2015	9	Product description	20	$21.95
3	21608	20.10.2015	9	Product description	20	$21.95
4	21608	20.10.2015	9	Product description	20	$21.95
5	21608	20.10.2015	9	Product description	20	$21.95

Figure 20. Orders details page screenshot

3.2. Subjects

The main participants that will be part of the experiment are frontend and fullstack developers with different levels of experience. The experience of each participant will be documented using a survey, shared prior to the experiment. The evaluation will be done among participants with and without experience in micro-frontend development. The recruitment of participants will be made possible by directly contacting them via Microsoft Team and Slack. Participants will be mainly from work environment and known professional circle. There is no need to organize meetings with the participants as the experiment will take place remotely.

3.3. Methodology

The experiment will take place not in a specific place as it will give the participant the opportunity to get informed and learn the micro frontend approach first if needed. Prior to the experiment, the participant is required to complete a survey that will quantify its experience so the evaluation of the experiment can be based on how familiar this method is for the participant. The experiment will be using a qualitative research method as the participants will be asked to answer a survey that will summarize their experience, and their preferable frontend development approach.

3.4. Procedure

Prior to the experiment, the participant will answer a survey related to its qualification and experience in general frontend development including Single Page Applications and Micro frontend. Survey questions marked as **Ax** are listed in section 3.5.

After the first step, each test participant will have access to a PDF file, shared privately that will contain the following information:

1. Experiment background information
2. What is the purpose of the experiment and a simple experiment scenario
3. Requirements for single page application and micro frontend approach
4. Link to Github repository. The participant can download the static frontend setup and transform it to a fully functional single page application and micro frontend application. To simplify the process, participants can make use of a fake REST API. Instructions on how to setup the server will be provided in the Github repository.

The experiment should take about 2 - 4h of work for each approach to be completed successfully but this may vary depending on the participant experience and familiarity with micro frontend and/or single page application. After the experiment is successfully completed (or not), the participant is asked to complete a survey to give his feedback about the experiment, any unforeseen complications, how did they manage to complete it, and a short evaluation of the chosen approach. Survey questions marked as **Bx** are listed in section 3.5. In the same survey, participants are asked to share a PDF or PowerPoint presentation highlighting key details of the architecture and implementation *(B16)*. They are asked to explain how they implemented the solution, what framework (if any) did they use, and what challenges did they encounter during the development. The participant can choose his preferred technology stack while setting up the architecture on the application and delivering the assignment.

To get started with micro frontend development, a few articles and tutorials are provided for the experiment participant to jump into.

3.5. Survey Data Collection

During the experiment, will be gathered 2 types of data from test users. From the first survey, we'll have qualitative data that will quantify the participant experience and familiarity with single page application and micro frontend approach. The list of questions are:

- **A1:** *What would best describe your profession?*
  The respondent can choose among *fullstack developer, frontend developer* and *backend developer*.
- **A2:** *What level of seniority do you have?*
  The respondent can choose among *novice(0-1 year, advanced beginner (1-2 years), competent (2-5 years), proficient (5-10 years,* and *expert(10+ years)*
- **A3:** *Please list your familiar tech skillset stack.*
- **A4:** *What describes best your level of experience with microfrontends?*
  The respondent can choose among *less than 1 year, more than 1 year* or *no experience*
- **A5:** *If you have experience with micro frontend architecture, what technology did you use to implement it.*
  The respondent can choose among *Iframes, through NGINX, web components, monorepos, customized orchestration, micro frontend framework* and *other*
- **A6:** *If you used a framework to implement the architecture, please specify the name of the framework.*

From the second survey, we'll also have qualitative data that will compare which approach is more preferable among our participants and how experience is related to delivering the experiment. The list of questions are:

- **B1:** *How much time (in hours) did you spend to prepare prior to implementing the SPA experiment?*
- **B2:** *How much time (in hours) did it take to design the single page application architecture?*
- **B3:** *How much time (in hours) did it take to implement the single page application?*
- **B4:** *How much time (in hours) did you spend to prepare prior to implementing the micro frontend the experiment?*
- **B5:** *How much time (in hours) did it take to design the micro frontend application architecture?*
- **B6:** *How much time (in hours) did it take to implement the micro frontend application?*

- **B7:** *On a scale of 1 to 5, with 1 being very easy and 5 being very difficult, please rate the application architecture?*
  The respondent can input values for both single page application and micro frontend from a range of 1 - 5

- **B8:** *Please explain the reasoning behind your SPA score. What positive and/or negative elements influenced your response scale?*

- **B9:** *Please explain the reasoning behind your micro frontend score. What positive and/or negative elements influenced your response scale?*

- **B10:** *Do you think that the micro frontend approach that you chose would have any benefits compared to the SPA implementation? If yes, please explain.*

- **B11:** *Based on your opinion, what are the advantages and disadvantages of using a micro frontend approach in front-end applications?*

- **B12:** *Based on your opinion, what are the advantages and disadvantages of using a SPA approach in front-end applications?*

- **B13:** *While comparing SPA and micro frontend approach, which approach do you think is better if we want to achieve: better team productivity, better scalability, better continuous integration, development and deployment, better performance, better UI/UX experience, technology independence, small scoped project, medium scoped project and large scoped project*

- **B14:** *SPA VS Micro Frontend: Which one do you choose and why*

- **B15:** *Would you consider using micro frontend approach in the future? Explain your reasoning briefly.*

- **B16:** *Please attach your presentation file (PDF or PP) containing short information about the key details of the architecture and implementation. Shortly explain how you implemented the solution, what tech stack and micro frontend framework (if any) did you use, and what challenges did you encounter during the development.*

4. RESULTS AND DISCUSSION

This chapter takes the results of the experiment and analyzes them. Then it reflects on the experiment results and how they relate to the existing research. The main focus is to tie in how the developers perceive micro frontends in general. In more detail, this chapter analyzes the benefits found in micro frontends, check if the benefits match between users with different levels of experience, what do the users find hard about the micro frontends, and the readiness of using micro frontend for their next project.

