Son Bui

Micro frontend : Microservice implementation on Web Development

Metropolia University of Applied Sciences

Bachelor of Engineering

Name of the Degree Programme

Bachelor's Thesis

1 November 2021

Abstract

Author:	Son Bui
Title:	Title of the Thesis
Number of Pages:	52 pages
Date:	1 November 2021
Degree:	Bachelor of Engineering
Degree Programme:	Degree Programme in Information Technology
Professional Major:	Mobile Solutions
Instructors:	Janne Salonen, Supervisor (e.g. Project Manager)

With the increasing relevancy of microservice architecture style in software development, this paper aims to examine the applicability of utilizing microservice architecture style for frontend development, or micro frontend. Through going over an array of literature concerning frontend development and microservice architecture, this paper aims to document the definitions along with various advantages and disadvantages of micro frontend against monolithic architecture style, thereby trying to establish a best practice concerning when to adopt micro frontend architecture and when to migrate a monolithic to micro frontend.
Moreover, this paper is also accompanied by an example project, which is a pseudo social media page, with the aim of demonstrating the migration process using the single-spa package, migrating from React framework to micro frontend system utilizing both React and Framework for its multiple micro applications.
Conclusively, the paper is able to draw out the differences between the two architectures with the micro frontend proving to be significantly beneficial when applications growing into later development cycles. The example project also succeeds in documenting and demonstrating the migration process, albeit the final micro frontend system not a complete replica of the original due to limitation of documentation availability.
Overall, micro frontend is still a relatively new concept and should be explored further since it advantages are of meaningful significance for large scale application.

Keywords: Frontend Architecture, micro frontend, Microservice

Contents

List of Abbreviations

AJAX:	Asynchronous JavaScript and XML
API:	Application Programming Interface
CSS:	Cascading Style Sheet.
HTML:	Hyper Text Markup Language.
HTTP:	Hypertext Transfer Protocol
HTTPS:	Hypertext Transfer Protocol Secure
MA :	Monolithic Application
MSA :	Microservice Application
npm:	Node Package Manager
SEO:	Search engine optimization
SPA:	Single Page Application
W3C:	World Wide Web Consortium
XHTML:	Extensible Hyper Text Markup Language
XML:	Extensible Markup Language

1 Introduction

In recent years, there has been a growing trend moving towards replacing monolithic model with new microservice architecture among considerable sized tech corporations, which include Netflix (Mauro, 2015), LinkedIn (Ihde and Parikh, 2015), and Amazon (Kramer, 2011). This is understandable considering the diverse benefits provided to companies who possess large scale operations that require highly flexible and easily maintained solutions (Nadareishvili *et al.*, 2015). Following the emergence of microservice design principles, the term micro frontend starts to develop as an alternative structure for web application on the client side.

Micro frontend, or the adaptation of microservice for frontend development, emerges with the aim of taking advantage of the strengths of microservice. In this method, typical monolithic application is segmented into separate components that can be programmed and deployed almost independently by decompartmentalized teams (Jackson, 2019). With the current high number of cloud application which still employs the more traditional model, there is a growing demand for migrating from the monolithic to micro frontend architecture design pattern.

This thesis paper's aim is to explore the concept of microservice, micro frontend, and the implementation of microservice on frontend development. The paper is also accompanied by a project which demonstrates the method of Micro frontend migration and make comparisons on these architecture model based on the development process. Through the outcome of the comparison, this paper will strive to determine the best practices when deciding on initialize a micro frontend migration or start the process of building a micro frontend application.

2 Cloud Application Architecture Overview

This chapter will elaborate on the concepts of microservices and its counterparts as monolithic, their distinguished attributes, and advantages and disadvantages within the context of software development. Thereafter, the adopt of these concepts onto frontend development will be further expanded upon, whereas its architecture design pattern will be fully elaborated, and its array of beneficial and potentially detrimental aspects will be analysed.

2.1 Definition of Monolithic Architecture

By definition, a monolithic software application is one wherein all its functional codes are confined within a single application, code base, repository, or combinations thereof (Villamizar *et al.*, 2015). It is the typical architecture that almost all programs start out as. Even though components within the monolithic may be compartmentalized in a modular fashion, they are still regarded as a single program where they are packaged, tested, or deployed as a single entity. The purposes of these components are diverse and specific to each application, be it authorization, presentation on the web frontend, business logic of the application, database for accessing and writing data, or application integration with other services. (*Introduction to Monolithic Architecture and MicroServices Architecture* | *by Siraj ul Haq* | *KoderLabs* | *Medium*)

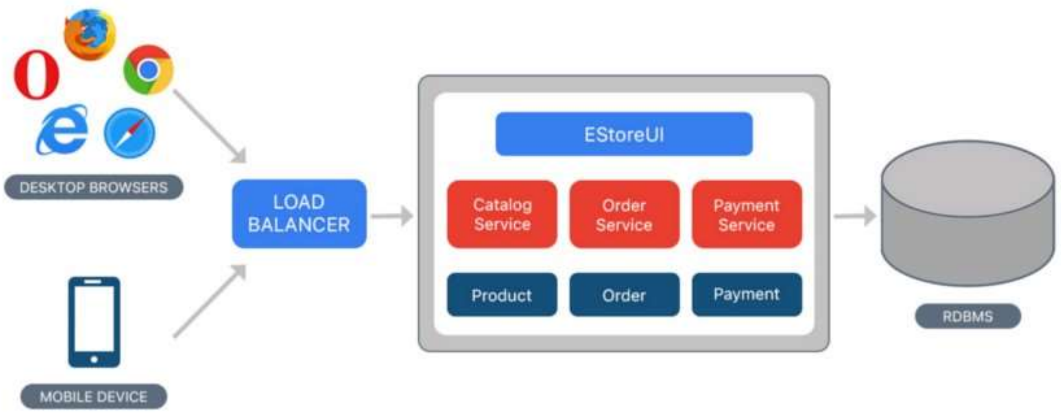


Figure 1. Example architecture of a monolithic application. (Introduction to Monolithic Architecture and MicroServices Architecture | by Siraj ul Haq | KoderLabs | Medium)

Figure 1 shows an example of a monolithic application. Despite the fact that multiple services, components, and modules are utilized, the application is deployed as one single entity providing services to multiple platforms - desktop browsers and mobile devices in this case - using one single RDBMS database.

2.2 Monolithic: The traditional way

In a general term, monolithic architecture is the more traditional design pattern for building application. When a monolithic application is still small; development, testing, deployment, and scaling are all relatively straightforward (Bogner *et al.*, 2019). There are a few beneficial characteristics of the monolithic architecture that renders it still a viable option as a design pattern, with most of these are derived from the fact that the codes are all contained in a single code base (*Monolithic & Microservices Architecture* | *by Henrique Siebert Domareski* | *Medium*, no date):

- Simplicity: All the codes of the application are contained within a single repository. This renders the process of locally running the application or changing the code simple, which allows for fast and comfortable development experience.
- Easy to deploy**:** Since all the codes are confined within a single project, it can be deployed in a simplistic manner. This is also quite convenient with each of the added features or fixed bugs only require a single deployment.
- Familiarity**:** Because almost all developers have already substantial experience working in a monolith environment, the onboarding process is more straightforward, and it is relatively quicker to get started contributing to the project, provided that the size of the code base is still small.
- Easy for monitoring**:** Since there is only one project, there is only the need to monitor the single application. Should undesirable errors occur, there is only one repository for the error to exist in.
- Easy to debug**:** With all the codes already enclosed within the scope of one single monolithic solution, there is only the need to run the application locally and focus on identifying the faulty code without any further configurations.
- Easy to test**:** With the whole monolithic application in one single location, it’s easy to perform tests without any need to set up the communication applications or perform checks on the operating status of any other applications, inter alia. Implementation of end-to-

end testing in this design pattern is also effortless without testing tools like Selenium or Cypress.

- The upsides of the monolithic design pattern allow for quick and seamless development experience.

Unfortunately, the benefits of the monolithic will gradually subside as these applications grow larger and complicated; this is a direct result of the sheer size of the application impeding and slowing down development due to longer start up time, longer testing time, longer deploy time, inter alia. Code incomprehensibility and redundant sophistication put forth a stranglehold on bug patching and feature implementations (Chen, Li and Li, 2018).

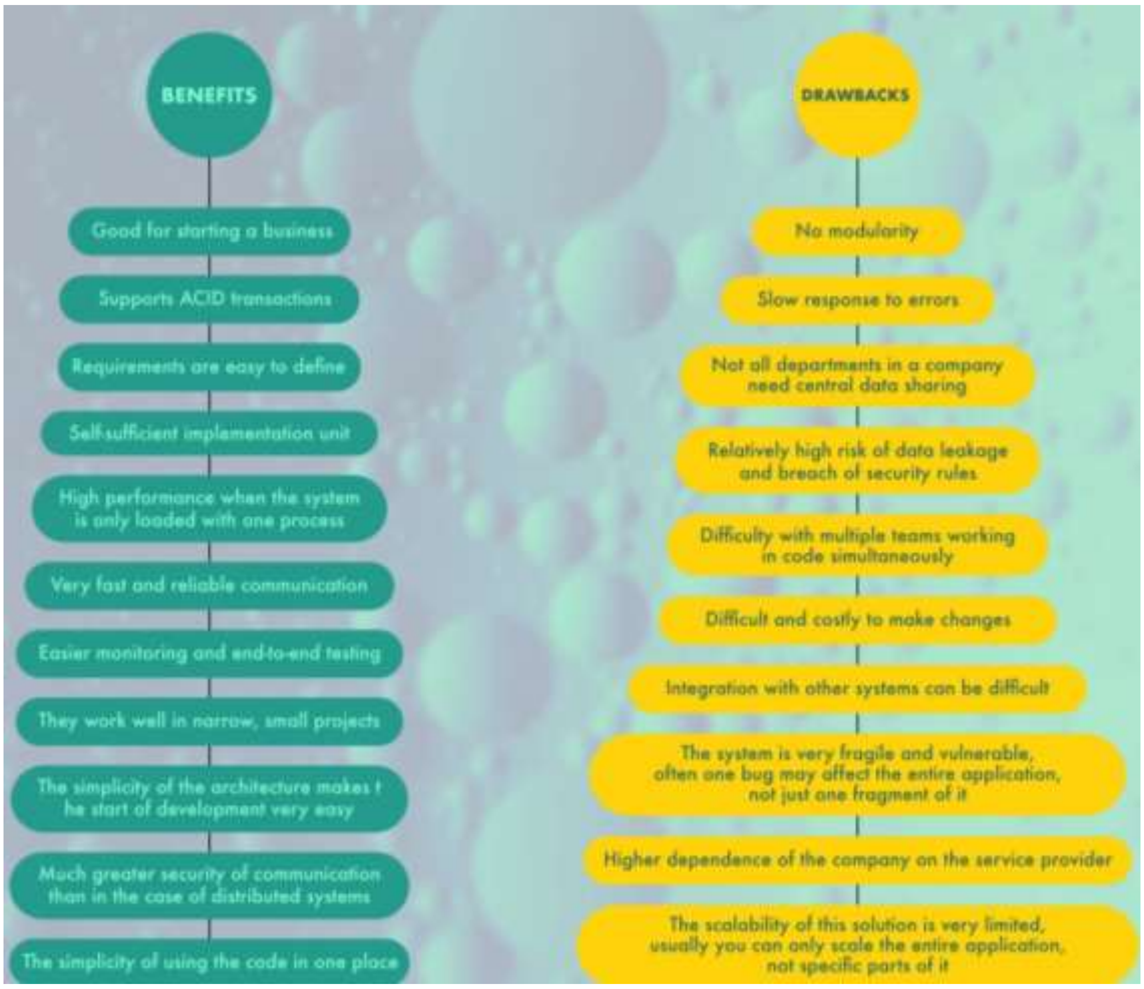

BENEFITS
Good for starting a business
Supports ACID transactions
Requirements are easy to define
Self-sufficient implementation unit
High performance when the system is only loaded with one process
Very fast and reliable communication
Easier monitoring and end-to-end testing
They work well in narrow, small projects
The simplicity of the architecture makes t he start of development very easy
Much greater security of communication than in the case of distributed systems
The simplicity of using the code in one place
DRAWBACKS
No modularity
Slow response to errors
Not all departments in a company need central data sharing
Relatively high risk of data leakage and breach of security rules
Difficulty with multiple teams working in code simultaneously
Difficult and costly to make changes
Integration with other systems can be difficult
The system is very fragile and vulnerable, often one bug may affect the entire application, not just one fragment of it
Higher dependence of the company on the service provider
The scalability of this solution is very limited, usually you can only scale the entire application, not specific parts of it

Figure 2. Monolithic architecture Benefits and Drawbacks. (J. Lewis and M. Fowler, 2014)

Figure 2 as above listed some of the benefits and drawbacks of the monolith application. Within the context of this paper, some of the most outstanding disadvantages with such a large and complex monolithic application are to be accentuated on:

- Centralized code base: Since there is only one centralized application, the technology choses is primarily focused a one-size-fit-all situation instead of the best solution for the requirements, which means unoptimized codes and underperformed efficiency. Different teams are also enforced with the same technology, rendering API and system integration challenging.
- Prone to crash and expensive to repair: One small error in the application can impede the working of the application or even crash it. This cascading effect is typically due to the interconnectedness of components within the application. Though it is possible to pinpoint the error to the latest update or patch, it usually takes a considerable amount of time to identify and fix these bugs due to the size and entanglement of the monolithic code base.
- Susceptible to external attacks: Since the services within the monolithic applications are not fully decompartmentalized from each other, it is possible for external attacks to gain access to the whole system from a data leak.
- Low scalability and lack of flexibility to changes: These are ones of the major shortcomings of monolithic architecture and those rigidities are a decisive factor for microservice migration. Since the many elements within the monolithic application system is deeply interwoven and integrated into each other, it is of insurmountable difficulty to switch one part of the system to a new technology due to the fact that figuring out all dependencies and changing them can be time-consuming. Scaling with the application is also harder since the whole monolith needs to scale at the same time, even though each of the service within the monolith may need to scale differently due to high deviations in uses.
- Slow deploying time and slow onboarding time: Another problem with such large code base is long deployment time. Even with minimal changes confided within one or two files in the repository, the whole monolith deployment needs to be re-enacted and all the tests need the be run again, taking away development time. In addition, another issue might rise is large code base will slow down onboarding time for new developers to join and contribute to the project since it takes substantial amount of time and effort to gain in-depth understanding of the code base.
- Low customization: Since the monolithic is one single application entity, once it scales up all of its components are duplicated across the many servers thereby impeding the application capacity to adapt

to each of the server’s need. With microservice, since the services are separated, they can be adopted based on the demands of each of the servers, thus increasing customizability.

