INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Micro Frontends para Aplicações Web

DANIEL FILIPE FERNANDES ALMEIDA

Outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Micro Frontends for Web Applications

Daniel Filipe Fernandes Almeida

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Supervisor: Prof. Ana Madureira

Porto, October 14, 2021

Dedicatory

To everyone that has supported me throughout my life's journey.

Abstract

Over the course of the last ten years, web applications have stood out as the go to when it comes to software development, replacing the traditional desktop apps. As a result of this, customer demand has been on the rise, leaving companies with their hands full as they try to meet expectations.

In the year 2011, the world was introduced to the concept of Microservices, which came to revolutionize the software development industry by splitting typical backend monolithic codebases into smaller, more manageable chunks that were able to be developed, deployed and tested independently. This in turn facilitated customer deliveries as it opened the door to iterative and incremental developments, preventing customers from having to wait months, or even sometimes years, to receive a fully functional product since they could now be handed constant updates in time periods as short as 15 days.

However, even though this expedited today's backend developments, a web application is not only composed by a server-side layer, but also a client-side one. Today's application frontends still rely on the traditional monolithic architecture which presents the same issues that backend developments once did and, due to the ever-increasing popularity of the online world, these types of solutions have been growing massively and escalating those problems. By having these architectural discrepancies between frontend and backend codebases, iterative and incremental software developed can never be fully realized as all microservices still depend on the same monolithic frontend. This is where Micro Frontends come in.

The concept behind this model is to take the teachings microservices provided and expand them to the client-side. By organizing applications into several components, each with its respective business sub-domain, vertical teams can then be assembled and take full ownership of said sub-domain, ultimately implementing the component from end-to-end.

Since a Micro Frontend based architecture is still a somewhat recent concept, information is lacking and makes the assessment of its viability hard to reach. As such, this dissertation aims to study its feasibility by studying and comparing possible alternatives, ultimately describing its advantages and disadvantages, and finally presenting a proof-of-concept application.

Keywords: Web Applications, Micro Frontends, Microservices, Architecture

Resumo

Ao longo dos últimos dez anos, as aplicações *web* têm-se destacado no que diz respeito ao desenvolvimento de *software*, substituindo as aplicações *desktop* tradicionais. Consequentemente, a quantidade de clientes neste mercado tem vindo a aumentar, deixando as empresas com dificuldades ao tentar atender às expectativas.

Em 2011, o mundo tecnológico foi apresentado ao conceito de Microsserviços, o qual veio revolucionar a indústria de desenvolvimento de *software* ao dividir soluções monolíticas em peças de menor dimensão e mais facilmente geridas que possibilitaram os processos de desenvolvimento, implantação e teste de forma independente. Deste modo os microserviços permitiram a adoção de desenvolvimentos iterativos e incrementais, que por sua vez vieram evitar morosas esperas por parte dos clientes para receber o seu produto desejado em troca de entregas de, por exemplo, 15 em 15 dias (a duração por defeito de um sprint na maioria das metodologias de desenvolvimento de software ágeis).

No entanto, embora isso tenha acelerado o desenvolvimento de código relativo ao *backend*, uma aplicação *web* não é composta apenas por uma camada do lado do servidor, mas também do lado do cliente. Atualmente, o desenvolvimento de aplicações de *frontend* tende a utilizar uma arquitetura monolítica tradicional, a qual apresenta os mesmos problemas que os desenvolvimentos de *backend* anteriormente apresentavam. Devido ao aumento de popularidade das aplicações *web*, essas mesmas soluções monolíticas têm crescido exponencialmente, agravando os problemas. Com as discrepâncias arquiteturais entre projetos de *frontend* e *backend*, o conceito de desenvolvimento de *software* iterativo e incremental nunca pode ser cumprido na totalidade, pois todos os microsserviços implementados dependem do mesmo *frontend* monolítico. Para solucionar os problemas supramencionados, surgiu o conceito de *Micro Frontends*.

O principal propósito da arquitetura de *Micro Frontends* é aplicar os conceitos providenciados pelos microsserviços e expandi-los para o lado do cliente. Ao organizar as soluções em vários componentes, cada um com seu respectivo subdomínio do negócio da aplicação, torna-se possível a criação de equipas verticais, as quais assumem controlo total do referido subdomínio, implementando o componente de ponta a ponta.

Visto que uma arquitetura de *Micro Frontends* ainda é um conceito relativamente recente, avaliar a sua fiabilidade torna-se uma tarefa difícil. Deste modo, esta dissertação procura estudar este modelo e comparar as diferentes alternativas de implementação, descrevendo as suas vantagens e desvantagens e, por fim, apresentando uma prova de conceito.

Palavras-chave: Aplicações Web, Micro Frontends, Microsserviços, Arquitetura, Desenvolvimento de Software

Acknowledgement

I would like to first thank my supervisor, Professor Ana Madureira, for all her guidance and patience throughout the development of the present dissertation.

My parents, for making this possible and my sister for egging me on.

All my friends for walking this journey with me.

All the teachers that had an impact on my education.

Everyone at Konkconsulting, that make me wake up and go to work with a smile on my face everyday.

Finally, my girlfriend, for being my rock.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter contextualizes the problem being tackled in the current document while also providing a brief description, the objectives and the approach used for the project's development.

1.1 Context

Traditionally, when referring to Web Applications, it is safe to categorize them into two parts: back-end (server), that is hidden from the user and in most cases responsible for data processing and storage, and front-end (client), that provides an interface for the user to interact with and communicates with the server.

Currently, there are multiple architectural approaches that can be applied to develop back-end components, one of them being Microservices, which is often referred to as the most suitable for scalable systems while also being the most versatile. Unlike monolithic architectures, Microservices structure an application as a collection of services that are loosely coupled, highly maintainable and testable, independently deployable, and capable of being developed by a small team (Richardson, 2018). This makes the Microservices architectural style the go-to approach for large/complex applications.

On the other hand, client-side applications have been suffering from continuous increases to complexity and overall size. This makes the standard front-end monolith approach less suited for their development, upping the demand for an alternative architectural style, leading us to micro frontends. The concept is similar to the one behind Microservices, break up the application into many smaller, more manageable pieces which aims for increased effectiveness and efficacy of front-end development teams.

The main goal of this dissertation is to examine the benefits of Micro Frontends when applied in the development of Web Applications, with a particular interest in single-page applications (SPA).

1.2 Problem

In recent years, microservices have exploded in popularity, with many organizations using this architectural style to avoid the limitations of large, monolithic backends. While much has been written about this style of building server-side software, many companies continue to struggle with monolithic frontend codebases (Richardson, 2018).

To this day, developers are still used to creating fully featured browser applications that tend to grow over time, making them harder to maintain and work on. Any backend change, more specifically on one of the services used by the client, propagates directly to said client since the new feature/update needs to be reflected on it. On top of that, changes to a single service might sometimes impact multiple components, leading to a more time consuming and tedious update process.

The concentration of knowledge in a single codebase is also concerning given that the frontend development team must be able to cope with the entirety of the backend structure. In a scenario where multiple teams are working on distinct Microservices responsible for providing new or updated features to the same Web Application, an increase in pressure on the frontend development team is set to occur since the application has to be deployed in one go. Having multiple teams working independently on the frontend codebase could not nullify the issue since the new changes must be integrated into the same project and duly tested given that they are at a liability to break other features. This leads to the conclusion that the teams are not actually working independently.

Other issues also rise when dealing with a multi-environment setup (e.g. having a development, quality, and production instance) since our client application can become blocked for a certain environment when the underlying changes to a specific microservice have not gone live yet. To address these and other issues, we are seeing more and more patterns emerge for decomposing frontend monoliths into smaller, simpler chunks that can be developed, tested, and deployed independently, while still appearing to customers as a single cohesive product (Richardson, 2018).

This technique is called Micro Frontends and is defined as "An architectural style where independently deliverable frontend applications are composed into a greater whole" (bit.dev, 2021).

1.3 Objectives

This dissertation's main goal is to assess the value of a Micro Frontend architecture when developing large web applications. By weighing its benefits and downsides, the idea is to then decide whether a move from the traditional monolithic frontend is justified and advantageous.

The idea behind Micro Frontends is to think about a website or web app as a composition of features which are owned by independent teams. Each team has a distinct area of business or mission it cares about and specializes in. A team is cross functional and develops its features end-to-end, from database to user interface.

To properly evaluate the applicability and effectiveness of such a model, an initial research stage will take place to provide added context to current frontend development techniques and subsequently emphasize and describe Micro Frontends. The analysis itself should also consider monetary costs since they motivate a great deal of architectural compromises.

It is important to highlight which conditions/situations incentivize the usage of Micro Frontends in addition to the ones where a more traditional approach would be most suited. This is especially true since design models are almost never a universal fit for specific developments but instead vary with the project's context and constraints.

Finally, a proof of concept should be designed and implemented to facilitate the evaluation of the architecture's merit and place eventual difficulties/hurdles into perspective. The application should be composed by small components that can be developed, tested, and deployed independently.

1.4 Approach

As mentioned in the objectives section of the current dissertation, a possible solution for the presented problem revolves around applying a Micro Frontend architecture. This approach would enable the creation of development teams which would take full ownership of a specific business sub-domain, becoming subsequently responsible for designing and implementing its features from end-to-end.