4.1. Experiment Contributors Overview

The experiment details were handed over to 10 developers with different experience in the field starting from few months of experience up to more than 1o years of experience. At the same time, an email targeting students in computer science was sent. An overview of participants and their experience is displayed in Table 2.

Participant reference	Experience
Participant 1	10+ years
Participant 2	10+ years
Participant 3	5 years
Participant 4	4 years
Participant 5	1.5 years
Participant 6	9 months

Table 2. Participant are referred with the keyword participant followed by an order number

Only 6 out of 10 developers could complete the experiment. The main reason among developers that were asked to complete the experiment and were not able to deliver it was time limitation since the experiment required a dedication that was equal to one day of work or more based on the experience. Regarding the email that targeted computer science students, the response rate was 0. This can be related to the lack of experience from students in this specific topic.

4.2. Implementation Time

This section will describe the findings coming out from the implementation time. The participants in the experiment were required to reply about the time they spent preparing or learning about single page applications and micro frontends (**B1 & B4**), choosing the architecture (**B2 & B5**) and time spent implementing it (**B3 & B6**). The results will reflect how time is spent in different sections to compare the two approaches.

4.2.1. SPA

The data gathered in this experiment show that the time varies between different levels of experience. A summary of how much time did participants take to complete the single page application project in each phase is displayed in Table 3:

Subject	Experience (years)	Preparation time	Architecture design time	Implementation time	Total time
Participant 1	10+	0 (0%)	0.5 (16.7%)	2.5 (83.3%)	3
Participant 2	10+	1 (20%)	1 (20%)	3 (60%)	5
Participant 3	5	0 (0%)	0 (0%)	2 (100%)	2
Participant 4	4	0 (0%)	0.5 (25%)	1.5 (75%)	2
Participant 5	1.5	0.5 (16.7%)	0.5 (16.7%)	2 (66.6%)	3
Participant 6	9m	2.7 (50%)	0.5 (9.2%)	2.2 (40.8%)	5.4

Table 3. Overview of how much time (in hours) did every participant spent in each of the phases

What can be seen is that the SPA architecture is that among participants with three or more years of experience the time of preparation is close to zero. This reflects the fact that the participants are already familiar with single page applications and have a well-known path to follow which was underlined in their answers. Another thing that can be seen from the results is also that the time spent choosing the architecture was also less than one hour. As common reasons that were given when answering question **B8**, the familiarity of single page application due to previous working experience was highlighted. Participants said that "*Previous experience in implementing production-grade SPA applications*" (P1), ""SPA is much easier to implement and everything is in one place. (P2), "*I have done previous spa applications before*" (P3), "*Application architecture of SPA was relatively easy due to my working experience*" (P4), "*I was familiar with setting up a new project*" (P5), "*I have been working with SPA*" (P6).

The same outcome might not be applied in company projects. Usually, before taking a project into development depending on the size of a project, more time is invested in the architecture design phase. This is usually the most important phase alongside implementation where a conceptual architecture infrastructure is built. This phase requires the input of an architect as there are several key frontend decision that needs to be made at the beginning of the project as the cost of making wrong decisions it's usually high.

Among our participants, a common reason for investing little time in the preparation and architecture phase was the existence of project starters in each of the implementations. The frameworks used were React, Angular, and Svelte. In each of the solutions, the participants replied of having a well-known way of starting their project with the React users using "*create-react-app*" (P3, P6), a common way of creating new react apps[58], Angular users implementations are using "*Angular CLI*" (P2, P4, P5) to setup their projects and one implementation with Svelte using "*Vite's Svelte template*" (P1). An overview of single page application frameworks chosen by the participants is shown in the Figure 21.

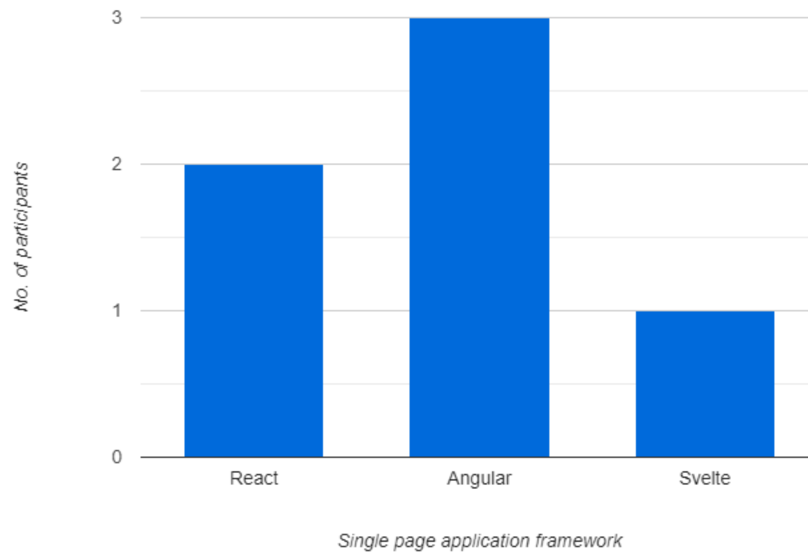

Figure 21. Single page application framework decision among participants

An interesting result that can be seen is that participant 5 and participant 6 each with less than two years of experience did not take a long time to think about the project architecture and stated that they followed the structure of the projects that they had previously used in their workplace.