Conclusively, monolithic architecture is appropriate for certain situations, which can be smaller applications, development is still the early stage, requirements are not yet clearly defined, or development speed is of a higher priority than scaling. For larger and more complex application, where its size and complexity reach a point where the disadvantages of maintaining a monolith outgrow its benefits, microservice becomes a more applicable option for application architecture. (*Monoliths vs. microservices — benefits and drawbacks [a comparison]* | *by Transparent Data* | *Blog Transparent Data ENG* | *Medium*, no date)

2.3 Microservice: Definition

Even though there has not been a consensus on definition for microservice (Nadareishvili *et al.*, 2015), one of the most referenced definition for microservice is to imply the design pattern whereas an application is developed through an array of smaller scale services. Each of these services has their own business capability, logic, message-based communication mechanism, and is entirely capable of independent deployment; together, these services become the basic components that form the entirety of the microservice architecture (Nadareishvili *et al.*, 2015). These microservices can even utilize different languages, frameworks, database technology as long as they are proven sufficient for the requirements (J. Lewis and M. Fowler, 2014).

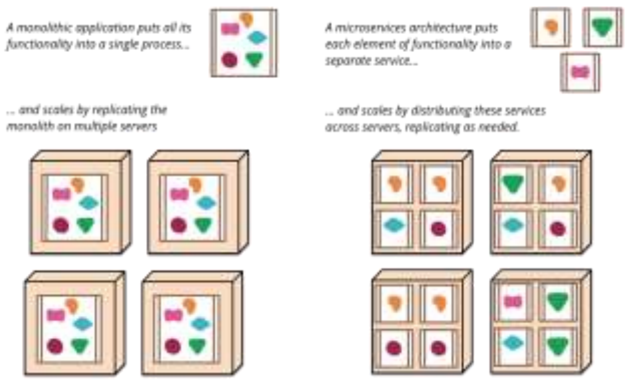

A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

Figure 3. Monoliths and Microservices. (J. Lewis and M. Fowler, 2014)

Figure 3 showcases some of the basic differences between monolithic and microservice design pattern. While monolithic application has all its functions contained within one single unit, microservice segregate each of its functions into separate units (services) and distribute these services as per requirements. The figure also implies the shortcomings that monolithic structures might encounter with upscaling (cf. subchapter 2.2).

2.4 Hallmarks of a microservice system

There are quite a considerable number of characteristics which may have contributed to microservice system's popularity. Due to the structural design of microservice, these advantages are attained at a higher degree (Newman, 2015). This section will list out some of the key aspects of microservice:

- Diversity of technologies: Due to the segregation of the functions of the services, it is easier to utilize different technology solutions for different problems in microservices (Newman, 2015). The size of these services is also considerably small, which allows for smooth replacement or even depletion should the need arises (De Lauretis, 2019). Microservice design pattern allows teams to adopt new technology stack with more desirable performance or higher efficiency. This is not without risk when multiple technologies are involved, but in a microservice system, mitigating and limit the potential the damage to the entire system is more manageable (Newman, 2015).
- Resilience: Since each of the services is confined, the failure of one unit should not affect the whole system. This is one apparent benefit of microservice over monolithic structure; one deficit in one function can collapse the integrity of the monolithic system. (Newman, 2015)
- Scalability: In monolithic architecture, all components in the application needs to be scaled together. In smaller sized service, it is possible to scale each of these services on machines appropriate for the performance and demands of each service. (Newman, 2015)
- Deployment ease: With the architecture design of microservice, each microservice is deployed individually, which is inherently faster and more lightweight than typical monolithic structure, where each deployment requires the whole application to be redeployed. Should problems arise, it is also less burdensome to isolate and rollback the faulty changes in a microservice design. (Newman, 2015)
- Replaceability: One of the most important aspects of microservice architecture design lies within its capacity to adapt to changes

(Nadareishvili *et al.*, 2015). With each service unit deployed independently, replacing them with better and appropriate tools and designs are more streamlined. It is easier for companies to adopt technologies in faster and more modular ways.

In short, microservice is an architecture design pattern that incentivizes small changes. Apparently, with all these advantages that microservice provides, there exist a wide array of disadvantages which companies and organizations need to account in when making architecture decision.

- Architecture complexity: While monolithic testing, continuous integration/continuous deployment, and deployment can be streamlined, it is not so with microservice. With services segmented down in sizes and scales, the architecture structure of the application tends to grow up in complexity, where each of these service units needs different testing and deploying handling.
- Overhead cost: In addition to the daunting application complexity, the diverged code base, diverse frameworks, multiple languages, and multitude of repository adds extra layers of burden upon maintenance. This renders microservice architecture style quite an considerably expensive system due to elevated maintenance cost (Nadareishvili *et al.*, 2015). This is something companies might need to take into consideration when adopting microservice architecture as the benefits may not be worth the cost imposed on the companies.

3 Frontend development Background in relevancy to Microservice

Since this paper primarily concerns adopting microservice into frontend development, this section will briefly cover some basics of frontend development and some frequently used concepts. Moreover, some of the basic concepts will be elaborated upon

3.1 Frontend development: An overview

As a concept, frontend development refers to the development of websites and applications on modern web platform where users can view and interact with, utilizing pure Hyperlink Text Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript or web application frameworks such as React, Vue, or Angular (*What Is a Front-End Developer? · Front-End Developer Handbook 2018*, 2018).

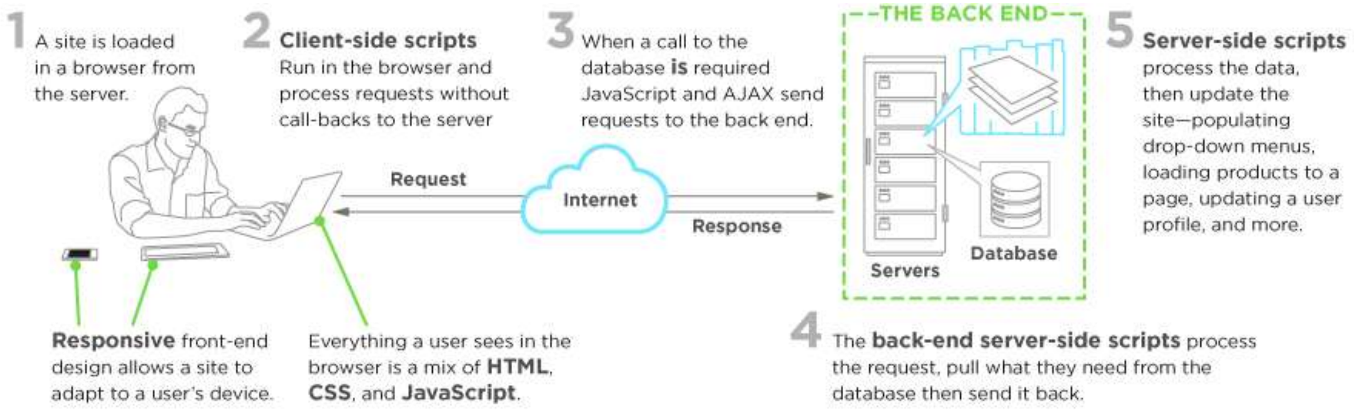

1 A site is loaded in a browser from the server.
2 Client-side scripts Run in the browser and process requests without call-backs to the server
3 When a call to the database is required JavaScript and AJAX send requests to the back end.
THE BACK END
5 Server-side scripts process the data, then update the site—populating drop-down menus, loading products to a page, updating a user profile, and more.
Request
Internet
Response
Servers
Database
Responsive front-end design allows a site to adapt to a user's device.
Everything a user sees in the browser is a mix of HTML, CSS, and JavaScript.
4 The back-end server-side scripts process the request, pull what they need from the database then send it back.

Figure 4. Simplified web application process (What Is a Front-End Developer? · Front-End Developer Handbook 2018, 2018)

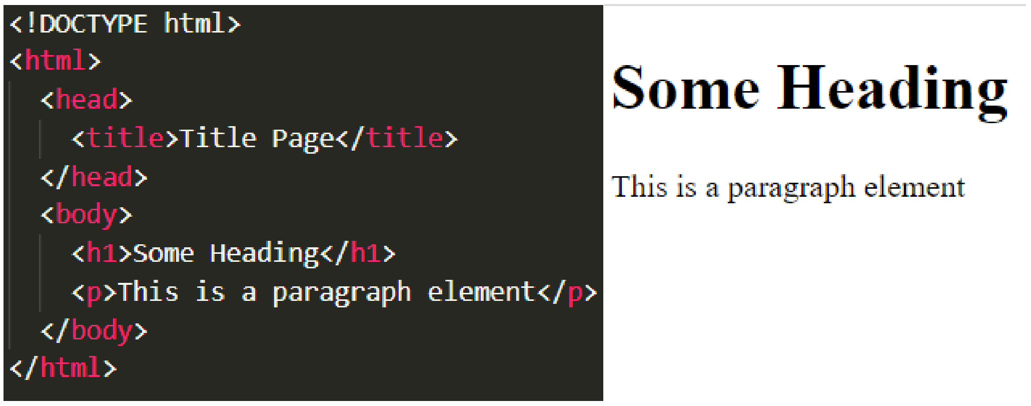
As illustrated in figure 4, the process of modern web applications' working with the frontend (the client-side scripts) and the backend (server-side scripts). The process starts with users browsing through the application contents and interacting with its graphical interfaces via mouse actions (clicking, scrolling, etc) or keyboard actions (typing, pressing certain keyboard buttons, and so on). The web application then takes in the inputs from the users, packages those inputs into data, renders the data into requests and send those requests to the server backend(s) through the internet. The backend server then processes these requests, either running some queries to the database or run some logics with the request, and then send responses to the client-side frontend with the requested data. The frontend then receives these responses, consumes the data packages, and thereafter, update the users' interface in accordance with the server responses.

Frontend development revolves primarily around HTML, CSS, and JavaScript. It typically employs different frameworks, such as React, Vue, or Angular, for ease of development. These concepts and frameworks that are employed in this paper are to be explored in the next sections.

3.2 Concepts and definitions

Listed below are some of the fundamental concepts required for the understanding of the frontend development and software development process in general:

- HTML, acronymized for Hyper Text Markup Language, is the standardized markup language used for composing webpages and displayed on a web browser. HTML composes of various elements, called tags, which form the structure of the web page, the content to displays, and instruct the web browser to render the page. (Introduction to HTML)



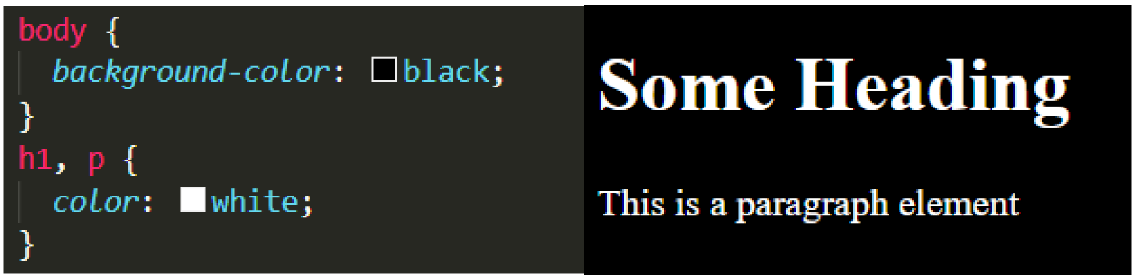
```html
<!DOCTYPE html>
<html>
  <head>
    <title>Title Page</title>
  </head>
  <body>
    <h1>Some Heading</h1>
    <p>This is a paragraph element</p>
  </body>
</html>
```

Some Heading

This is a paragraph element

Figure 5. A simple HTML file with the title element of “Title Page”, a heading element with “Some Heading”, and a paragraph with content “This is a paragraph element”. The resulted rendered can be seen on the right image.