Logically, before ensuing with the actual adoption of such a methodology, one must assess its viability and effectiveness by analysing its advantages and disadvantages, as well as comparing it to possible alternatives. To do so, an initial research stage will be performed to fully understand its architectural drivers while also analysing companies that are currently making using of Micro Frontends and reading up on their experiences.

Afterwards, in order to apply the knowledge acquired in the research stage, a proof of concept web application is to be developed, possibly with multiple versions that use several composition techniques, to finally evaluate them against each other and draw the respective conclusions.

1.5 Document Structure

This dissertation is composed by 5 chapters.

Chapter 1 (Introduction) provides context on what motivated this dissertation while also explaining the problem and objectives.

Chapter 2 (State of the Art) presents existing solutions for the problem in addition to a detailed description of the Micro Frontends architecture.

Chapter 3 (Value Analysis) presents the value a Micro Frontend architecture can offer from both business and user perspectives.

Chapter 4 (Solution Description) presents the a detailed description of the design for the proposed solutions regarding the development of web applications through a micro frontend-based approach.

Chapter 5 (Solution Implementation) details the actual implementation of the proof of concept application, providing a step by step description of said process.

Chapter 6 (Evaluation) details which evaluation indicators were utilized in conjunction with the assessment methodologies.

Chapter 7 (Conclusion) delineates this dissertation's conclusions, providing a brief summary on the results found and putting forward some future work.

Chapter 2

State of The Art

In this chapter, a quick introduction is provided for the main topics of interest for this thesis, followed by a detailed description of Micro Frontends and how they can be integrated with Microservice based architectures. It is important to understand the issues that microservices are intended to solve and translate them into the current problems with frontend development.

2.1 Client Server Architecture

As the name indicates, a client-server architecture splits an application into two parts: the client and the server.

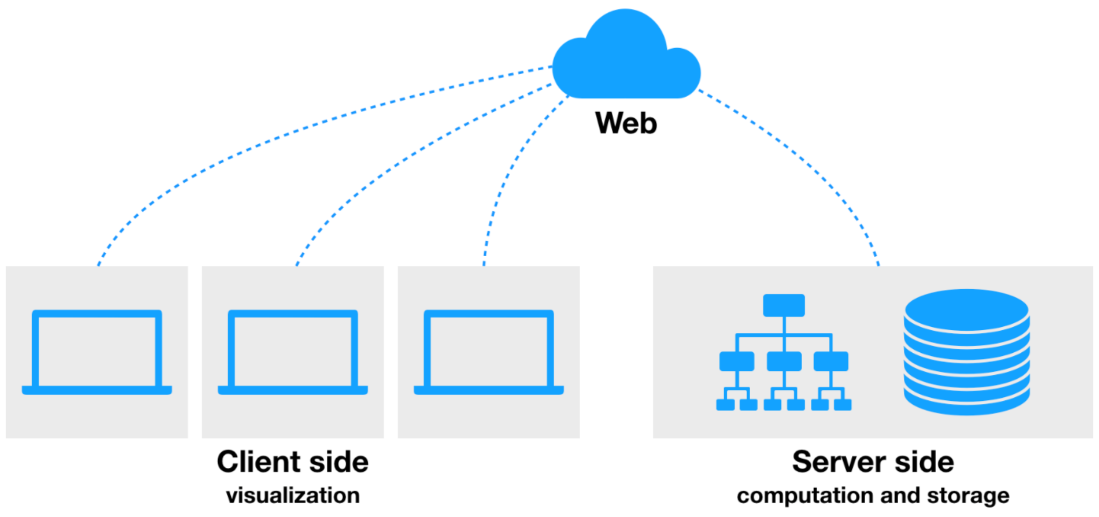


Figure 2.1: Client Server Architecture
(ASIAA et al., 2019)

The server part of this architectural model is responsible for providing the application's central functionality, allowing multiple clients to connect to it and request for any supported task to be executed.

When a request is performed, the client then promptly waits for the server to return a response which varies according to the executed task and its result. On the other hand, the client provides an interface that the user can interact with and use to trigger the requests. The same interface is also used to display the results the server returns.

A good thing to note is that, even though the server is usually responsible for data processing, this does not mean the client cannot perform some processing for itself.

When developing Web Applications, the two parts of this architecture are often referred to as frontend (client) and backend (server).

2.2 Backend Architecture

When it comes to backend development, there are two prevalent architectural styles that tend to be brought up for discussion: monolithic and microservices.

In the following subsections, both these alternatives will be thoroughly detailed in accordance to their principles, advantages, and disadvantages.

2.2.1 Monolithic Architecture

With the advancements in software architecture and the novelties that arose for developing applications, the definition of monolithic architecture returned as a major topic of discussion, mostly with the purpose of being compared with microservices oriented solutions.

According to Google, a monolith is "a geological feature consisting of a single massive stone" which is why it is used to define an architectural style where the entirety of the application's features is built in a single process. Aside from being part of the same codebase, the application's functionalities share the same machine's resources, be it processing assets or memory.

Usually, a solution that follow these architecture's guidelines is comprised of a client-side user interface (e.g. consisting of HTML pages and JavaScript), a server-side application, and a database (consisting of several tables in a relational database management system), all of which are unified and served in one place.

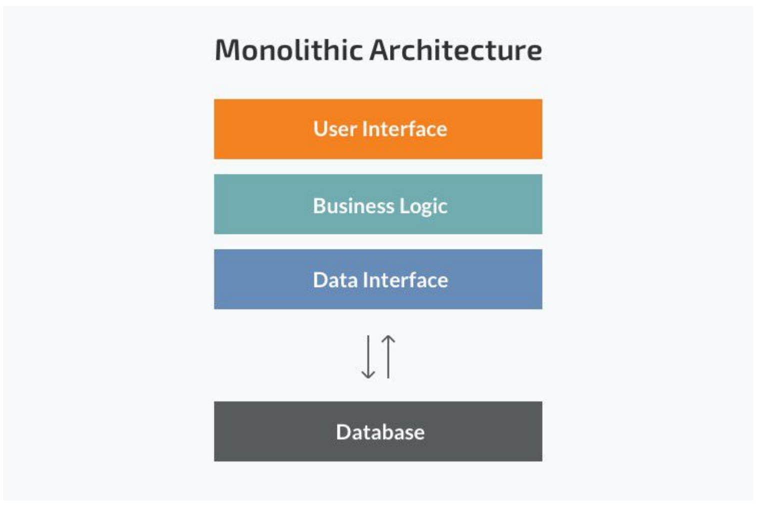


Figure 2.2: Monolithic Architecture
(Gnatyk, 2018)

Advantages

When compared to other models, it is safe to state that monolithic applications are, most of times, easier to implement. This is usually the case due to the absence of orchestration concerns which one faces when following a microservice based architecture.

The existence of a single application, as opposed to several running services, results in a lot less cross-cutting concerns since tasks such as performance monitoring, caching, and logging are made easy and more straightforward. Without the concern of different run-time environments, testing the application also becomes easier as well as debugging it when in the development stages given that you can run end-to-end tests a lot faster.

Another advantage comes in the form of simpler deployment proceedings that are comprised of a simple application build and subsequent release of its entirety. Horizontal scaling is also made easy given the option of running multiple copies of the application under a load balancer.

Disadvantages

Like all the other models, the monolithic architecture does not come without its weaknesses. Although its singular codebase/application concept makes it easier to develop, it presents a major drawback when faced with scaling. A sizeable and complex monolithic solution leads to tight coupling processes resulting in convoluted code as it grows, becoming harder to isolate certain areas for independent scaling since you cannot scale components independently.

The high dependency levels that are also a by-product of this model make it so changes are more time consuming given the extra need for attention in regard to ensuring the new features do not affect existing ones which might break the application.

Although the simpler deployment process was presented as an advantage, there is an inherent drawback to releasing the solution as whole that comes in the form of any change, no matter how small or insignificant, requiring a redeployment of the entire application.

Another disadvantage is made clear when presented with a scenario where one would like to apply a new technology, say programming language, to the project given that in most cases this would require the entire application be rewritten.

2.2.2 Microservices

By contrast with monolithic based applications, in a microservice architecture business logic is broken down into lightweight, interconnected, single purpose services. As such, the project's structure is modular and can be compared to a jigsaw puzzle, where each service represents a single piece. Service-to-service communication is often ensured by means of API.