Among these participants, participant 5 with an experience of 1 year and a half took 16.7% of the total time for preparation and another 16.7% for the architecture stating that "*I followed the same architecture that is implemented in the project I'm working, so I was familiar with setting up a new project*".

Participant 6 had less than one year of experience and spent 50% of the total time on the preparation and then 40% of total time to implement the architecture. The responses also outlined that the ease of starting the SPA application was common across all levels of experience.

When we delve deeper, we can see that the average time for the participants with three or more years of experience took an average of 2.5 hours to implement the SPA application and chose a platform they use in their daily work, with the exception that participant 1 used Svelte, a platform they didn't use before at work. Participant 5 and participant 6 with less than two years of experience took also approximately two hours both in the implementation phase. As a summary, the total time was similar across all participants with the exception of the novice developer, taking a total of 5.4 hours to complete the experiment. An overview of participants experience and total time they spent to implement single page application approach is shown in Table 4.

Subject	Experience	Total time (hours)
P1	10+	3
P2	10+	5
P3	5	2
P4	4	2
P5	1.5	3
P6	9m	5.4

Table 4. Overview of participant experience and the time it took to complete the assignment

4.2.2. Micro Frontend

Micro frontends results reflect a total opposite of the SPA results. While the SPA results were consistent the micro frontend ones vary between participants and have no direct connection to the years of experience but more with the fact that they have worked before or not with micro frontends. A graphic showing the participants' experience with micro frontend is shown in Figure 22.

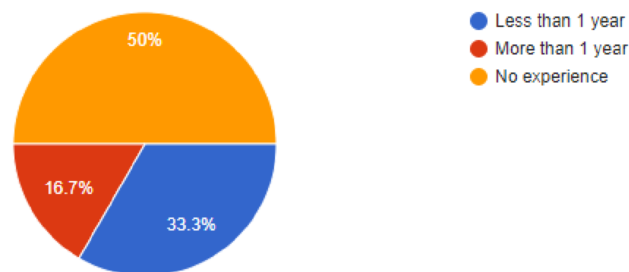

Figure 22. Graphic of the experiment participants experience with micro frontend

The preparation time, architecture planning and implementation can be seen in the Table 5 for all participants.

Subject	Total exp.	MF exp.	Prep. time	Arch. design time	Impl. time	Total time	Success rate
P1	10+	< 1y	1	1	4.5	6	100%
P2	10+	-	2	3	7	12	100%
P3	5	1+	2	1	1	4	80%
P4	4	< 1y	2	0.5	3+	5.5+	60%
P5	1.5	-	3	1	2+	6+	50%
P6	9m	-	2	2	-	4	0%

Table 5. Overview of time spent in each of the phases for micro frontend approach

The developers with more than ten years of experience, participant 1 and participant 2 were the only ones to fully complete the micro frontend setup taking respectively four and half hours and seven hours to complete it. Among other participants, participant 3

completed the experiment partially with the implementation being able to run locally and having a well-defined structure but commented that they couldn't get the setup into a production-ready state and it wouldn't be able to run tests across all the packages. Participant 3 completed the micro frontend after two hours of preparation, one hour of architecture, and one hour of implementation, totaling four hours. They had previous experience in using micro frontends. Furthermore, it adds "*it was a process of trial and error and still very complicated to grasp*"

Participant 4 said that "*microfrontend application was more difficult to jump to and not as straightforward compared to SPA approach*". Participant 4 chose to implement the assignment using single-spa framework and said that "*while still following the official documentation and recommended setup, I had difficulty to have the application up and running*". Participant 4 spent 2 hours to prepare for micro frontend, 1 hour for architecture design and more that 2 hours for implementation.

Participant 5 responded they could setup the architecture and get the application to a semi-working state but not being able to run it yet and responded that it could take a substantial amount of time for them to finish it. He spent a total of six hours spread across 3 hours of preparation, 1 hour of architecture design, and more than 2 hours of implementation.

The novice developer, participant 6 couldn't implement the micro frontend and responded that they gave up after studying the micro frontend approach for two hours and trying to architect and run an application with it for two other hours. While answering question **B6**, participant 6 said: "*I tried to implement the micro frontend but it seems to be a little difficult for me to apply the new concepts.*"

4.2.3. Comparison

By comparing the data gathered we can conclude that the time to prepare and plan the architecture for the micro frontend implementation is higher. Figure 23 and Figure 24, gives an comparison between two approaches for preparation time and architecture design time.

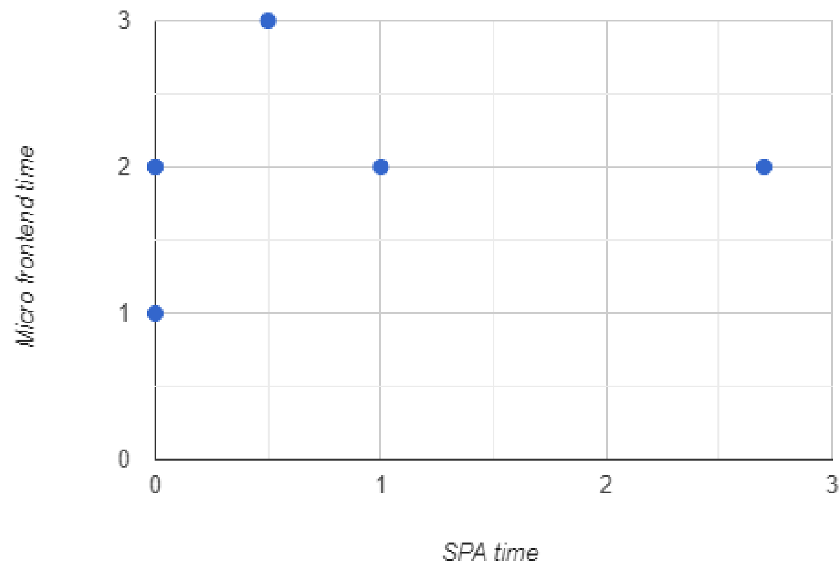

Figure 23. Scatter plot graph that shows a comparison of how much time it took participants to prepare for single page application vs micro frontend