- CSS, short for Cascading Style Sheet, is considered as the most essential style languages used in modern web development and one of the three technologies contributing to the modern web development. CSS’s role is to dictate the visual facet of the HTML elements, including but not limited to background colors, font styles, font sizes, or positions within the web page.



```css
body {
  background-color: black;
}
h1, p {
  color: white;
}
```

Some Heading

This is a paragraph element

Figure 6. A simple CSS file and its effect on previous HTML example.

- DOM, or Document Object Model, is a programming interface for representing and interacting with objects in web documents, which include HTML, XML (extensible markup language), or XHTML (extensible hypertext markup language). It is designed to be cross-platform and language-independent, rendering the representation of the web document as an application programming interface (API). The web page document and its elements are defined within a tree structured called the DOM trees, wherein the objects can be addressed and manipulated through different methods. (Introduction to the DOM - Web APIs | MDN)
- JavaScript, one of the most used programming languages, is the main language used for web development and allows for some of the more dynamic functionalities within the web. Originally created by Brendan Eich as server-side language and then employed for running dynamic client-side script on the web pages. There exist some downsides to JavaScript as programming language such as the lack of typing, its being performant and easy to learns deem it a good language for web development. One particular use for JavaScript is to access and manipulate elements within the DOM of the web pages, allowing them to have more dynamic functionalities. (JavaScript - MDN Web Docs Glossary: Definitions of Web-related terms | MDN)
- SEO, short for Search Engine Optimization, is a process of enhancing a web page to better its visibility on search engines such as Google, Bing, Duck Duck Go. This is based upon the working mechanism of search engine, wherein a crawler bot will visit web pages one after another, sequentially index and rank them in accordance with the engine internal algorithms. Better rank and index lead to a page appearing more on search engine and thereby attracts more users. Though these engines are constantly changing, there are common denominators shared amongst them that developers can optimize them for better results. (*What Is SEO / Search Engine Optimization?*)
- AJAX is the acronym for Asynchronous JavaScript and XML, is one important web development technique that utilizes browser built-in XMLHttpRequest objects to send request to servers and JavaScript in conjunction with HTML DOM to display the response XML data. In more modern implementation, JSON has replaced XML for standardised usage. AJAX allows for asynchronous loading of contents on web application as the script is run in the background without disrupting the flow of the web application. Despite popular misconception, AJAX is not a programming language. (*What is AJAX*)
- Node: Node is an asynchronous event-driven JavaScript runtime environment. It is used mostly for server-side application for constructing scalable applications ranging from JSON API server to Single Page Application(*Node.js - Introduction*, no date).

- NPM (or node package manager) is a library system for sharing, downloading, and managing packages.(*About npm* | *npm Docs*, no date)

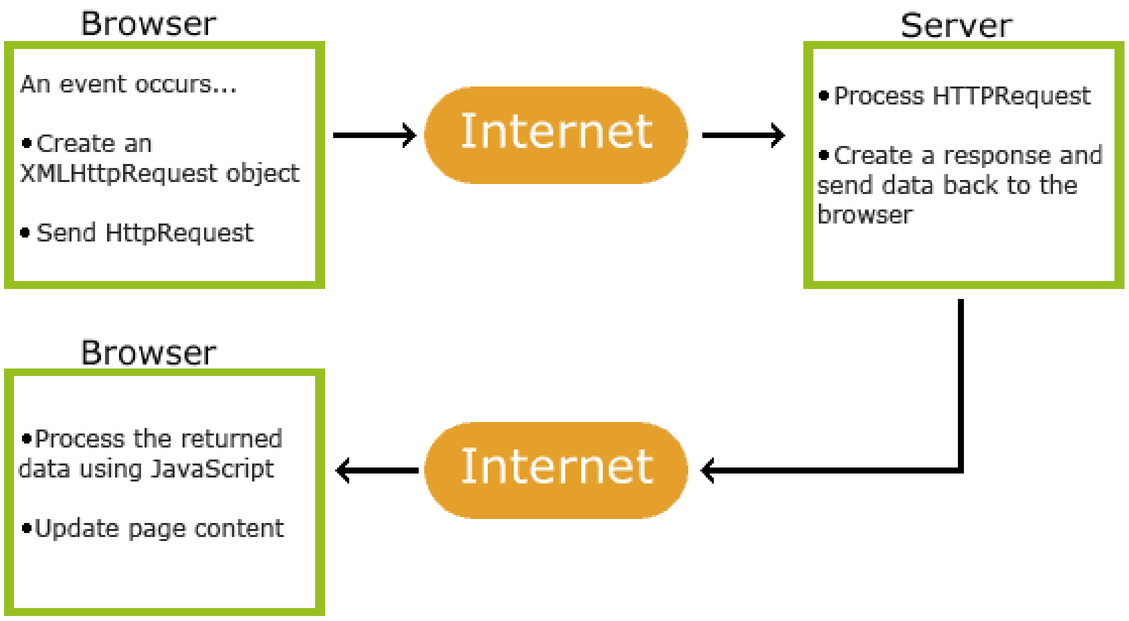

Browser
An event occurs...
•Create an XMLHttpRequest object
•Send HttpRequest
Internet
Server
•Process HTTPRequest
•Create a response and send data back to the browser
Browser
•Process the returned data using JavaScript
•Update page content
Internet

Figure 7. How AJAX works. (What is AJAX)

3.3 Single Page Application

Single Page Application, or SPA for short, is one the modern type of web application. In SPA, the web app loads the web document only once and the web application content will be dynamically loaded with through the use of the JavaScript code. Therefore, there is never a need for page reloads and the web application users can enjoy a multitude of benefits that the SPA model provides. (*SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms* | *MDN*)
Listed below are some of the advantages of Single Page Application: (*Single-page application vs. multiple-page application* | *by Neoteric* | *Medium*)

- Speed: Since most of the resources (HTM/CSS/Scripts) used for the web application are only loaded once when the application is started, it is relatively fast.
- Simplified and streamlined development: It is significantly straightforward to develop the application since the developers are not required to write code for the web page to be rendered server-

side anymore. Development can be initiated through a single without the need of setting a running server.

- Debugging ease: SPAs are easy to debug with modern web browsers. It is possible to monitor network activities, inspect page elements and all associated data.
- Easier transition to mobile application: this is due to the fact that the server-side code can be reused for both web and native mobile applications.
- Effective local storage caching: The web application only needs to send one request, store the respond data, and then will be able to use the data for running the application even without internet access.

Though it comes with a diverse array of benefits, SPA does come with a certain number of drawbacks, even though some of them have been addressed and improved upon: (*Single-page application vs. multiple-page application* **|** *by Neoteric* **|** *Medium*)

- Single Page Applications, by their nature, are not optimized for SEO- Since SPA utilizes AJAX to asynchronously populate its web content, the page is not yet fully loaded when search engine bots crawl into the SPA pages. This results in the bots not being able to read the content in the pages on start-up and this can detrimentally influence the SEO scoring process. This used to be an issue in the past but of this writing, this is largely irrelevant since there are different methods to optimize SPA for SEO such as rendering the page on the server side. In addition thereto, Google has engineered their search engine to be able to browse through AJAX pages, thus making this point irrelevant, at least in the context of Google search engine (*Deprecating our AJAX crawling scheme* **|** *Google Search Central Blog*, 2015).
- The web page application can be slow in its downloading process due to the required client framework being demanding.
- JavaScript being functional in the web browser is mandatory for SPA to work. Should the users disable JavaScript in their browser, these applications will not be able to load or operate correctly. There exist workarounds for this issue, but they are unnecessarily complex to orchestrate, especially just with HTML and CSS components.
- SPA is less secure than traditional web application since they are more prone to Cross-Site Scripting attacks where client-side scripts can be injected into web application. There are methods to prevent this, but they can extra time and effort to implement instead of new features during development.

Some good examples of single page application include Facebook, Instagram, Gmail, Google Maps, et cetera. They also typically employ different frameworks to achieve the end result application, the most frequently used of which include React, Angular, and Vue.

3.4 Micro frontend: The adoption of microservice for Frontend Development

Like any other software development process, frontend development often utilizes monolithic architecture for the early stages. This typically translates to the web application development will be contained within one single repository. This is convenient in the early phases of development as the code base is still relatively small, features are fast to develop and deploy.

As these applications grow, the many problems of monolithic architecture also affect their development process. With Microservice architecture becoming a growing trend among many SaaS companies such as Amazon (Kramer, 2011) or Netflix (Mauro, 2015), the adopting of microservice for frontend development emerges. Coined micro frontend, Cam Jackson defined this as:

> An architecture style where independently deliverable frontend applications are composed into a greater whole. (Jackson, 2019)

Inheriting the same core architecture philosophy from microservice, micro frontend shares many of its characteristics such as (Jackson, 2019):

- Smaller fragmented codebases instead of one single large monolithic structure
- Better scalability with more self-contained teams
- Independent deployment of micro frontend unit
- Improved replaceability, be it upgrade, remove, or replace, in a more incremental fashion.

Micro frontend also comes with the increased repository complexity and overhead cost, not to mentioned the increased number of data bytes that needs to be downloaded for applications to work on the client side (Jackson, 2019).

3.5 Reasoning for Microservice Adoption and Migration

One of the common strategies involving microservice is to segregate a monolithic application into microservice design instead of building a microservice immediately from the onset. Despite there being records of microservice application which are built from scratch, a majority of applications are initially built using monolithic style where quick development, deployment, and creation of new essentials feature are compulsory for the success of early start-up or new founded business (J. Lewis and M. Fowler, 2014). As business grows and scales up in size, the code monolith reaches its eventual size where the code becomes excessively complex for any one single individual to comprehend and all development, bug fixing, deployment slow down considerably. There are certain breakpoints that push business to lean towards decision of microservice migration which includes insufficient maintainability, problematic operability, inadequate performance, deprecated technologies, outdated design structure, long time to market, or combination thereof (Fritzsch *et al.*, 2019). Microservice is an appropriate solution for these issues due to a variety of its advantages, including but not limited to scalability, manageability, maintainability, quality attributes interoperability, reliability, swift time to market, and faster decision-making (Fritzsch *et al.*, 2019).

Since there are no solidified guidelines and codified rulesets for when a service should swap their existing monolithic to the microservice, the architecture switch decision is unique to each case and business requirements. There are some of the points that may help with the decision that project architect and product manager should consider, which include but not limited to: long deployment time, long feature development time, frequency of system crashes caused by hard to identified errors, performance decrease after each feature update, or low productivity of project developers (*Microservices architecture: Moving to microservices | Lightstep blog*, no date). There are also other elements that can contribute to the requirements such new feature demands from clients, dependency and technology updates for security reason or development quality of life, or the application is nearing its limit and needs scaling. In the end, it’s a case-by-case problem and each company should take their business strategy into account, investigate whether the pecuniary and effort investment is worth the potential return value.

3.6 Micro frontend implementation approach

Listed below are five different approaches to Micro frontends in real life scenarios (Jackson, 2019).

- Server-side composition: this approach is achieved through assembling the different micro-frontend application as fragments into one container HTML page during the generation of the HTML on the server. This method is lightweight, performant, and simplistic, though it comes at the cost of losing all the advantages of Single Page Applications since the HTML page is regenerated after each of the user's request
- The build-time integration: in this approach, each micro-frontend application is bundled into separate JavaScript package and included in the main container application's dependency library. The process renders a single deployable JavaScript bundle without any duplication of any same dependency among the micro-frontend application. The major downside of this method derives from the lockstep release process, where each time one of the micro-frontend packages is updated, the entire application needs to be recompiled and each of the packages need to be released. This in turns creates undesirable coupling, which voids one of the main reasons why microservice infrastructure was selected in the first place.
- Run-time integration via iframes: this method takes advantage of the iframe tag to orchestrate the isolated micro frontend applications into the final built container application. Integration via iframes gains the benefits of simple implementation and complete separation of each of the micro frontends and the container, disallowing global variables or stylings bleeding from one application to another. Utilizing iframes also means that sharing common dependencies between micro frontends is virtually impossible, integrating different micro frontend becomes a challenging task, and page history, routing, and linking add more complexity to the application. In the end, despite iframe being a good candidate for micro frontend, inflexibility remains its major drawback in term of micro frontend implementation.
- Run-time integration via JavaScript: this is the more flexible and commonly used solution amongst all the implementation methods, this is achieved through each application is separately bundled and then loaded and mounted on demand through the main container application rendering logic. There are a wide variety of upside in this approach: (i). The capability to pre-load shared styles, dependencies, or libraries help minimalize the application size; (ii).

Separately deployed bundle allows teams to update functionalities independently from each other. One noteworthy variation of this approach is the Run-time integration via Web Component, which is enabled through the new HTML standards issued by W3C, which permits developers to create customized HTML elements, determine their characteristics, and dynamically instantiate them. In this approach, the browser keeps the isolation in a similar fashion to the iframe approach while allowing communication between components. As good as it may sound, the new standards are not yet supported by all browsers, so the functionalities are not guaranteed to operate as intended all the time. There is the option to utilize polyfills to allow new JavaScript features to work older browsers, but this will render the content size of the package.