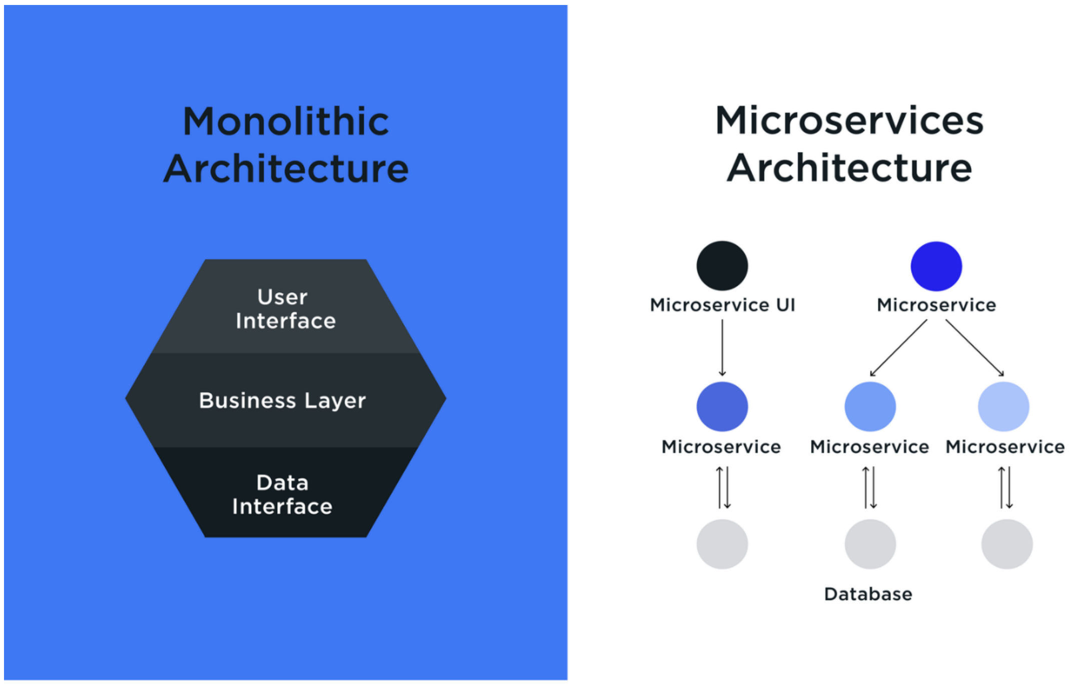


Figure 2.3: Monolithic Architecture vs Microservices Architecture
(Spoonhower, 2020)

Another major difference with this architectural pattern presents itself through the application to database relationship. Whereas in a monolithic application one would typically find a single database instance, with microservices each service has its own database schema which only contains information relevant to the business logic it respectively handles.

Advantages

Given its origin as a solution to most of the issues faced when applying a monolithic architecture, microservices excel in areas where monoliths would struggle. A good example of this juxtaposition becomes evident when analysing the application's complexity. By decomposing the project into smaller, more manageable services, new developments become faster to implement, understand, and even maintain. Releasing changes is also simpler since there are fewer dependencies and microservices can be independently deployed, enabling engineering organizations to use continuous deployment (CD) for their applications.

Unlike the monolithic architecture, microservices present amazing scalability considering the separation of concerns and the ability to develop, enhance, and deploy a single service as opposed to being restricted to scale the entire system. Moreover, there is nothing preventing developers from using separate teams and work with different programming languages/frameworks and storage technologies for the system's different features.

When compared to monolithic architectures, microservices are often referred to as more reliable since a fault in one of the services would not affect the remaining ones, eliminating an application's susceptibility to a single point of failure. Aside from being autonomous, they also promote reusability provided each service is properly documented to facilitate one's understanding of its purpose, often decreasing the project's overall technical debt.

Disadvantages

When analysing the drawbacks associated with a microservices based approach, the first one presents itself at the early stages of a project when the team is faced with the task of designing a distributed system. This requires thorough planning and a great deal of effort since a good design is the foundation of a solid project. Connections between every module and database must be drawn out and set up, resulting in a larger overhead in the beginning stages of the development process.

Cross-cutting concerns are more prominent when enforcing this architecture given the set of tasks that need to be performed independently for each component such as logging, caching, and performance monitoring. There is also a directly proportional relationship between the complexity of the application and the technical debt that needs to be met to fully test it.

2.2.3 Summary

Although faced with a recent decrease in popularity, it would be ill-advised to discard monoliths as a reliable solution to some problems, namely the development of small applications.

On the other hand, a microservice based architecture would be the way to proceed when dealing with large-scale solutions that are prone to evolve as time passes or are comprised of multiple workflows. This architecture is also recommended when an organization has many assets available and wishes to enable the existence of multiple development teams and the usage of different programming languages.

2.3 Single-Page Applications vs Multiple-Page Applications

As evidenced by the increase in demand, web applications (web apps) have been replacing desktop applications and continue to do so in current times. This shift mainly comes from the standard user's desire to have everything available online to access their applications of choice across all devices, as opposed to being limited to a single machine (Neoteric, 2016).

Although there are multiple design patterns available that address web app creation, two of them are often referred to as the main ones: Single-Page Application (SPA) and Multiple-Page Application (MPA). As with all models, each has its benefits and downsides.

However, before deciding which of these design patterns best suits a specific web application, an important first step would be to carefully weigh what content is going to be displayed, what presents the most value to the user and how said content will be presented. This heavily influences the effectiveness of an application namely because users want an intuitive experience when using it, on top of easily being able to find interesting and useful content in a timely manner. This is often referred to as a content-first approach.

As mobile applications thrive, some enterprises are opting for a mobile-first approach when designing their applications, rather than the one previously noted. Thus, added focus is given to smaller screen sizes with a limited number of pixels, shaping the web app. This is then scaled up to desktop size for users on other devices.

Now that we have established the importance of an application's content, let's take a closer look at two main design patterns used in web app creation by briefly analysing them in the coming sub-sections.

2.3.1 Single-Page Applications

Single-Page Applications, or SPAs for short, are web applications that interact with the user by rewriting the web page in a dynamic fashion with data fetched from the web server (Neoteric, 2016).

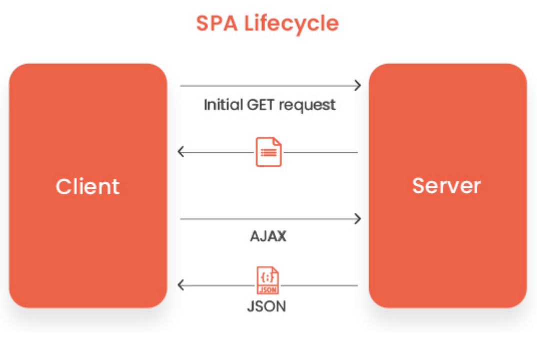


Figure 2.4: SPA Lifecycle
(Shah, 2020)

By doing so the web app avoids page reloads and, consequentially, extra wait time, providing an outstanding user experience. These applications rely heavily on JavaScript (JS) which they use to load data and render the pages in the browser, something made possible through advanced JS frameworks such as Angular and React (Joseph, 2015).

Advantages

By avoiding page reloads, SPAs are generally fast than traditional web applications. Most of necessary resources that comprise the web app (HTML, CSS, Scripts) are loaded a single time throughout the lifespan of the application, only requiring additional requests to attain or provide data.

Aside from the performance improvements, by steering clear of successive roundtrips to load additional logic and render views, SPAs provide a much better user experience with fluid navigation and seamless page transitions.

Debugging is also simplified as the developer can easily monitor network activity, investigate the page's elements, and perform checks on the data associated with each component through natively available tools like Google Chrome's developer's console.

Another upside of Single-Page Applications is the ability to cache information through local storage, increasing performance and even permitting the application to function while offline after the initial request.

Finally, by maintain server logic isolated, SPAs promote backend reusability in instances where a company wishes to develop both a web and mobile application that utilize the same business data.

Disadvantages

When compared to traditional web apps, SPAs are often less secure. A clear example of this is their increased susceptibility to Cross-Site Scripting (XSS), which consists in injecting malicious client-scripts into the application, raising safety concerns. Exposure of sensitive data is another aspect to have in mind since unintended data may be included in the initial request which should not have been exposed to the user. This does not refer to information being easily presented to users unintentionally, but instead to more tech-savvy individuals finding something sensitive in the browser's storage.

Another downside of Single-Page Applications is applicable to public projects that rely on having good search engine optimization (SEO) to raise website traffic and grow. This is due to lack of separate URLs for ever page which prevents most search bots from scanning said pages, complicating optimization proceedings. In recent years, some strides have been made to minimize this issue including Google's support for dynamic page indexing, provided the developers take the necessary precautions to enable this.

2.3.2 Multiple-Page Applications

Multiple-Page Applications, MPAs for short, are often referred to as the traditional web applications. Unlike SPAs, with this model pages are entirely reloaded whenever a user interacts with the web app and triggers a data exchange, both towards the server or back.

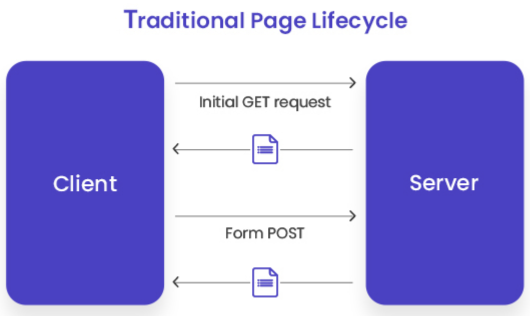


Figure 2.5: Traditional Page Lifecycle
(Shah, 2020)

This behavior could negatively impact user experience if not properly managed, however AJAX introduced the possibility of rendering specific application components instead of the whole page. Nevertheless, this does complicate application development by adding increased complexity and can still be mitigated by network traffic/delay.

Advantages

One of the areas where Multiple-Page Applications excel as opposed to Single-Page Applications is SEO. This design's architecture is native to typical search engine crawlers, providing better control with its multiple pages and specific URLs.

MPAs can also easily include any amount of information regarding services or products with little to no concerns and no built-in page limitations. This in turn makes them better suited for projects that need to display high volumes of data while SPAs might struggle performance-wise.

Another benefit of this model revolves around its native support for browser history. Users are used to a timely response whenever they perform an action such as pressing the back button in their web browser which is facilitated by its mechanisms that attempt to always load a cached version of the previous page, preserving its state. Although SPAs can achieve a similar behaviour, this would require extra developments since it does not take advantage of any existing mechanism, meaning it needs to be mimicked. To do so, things like storing pages in memory and client-side databases are required (Pupius, 2013).