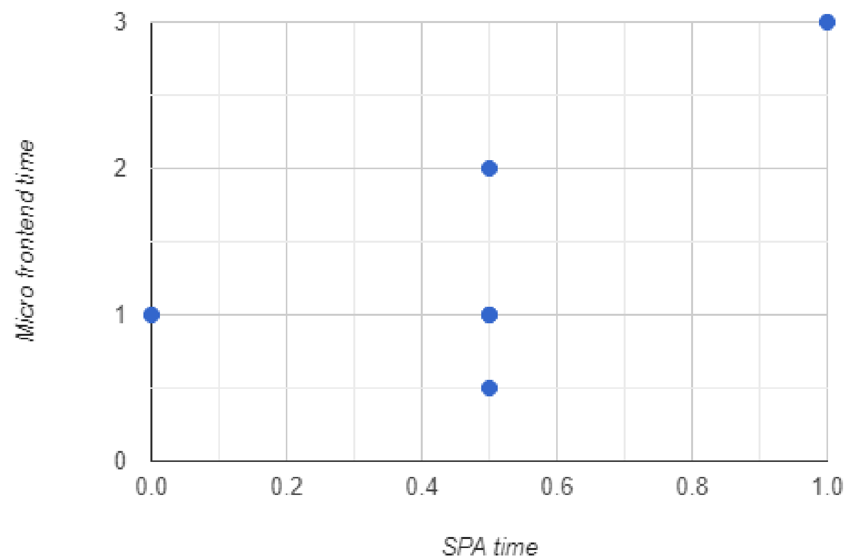

Figure 24. Scatter plot graph that shows a comparison of how much time it took participants to design the architecture for single page application vs micro frontend

Paired t-test results for preparation time showed a P-value and statistical significance of the two-tailed P value equaling **0.0393** hours. The mean of single page application

is **0.700** hours and the mean of micro frontend is **2.000** hours resulting in a confidence interval of **1.300** hours meaning that micro frontend took longer to prepare.

Paired t-test results for architecture design time showed a P value and statistical significance of the two-tailed P-value equaling **0.0284** hours. The mean of single page application is **0.500** hours and the mean of micro frontend is **1.417** hours resulting in a confidence interval of **0.917** hours meaning that micro frontend took longer to design the architecture.

The data gathered on the implementation part gives mixed results with participants realizing the implementation phase by reusing pieces they built during the implementation of the SPA and with that shortening the time for the micro frontend implementation. Paired t-test results for implementation time showed a P value and statistical significance of the two-tailed P-value equaling **0.2053** hours. The mean of single page application is **2.200** hours and the mean of micro frontend is **3.500** hours resulting in a confidence interval of **1.300** hours meaning that micro frontend took longer to implement.

Participant 2 stated that "*I started with the SPA part, so there was a lot of code that I was able to reuse in the micro frontends (MF) part*". It took him seven hours to complete the micro frontend compared to three for the SPA regardless of the experience. But a visible result is that all the participants "*started with the SPA implementation first and reused it later for building the micro frontend*" (P2, P3, P4). This aligns with the findings in the research section that the micro frontend is a good approach when migrating already existing single page applications.

While the SPA implementation showed that the time to setup and implement decreases with the increase of the years of experience the micro frontend doesn't reflect that fully, but it shows that the years of experience have an effect on the ability to build the functionality end to end with it. This is tied to the responses of the users that they have not needed to build the architecture themselves in less experienced participants and has a higher prevalence on the ones with more experience. Previous micro frontend experience is an important indicator of the ability to plan and execute the infrastructure. The users that have used it previously have a clear path of executing it and it is reflected in the total time. Two of the users that had previous experience with it, participant 1 completed the micro frontend part respectively in four and half hours and participant 3 in one hour but not being able to run it in production.

4.3. Ease of Development

This section will describe the findings of the analyzed written responses from the participants. Participants responded to how difficult it was for them on a scale from one to five to plan the SPA and micro frontend architectures **B8**. They also were asked about how they feel about the micro frontends compared to single page applications **B13, B14**. The results are used to see how ready the developers would be to jump into the micro frontends. In this section it is also analyzed how the responses vary based on general experience and the experience with micro frontends. Furthermore other aspects such as if previous experience with microservices have helped the participants are analyzed. The findings are then also put in comparison with the findings of the research.

4.3.1. How Difficult Are SPA Vs Micro Frontends

Subject	Single page application difficulty	Micro frontend difficulty
Participant 1	1	2
Participant 2	2	4
Participant 3	1	3
Participant 4	1	5
Participant 5	2	5
Participant 6	2	5

Table 6. Participants were asked to rate single page application difficulty and micro frontend difficulty

The first response is that the users have scored the SPA approach with an average difficulty of 1.5 with three participants scoring the difficulties with two and the other three participants with one. In comparison, the micro frontend was scored with an average of 4. Among the participants, only participant 1 scored it with a two and responded that they are very familiar with it, have used it before, and even gave a talk about micro frontends.

Participant 2 took the most time with the micro frontend and said that "*it was a process of many trial and error*" with "*no well-defined path of how to setup micro frontends*".

Participant 3 responded with a three and was using micro frontends in their daily work. The reasoning was that the micro frontends can be "*started easily*" but it takes more work and experience to bring it to a production-ready state.

Participants 4, 5, 6 scored it with a five. Participant 6 couldn't finish the setup at all and "*read about Microfrontend, but don't have the ability yet to implement it. I can just think how it works in thick lines*".