- There is another approach utilizing webserver to distribute micro frontend application, but this makes the page reloading whenever the user opens a new application so will not be further discussed in this paper.

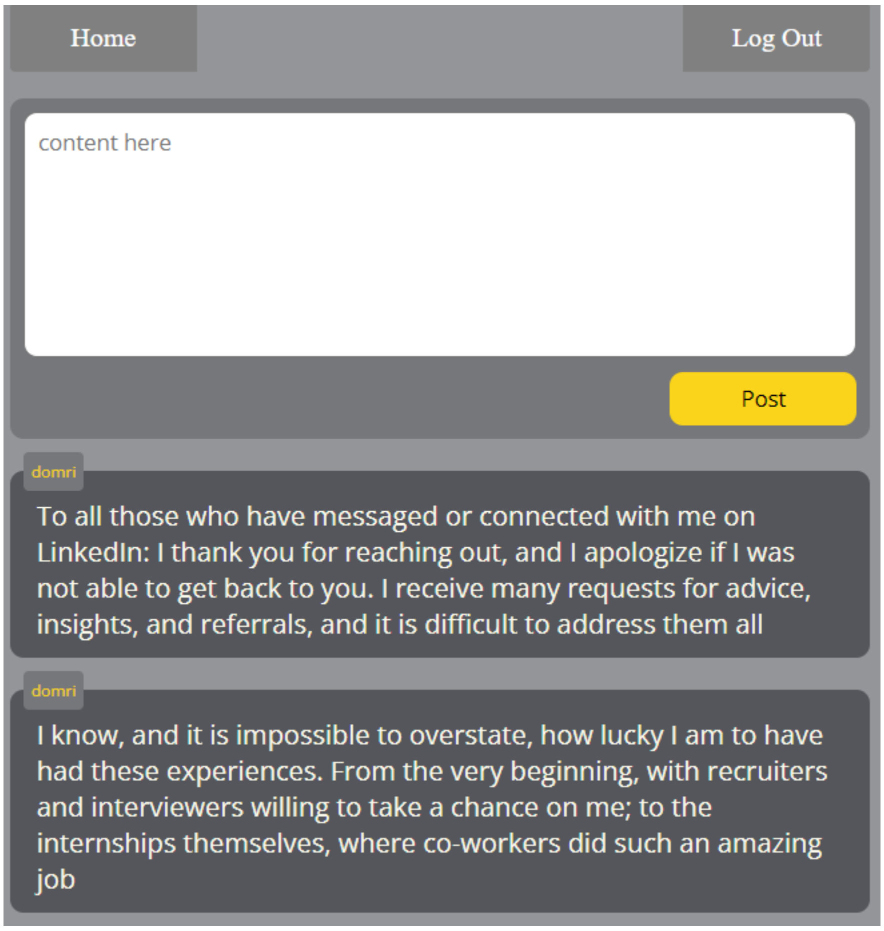
4 Micro frontend migration implementation

The aim of this chapter is to present a sample migration project, wherein the original web application project is a monolithic React application with multiple functionalities. The first step of the project is to conduct an analysis of the different components and functionalities of the application, wherefrom the micro frontends are to be decided and abstracted based on logical reasonings. Subsequently, a migration strategy is to be appropriated from the requirements of the original application and the new micro frontends. A posteriori, each of the micro frontends is to build using different frameworks, e.g., Vue, Angular, or React, to demonstrate the capacity to tolerate various technologies of microservice architecture. In addition, a container application will be built to serve the microservices, thereupon finalizing the form of the microservice. The whole migration process will also be documented along with the provided code snippets. After the migration is complete, performance tests will be conduct for the purpose of discerning a variety of performance statistics between the monolithic and newly migrated microservice application, whereupon a comparison will be drawn.
As a sidenote, the project in this thesis is not on the same scale as typical large corporate monolithic code base since it aims to demonstrate the migration process, so a simpler project might provide a more comprehensible and easier to follow example. Realistically, each case of monolithic to microservice migration needs to be examined on a case-by-case basis and the solution provided in this paper might not be a universal application for many different situations.

4.1 Original application details, characteristics

The original frontend application used as a sample in this thesis is a create-react-app single page application which is a pseudo social media page which allows the user to update their status and see other people’s posts. The data that the users put on the application are to be stored in a backend and will be fetched by the application on start up. Since the project is only for demonstration purpose, the backend and the authentication process are out of scope of this thesis so they are simplified using rudimentary development version so the frontend migration can remain the focus. It should be borne in mind that backend microservice migration and authentication on micro frontend application are complex topics on their own, and hence, they would not be apt to the scope of this thesis.

The contents of the application consist of multiple pages and components, which are all built using React, which can be detailed as below:



To all those who have messaged or connected with me on LinkedIn: I thank you for reaching out, and I apologize if I was not able to get back to you. I receive many requests for advice, insights, and referrals, and it is difficult to address them all

I know, and it is impossible to overstate, how lucky I am to have had these experiences. From the very beginning, with recruiters and interviewers willing to take a chance on me; to the internships themselves, where co-workers did such an amazing job

Figure 8. The Home page with the Upload Component and Content Component

- The Home Page: This page will display the content that all the other users have posted, which include both texts and images. The user can browse through these as they are synchronously organized. The user can upload their images and upload their contents in this page. They can also delete posts so long the posts belong to them. On top of the home page is the navigation bar which allows the user to log out.

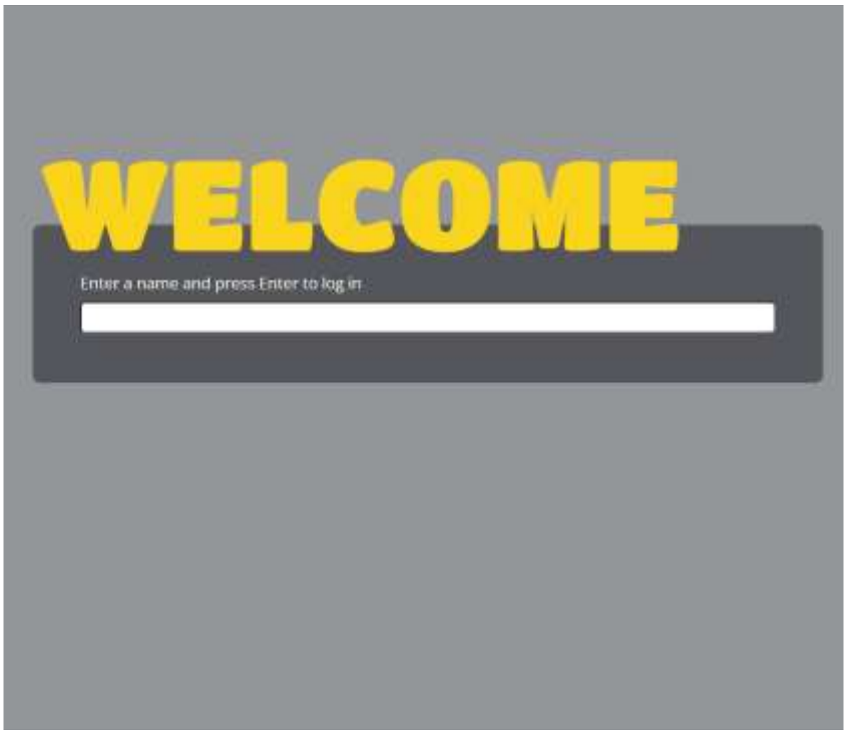


Figure 9. Authentication Page

- Authentication Page: This is the first page the user will see when they open the web application. For simplified reason, it will only ask for the name of the user. Once the user have entered their name and press Enter, they will be taken to the Home page.

The flow of the application is as followed: When the user starts up the application, it will check in local storage in the web browser whether the user is logged in or not. If not, the user is redirected to the authenticate page, for simplifying in this example, the user only needs to enter a name to log in. Otherwise, they will be taken to the Home Page, wherein the user can upload new content or view the contents posted by all the users, which are fetched from the database backend. Though straightforward in nature, the application does utilize a number of dependencies and packages to accomplish a variety of it tasks:

- json-server: this is a REST API package which simulates a working database server for prototyping and mocking purposes, which comes with all the capacities of a functional REST API including GET, POST, UPDATE, and DELETE. It is lightweight and easy to set up, which only requires a db.json file so it is perfect for the purpose of this example project. (*json-server - npm*)

- Reactjs: React is popular modern JavaScript library developed by Facebook for developing web application. It is component-based, declarative, and has state management. Being predictable, easy to learn, straightforward to set up, and have a good array of dependency supports mean it is efficient to use React to build new application with. React was chosen for this application due the aforementioned purposes. (*React – A JavaScript library for building user interfaces*)
- Axios: axios is a promise based HTTP client, in the case the current application project, is employed to communicate with the json-server backend. It is fast, lightweight and easy to employ. (*axios - npm*)
- Emotion: Emotion, or Styled Emotion is a library that enables composing CSS styles with JavaScript with its advantages being accessibility and predictability. There are other features such as source maps, labels, and utilities for testing capacities. It supports both string and object styles so it is up to the developers to choose which styles for the application (*Emotion - Introduction*). String style is to be utilized within this project due to its ease of transition from normal css styling.

4.2 Requirements for the micro frontend application

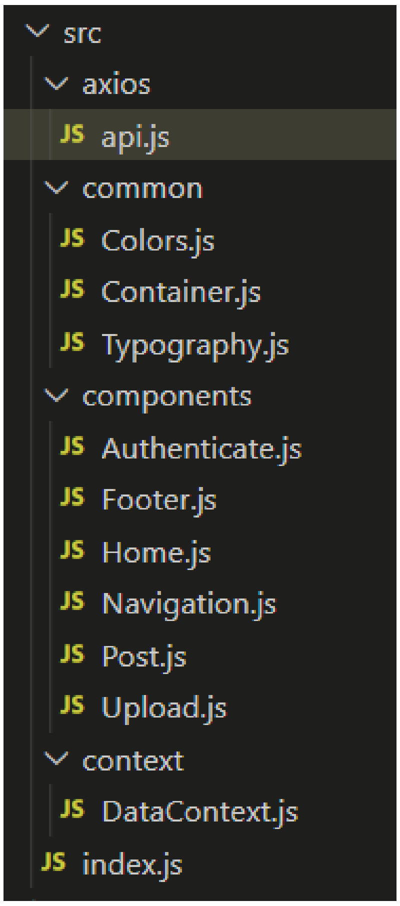
In the scenario of this project, the original monolithic application is to be dissected into three applications where three different teams will work on them separately. Assuming the original monolith application is a prototype application, and the management wants to form different teams to handle the separate functionalities of the application:

- Team Core will work on the container application, where it will contain the other applications and handle the authentication process.
- Team Content will work on the Content component of the Home page, where the user sees the posts of other users, using the fetched data from the server. Since the team is proficient with Vuejs, the Content application is going be using Vuejs as its framework.
- Team Upload will work on the Upload component of the Home page, which is where the user uploads their content. The idea behind this segregation is for Team Upload to synergise between what the user can upload and the team can manage the storage on the backend.

Aside from the team separation, the post-migrated application should work identically to how it was working before. Since the scope of this paper is only limited to the process of micro frontend migration, details on how these applications will be developed, such as how actual authentication will be handled or how improvements on content uploads and displays, are not to be discussed.

4.3 Requirements for the micro frontend application

When examining the file structure of original monolithic application, different components are created on the basis of single responsibility, wherein each component is responsible for one single functionality within the application. It is possible to inspect the components in Figure 10:



```
src
  axios
    api.js
  common
    Colors.js
    Container.js
    Typography.js
  components
    Authenticate.js
    Footer.js
    Home.js
    Navigation.js
    Post.js
    Upload.js
  context
    DataContext.js
  index.js
```

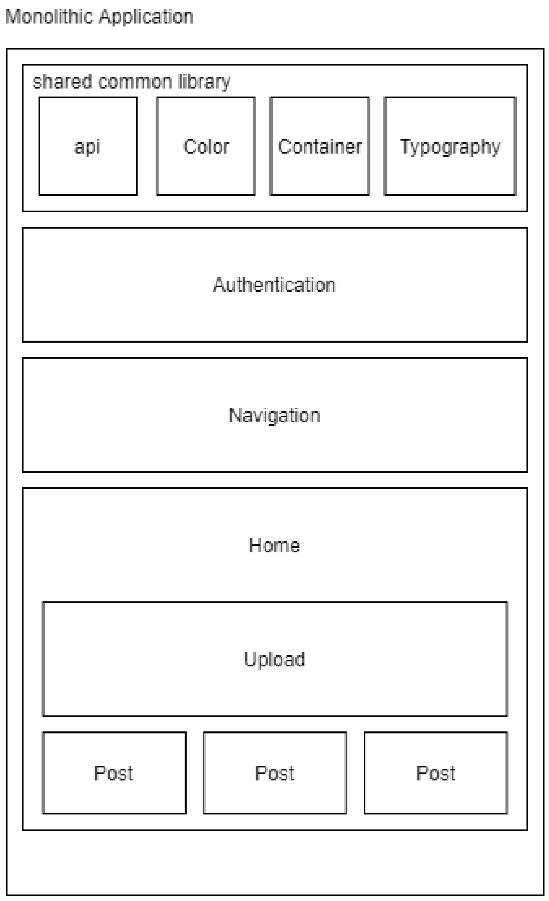


Figure 10. File structure and the nesting of components with the original monolithic application

With the current tessellation of the components within the application, it is straightforward to migrate them to their designated Microfrontend: Upload will be migrated to the Upload application, the Post component will be migrated to the Content application, whereas the rest will be converted to the Container application, where the Authenticate component will stay relatively the same and it will be modified to house the content from the other micro frontend. The final architecture can be seen in figure 11 as below.