Disadvantages

One of biggest downsides of this design pattern given the current emphasis on mobile applications is the added technical debt to develop them since in most cases this requires the backend of the application be built from scratch. Even when trying to introduce updates to the system, developments often take longer and raise additional concerns like independently ensuring each page is properly secured. Moreover, this model's inherent tight coupling between frontend and backend almost entirely prevent reusability.

As previously stated, performance is another problem faced with MPAs due to the app's need to constantly reload upon user interaction and retrieve multiple resources (e.g. HTML, CSS). AJAX does attempt to lessness the impact of these operations by reducing the scale at which they are performed but it can very rarely achieve the performance levels one sees with Single-Page Applications.

2.3.3 Summary

By evaluating both design patterns for web applications we can first and foremost conclude that neither model is perfect. As most opposing architectural styles in software engineering, each of them excels when faced with specific conditions and have different trade-offs.

If the goal of a company is to develop a dynamic application that uses small data volumes and looks to also work as a base for a future mobile application, then an SPA would be the advisable choice. On the other hand, if a project relies heavily on SEO to thrive and needs to present data on a broad range of services/products (e.g. an online store), perhaps an MPA is best suited for it.

This conclusion serves to further illustrate the importance of an application's content and why due diligence is required in a project's early stages to make such impactful decisions, as it can shape its effectiveness.

2.4 Micro Frontends

Although the debate between monolithic and microservice architectures has become a hot topic of discussion and major strides have been made to make microservices the norm for backend development, frontend applications are still lagging architecture-wise. The unprecedented speeds companies are currently expected to deliver upon also generate a necessity for adoption of agile methodologies.

Unlike backend programmers, frontend developers and architects are presented with fewer options to choose from, mainly relying on single-page applications (SPAs), server-side rendering (SSR) applications or static HTML pages. As a project grows, all these models inevitably end up becoming monoliths comprised of large codebases, tight coupling, and countless dependencies.

Much like a microservices architecture, micro frontends are looking to apply modularization to client side solutions in order to address the aforementioned issues as well as others that triggered the changes to server-side software development by transforming the monolith into smaller chunks that can be independently run, developed and deployed. By achieving this goal, development teams would be able to build loosely coupled services on their own without all the cross-development concerns when working on a single codebase.

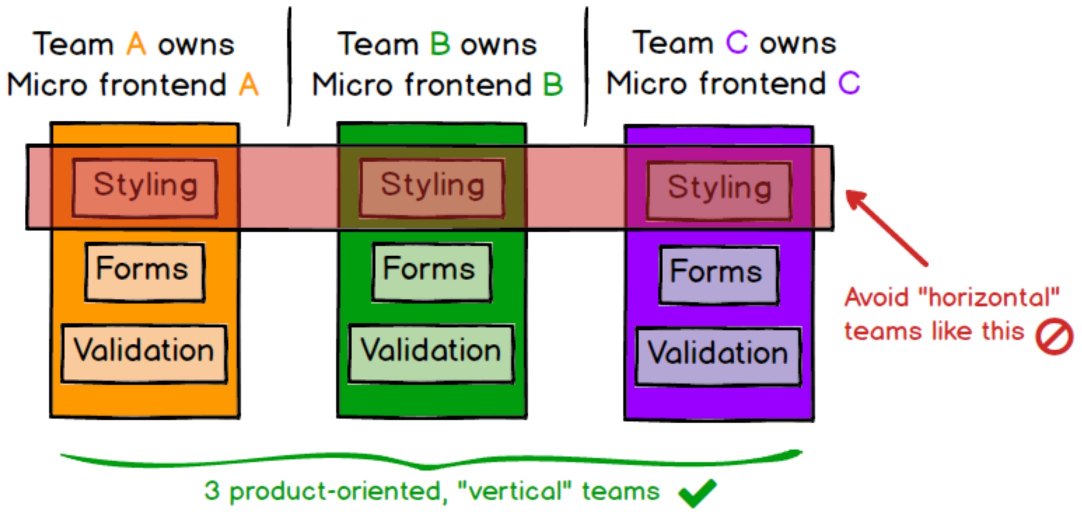


Figure 2.6: Micro Frontend Team Composition
(Jackson, 2019a)

As such, the goal would be to divide an application vertically, as demonstrated by Figure 1, as opposed to the traditional horizontal model that separates developers by their technical capabilities. With such a level of autonomy, teams can take full ownership of a specific business sub-domain and manage it from the initial design stage all the way through its production.

Before presenting the available approaches that could be labelled as micro frontends, let's discuss this architecture's advantages and disadvantages (Jackson, 2019a).

2.4.1 Advantages

As previously stated, the micro frontend architectural pattern is loosely based on the ever so popular microservices approach for backend development, therefore most of, if not all its benefits, will be conjoint. Here are the main advantages this model provides:

Scalability

Monolithic frontends often weigh down development teams when attempting to perform changes or implementing new features. The added technical debt to complete these tasks when faced with a large and complex codebase ends up being quite costly both resource-wise and timewise. Some companies are even faced with the decision of rewriting an entire application due to accumulated developments that have since become obsolete.

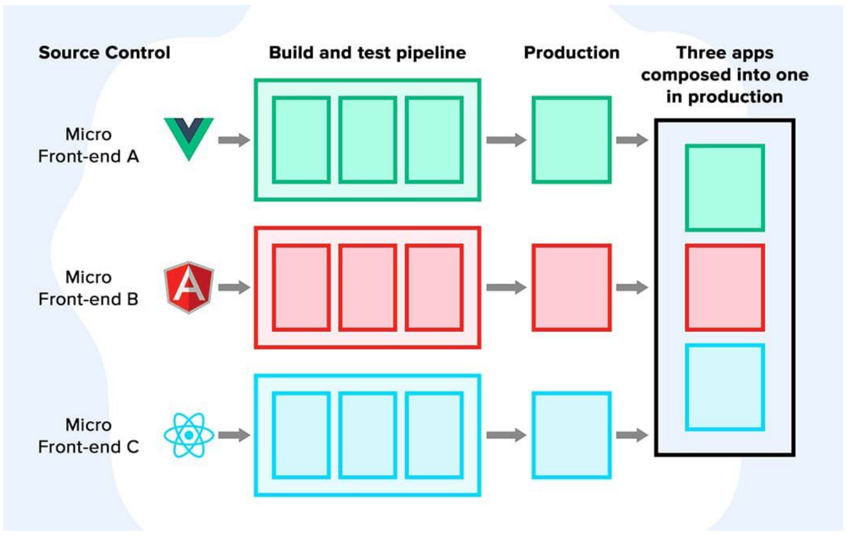


Figure 2.7: Micro Frontend Deployment
(Media, 2020)

By splitting the project into smaller independent components, teams can work in different codebases, release their changes through a separate build pipeline and deliver constant incremental upgrades. They are even able to make case-by-case decisions on each component that address its specific needs. Using different technological stacks also becomes an option since everything is now isolated.

Future Proofing

As teams are free to choose their own technology of choice, the appearance of a new coding standard or framework can be easily analysed, tested, and integrated into a micro frontend architecture. Abandoning old practices or updating them is also easier in such a smaller scale contrary to the effort of rewriting an entire monolith. This in turn makes the application future proof.

Deployment

In agreement with microservices, the ability to independently deploy micro frontends is of paramount importance. When a given component is finished and ready to release, the team responsible for it should be permitted to do so with little to no concern regarding parallel developments.

Each micro frontend must have its own designated continuous delivery pipeline responsible for building, testing, and deploying it. This reduces the associated risk given the smaller scopes of deployment per component.

Testability

Performing changes and subsequentially testing them in a monolithic architecture is often more complex due to the possibility of side effects throughout different parts of the solution.

Much like the deployment advantages, the reasoning behind the statement “testing is easier with micro frontends” mainly surges from the decreased scope one would have to test and the lack of interdependencies across codebases.

Reusability

Although reusability is an overall benefit that presents itself with micro frontends, it becomes more indisputable for enterprises that build multiple applications that possess common workflows.

Organizations can easily take advantage of single components that address these workflows, saving valuable time and effort in the meantime, instead of recreating them. An example of this might be in enterprises that often develop applications that require a checkout/payment process which trigger multiple business validations.

Onboarding

When discussing the benefits of a micro frontend architecture, an often-overlooked advantage is the facilitation of the onboarding process.

Introducing a developer to a monolithic codebase, from business logic, to the application’s patterns, logic, and common practices is traditionally a lot more tasking and takes a set amount of time (depending on the individual’s proficiency). It is safe to assume that the duration for this sequence of events is directly proportional to the project’s proportions meaning that, as the solution grows, so does the length of the onboarding procedures. Bearing that in mind, micro frontends expedite the enrolment of new team members by minimizing the time and resources required for them to fully understand the application.

Team Composition

The main purpose of a development team is to provide value to the customer as quickly and effectively as possible while meeting all the quality standards. Figure 2.8 presents two known types of teams, Specialist Teams and Cross Functional Teams.