Participant 4 had "*difficulty to have the application up and running*" with the compilation steps while trying to setup their micro frontend.

Participant 5 had less than two years of experience and used a micro frontend starter but couldn't get it fully running.

What we can see from the results is that all the participants no matter how little or much experience they had, they still could easily set up single page applications. This is also reflected in the findings of the research. The answer for a high difficulty for most participants in setting up the micro frontend correlates to the findings that micro frontends are harder to setup and orchestrate[47]. The main reason behind the ease of the setup of SPA was that they have a well-defined template for setting up those. This was also one of the main reasons why they had difficulties to setup the micro frontends. The lack of well-defined starters or templates troubled all the participants and the ones who were able to implement it needed to do extra work even when they used templates and participant 1 did their own orchestration.

4.4. The Tool for the Job

When analyzing the findings the participants were quite positive about using single page applications for small and medium projects, with all participants saying they would use single page applications for that purpose **B13**. For large projects, the answer was micro frontend for all users. Participants responses are displayed in Figure 25.

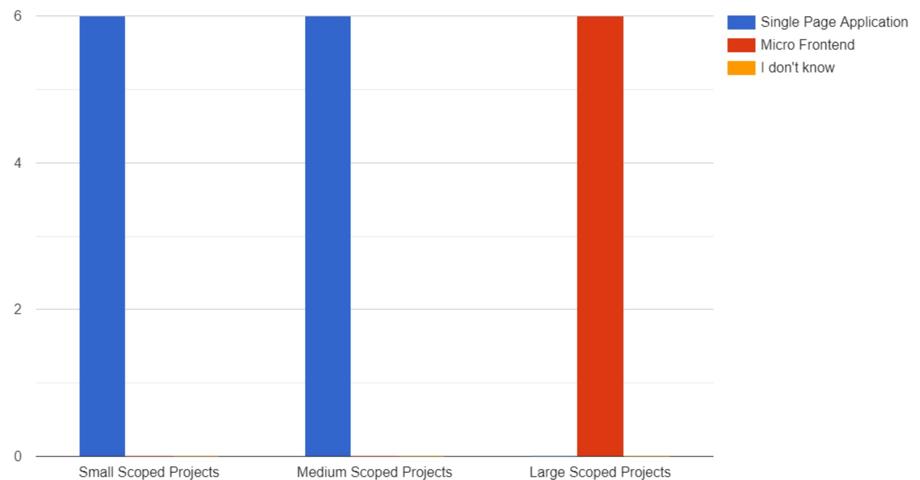

Figure 25. Participants opinion of choosing single page application for small to medium size project and micro frontend for large scoped projects

The consensus from the participants on the approach reflects what can be found also from the research. Micro frontends are a tool that only serves certain needs. Even participants who had high difficulty on setting it up agreed that it would be "*worth the difficulty*" (P3) of setup when considering the different benefits. Also, all participants mentioned that the approach is too complex for smaller scale and it's "*only for large systems with many different programmers and teams*" (P1, P2). More specifically participants answered that "*depends mainly on the size of the team and the project*" (P3) and "*if we know in advance that the application will be heavily extended in the future*" (P1). In other words, team size and the life-cycle of the software would be key to this decision. The participants said that the reasons for not choosing single page applications are that the different parts of the project would be more dependable on each other and that "*maintenance becomes harder*" (P4), resembling that of traditional monoliths.

4.5. Implementation Details

This section describes which tools, languages, and platforms were chosen for implementing the single page applications and micro frontends. Through the analysis of the tools, it is aimed to show how custom the solutions are for the SPA compared to micro frontend, which are the popular choices for each, and check if the choices in micro frontends play into the strengths of the approach. The analysis here shows if the choice of SPA has a direct effect on the selection of the micro frontend approach. Furthermore, the analysis shows if the micro frontend implementations considered the usage of other frameworks.

From the data we can see that two participants chose React with create-react-app as their app starter, three users chose Angular with Angular-CLI to start their project, and one chose Svelte with Vite's Svelte template. In each of the implementations of the single page applications users responded that they chose a language they have used in their daily work and chose popular starters instead of using custom templates for their project as can be seen in Figure 21. This was mentioned as one of the benefits of the single page applications when compared to micro frontends. The simple setup process for single page applications relates to the low difficulty score in Table 6.

An overview of the tool used to implement micro frontend implementation is displayed in Table 7.

Subject	Micro frontend implementation tool
Participant 1	Custom orchestration
Participant 2	Module Federation
Participant 3	Custom orchestration
Participant 4	Single-spa
Participant 5	Single-spa
Participant 6	-

Table 7. Micro frontend implementation choices among participants

Participant 1 used a custom solution using Svelte.

Participant 2 chose module federation from Webpack[40], a tool that allows custom orchestration and can be used to built micro frontends.

Participant 3 used another custom setup with the smartly/micro-frontend-starter template[59].

When it comes to the choices of micro-frontend implementation participant 4 and participant 5 were the only ones to choose a framework for setting it up. The framework used was single-spa[45].

What can be seen across all participants the only choices which conform to the ability to use different frameworks for each micro frontend are the single-spa implementations. While the participants have answered that one of the benefits of the micro frontends is the "*freedom of choice*" (P2, P3, P4) the implementations don't reflect that. Another line that can be drawn from the data is that the participants with more experience tended to choose the more custom solutions and the ones with less experience tend to choose frameworks even though they still choose set frameworks for their single page applications. This is related to the difficulty of setting up the micro frontends and the infancy of the micro frontend framework. But even when

choosing a set framework for building the micro frontend the score about the difficulty remains high among participants 4 and 5. The micro frontends of all implementations used pieces of the code built from the SPA. This is also what can be seen in the industry where [47][50] say that the implementation usually uses the same framework, usually React for all the micro frontend pieces. In conclusion, the participants use proved frameworks for building single page applications but the tendency is different in the micro frontend implementations with users going with more custom solutions.