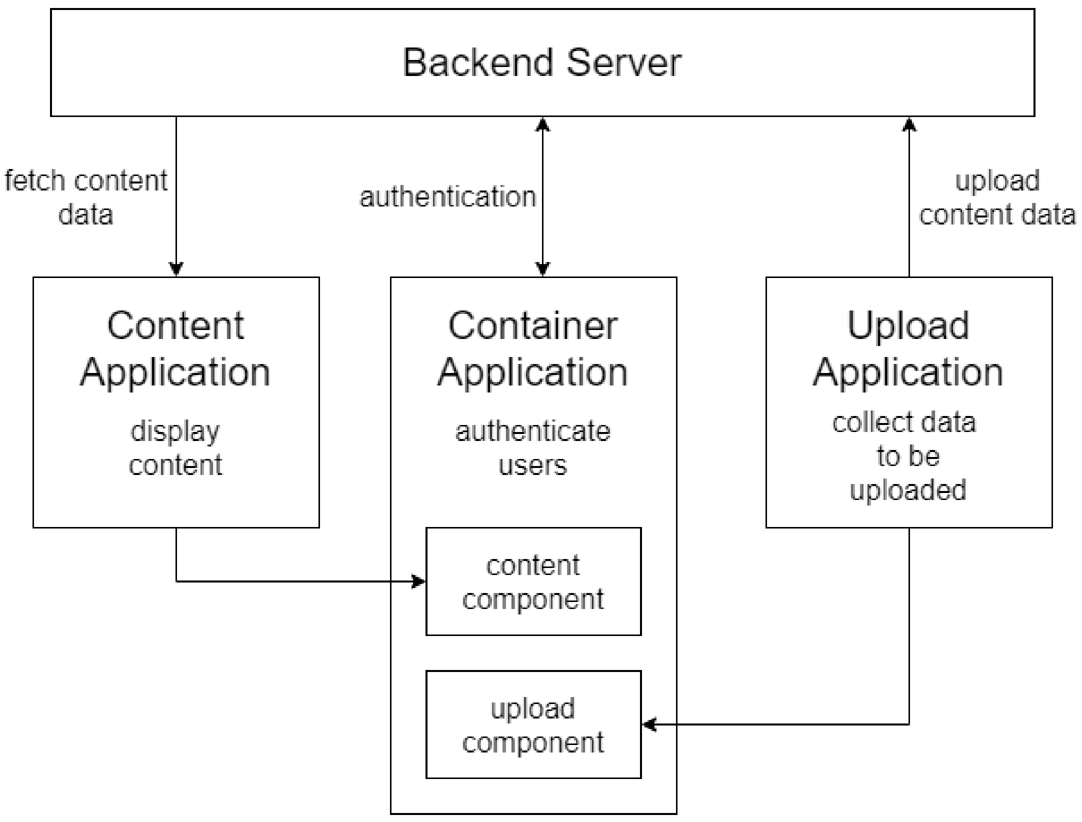


Figure 11. An architectural overlook of the segregated micro frontend applications.

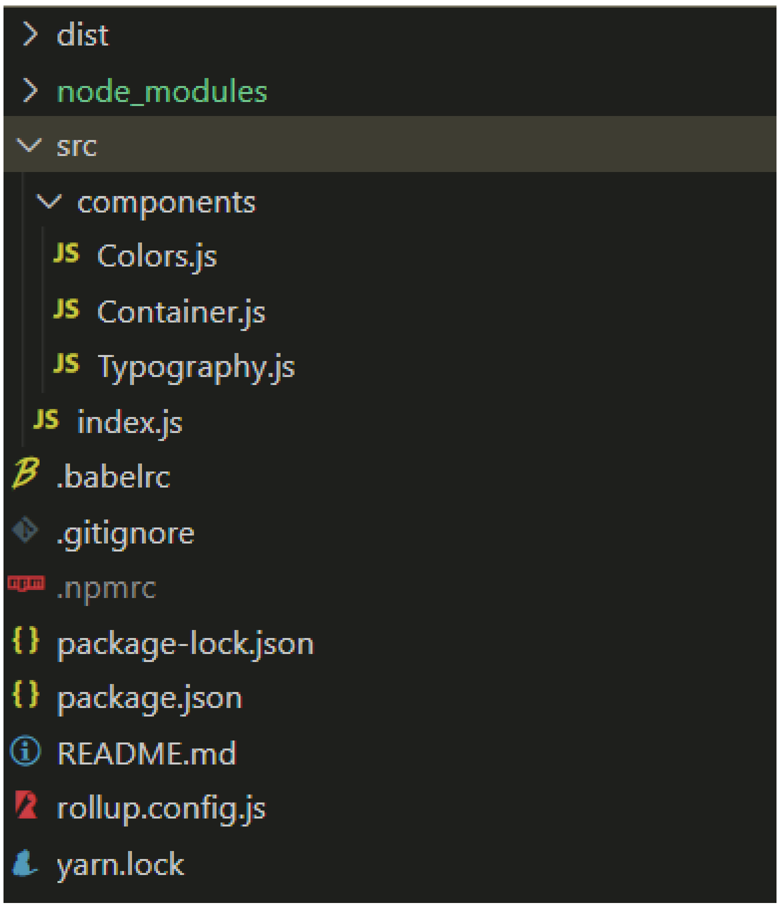
There is also matter of the shared common component and function library used in the application, which consists of a variety of some shared components such as Color for the different colors used in the application, Container for the different configured containers, and Typography for setting up the stylings of typogragraphy. The purpose of the shared components is dictate and superimpose a singular styles within the application due to the importance of maintaining a conformity of styles across the array of micro frontend applications. For this purpose, a shared component library is to be created to house the shared components and serve them to the other micro frontend applications.

4.4 Implementation of the shared common component library

Even though this is not mandatory but can be a common issue occurring during the micro frontend migration process. This is due to the fact that typical frontend application usually has a shared in-app library for common small components such as buttons, containers, or typographies that are used by other components. Within the context of the example application in this thesis, the common components include different variations of containers and typographies with their main purpose being to standardize the visual aspect of the application. Since the monolithic application is to be migrated into smaller micro frontend applications, these shared components also need to be migrated in order for them to be used in the micro frontend application.

In this project, the chosen solution is to use a shared component library, which, in this case, means creating a new repository aside from the repositories for each of the micro frontend. This repository employs rollup, which is a JavaScript module bundler, to compiles the different components into a single library and publish it to a GitHub registry package. This package then can be installed, and the shared components can be utilized by the micro frontends.

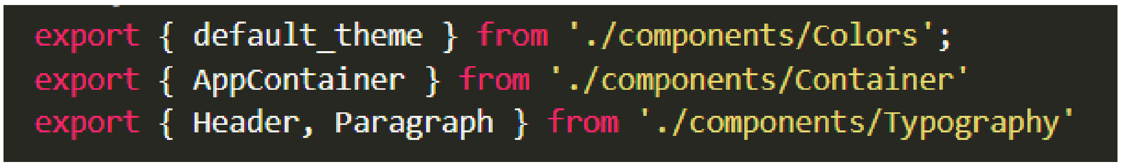
The next part of this section is to provide a generalized documentation of the creation of the common component library to be used within this example application. The overall file structure of the shared library can be seen in figure 12 below.



```
> dist
> node_modules
v src
  v components
      Colors.js
      Container.js
      Typography.js
    index.js
  .babelrc
  .gitignore
  .npmrc
  package-lock.json
  package.json
  README.md
  rollup.config.js
  yarn.lock
```

Figure 12. Repository structure of the common-component library

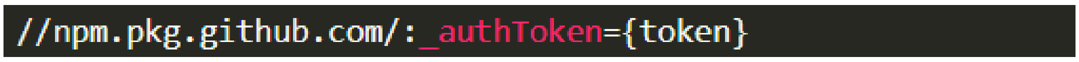
The shared components are the Colors, Container, and Typography files, which are then exported by the index.js:



```
export { default_theme } from './components/Colors';
export { AppContainer } from './components/Container'
export { Header, Paragraph } from './components/Typography'
```

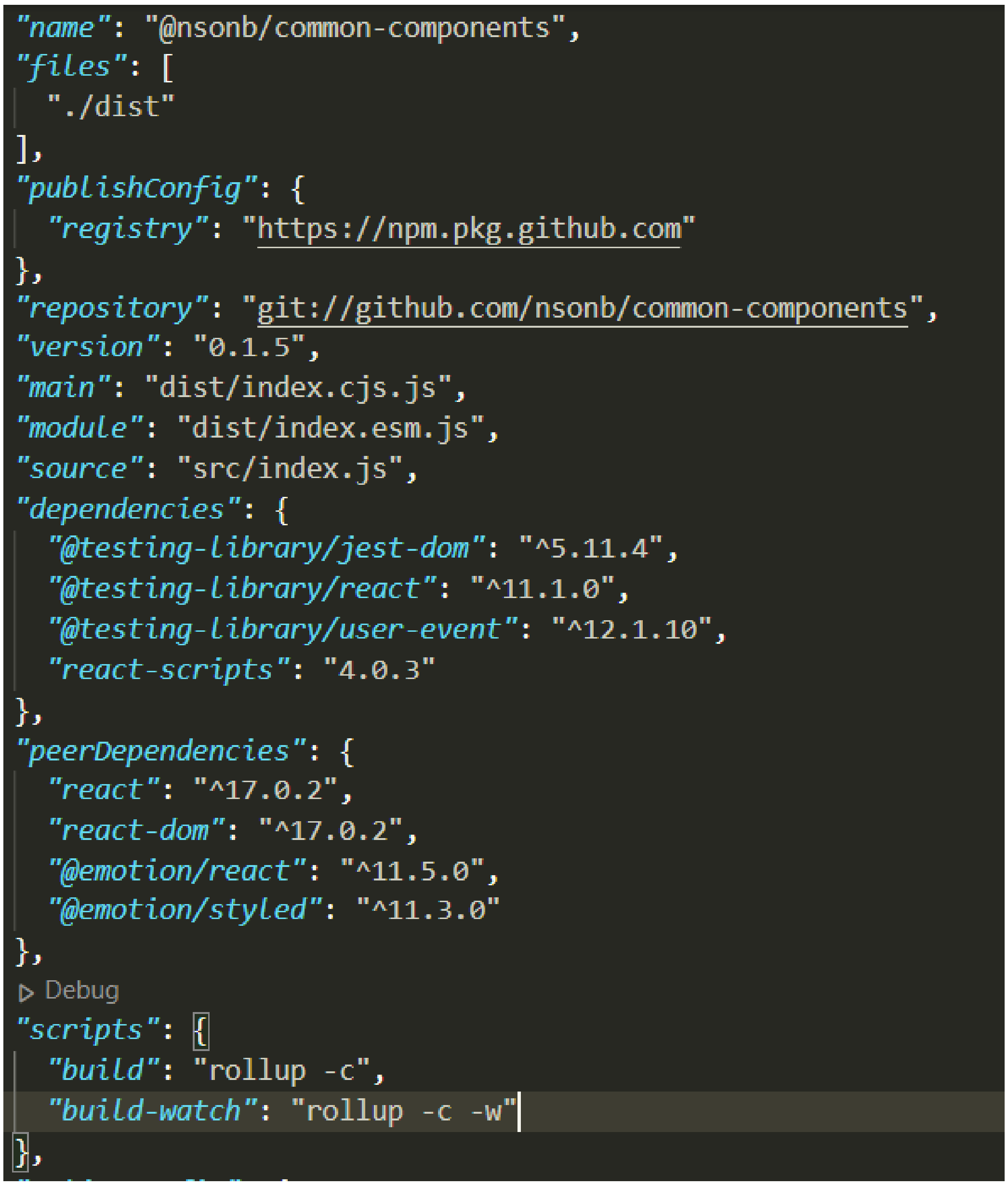
Figure 13. Content of the index file

Then with the rollup, babel, and other dependencies installed, the package.json file is to be reconfigured so rollup can build the files and then npm can publish to GitHub. Since this npm package is private, one needs access to GitHub and create appropriate token to push new version of the package. The token information is then stored in the .npmrc file, which looks as followed:



```
//npm.pkg.github.com/:_authToken={token}
```

Figure 14. Content of the .npmrc file



```
"name": "@nsonb/common-components",
"files": [
  "./dist"
],
"publishConfig": {
  "registry": "https://npm.pkg.github.com"
},
"repository": "git://github.com/nsonb/common-components",
"version": "0.1.5",
"main": "dist/index.cjs.js",
"module": "dist/index.esm.js",
"source": "src/index.js",
"dependencies": {
  "@testing-library/jest-dom": "^5.11.4",
  "@testing-library/react": "^11.1.0",
  "@testing-library/user-event": "^12.1.10",
  "react-scripts": "4.0.3"
},
"peerDependencies": {
  "react": "^17.0.2",
  "react-dom": "^17.0.2",
  "@emotion/react": "^11.5.0",
  "@emotion/styled": "^11.3.0"
},
▷ Debug
"scripts": {
  "build": "rollup -c",
  "build-watch": "rollup -c -w"
},
```

Figure 15. Setup of the package.json file for the common-component repository

Once this is set up, it is relatively straightforward to publish to GitHub package as there are few steps needed to accomplish thus:

- If a change has been made to the files in the repository, the version in the package.json needs to be updated (in this case in figure 15, the version needs to be updated from 0.1.5 to the next number, for example 0.1.6)
- Run the command ‘npm update && npm build’ so rollup can build the dist folder and the package files.