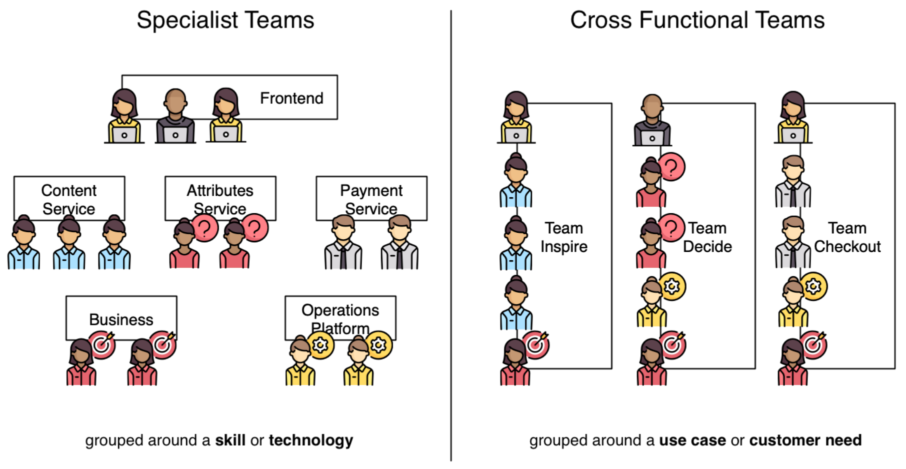


Figure 2.8: Specialist Teams vs Cross Functional Teams
(Geers, 2020)

By having vertical teams that address a specific business sub-domain for a given project, their members can plan and coordinate the necessary work with little to no outside interference. If a team were to be built by focusing on its members technical capabilities, cross-team dependencies would rule the overall project and translate into decreased effectiveness and make the application harder to maintain.

Incentivizing cooperation between the members of an interdisciplinary team can also lead to increased productivity as different points of view may result in more creative and fruitful solutions.

2.4.2 Disadvantages

Much like the advantages, micro frontends also share most of microservices' issues. No architecture is ever without fault and these are this model's hurdles:

Dependencies and Payload Size

Having different components that make use of different, or even the same, technological stacks can severely cripple application performance. For example, if a project is comprised by multiple micro frontends, some built on Angular and others on React, the size of the payloads that need to be sent to the browser over network calls for them to function is much larger. Ensuring the same framework is used across the different components doesn't solve this issue either since each individual instance must load the respective dependencies (e.g. five micro frontends would load a copy of Angular five times).

Consumers have become more demanding and judgemental towards the overall experience when using a service and research shows user engagement is heavily influenced by page performance. As an example, in 2016 Pinterest rebuilt its pages with a focus on performance and achieved 40% decreases in Pinner wait time which presented a 15% increase in conversion rate to sign up and SEO traffic (Meder et al., 2017).

These micro frontend performance concerns are not an easy issue to address without going against some of the architecture's guidelines. On one hand, teams are intended to be completely autonomous and should be able to independently develop and deploy their applications, on the other, common dependencies should be shared between different components to avoid duplication. However, by sharing dependencies, a coordination effort is introduced to the project since something like a version update or breaking change could result in a full-scale upgrade that englobes every component.

Although this sounds like a serious problem, there is still a silver lining to it. Even with duplicate dependencies, page load times might possess better performance than a monolithic frontend would. In the latter, the standard use case is to download the source code in its entirety, along with every dependency, by the first page the user lands on. By splitting this throughout different components, the browser will only load the specific page's source code (in a project with a micro frontend per page), often fulfilling the task at much faster rates.

User Interface Consistency and User Experience

Ensuring a consistent user interface (UI) across an application may prove challenging in the absence of guidelines since different teams may decide upon the usage of distinct frameworks. It is important that, even though the application is split into separate micro frontends, the resulting product be a seamless integration of every component and provide a compatible user experience (UX) throughout.

A somewhat simple approach to this issue would revolve around having a shared CSS stylesheet which could attempt to maintain a component's look and feel in accordance with the rest of the application. However, this does come at a cost performance-wise as it creates dependencies and redundancy when it comes to loading said stylesheet to the browser, raising similar concerns to the ones described in the previous sub-section of this paper.

A universal solution, and one that would avoid the overhead presented by the previous one, would be to use a style guide (e.g. Bootstrap, Material Design) along with a list of rules/standards to develop by. Each team would then follow these regulations with an emphasis on cross-team communication to achieve a consistent UX, while sacrificing some of its autonomy regarding style choices as a trade-off.

Governance

As with microservices, having a system use a more distributed architecture as opposed to a single monolith will create increasing governance concerns. While it does avoid other issues, namely having a single point of failure, it creates more items to manage and monitor (e.g. repositories, pipelines).

2.5 Micro Frontend Integration

Since the idea behind Micro Frontends is to break down a monolith into several smaller, more manageable components, orchestrating said components is required to have them work together as a whole, giving the user the impression of a single application.

Micro Frontend integrations can be divided into two categories: build time integration and runtime integration. As the names would indicate, approaches vary according to when the actual integration happens, and while the former does this at build time, the latter executes it at runtime.

2.5.1 Build Time Integration

A build time integration based approach looks at Micro Frontends as packages that can published and later on included as a dependency into a single container responsible for their availability.

Although having a single bundle that contains every dependency may initially look like a good idea as it prevents code duplication, it reverts the application back to a monolith at its release stage since every component must be re-compiled to introduce a change in a single one.

This makes build time integration a less than optimal approach since it, in a way, goes against the Micro Frontend guidelines by re-introducing tight coupling in one of its integral stages (release). Given that, it's advisable to move past this technique and take a deeper look into runtime integration and its variants.

2.5.2 Runtime Integration

As previously stated, Micro Frontends can also be orchestrated at runtime. This approach is a lot more versatile than the aforementioned one since it can be divided even further as either server-sided, edge-sided, client-sided or a combination of each.

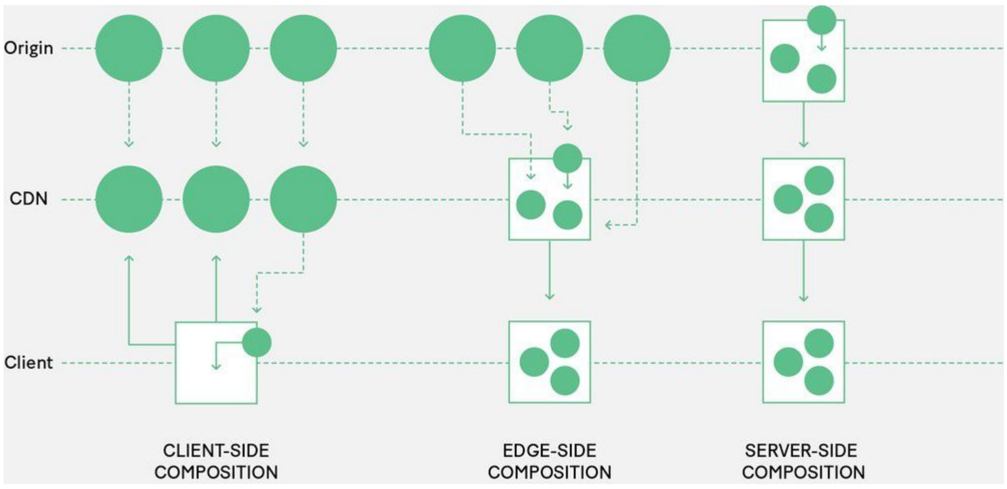


Figure 2.9: Micro Frontend Composition
(Mezzalira, 2020)

In the following subsections, some of the most established techniques to employ these types of integrations will be described, along with their benefits and downsides.

Server-Side Composition

Server side composition is made up of HTML rendering mechanisms that are responsible for assembling multiple view fragments/templates into the requested page, posteriorly returning it to the browser when finished. As the name would indicate, this type of processing occurs in the server-side and only provides the finalized product to the client, allowing for faster page loading times and avoiding the excessive use of JavaScript to handle component rendering (Jackson, 2019b).

Server side composition allows for full control over the web app. It enables developers to attach data directly on the server as opposed to having the frontend rendering and network requests that retrieve said information be separate actions, improving performance in the meantime. Highly cacheable content can also be delegated to a Content Delivery Network (CDN) to avoid roundtrips to the application's server, once again improving performance and overall user experience.

Edge-Side Composition

Unlike with server-side composition, edge-side composition assembles and renders view fragments at the CDN level by obtaining each micro frontend for their respective source and delivering the final assembled page to the client (Mezzalira, 2020).

Edge Side Includes (ESI) is a markup language developed by Akamai and Oracle whose main purpose is to enable edge level dynamic content assembly (Tsimelzon et al., 2001) in order to promote scalability. By inserting ESI element tags into HTML (e.g. esi:include), also known as directives, a set of instructions are being provided to the processor to later on execute. With the aforementioned example tag, a request would be carried out to retrieve the include tag's content and then insert it in the page.

Although it might seem like a good approach, there are some inherent disadvantages with, starting from the lack support for ESI by multiple CDNs or even different implementations that can lead to refactors. Moreover, CDNs are most adequate for web applications that are comprised of a lot of static content since they can take full advantage of their caching mechanisms.

When developing an application that's constantly retrieving up to date information, the introduction of a CDN as it acting as a simple reverse proxy, creating unnecessary middlemen and, consequentially, adding latency to network requests. Another aspect to bear in mind is, given this approach's nature, it makes it more fitting for horizontal teams that handle different aspects of a specific page simultaneously, going against one of Micro Frontend's principles of having vertical teams specialized in a specific business sub-domain.

An example of properly enforcing Edge Side Composition would be an e-commerce website that needs to display a large product catalog, as is the case with IKEA which uses this technique to great effect.