Implementation

Participants were asked to submit a PDF file or a power point presentation of their implementation. Examples of document submissions are displayed in Figure 26, 27 and 28.

For the spa architecture

Used create-react-app

The styles are used with bootstrap cdn and incorporated in the public html file
Each functionality is under a separate folder -> overview, sales etc
Each functionality was split into their own component and each piece of repeating html was componentized.
The rest api call was made under a generic function that can fetch the different data
These were put all under different routes.

react-scripts do starting locally, building and testing

Main issues:
- Need to add router and change routing logic
- Html to react is not 100% straightforward
- Componentizing is limited if you are using bootstrap as css
- Routing works differently regarding the path matching

Micro frontend

smartlyio/micro-frontend-starter was used from smartly.io github
Since I did the spa first the transition was easy
Each of the folders to the spa moved in a separate project which required its own tsconfig and package.json to manage the dependencies
The overview was moved to the main project that is used to aggregate all the pieces
It uses a monorepo approach. So code reuse is possible by relative folders.
All projects use react,
There is a main project called workbench which has a router that will render each project by lazy loading the main component of that into a route
There are ready made scripts for starting and building the project. There is a lot of complexity there so I didn't get to experiment with those.

Main issues:
- build script doesn't run
- testing is hard to setup
- difficult setup in the webpack setups for local and production environments.
- dependencies are old
- feels very custom made and not sure if I would use this as judging from the dependencies is not very actively maintained

Good stuff:
- easy to move away. No need to code anything extra
- the structure is easy

Figure 26. Participant 3 submitted document

For SPA, I used Angular framework. I initialized the app using Angular CLI. I used Angular built-in routing, server communication using Angular HTTP API. It didn't have any
challange that required extra time thinking, and it was very straightforward process.

I created the micro-frontend application using single-spa framework. I heard it was the most popular framework among others and the documentation of the framework covered details information how to setup the application using different spa frameworks. Angular framework was the ecostem from which micro apps were created. To initialize the application I used ng `new admin-dashboard --routing --prefix admin-dashboard`. To create add Angular microfrontends I used the command ng add single-spa-angular. Each of the pages were created as separate microfrontends: visitors, subscribers, sales and orders.

Figure 27. Participant 4 submitted document

Fist of all i crated the react app using yarn create-app with typescirpt
After i design the structure how it should look like, the app is separated in components.
In the fist folder i kept the main structure of the html in files .After i created a foleder called DATA-TABLE there I separated the tables with
the datas for each functionality (Ex. Sales-table-data). After i implemented in the main
structures for each component.

Figure 28. Participant 6 submitted document

5. CONCLUSIONS

The study took a peek into traditional frontend development underlining the issues that it faces and gave an introduction to what can be the solution to those issues, micro frontends. To analyze how micro frontend is perceived, an experimental study was designed and conducted by 6 developers with different levels of experience ranging from 9 months of experience to +10 years of experience. The experiment consisted of a simple MVP frontend application to be developed in two different approaches, single page application, and micro frontend. Participants tracked the time of delivering each of the approaches as well as their opinions about the topic from which the data was later analyzed.

This chapter presents the findings and covers the limitations of this study. Finally, it peeks into future work.

5.1. Summary of Results

This section gives a summary of the results found in the discussion section.

One of the results that come from this study is that the experience mattered when developing single page applications and had a correlation based on the gathered data. It showed that the participants tend to take less time planning and implementing single page applications when they have more experience. On the micro frontend counterpart, the data is varied and doesn't show a trend on it as different levels of experience didn't impact the time. A more visible factor was exposure to working with micro frontends before. It had a lowering effect on the time it takes to develop the micro frontend solution. Another point is that users have a perception of single page applications being simple even when the experience on it is less than one year. The maximum difficulty score on a scale of one to five was two. While it showed that the micro frontend approach is perceived as difficult with only one participant with vast experience scored it with two and another which uses it in their daily life with three and other scoring it with four and five. This result tied to the lack of tooling in creating micro frontend. Even though the perceived difficulty remains high the results show that micro frontends are the choice for big-sized projects and even the participants who couldn't complete the experiment were positive about using micro frontends for such projects and ready to jump the difficulty gap. While for medium and small-sized project all the developers stated that they would choose single page applications. The main advantages of the single page applications and micro frontends were similar to findings in the research section with participants listing the simplicity of setup, better tooling currently, and that the approach would fit for smaller and medium-sized scopes. The micro frontends were seen as advantageous when it came to bigger scale projects, freedom of language and framework choices, better team composition, better deployment control, and more maintainable. The most prominent issue was the lack of tooling for the micro frontends which was a visible result in this experiment. It was also interesting to see that the results showed that migrating from the single page application to the micro frontend approach was a common approach in the experiment. In summary, the results showed that developers would only use the micro frontends if they knew forehand that the project size would be big and that multiple teams would

be required, developers perceive the micro frontends as harder to approach, previous experience in micro frontends is an important factor while the development experience doesn't always help with this and that the tooling for setting up micro frontends is still not very mature.

5.2. Limitations

This section describes the limitations of the work. It takes into account the issues around the sample size, sample selection, method, and data collection.