- Run the command ‘npm publish’ and the contents of the dist will be pushed with the package updated to the next iteration (0.1.6 in this situation)

Once the package is published, there is some needed steps of configuration on the micro frontends for them to utilize the components from the common library:

- Create a .npmrc file in the micro frontend repositories, in which will contain the GitHub token that allows access to the GitHub package. The TOKEN is the generated token from GitHub and the registry is the URL of the GitHub package that the application needs to access. In this case, nsonb is the owner of the package:

```
//npm.pkg.github.com/: authToken={TOKEN}
registry=https://npm.pkg.github.com/nsonb
```

- Run the command ‘npm install @nsonb/common-components@0.1.5’
- Import the component from the installed package and use them normally as a component created inside the application. For example, in this next line a component named AppContainer is imported from the package and can be used as a normal React component.

```
import AppContainer from '@nsonb/common-components'
const App = () => {
  return (
    <AppContainer>
      <Upload/>
    </AppContainer>
  );
}
```

There are some downsides in this approach, such as when a change is made in the common-component package, all the micro frontends will also need to update their common components at the same time to benefit from the new version. This may be inconvenient in the long term as dependency issue or breaking changes from the shared common library may occur, but within the small confide of this current project, this approach is acceptable since the scope of this current project is still simple and changes can be fixed easily.

4.5 The micro frontend solution and the setup process of the single-spa root config repository

Even though there are a good number of micro frontend frameworks such as Bit or Webpack/Module Federation, the solution utilized for this project is the single-spa framework. The reasoning behind this choice is due to single-spa's capacity to integrate multiple frameworks into one single DOM (both React and Vue in this case) without the need for refreshing the page. This section will focus on the process of setting up single-spa for the migration process and explaining various details of the working of single-spa.

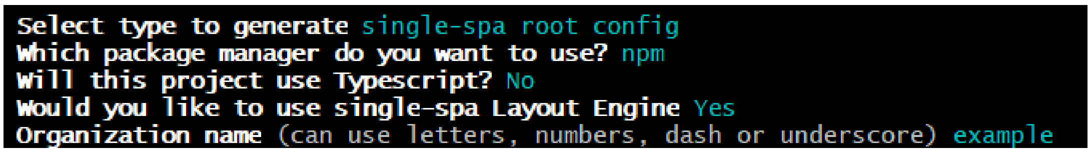
The process of installation of single-spa is straightforward, simple type the following command in the console:

```
npm install --global create-single-spa
# or
yarn global add create-single-spa
```

Once the installation is complete, there is a variety of customizability and options for single-spa initiation, in this case following command is used:

```
create-single-spa
```

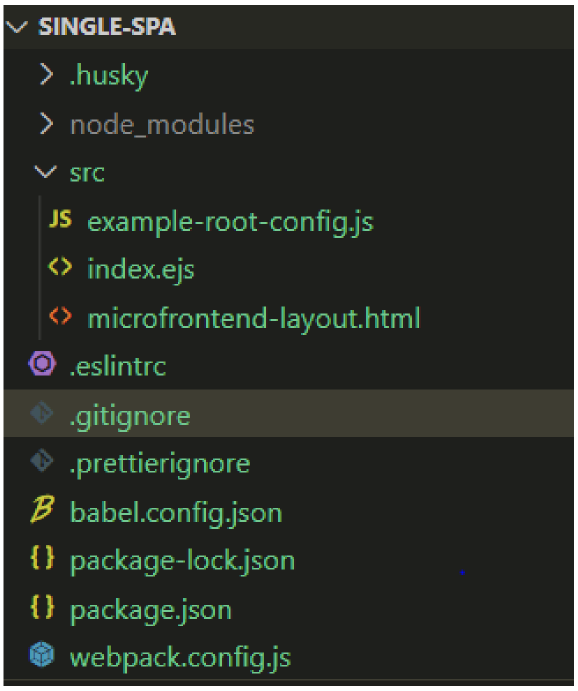
A selection of choices will be presented, wherefrom the choices are selected as below, with the goal is to generate a root repository. This will host the configuration files, act as the container for the micro frontend applications through importing them and dictate how they will display on browser. It will also take advantage of the single-spa Layout Engine to render multiple micro frontends on one single HTML page. The various chosen options can be regarded in figure 18.



```
Select type to generate single-spa root config
Which package manager do you want to use? npm
Will this project use Typescript? No
Would you like to use single-spa Layout Engine Yes
Organization name (can use letters, numbers, dash or underscore) example
```

Figure 16. Options chosen during the setting up the single-spa root config repository

After the setup finalizes, the result repository structure should resemble the following folder structure in figure 19. With this set up, most of the configurations have been set up in the initial process and the main focus for getting the repository to work with the micro frontend is with the files in the src folder.



```
SINGLE-SPA
  .husky
  node_modules
  src
    example-root-config.js
    index.ejs
    microfrontend-layout.html
  .eslintrc
  .gitignore
  .prettierignore
  babel.config.json
  package-lock.json
  package.json
  webpack.config.js
```

Figure 17. Project structure and files generated through the create-single-spa command

There are three files in this folder:

- microfrontend-layout.html is the template file that contains the template that dictates the routing and compositional juxtaposition of the micro frontend applications on the web page. This is read by the single-spa layout engine and used to render the web application.
- example-root-config.js in the central configuration JavaScript file that is responsible for importing, mounting and control the layout to render.
- Index.ejs is the root HTML file of the entire project whereon the contents of the micro frontend applications are to be rendered.

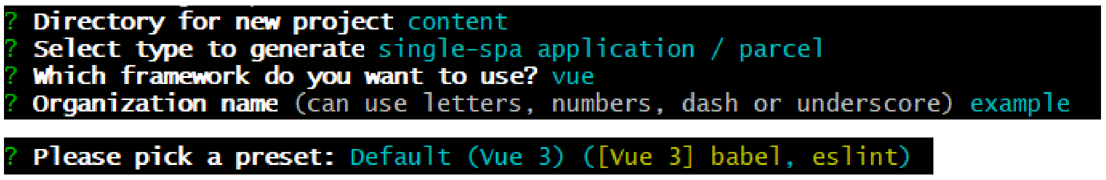
4.6 Initiation of the micro frontend applications

In this project there are two frameworks that are to be utilized for composing the micro frontend applications – which are React and Vue. Though these two frameworks share no similarities aside from using JavaScript and HTML technologies, the setup methods for both of them are going be quite similar despite some minor discrepancies.

In a similar fashion with the root configuration repository set up process and still taking advantage of the benefits of single-spa, the command to use is the same as before:

```
create-single-spa
```

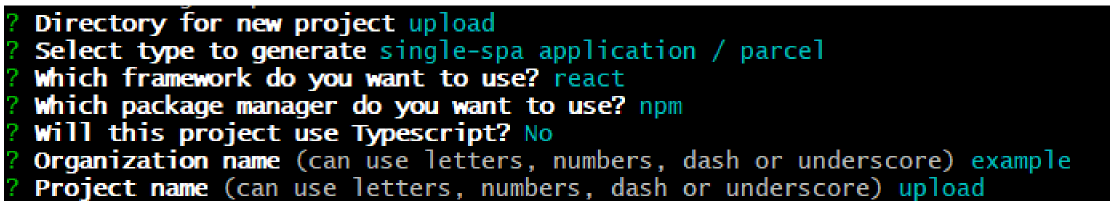
Consequentially, an array of options will emerge which are similar to the root configuration albeit a few differences. These include choosing single-spa application / parcel choice for type of project to generate and framework to use, Vue for the content project and React for the upload projects. The choices for both of these repositories can be observed in the following figure:



```
? Directory for new project content
? Select type to generate single-spa application / parcel
? Which framework do you want to use? vue
? Organization name (can use letters, numbers, dash or underscore) example
```

```
? Please pick a preset: Default (Vue 3) ([Vue 3] babel, eslint)
```

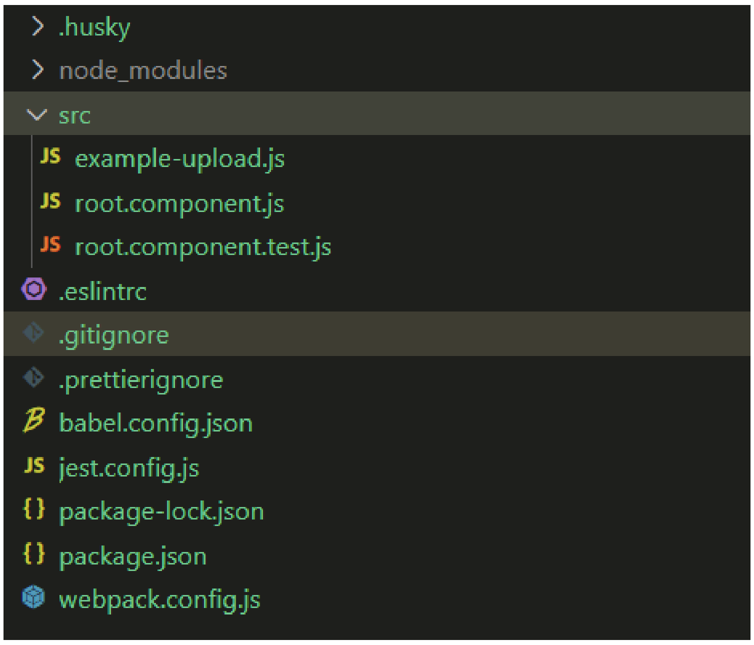
Figure 18. Choices for the content project. For this project the default Vue 3.0 will be utilized.



```
? Directory for new project upload
? Select type to generate single-spa application / parcel
? Which framework do you want to use? react
? Which package manager do you want to use? npm
? Will this project use Typescript? No
? Organization name (can use letters, numbers, dash or underscore) example
? Project name (can use letters, numbers, dash or underscore) upload
```

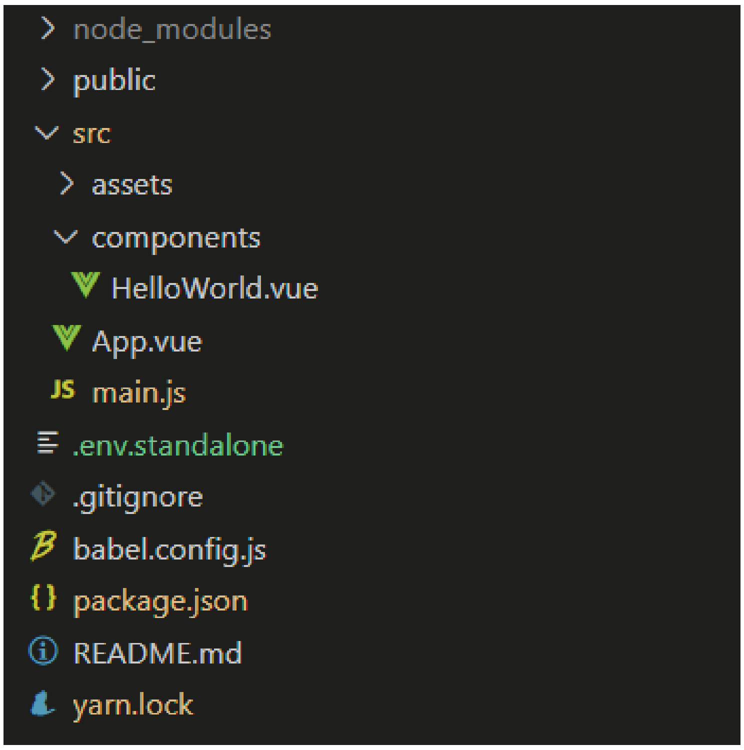
Figure 19. Choices for setting the upload project, which utilizes React and npm to manage its installed packages.