Client-Side Composition

The two available variants of runtime integration are often divided according to the application's setting and its priorities based on predominant usage. Client-side orchestration excels in web apps that need to promptly react to user input due to faster response times since a roundtrip to the server is avoided.

The idea behind this type of composition is to have a container application load the required Micro Frontend according to necessity by updating the HTML structure of the web app dynamically (Miller & Osmani, 2020).

iFrames

An HTML Inline Frame Element, or iFrame for short, constitutes a nested browsing context by means of embedding a separate HTML document inside the current one (Mozilla, 2020).

This element enables the approach most would deem as the simplest one since its purpose is already to include smaller components into a page. Another positive with iFrames is the built in isolation capabilities since each embedded context has its own DOM, preventing mix-ups with stylesheets, variables or even namespaces, promoting low coupling across the application.

Managing an iFrame layout does present some caveats, namely when it comes to its height since an explicit value is required. However, there are existing workarounds through some additional scripting that emulate dynamic behavior.

Moreover, cross-container communication requires dedicated mechanisms since the built in isolation capabilities prevent direct information flow.

JavaScript

A JavaScript based approach leans on a very common technique in current web developments which is to update by means of replacement or appendage of script tags to the Document Object Model (DOM).

Each tag represents a separate Micro Frontend which exposes an entry level function responsible for rendering it upon being called by the top-level container application that also decides where the component is rendered (Jackson, 2019b).

As opposed to the deployment dependencies that surface with build time integration, each JavaScript file can be independently released, avoiding the need for a conjoint re-compilation before doing so. Another benefit, this time in regard to the iFrame approach, is the ability to have data flow into or out of the JavaScript functions, facilitating integration and ridding the project of a dedicated communication mechanism unless of course, the team decides it would be advantageous.

Angular and React are a couple of examples frameworks that take advantage of this procedure and have since popularized it in the web application development industry.

Web Components

The term Web Components refers to "a set of web platform APIs that allow you to create new custom, reusable, encapsulated HTML tags to use in web pages and web apps" ("What are web components?", 2019). These same custom components can then be used across any framework, as they are globally supported, irrespective of the environment.

There are four main concepts attach to Web Components:

- Custom Elements – allow the development of custom components, including their behavior, that can posteriorly be used through their respective HTML tag.
- Shadow DOM – attaches an isolated DOM tree to an element to prevent cross-element concerns (e.g. stylesheets) and keeps its features private.
- ES Modules - defines the inclusion and reuse of JS documents in a standard based, modular, performant way (WHATWG, 2021).
- HTML Templates – allow the specification of markup templates than can be instantiated at runtime at will. Web Components provide full freedom to create proprietary components while maintaining interoperability. Although this presents an initial development overhead when compared to the usage of proprietary component libraries, it promotes reusability which will in turn compensate the added technical debt. The absence of technology barriers, enables the development of components with different stacks, consequently falling in line with one of the guidelines for Micro Frontends.

A drawback to take note is that some older browsers do not support the usage of a Shadow DOM.

Routing Distribution

Routing distribution based solutions are comprised of several distinct applications (Micro Frontends), each with their respective routes, that depend on navigation to become integrated. This implicates the existence of a component per page which requires a full refresh to render, and a tool that handles the actual routing distribution, Nginx being one of the most popular ones.

This methodology suffers from recurring code duplication, from stylesheets to actual business logic, since the components are often not granular enough given their proportions (a full page). The inability to share a framework's source code may also result in a user having to constantly download dependencies whenever a page navigation occurs.

Additional management and orchestration issues also surface given that cross-team communication is required when performing actions like a user interface style update.

2.5.3 Summary

In a very similar fashion to the microservices vs monolithic debate, when faced with the advantages and disadvantages provided by Micro Frontends and their monolithic codebase counterpart, the former appears to come out on top. However, a somewhat identical conclusion applies as there is no ideal solution when it comes to frontend architecture.

Enterprises should consider whether Micro Frontends is the best approach to use on a given project by carefully analysing important variables like the following:

- Do you possess the required resources to successfully manage this kind of application?
- Does the project justify taking a micro frontend-based approach or would a monolithic architecture suffice?
- If the decision has been made to proceed with micro frontends, which parts of the application will be promoted to one as opposed to being a generic frontend component?
- Do you have guidelines to ensure UX consistency across the application?

A lot more examples of important questions such as the previous ones could be presented but the main point to get across is that, before diving head first into a new type of architectural style, each organization should perform their due diligence and arrive to a decision. There's no universal “one size fits all” approach in software architecture and each project should be looked at objectively.

Chapter 3

Value Analysis

By definition, a Value Analysis (VA) is "a systematic and function-based approach to improving the value of products, projects, or processes" (Rich & Holweg, 2000). It uses a combination of analytical techniques with the sole purpose of identifying possible ways to achieve specific objectives.

This section aims to assess the value of this dissertation by initially analyzing the front-end of innovation using the New Concept Development Model (NCD). Secondly, the solution's value and perceived value are presented, followed by a value proposition. The business model canvas is then introduced, finalizing the chapter with the application of the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS).

3.1 Innovation Process

Due to the growth of the software development industry, enterprises are often placed under severe pressure to finalize projects in a swift manner while still maintaining a certain degree of quality. The ability to keep up with customer requests becomes imperative, as they can often easily transition to another company to meet their demands. The keyword for success in such an industry is innovation, as it is regularly used to gain the upper hand on other contenders, which is the main reason behind many companies betting on improving their practices to attain a competitive edge.

To facilitate this endeavor and attempt to guarantee the development of a valuable product, a model was designed, demonstrated in Figure 8. The model itself presents the innovation process as three separate stages: Fuzzy Front End (FFE), New Product Development (NPD), and Commercialization.

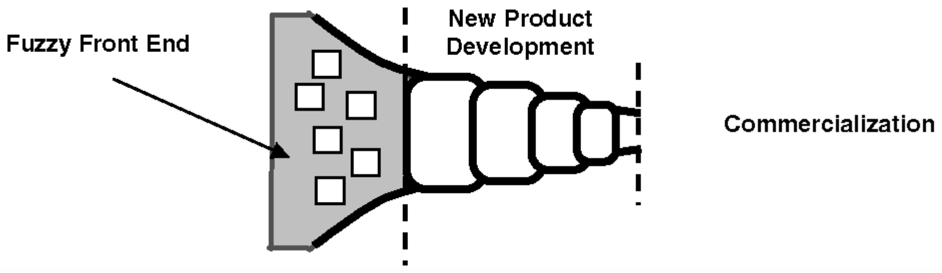


Figure 3.1: Innovation Process by P. A. Koen
(Koen et al., 2002)

As shown in the illustration, the starting point of the innovation process is the Fuzzy Front End stage, which looks to originate a concept based upon an idea, opportunity, or actual business. This first step is often regarded as "one of the greatest opportunities for improvement of the overall innovation process" (Koen et al., 2002).

The New Product Development stage addresses the implementation of the conceptualized ideas that originated from the preceding step by transforming them into actual products. NPD requires an understanding of the targeted market, including its nature and customer demand.

When comparing both stages, one could state they are intrinsically different as the FFE requires thorough understanding and disparate reasoning, being often categorized by its unpredictability and uncertainty, as opposed to the sequential, well-practiced routine which the NPD thrives upon, making it a more disciplined stage. The final stage in the innovation process is Commercialization whose main purpose is to sell the finalized product to the studied market segments.

3.1.1 New Concept Development Model

To account for the lack of coinciding terms when comparing FFE practices and mitigate interpretation/understanding complications, a theoretical construct was created whose designation is New Concept Development (NCD) model.

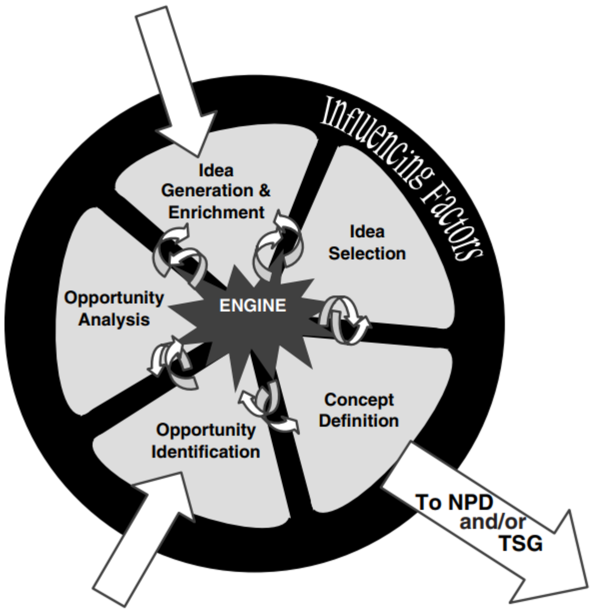


Figure 3.2: New Concept Development Model
(Koen et al., 2002)

The NCD provides a more holistic perspective of the front end and divides it into three distinct areas:

- The engine (or center of the model) which accounts for the primary business aspects that include culture, vision, leadership, and strategy while also being responsible for driving the five activities of the front end.

- The Fuzzy Front End elements, comprised of five activities: opportunity identification, opportunity analysis, idea genesis, idea selection and concept and technology development.
- The influencing factors which are mostly of external nature, often unpredictable, and can vary from competitor and customer constraints to political, economic, social, technological, environmental, or legal concerns.