One of the limitations of this study is the relatively low number of participants. The experiment was executed by six developers. The low number was due to circumstances such as time for the development of this study, unwillingness to participate, and the fact that the experiment would require studying and going out of the comfort zone for the sample group. The selection of the sample was partially random and the experiment was sent to student groups asking for their interest to participate but there was no response. The other sample group was direct coworkers and people which the author knew personally. While the selection wasn't random in that part, it proved that the sample group had different levels of experience, ranging from a person with less than one year of experience to persons with ten or more years. The experience in micro frontend development was also spread well in the group, where two of the developers had previously used it in their work and the others had not worked with micro frontends before. The method selection was also limited due to the small sample size anticipation and time. The paper incorporates a combination of quantitative and qualitative research. Due to the small sample size, a more thorough level of experimenting was required but with fewer data. This is one of the more limiting factors which decreases the significance of the data. Data collection was also a combination of the two methods. With participants clocking the times it took for experiment development and answering free-text answers, a part of the burden of interpreting free text or interview answers rests on the author and these may not always have an objective interpretation. Finally a clear discrepancy is the size of the experiment which does not necessarily reflect the reality. In reality software projects can last thousands of hours and trying to minimize that to an experiment is difficult and not fully reliable.

5.3. Future Work

This section summarizes key ideas as how this study could be developed further in the future.

A key point about the future work would be a better sample size to give significant data and adding more objectivity. That would also help shape the questionnaire that the participants required. A bigger size sample and a more controlled sample would help analyze the micro frontend tools of choice and shine a light as to what are the most common tools and approaches developers choose when it comes to micro frontends. A further point to improve the study would be to have a more in depth interview or questionnaire about the user perception towards micro frontends and

single page applications, rather than having to analyze these results mainly from the implementation details.

6. REFERENCES

[1] Tilak P.Y., Yadav V., Dharmendra S.D. & Bolloju N. (2020) A platform for enhancing application developer productivity using microservices and micro-frontends. In: 2020 IEEE-HYDCON, IEEE, pp. 1–4.

[2] Hamerník B.M. (2020) Development of modern user interfaces in angular framework .

[3] Microfrontends - extending the microservice idea to frontend development. https://micro-frontends.org/. (Accessed on 02/29/2020).

[4] Fritzsch J., Bogner J., Zimmermann A. & Wagner S. (2018) From monolith to microservices: a classification of refactoring approaches. In: International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment, Springer, pp. 128–141.

[5] Chen R., Li S. & Li Z. (2017) From monolith to microservices: A dataflow-driven approach. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), IEEE, pp. 466–475.

[6] Escobar D., Cárdenas D., Amarillo R., Castro E., Garcés K., Parra C. & Casallas R. (2016) Towards the understanding and evolution of monolithic applications as microservices. In: 2016 XLII Latin American Computing Conference (CLEI), IEEE, pp. 1–11.

[7] Poccia D. (2016) AWS Lambda in Action: Event-driven serverless applications. Manning Publications Co.

[8] Serverless architectures. https://martinfowler.com/articles/serverless.html. (Accessed on 03/29/2020).

[9] Serverless architecture the future of business computing. https://marutitech.com/serverless-architecture-business-computing/. (Accessed on 03/29/2020).

[10] Bila N., Dettori P., Kanso A., Watanabe Y. & Youssef A. (2017) Leveraging the serverless architecture for securing linux containers. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), IEEE, pp. 401–404.

[11] Adam Polak A.Z. (2019), Microservices design patterns: Api gateway, backend for frontend (bff) | tsh.io. https://tsh.io/blog/design-patterns-in-microservices-api-gateway-bff-and-more/. (Accessed on 04/02/2021).

[12] Sandoval K. (2017), Building a backend for frontend (bff) for your microservices | nordic apis |. https://nordicapis.com/building-a-backend-for-frontend-shim-for-your-microservices/. (Accessed on 04/02/2021).

[13] Wickramarachchi V. (2021), The bff pattern (backend for frontend): An introduction | by viduni wickramarachchi | feb, 2021 | bits and pieces. https://blog.bitsrc.io/bff-pattern-backend-for-frontend-an-introduction-e4fa965128bf. (Accessed on 04/02/2021).

[14] A recap of frontend development in 2019 - level up coding. https://levelup.gitconnected.com/a-recap-of-frontend-development-in-2019-1e7d07966d6c. (Accessed on 03/29/2020).

[15] Wanyoike M. (2018), History of front-end frameworks - logrocket blog. https://blog.logrocket.com/history-of-frontend-frameworks/. (Accessed on 04/04/2021).

[16] Ajax (programming) - wikipedia. https://en.wikipedia.org/wiki/Ajax_(programming). (Accessed on 04/04/2021).

[17] Jaman S. (2019), Everything you need to know about “single-page-application” | by shifat jaman | medium. https://shifat-jaman.medium.com/single-page-application-everything-you-need-to-know-6f00d87e5130#:~:text=It%20was%20introduced%20to%20the,handling%20the%20complicated%20User%20Interfaces. (Accessed on 04/04/2021).

[18] Pośliński D. (2020), Microfrontends to the rescue of big single page application monoliths. https://selleo.com/blog/microfrontends-to-the-rescue-of-big-single-page-application-monoliths. (Accessed on 03/20/2021).

[19] 10 best javascript frameworks to use in 2021. https://hackr.io/blog/best-javascript-frameworks. (Accessed on 05/23/2021).

[20] Angular. https://angular.io/. (Accessed on 04/02/2021).

[21] React - a javascript library for building user interfaces. https://reactjs.org. (Accessed on 04/20/2021).

[22] Vue.js. https://vuejs.org/. (Accessed on 04/02/2021).

[23] Ember.js - a framework for ambitious web developers. https://emberjs.com. (Accessed on 04/27/2021).

[24] Stack overflow trends. https://insights.stackoverflow.com/trends?tags=reactjs%2Cangular%2Cvue.js%2Cember.js. (Accessed on 04/04/2021).

[25] Choosing the best front-end framework | toptal. https://www.toptal.com/javascript/choosing-best-front-end-framework. (Accessed on 06/07/2021).