The result the generation command is the creation of two different repositories, one using React and the other utilizing Vue, which are both pre-configured by single-spa to be used in conjunction with the root project. The project structures of the repositories can be regarded below.



```
> .husky
> node_modules
v src
    example-upload.js
    root.component.js
    root.component.test.js
  .eslintrc
  .gitignore
  .prettierignore
  babel.config.json
  jest.config.js
  package-lock.json
  package.json
  webpack.config.js
```

Figure 20. Generated project file structure of the upload micro frontend application, which uses React.



```
> node_modules
> public
v src
  > assets
  v components
      HelloWorld.vue
    App.vue
    main.js
  .env.standalone
  .gitignore
  babel.config.js
  package.json
  README.md
  yarn.lock
```

Figure 21. Generated project file structure of the content application, which uses Vue framework.

The subsequent two sections will briefly go through the files in figure 20 and 21 to provide a brief understanding of the functions of the files and their configuration along with the changes made to the generated files. This is to migrate the corresponding components in the original project to micro frontends in the new application system.

4.7 Migrating the upload component to React micro frontend

This section will elaborate further on the details of the upload micro application, which uses React framework. Of the generated files seen in figure 20, the most important is the example-upload.js file in the src folder. The content of the file can be observed as followed:

```
import React from "react";
import ReactDOM from "react-dom";
import singleSpaReact from "single-spa-react";
import App from "./App";

const lifecycles = singleSpaReact({
  React,
  ReactDOM,
  rootComponent: App,
  errorBoundary(err, info, props) {
    return null;
  },
});

export const { bootstrap, mount, unmount } = lifecycles;
```

This file is generated by the create-single-spa command which installs single-spa-react and instantiate the single-spa version of React with everything already set up. The file is responsible for exporting the lifecycle functions (bootstrap, mount, and unmount functions in this case) so that the upload micro application can be registered to the root-config application. These functions are then to be called by the root-config application through the course of the application running which allows the micro frontends are bootstrapped (initialized), mounted, and unmounted (putting the micro frontend contents on the DOM and removing them therefrom respectively). These are handled by the single-spa package and can be customized for advanced usage. Within the context of this project, the example-upload file does not need to be changed except for root component changed from the generated Root file to the newly written App file.

Since the original project also uses React which is the same framework as the newly created upload micro application, migrating to the upload component is rather straightforward. There is only the matter of copying the code of the components and installing the needed dependency packages for it to function identically to its original counterpart.

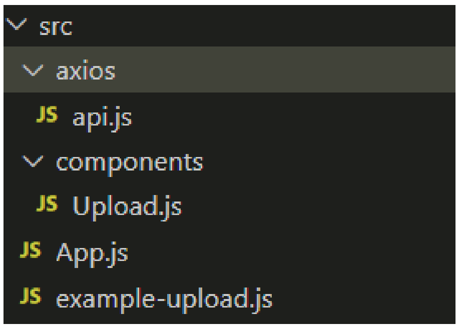


Figure 22. File structure of the upload micro frontend application.

As seen from the figure 22, the Upload.js and the api.js are directly copied from the original monolithic application and the App.js file is used as the wrapper component for the files. The content of the App.js is as followed:

```
import Upload from "./components/Upload";

const App = () => {
  return (
      <Upload/>
  );
}

export default App;
```

Aside from the previous changes, the package.json file also needs to be set up so the application will run on a local port when using the npm run start command, which is 8080 in this case.

```
"start": "webpack serve --port 8080"
```

It is also possible to run the micro frontend as a separate instance through the command line, which helps with the development process of the micro frontend application:

```
npm run start: standalone
```

The application can then be visually seen and functionally tested on any browser by typing http://localhost:8080. N.B., there is another file generated by the create-single-spa command named root.component.test.js which is used for testing purpose but has been deleted since testing is not within the scope of this project.

4.8 Migrating the content components to micro frontend using Vue framework

This section will detail the migrating process of the content components. Generally, this process is similar to generating the upload micro application, albeit the difference in framework, i.e., Vue against React. One of the first similarities in this case is that create-single-spa also generates a fully working repository but this time in react. There is also a generated file for exposing the different lifecycles functions which are mandatory for the micro frontend system to work. The name of the file is main.js and its contents can be observed below:

```
import { h, createApp } from 'vue';
import singleSpaVue from 'single-spa-vue';

import App from './App.vue';

const vueLifecycles = singleSpaVue({
  createApp,
  appOptions: {
    render() {
      return h(App);
    },
  },
});

export const bootstrap = vueLifecycles.bootstrap;
export const mount = vueLifecycles.mount;
export const unmount = vueLifecycles.unmount;
```

As above, in the same fashion as the React micro frontend, bootstrap, mount, and unmount are the three lifecycle functions that will be called by the root-config file in order to mount the application into the DOM. The difference from the upload application is that this utilizes Vue so simple copying and pasting codes will not apply. In this scenario, new code needs to be composed in order to achieve the same goals as the application counterpart in the original monolithic application.

The current implementation includes two files, Post.vue and Posts.vue, which respectively renders the content of one post and acts as the container component for the many instances of component generated from Post.vue. The following is the code of the Posts.vue file:

```
<template>
  <div>
    <div>This is contents</div>
    <div class="posts-container" v-for="post in posts" :key="post.id">
      <Post :post="post"/>
    </div>
  </div>
</template>

<script>
  import Post from './Post.vue'
  export default {
    name: 'Posts',
    props: {
      posts: Array
    },
    components: {
      Post
    }
  }
</script>

<style scoped>
  .posts-container{
    width: 35vw;
    min-width: 480px;
    margin: auto;
    margin-top: 1rem;
    box-sizing: border-box;
  }
</style>
```

And the Post.vue file with the styling mimicking the stylings in the original monolithic:

```
<template>
  <div class="post-container">
    <div class="post-author">
      {{post.author}}
    </div>
    <div class="post-content">
      {{post.content.text}}
    </div>
  </div>
</template>

<script>
  export default {
    name: 'Post',
    props: {
      post: Object
    }
  }
</script>

<style scoped>
  .post-container {
    position: relative;
    background-color: #54565B;
    color: #FAF8EB;
    padding: 1rem;
    padding-bottom: .6rem;
    font-family: 'Open Sans', sans-serif;
    width: 100%;
    margin-top: 1.2rem;
    height: fit-content;
    border-radius: .5rem;
    box-sizing: border-box;
  }

  .post-author {
    display: box;
    position: absolute;
    top: -.7rem;
    left: .5rem;
    padding: .3rem;
    font-size: .6rem;
    color: #FAD41B;
    width: fit-content;
    background-color: #76777B;
    border-radius: .2rem;
  }
</style>
```

The Posts.vue file is then imported by the App.vue and rendered on top of the application, which subsequently is imported by the main.js and exported to be used in the root-config repository. The App.vue is set up so it can fetch the data and refresh the data after certain interval, which in this case is 12000 milliseconds:

```
<template>
  <div>
    <div>Latest contents</div>
    <Posts :posts="this.posts"/>
  </div>

</template>

<script>
  import Posts from './components/Posts.vue'
  import PostDataService from './PostDataService'

  export default {
    name: 'App',
    data() {
      return {
        posts: Array,
        polling: null
      }
    },
    methods: {
      doSomething() {
        PostDataService.getAll()
          .then((res) => {
            this.posts = res.data.sort((a, b) => (a.id - b.id)*-1)
          })
        this.polling = setInterval(() => {
          PostDataService.getAll()
          .then((res) => {
            console.log(res.data.sort((a, b) => (a.id - b.id)*-1))
            this.posts = res.data.sort((a, b) => (a.id - b.id)*-1)
          })
        }, 12000)
      },
    },
    created() {
      this.doSomething();
    },
    beforeUnmount () {
      clearInterval(this.polling)
    },
    components: {
      Posts
    }
  }
</script>
```

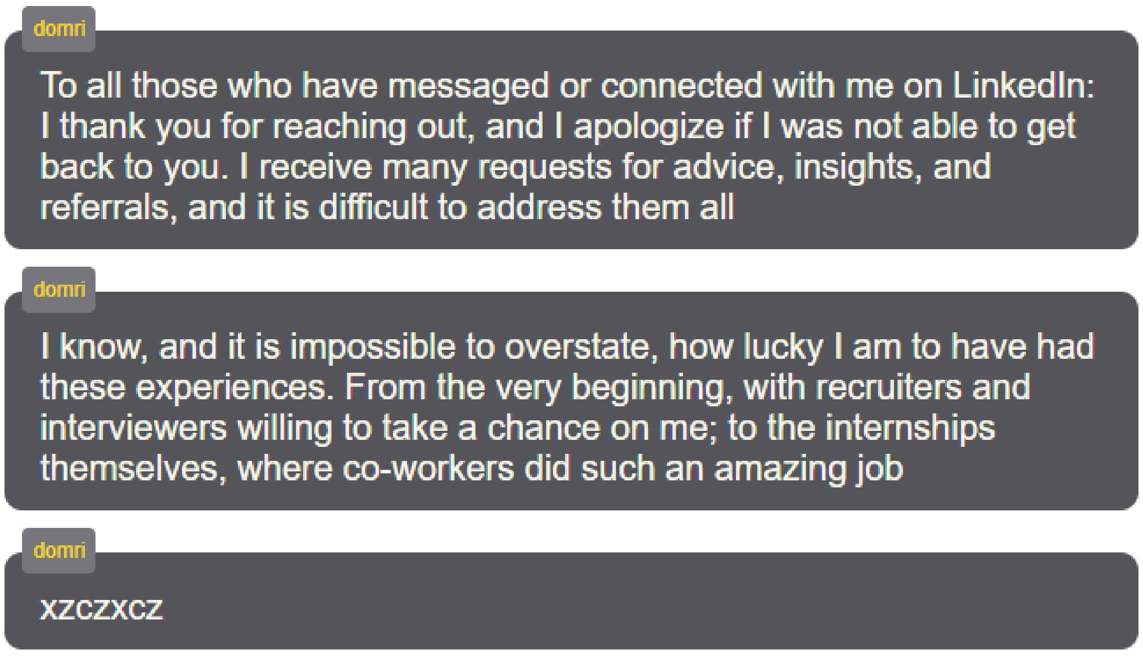
The package.json file also needs some minor reconfiguration to set up the which port the content application will be deployed:

```
"serve": "vue-cli-service serve --port 8081",
```

This will set the command ‘npm run serve’ to host the application host at the address http://locahost:8081. Similar to the React micro application, it is also possible to run the content application in standalone mode in order to develop it independently. The command to use is:

```
npm run serve:standalone
```

The resulted Vue content micro application functions the same as the original component where the different posts are displayed. The contents are fetched on initial startup and also refreshed every 12 seconds so it can fetch and display the latest posts. The visual of the application can be regarded below, which is identical to its monolithic counterpart:



domri

To all those who have messaged or connected with me on LinkedIn: I thank you for reaching out, and I apologize if I was not able to get back to you. I receive many requests for advice, insights, and referrals, and it is difficult to address them all

domri

I know, and it is impossible to overstate, how lucky I am to have had these experiences. From the very beginning, with recruiters and interviewers willing to take a chance on me; to the internships themselves, where co-workers did such an amazing job

domri

xzczxcz

Figure 23. Content micro frontend application display of post contents with standalone settings.

4.9 Assembling the final root-config repository

After setting up the upload and content micro frontends, the process is repeated for the nav component which acts as the top navigation bar and allows for logging out of the application. After all the micro frontends are set up, the root-config project needs to be set up for using the micro frontends. The repository takes advantage of the single-spa-layout package to set up the placement and micro frontend applications on the DOM. The first step is to initiate the layout engine in accordance to the documentation (*Layout Engine* | *single-spa*, no date) :

```
import {
  constructApplications,
  constructRoutes,
  constructLayoutEngine,
} from "single-spa-layout";

const routes = constructRoutes(document.querySelector("#single-spa-layout"));
const applications = constructApplications({
  routes,
  loadApp({ name }) {
    return System.import(name);
  },
});

applications.forEach(registerApplication);
```

This segment of code will populate DOM in accordance with the template with id ‘single-spa-layout’, which is present in the index.ejs file in the head section of the file:

```
<template id="single-spa-layout">
    <single-spa-router>
      <div class="main-content">
        <route path="/">
          <div class="nav">
            <application name="@example/nav"></application>
          </div>
          <div class="upload-container">
            <application name="@example/upload"></application>
          </div>
          <div class="content-container">
            <application name="@example/content"></application>
          </div>
        </route>
      </div>
    </single-spa-router>
  </template>
```

With this template, the nav application will be on top of the DOM, followed by the upload micro frontend and the content micro frontend afterwards. To use the micro frontends, the root-config needs to import, which is accomplished in the experimentalSPA-root-config.js file:

```
const layoutEngine = constructLayoutEngine({ routes, applications });
Promise.all([
  System.import("@example/content"),
  System.import("@example/upload"),
  System.import("@example/nav"),
]).then(() => {
  layoutEngine.activate();
  start();
});
```

This will load the needed micro frontends and activate the layout engine when all the importing is finished. Unfortunately at the moment of this writing, the actual authentication functionality of the final application is incomplete due to the limitation of the documentation and single-spa layout engine not yet configured for conditional routing. Without the authentication functions, the upload application does not work properly. Therefore, the final implementation utilizes a workaround to add the authentication capability into the upload application so it will function nearly identical to the original monolithic. This is done through copying the original Authenticate component to the upload application, but with some minor modification:

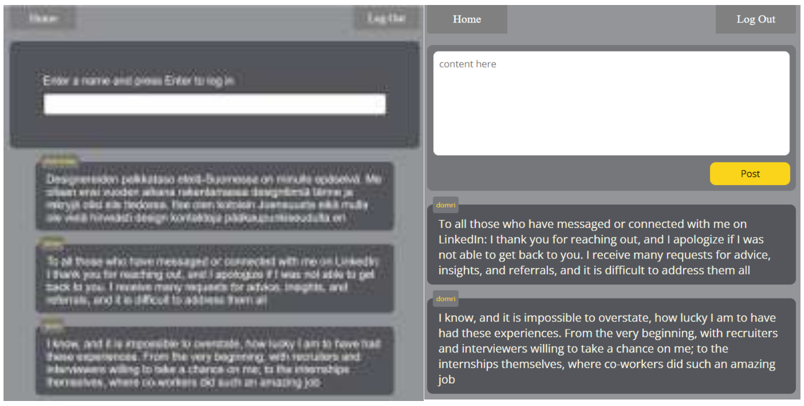
```
import React, { useState } from 'react'
import styled from '@emotion/styled'
import { default_theme } from '@nsonb/common-components'

const FormContainer = styled.form`
  position: relative;
  background-color: ${default_theme.dark_grey};
  padding: 3rem;
  padding-top: 3rem;
  height: 10rem;
  width: 50rem;
  margin: auto;
  margin-top: 15rem;
  box-sizing: border-box;
  border-radius: .5rem;
  color: ${default_theme.second};
  font-family: 'Open Sans', sans-serif;
`

const WelcomeHeader = styled.header`
  position: absolute;
  top: -5rem;
  left: .5rem;
  color: ${default_theme.main};
  font-size: 8rem;
  font-family: 'Titan One', cursive;
  line-height: 7rem;
`

const FormInput = styled.input`
  width: 100%;
  font-size: 1.2rem;
  line-height: 1.5rem;
  height: 2rem;
  border-radius: .3rem;
  margin-top: .5rem;
  box-sizing: border-box;
`

const Authenticate = (props) => {
  const [user, setUser] = useState('')
  const onSubmit = (e) => {
    e.preventDefault()
    window.localStorage.setItem('user', user)
    props.setUser(user)
  }

  return (
    <FormContainer onSubmit={onSubmit}>
      <WelcomeHeader>WELCOME</WelcomeHeader>
      <div>Enter a name and press Enter to log in</div>
      <FormInput type='text' value={user} onChange={(e) =>
{setUser(e.target.value)}}/>
    </FormContainer>
  )
}

export default Authenticate;
```

Subsequently, the App.js file in the upload application will also need to change, wherein it will check whether the user object is present in the localStorage on startup, if not it will load the Authenticate component and allow the user to log in, else it will display the Upload component.

```
import Upload from "./components/Upload";
import Authenticate from "./components/Authenticate";
import { useEffect, useState } from "react";

const App = () => {
  const [ user, setUser ] = useState()
  useEffect(() => {
    const _user = localStorage.getItem('user')
    setUser(_user)
  }, [])
  return (
    <div>
      { user? <Upload/> : <Authenticate setUser={setUser}/>}
    </div>

  );
}

export default App;
```

4.10 Final result

The finalized application functions in a near identical manner as the original monolithic application. The view of the current root application can be seen below:



Designereiden palkkataso etelä-Suomessa on minulle epäselvä. Me olisin ensi vuoden aikana rakentamassa designtiimiä tänne ja mietyjä olisi siis tiedossa. Itse olen toiminut Joensuussa eikä mulla ole vielä hirveästi design kontakteja pääkaupunkiseudulla en

To all those who have messaged or connected with me on LinkedIn: I thank you for reaching out, and I apologize if I was not able to get back to you. I receive many requests for advice, insights, and referrals, and it is difficult to address them all

I know, and it is impossible to overstate, how lucky I am to have had these experiences. From the very beginning, with recruiters and interviewers willing to take a chance on me; to the internships themselves, where co-workers did such an amazing job

To all those who have messaged or connected with me on LinkedIn: I thank you for reaching out, and I apologize if I was not able to get back to you. I receive many requests for advice, insights, and referrals, and it is difficult to address them all

I know, and it is impossible to overstate, how lucky I am to have had these experiences. From the very beginning, with recruiters and interviewers willing to take a chance on me; to the internships themselves, where co-workers did such an amazing job

Figure 24. Finalized micro frontend root application in juxtaposition to the original monolithic.

In figure 24, the upload application is disabled and replaced with the authenticate component which allows the user to authenticate. Once they enter a username, they can use the upload functionality. All the Vue and React applications are all loaded, composed, and rendered onto the DOM independently at the same time. Other than the difference in authentication scheme, almost all functionalities are the same. In general, the migration can be considered a near success even though the authentication process is not a complete overhaul. Since all the micro frontend applications are now properly segregated, development on each of them can be started separately without affecting each other. Moreover, changes can be made to each of the micro frontend can be deployed independently and the root application will be updated with the new version of the changed micro frontend without the need to be redeployed.

Other than the result, even though the migration process itself is rather straightforward, the process of learning how to handle to migration is troublesome and time-consuming due to single-spa documentation being ambiguous and its explanation being unclear at the moment of this writing. This is understandable due to the fact that micro frontend is still relatively new and there is not yet an agreed upon migration method, researching and figuring out how to accomplish migration on each monolithic can take time and effort, especially with large scale project with entangled functionalities with highly intertwined components. Throughout the process, it can be observed that the migration process can work in a small application setting such as the example project as it is still complex enough to define the boundaries between core functionalities and simple enough for the different components not being too entangled to impede the migration process.

Performance-wise, the resulted migrated application functions as nearly the same original monolithic. On the other hand, since the applications are all run in the local machine, the performance can be different when deployed onto cloud service since the connection between the applications can be affected by internet connection between the application deploy server. It is more optimal to test these configurations on a deployed server but that is beyond the scope of this paper.

5 Conclusion

This paper takes closer look at microservice and briefly going through the basics of frontend development, thereby drawing out the advantages and disadvantages of adopting microservice architecture for frontend development. This paper also comes with a project for demonstrating the monolithic to micro frontend migration process using the single-spa package and various functionalities it provides for the migration process.

In the end, the paper is able to discern the mandatory pros and cons of the micro frontend approach. In addition, it is successful in showcasing an example migration project with the end result having multiple frameworks, React and Vue in this case. These frameworks work concurrently with nearly identical functionalities with the original monolithic project.

Notwithstanding, there exists certain limitation on the result of this paper, which include restraint on time, effort, availability of documentations, and shortcomings of the technologies themselves. These hindrances, once be delimited in the future, may yield some substantial impact on the effectiveness of micro frontend utilization. Even with these aforementioned limitations, the micro frontend architecture shows promising results which encourages further explorations and experimentation, such as using other micro frontend frameworks such as Bit or Webpack Module Federation.

Conclusively and in opposition to many of the literature, this paper finds the process of doing the migration early in the development can be advisable instead of waiting until the monolithic grows too large since some of the benefits of utilizing micro frontend can already be applicable with smaller application. Though it is better if the application has already a better boundary definition between its functionalities.

References

About npm | npm Docs (no date). Available at: https://docs.npmjs.com/about-npm (Accessed: 5 November 2021).

axios - npm (no date). Available at: https://www.npmjs.com/package/axios (Accessed: 12 October 2021).

Bogner, J. *et al.* (2019) 'Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality', in *Proceedings - 2019 IEEE International Conference on Software Architecture - Companion, ICSA-C 2019*. Institute of Electrical and Electronics Engineers Inc., pp. 187–195. doi: 10.1109/ICSA-C.2019.00041.

Chen, R., Li, S. and Li, Z. (2018) 'From Monolith to Microservices: A Dataflow-Driven Approach', in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. IEEE Computer Society, pp. 466–475. doi: 10.1109/APSEC.2017.53.

'Deprecating our AJAX crawling scheme' (no date) *Official Google Webmaster Central Blog*. Available at: https://webmasters.googleblog.com/2015/10/deprecating-our-ajax-crawling-scheme.html (Accessed: 26 September 2021).

Emotion - Introduction (no date). Available at: https://emotion.sh/docs/introduction (Accessed: 12 October 2021).

Fritzsch, J. *et al.* (2019) 'Microservices Migration in Industry: Intentions, Strategies, and Challenges', in *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*. Institute of Electrical and Electronics Engineers Inc., pp. 481–490. doi: 10.1109/ICSME.2019.00081.

Ihde, S. and Parikh, K. (2015) *From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture*, *March*. Available at: https://www.infoq.com/presentations/linkedin-microservices-urn/ (Accessed: 1 April 2021).

Introduction to HTML (no date). Available at: https://www.w3schools.com/html/html_intro.asp (Accessed: 21 September 2021).

Introduction to Monolithic Architecture and MicroServices Architecture | by Siraj ul Haq | KoderLabs | Medium (no date). Available at: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63 (Accessed: 21 September 2021).

Introduction to the DOM - Web APIs | MDN (no date). Available at: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (Accessed: 21 September 2021).

J. Lewis and M. Fowler (2014) *Microservices*. Available at: https://martinfowler.com/articles/microservices.html (Accessed: 1 April 2021).

Jackson, C. (2019) *Micro Frontends*. Available at: https://martinfowler.com/articles/micro-frontends.html (Accessed: 1 April 2021).

JavaScript - MDN Web Docs Glossary: Definitions of Web-related terms | MDN (no date). Available at: https://developer.mozilla.org/en-US/docs/Glossary/JavaScript (Accessed: 21 September 2021).

json-server - npm (no date). Available at: https://www.npmjs.com/package/json-server (Accessed: 12 October 2021).

Kramer, S. D. (2011) 'The Biggest Thing Amazon Got Right: The Platform', pp. 0–3. Available at: https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/ (Accessed: 1 April 2021).

De Lauretis, L. (2019) 'From monolithic architecture to microservices architecture', in *Proceedings - 2019 IEEE 30th International Symposium on Software Reliability Engineering Workshops, ISSREW 2019*. Institute of Electrical and Electronics Engineers Inc., pp. 93–96. doi: 10.1109/ISSREW.2019.00050.

Layout Engine | single-spa (no date). Available at: https://single-spa.js.org/docs/layout-overview (Accessed: 4 November 2021).

Mauro, T. (2015) 'Adopting Microservices at Netflix: Lessons for Architectural Design', *Nginx Blog*, pp. 1–5. Available at: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/ (Accessed: 1 April 2021).

Microservices architecture: Moving to microservices | Lightstep blog (no date). Available at: https://lightstep.com/blog/microservices-architecture-when-and-how-to-move-to-microservices/ (Accessed: 17 September 2021).

Monolithic & Microservices Architecture | by Henrique Siebert Domareski | Medium (no date). Available at: https://henriquesd.medium.com/monolithic-microservices-architecture-239e8799d3e1 (Accessed: 21 September 2021).

Monoliths vs. microservices — benefits and drawbacks [a comparision] | by Transparent Data | Blog Transparent Data ENG | Medium (no date). Available at: https://medium.com/transparent-data-eng/monoliths-vs-microservices-benefits-and-drawbacks-a-comparision-9e7a462b8e3a (Accessed: 16 September 2021).

Nadareishvili, I. *et al.* (2015) *Microservice architecture : Aligning principles, practices, and culture*, *Microservices, IoT, and Azure*. Available at: http://oreilly.com/catalog/errata.csp?isbn=9781491956250 for (Accessed: 3 April 2021).

Newman, S. (2015) *Building Microservices - Designing fine-grained systems*. Available at: https://books.google.com/books?hl=en&lr=&id=jjl4BgAAQBAJ&oi=fnd&pg=PP1&dq=Building+microservices:+designing+fine-grained+systems.&ots=_BHT_k9_oM&sig=icBfgU4o20X77zbYjSYAkTt-Qo4 (Accessed: 6 April 2021).

Node.js - Introduction (no date). Available at: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm (Accessed: 5 November 2021).

React – A JavaScript library for building user interfaces (no date). Available at: https://reactjs.org/ (Accessed: 12 October 2021).

Single-page application vs. multiple-page application | by Neoteric | Medium (no date). Available at: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58 (Accessed: 23 September 2021).

SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN (no date). Available at: https://developer.mozilla.org/en-US/docs/Glossary/SPA (Accessed: 23 September 2021).

Villamizar, M. *et al.* (2015) ‘Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud’, in *2015 10th Colombian Computing Conference, 10CCC 2015*. Institute of Electrical and Electronics Engineers Inc., pp. 583–590. doi: 10.1109/ColumbianCC.2015.7333476.

What Is a Front-End Developer? · Front-End Developer Handbook 2018 (2018). Available at: https://frontendmasters.com/books/front-end-handbook/2018/what-is-a-FD.html (Accessed: 15 April 2021).

What is AJAX (no date). Available at: https://www.w3schools.com/whatis/whatis_ajax.asp (Accessed: 26 September 2021).

What Is SEO / Search Engine Optimization? (no date). Available at: https://searchengineland.com/guide/what-is-seo (Accessed: 26 September 2021).