3.1.2 Opportunity Identification

Opportunities can present themselves in multiple ways and are often driven by an enterprise's business goals. Examples of opportunities can include new product features, improvements to development processes (e.g. increased speed and reduced costs) or even advancements that allow the organization to attain a competitive edge.(Koen et al., 2002)

As described in a more detailed fashion on section 1.2, current frontend development practices still rely on the traditional monolithic architecture which is widely known for its limitations (e.g. scalability, tight coupling, among others). By transitioning to a Micro Frontend based approach, one could go about fixing these issues.

The adoption of this architectural model also falls inline with current emphasis on agile methodologies and incremental upgrades which make the development and deployment process more efficient and faster, something very important in today's high demand technological world.

3.1.3 Opportunity Analysis

After introducing the opportunity, the current stage of the NCD model looks to assess its potential and viability. Typical analysis is based upon market assessments, research, and statistical reports to detail the opportunity's value further and place it into perspective.

As the genesis of the Micro Frontend is still somewhat recent, thorough information that provides conclusive evidence on its value is difficult to find. A survey performed by the Software House saw 4500 developers give their opinion on the concept and confirmed that only approximately 24.4% had any previous experience with it (Mamczur et al., 2020). Nevertheless, only 20% of respondents agreed with a statement that dictated Micro Frontends would disappear in three years' time, showing promise for the concept.

Several large organizations have also come forward and confirmed their adoption of this architecture, among which are IKEA, DAZN, Spotify and Soundcloud to name a few. Having an enterprise's public backup helps promote the idea, which usually results in increased adoption and over-time improvements as different companies use different approaches and share their insights. This is something we saw with the rise of microservices when companies like Netflix and Amazon adopted the concept and raised the tech community's confidence in it. Moreover, the intrinsic connection between Micro Frontends and microservices can subsequently lead to conjecture that the adoption of one implicates the other.

According to a study performed by Camunda in 2018, nearly 63% of surveyed enterprises (which totaled 354) are building applications using microservices, 60% of which justified doing so "to achieve a faster time to market for new products and services". The provided justification falls in line with the mainstream adoption of DevOps throughout the software development industry, as its main purpose is to streamline software development processes by shortening their life cycle and enable continuous integration, testing, and delivery pipelines. This makes Micro Frontends the next logical step in application development as by adopting them, enterprises can be rid of the clunky frontend monolith and all dependencies that come with it, taking another step towards decentralization.

3.1.4 Idea Generation & Enrichment

When discussing an idea, one often correlates the term to the actual aim or purpose of a specific course of action. The existing issues originating from frontend monoliths seem to prevent some of the advancements offered by microservices since it becomes nearly impossible to prevent the inherent dependency of having a single client-side application for every service. A problem of this nature then triggered the question: How can we avoid this? Can we replicate what was done with microservices to the frontend of the application?

3.1.5 Idea Selection

It is uncommon for businesses to suffer from a shortage of new ideas, even in the face of adversity. The problem usually lies in the selection process when a decision must be made regarding which idea to pursue (Koen et al., 2002). Hence, the presented idea was analyzed based on its merit and what kind of value it could provide.

The intent with Micro Frontends is to look past the typical frontend monolith and move towards developing modular components that pertain to a designated business sub-domain in a similar fashion to microservices. By doing so, components present scalability, independency (from implementation to deployment) and can be developed by vertical teams to enable full ownership of its respective sub-domain leading (end to end implementation) while also incentivizing the usage of agile methodologies.

3.1.6 Concept Definition

The goal is to integrate the microservices and Micro Frontend architectural patterns to exploit the viability of such a solution in web app development. As such, research will initially be conducted to analyze current integration strategies, followed by a decision on which one to pursue for the proof of concept application.

The application itself will target the Human Resources Management thematic since it is the main area of interest for the company co-orientating the dissertation and whose nature engulfs multiple sub-domains (e.g. recruitment, compensation, talent management), making it a solid candidate for modularization.

3.2 Value

The term value can be defined in a plethora of ways since its definition may vary according to the context it is used upon. Contrary to popular belief, value is not always a matter of cost but instead ways of conveying utility.

Given that this dissertation tackles the adoption of Micro Frontends as a means to modernize web applications, there is no specific product or service being listed to outside customers. With that in mind, this chapter instead focuses what kind of value is offered to potential users of this architectural approach.

3.2.1 Perceived Value

The perceived value can be defined as the relationship between perceived benefits and sacrifices. For a customer to deem the solution has value or not, the overall benefits must have more weight than the sacrifices. It is important to note that each customer segment may possess different value perceptions regarding the same product or service (Ulaga & Eggert, 2006).

As discussed in section 2.4.1 of the current document, there are several advantages associated with a Micro Frontends architecture. Although the benefits were presented from a technological perspective, they inherently provide value to development teams and managers alike as they often simplify existing or future processes.

By isolating the application into separate components that target a specific business subdomain and have their respective team, tasks such as monitoring, and error handling become intrinsically faster. This type of improvement can then be translated to better customer support as speed is one of the most important metrics in problem-solving evaluation, while also reducing the allocated resources to perform such chores which impacts maintenance costs.

Moreover, a Micro Frontend architecture paves the way for continuous integration and continuous delivery pipelines that make delivering code changes a more frequent and reliable process, achieving faster time to market for the organization's new products and services. However, having these automated pipelines does come at an increased development cost. The same can be said infrastructure wise, as having several independent components with their respective foundations will increase costs.

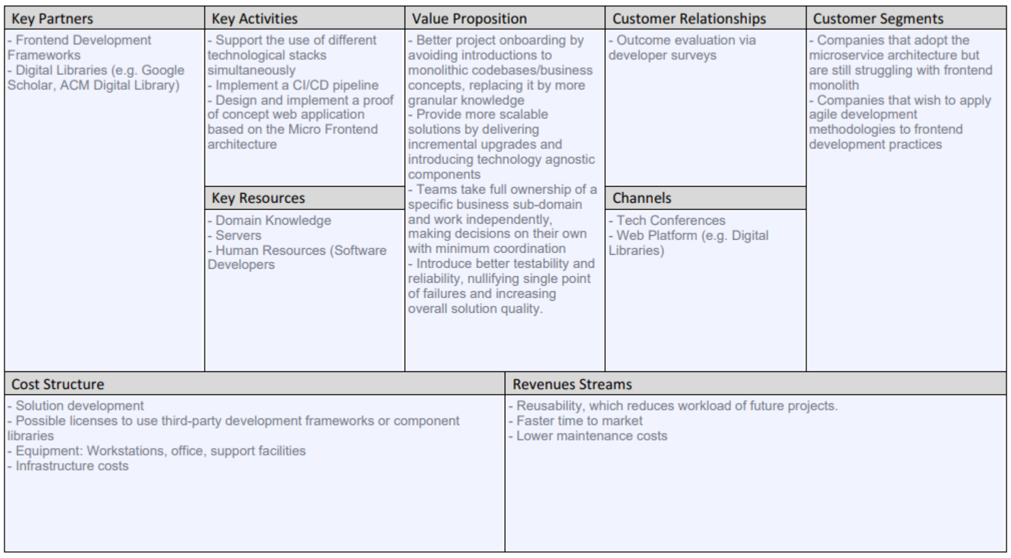
3.2.2 Value Proposition

The Business Model Canvas, proposed by Alexander Osterwalder, is a strategic management tool which uses nine pillars to visualize a business idea or concept. By examining the canvas, one should be able to describe how an organization creates, delivers, and obtains value to and from the market (Osterwalder & Pigneur, 2006).

It defines who is the customer, what is the customer's problem, how is the business going to address that problem, and how is it going to attain revenue.

Following is a description of each of the nine pillars that comprise the Business Model Canvas:

- Customer Segments - Identify the groups of customers that the business aims to serve.
- Value Proposition - The products/services that generate value for the identified customer segments and what is compelling/unique about them.
- Channels - The channels used to communicate and deliver value to the customer.
- Customer Relationships - How the organization interacts with its customers.
- Revenue Streams - The possibilities of revenue from its value proposition.
- Key Resources - The main strategic assets needed for the business model to succeed.
- Key Activities - The main strategic things needed for the business model to succeed.
- Key Partners - The partners and suppliers needed for the business model to succeed.
- Cost Structure - The main cost drivers.


Key Partners
- Frontend Development Frameworks
- Digital Libraries (e.g. Google Scholar, ACM Digital Library)

Key Activities
- Support the use of different technological stacks simultaneously
- Implement a CI/CD pipeline
- Design and implement a proof of concept web application based on the Micro Frontend architecture

Key Resources
- Domain Knowledge
- Servers
- Human Resources (Software Developers

Value Proposition
- Better project onboarding by avoiding introductions to monolithic codebases/business concepts, replacing it by more granular knowledge
- Provide more scalable solutions by delivering incremental upgrades and introducing technology agnostic components
- Teams take full ownership of a specific business sub-domain and work independently, making decisions on their own with minimum coordination
- Introduce better testability and reliability, nullifying single point of failures and increasing overall solution quality.