[26] Likness J. (2014), Model-view-viewmodel (mvvm) explained. https://www.wintellect.com/model-view-viewmodel-mvvm-explained/. (Accessed on 04/02/2021).

[27] Ng R. (2019), Patterns for javascript frontend applications | by richard ng | cloudboost. https://blog.cloudboost.io/the-state-of-web-applications-3f789a18b810. (Accessed on 04/02/2021).

[28] Fluxxor - what is flux? http://fluxxor.com/what-is-flux.html. (Accessed on 06/07/2021).

[29] Flux and redux. if you are working closely with… | by sidath asiri | medium. https://medium.com/@sidathasiri/flux-and-redux-f6c9560997d7. (Accessed on 05/25/2021).

[30] Micro frontends: The microservices approach to web app development. https://insights.daffodilsw.com/blog/micro-frontends-the-microservices-approach-to-web-app-development. (Accessed on 03/30/2020).

[31] Microfrontends: The benefits of microservices for client-side development - the new stack. https://thenewstack.io/microfrontends-the-benefits-of-microservices-for-client-side-development/. (Accessed on 03/29/2020).

[32] Micro frontends | technology radar | thoughtworks. https://www.thoughtworks.com/radar/techniques/micro-frontends. (Accessed on 02/29/2020).

[33] Wang D., Yang D., Zhou H., Wang Y., Hong D., Dong Q. & Song S. (2020) A novel application of educational management information system based on micro frontends. Procedia Computer Science 176, pp. 1567–1576.

[34] Yang C., Liu C. & Su Z. (2019) Research and application of micro frontends. In: IOP Conference Series: Materials Science and Engineering, vol. 490, IOP Publishing, vol. 490, p. 062082.

[35] What is a micro frontend? | packt hub. https://hub.packtpub.com/what-micro-frontend/. (Accessed on 04/04/2021).

[36] Fowler M. (2003) Who needs an architect? IEEE SOFTWARE 20, pp. 11–13.

[37] Software architecture & design introduction - tutorialspoint. https://www.tutorialspoint.com/software_architecture_design/introduction.htm. (Accessed on 04/20/2021).

[38] May I. (2011) Systems and software engineering–architecture description. Tech. rep., Technical report, ISO/IEC/IEEE 42010, 2011.(Cited on page 12.).

[39] Micro frontends · docs. https://docs.bit.dev/docs/workflows/microfrontends. (Accessed on 04/05/2021).

[40] Module federation | webpack. https://webpack.js.org/concepts/module-federation/. (Accessed on 04/05/2021).

[41] Module federation architecture | rangle.io. https://rangle.io/blog/module-federation-federated-application-architectures/. (Accessed on 06/07/2021).

[42] Micro frontend deep dive - top 10 frameworks to know about - cuelogic technologies pvt. ltd. https://www.cuelogic.com/blog/micro-frontends-frameworks. (Accessed on 03/21/2021).

[43] What is a web portal? | liferay. https://www.liferay.com/resources/l/web-portal. (Accessed on 05/04/2021).

[44] Piral - portal solutions using microfrontends. https://piral.io/. (Accessed on 05/04/2021).

[45] Micro frontends - what is it and how to use. https://appsoft.pro/microservice-approach-in-web-development-micro-frontends/. (Accessed on 05/04/2021).

[46] Project mosaic—frontend microservices. https://www.mosaic9.org/. (Accessed on 04/18/2021).

[47] Colin J. (18), Front-end micro services. https://engineering.zalando.com/posts/2018/12/front-end-micro-services.html. (Accessed on 04/18/2021).

[48] In-demand talent on demand.™ upwork is how.™. https://www.upwork.com/. (Accessed on 04/02/2021).

[49] About us - upwork. https://www.upwork.com/about/. (Accessed on 03/29/2020).

[50] Momot E. (2020), Modernizing upwork with micro frontends | by eduard momot | upwork engineering | medium. https://medium.com/upwork-engineering/modernizing-upwork-with-micro-frontends-d5be5ec1d9a. (Accessed on 04/20/2021).

[51] Momot E. (2020), Upwork modernization: An overview | by eduard momot | upwork engineering | medium. https://medium.com/upwork-engineering/upwork-modernization-an-overview-92ac3403b63f. (Accessed on 04/20/2021).

[52] #1 meal kit delivery service | fresh meal delivery | hellofresh. https://www.hellofresh.com/. (Accessed on 04/02/2021).

[53] Hellofresh - who we are. https://www.hellofreshgroup.com/websites/hellofresh/English/100/investor-relations.html. (Accessed on 03/29/2020).

[54] De000a161408-q3-2019-eq-e-00.pdf. https://www.hellofreshgroup.com/download/companies/hellofresh/Quarterly%20Reports/DE000A161408-Q3-2019-EQ-E-00.pdf. (Accessed on 03/29/2020).

[55] Front-end microservices at hellofresh | by pepijn senders | hellotech. https://engineering.hellofresh.com/front-end-microservices-at-hellofresh-23978a611b87. (Accessed on 06/07/2021).

[56] Jonada1/admin-dashboard. https://github.com/Jonada1/Admin-Dashboard. (Accessed on 05/25/2021).

[57] typicode/json-server: Get a full fake rest api with zero coding in less than 30 seconds (seriously). https://github.com/typicode/json-server. (Accessed on 05/25/2021).

[58] Facebook, Create react app. https://create-react-app.dev/. (Accessed on 05/06/2021).

[59] Kivelä M., smartlyio/micro-frontend-starter: Micro frontend monorepo example setup using lerna, typescript and react. https://github.com/smartlyio/micro-frontend-starter. (Accessed on 05/15/2021).