Customer Relationships
- Outcome evaluation via developer surveys

Channels
- Tech Conferences
- Web Platform (e.g. Digital Libraries)

Customer Segments
- Companies that adopt the microservice architecture but are still struggling with frontend monolith
- Companies that wish to apply agile development methodologies to frontend development practices

Cost Structure
- Solution development
- Possible licenses to use third-party development frameworks or component libraries
- Equipment: Workstations, office, support facilities
- Infrastructure costs

Revenues Streams
- Reusability, which reduces workload of future projects.
- Faster time to market
- Lower maintenance costs

Figure 3.3: Business Model Canvas

As initially stated in section 3.2, given that this dissertation discusses the viability of an architectural approach in the web application development field, the business model canvas may deviate slightly from usual examples as there's not an exact product being proposed.

Bearing this in mind, the canvas in Figure 11 illustrates that the value proposition for this idea revolves around the advantages it provides for software development enterprises, which are the targeted customer segments, and displays the activities and resources required to implement it. The same applies to the cost structure and key partners represented.

Lastly, in regard to revenue streams, the focus is to show the monetary avenues that an architectural change can pave for a company, mainly from a point of reusability, faster time to market, and lower maintenance costs.

3.2.3 Technique for Order of Preference by Similarity to Ideal Solution

The process of decision making holds paramount importance both in a business and personal setting. In order to aid companies through this process several multi-criteria decision analysis methods have been developed, one of each is the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS). TOPSIS has a company indicate which attributes will be considered when performing a comparative analysis, each of which must have a weight associated to it. Two important concepts in this decision making method are:

- Positive ideal solution - the sum of all the best value that can be achieved for each attribute.
- Negative ideal solution - all the worst value achieved for each attribute.

The distance between alternatives and the two hypothetical solutions is then measured to attain a rank for each given that selected solutions should be the ones closer to the positive ideal solution while also being the farthest from the negative ideal solution (Rahim et al., 2018).

The first step one must take to apply the TOPSIS decision making method is to determine which evaluation criteria will be used and their respective weights.

Table 3.1: Criteria Weights and Rating

Criteria	Weight	Rating
Independent Development and Deployment	20%	5 (very good) - 1 (very bad)
Technical Support	10%	5 (very good) - 1 (very bad)
Performance	15%	5 (very good) - 1 (very bad)
Technical Complexity	10%	5 (very good) - 1 (very bad)
Development Costs	25%	5 (very good) - 1 (very bad)
UX Impact	20%	5 (very good) - 1 (very bad)

After taking into consideration all of the defined ratings and weights, it is expected that the decision maker rates all the criteria topics in accordance with the performed research and brief explanation presented on section 2.5.2.

As it can be concluded by observing the table 3.2, all the previously mentioned alternatives are classified in a qualitative scale regarding each of the considered criteria.

Table 3.2: Criteria Qualitative Rating

	iFrame	JavaScript	Web Components	Routing Distribution
Independent Development and Deployment	Very Good	Very Good	Very Good	Good
Technical Support	Very Good	Very Good	Good	Very Good
Technical Complexity	Good	Very Good	Good	Good
Development Costs	Good	Very Good	Average	Very Good
UX Impact	Very Good	Good	Good	Bad
Performance	Very Good	Good	Good	Bad

After classifying each attribute and defining its weight, the decision maker rates each alternative. A 5 to 1 conversion table was applied over the qualitative ratings with the following values:

- 5 - Very Good.
- 4 - Good.
- 3 - Average.
- 2 - Bad.
- 1 - Very Bad.

The post conversion decision matrix is then as follows:

Table 3.3: Criteria Rating: Qualitative to Numeric Translation

	iFrame	JavaScript	Web Components	Routing Distribution
Independent Development and Deployment	5	5	5	4
Technical Support	5	5	4	5
Performance	4	5	4	4
Technical Complexity	4	5	3	5
Development Costs	5	4	4	2
UX Impact	5	4	4	2

The qualitative to numeric translation enables the execution of all the TOPSIS steps to then attain a conclusion in regard to which of the considered alternatives would be best suited for the current problem.

The next step is to normalize the decision matrix by applying the following formula:

$$r_{ij} = \frac{x_{ij}}{\sqrt{\Sigma_{i=0}^{m}(x_{ij})^2)}} \tag{3.1}$$

Table 3.4: Normalized Decision Matrix

	iFrame	JavaScript	Web Components	Routing Distribution
Independent Development and Deployment	0.52	0.52	0.52	0.42
Technical Support	0.52	0.52	0.42	0.52
Performance	0.47	0.59	0.47	0.47
Technical Complexity	0.46	0.58	0.35	0.58
Development Costs	0.64	0.51	0.51	0.26
UX Impact	0.64	0.51	0.51	0.26

Post-normalization, the weighted normalized matrix is calculated by multiplying each rating element by the weight of its respective criteria:

Table 3.5: Weighted Normalized Decision Matrix

	iFrame	JavaScript	Web Components	Routing Distribution
Independent Development and Deployment	1.04	1.04	1.04	0.84
Technical Support	0.52	0.52	0.42	0.52
Technical Isolation	0.71	0.89	0.71	0.71
Technical Complexity	0.46	0.58	0.35	0.58
Development Costs	1.6	1.28	1.28	0.65
UX Impact	1.28	1.02	1.02	0.52

In order to obtain the distance between alternatives, the next logical course of action it to determine the positive ideal and the negative ideal solutions. The ideal solution set is comprised of the maximum values for each criteria, while the negative ideal solution consists of the minimum values.

- Ideal Solution = 1.04, 0.52, 0.89, 0.58, 1.6, 1.28
- Negative Ideal Solution = 0.84, 0.42, 0.71, 0.35, 0.65, 0.52

For the fourth step, the goal is to define the separation from the ideal solution S_i^+.

The result of the following formula application is present at the table 3.6.

$$S_i^+ = \Sigma_{j=1^n}(v_{ij} - v_{ij+}^2)^{1/2} \tag{3.2}$$

Table 3.6: Separation from the Ideal Solution

	iFrame	JavaScript	Web Components	Routing Distribution
Independent Development and Deployment	0.00	0.00	0.00	0.19
Technical Support	0.00	0.0	0.01	0.00
Technical Isolation	0.00	0.03	0.00	0.00
Technical Complexity	0.01	0.00	0.05	0.00
Development Costs	0.00	0.10	0.10	0.90
UX Impact	0.00	0.07	0.07	0.58

The next step regards the separation from the negative ideal solution. The following formula produces the result that can be observed on table 3.7.

$$S_i^- = \Sigma_{j=1^n}(v_{ij} - v_{ij-}^2)^{1/2} \tag{3.3}$$

Table 3.7: Separation from the Negative Ideal Solution

	iFrame	JavaScript	Web Components	Routing Distribution
Independent Development and Deployment	0.04	0.04	0.04	0.00
Technical Support	0.01	0.01	0.00	0.01
Technical Isolation	0.00	0.03	0.00	0.00
Technical Complexity	0.01	0.05	0.00	0.05
Development Costs	0.90	0.40	0.40	0.00
UX Impact	0.58	0.25	0.25	0.00

Next step is the calculation of the relative closeness to the positive ideal solution. This process involves the application of the following formula:

$$C_i = \frac{S_i^-}{(S_i^+ + S_i^-)} \tag{3.4}$$

Upon application, this is the resulting table (3.8):

Table 3.8: Relative Closeness to Ideal Solution

	iFrame	JavaScript	Web Components	Routing Distribution
S_i^+	0.01	0.20	0.23	1.67
S_i^-	1.54	0.78	0.69	0.06
$S_i^+ + S_i^-$	1.55	0.98	0.92	1.73
$\frac{S_i^-}{(S_i^+ + S_i^-)}$	0.99	0.80	0.75	0.03

In accordance to the relative closeness, one can conclude that the two best alternatives based on the proposed criteria are the iFrame (0.99) and JavaScript (0.80) as their results are the ones closest to the positive ideal solution, with Web Components following as close third contender.

3.3 Summary

Upon an initial presentation of the proposed solution through the application of the innovation process model, the opportunity's genesis was judged and its motivations explained, subsequently contextualizing it against current development practices.

Afterwards, although in a somewhat unorthodox fashion given the nature of the idea (no specific product or service listed to outside customers), the value and value proposition were properly detailed to synthesize its merit and lay out the means through which the user can capitalize on it.

With the range of alternative solutions that can be achieved with a Micro Frontend architecture, a selection was required to determine which approach was deemed the best to follow. As such, after conducting research on the various integration options, a multi-criteria decision model was applied over them to facilitate the choice in question.

The resulting conclusion fell in accordance to the advantages and disadvantages evidenced when researching the approaches, as it highlighted iFrame and JavaScript as the ideal solutions, setting the scene for which type of integration will be developed in the proof of concept application.

Chapter 4

Solution Description

This chapter presents a detailed description of the design for the proposed solutions in this dissertation regarding the development of web applications through a micro frontend-based approach.

Since the main goal is to assess the validity and worth of aforementioned architecture, an arbitrary web application subject matter was selected to contextualize the proof-of-concept applications and their functionalities. For this sole purpose, an administration dashboard was chosen for a hypothetical ecommerce application.

Initially, an analysis of the solution's requirement artifacts and its domain concepts is presented. Soon after, both the monolithic and Micro Frontend-based architectures are shown, finally followed by the use cases realizations.

4.1 Requirement Analysis

This section focuses on the solution's functional requirements, domain concepts and non-functional requirements.

4.1.1 Functional Requirements

Functional requirements define the system's basic functions which can vary when faced with a specific set of inputs or outputs. (Fulton & Vandermolen, 2014)

Each functional requirement is depicted as a Use Case (UC) and can be viewed in Figure 4.1 which consists of a use case diagram:

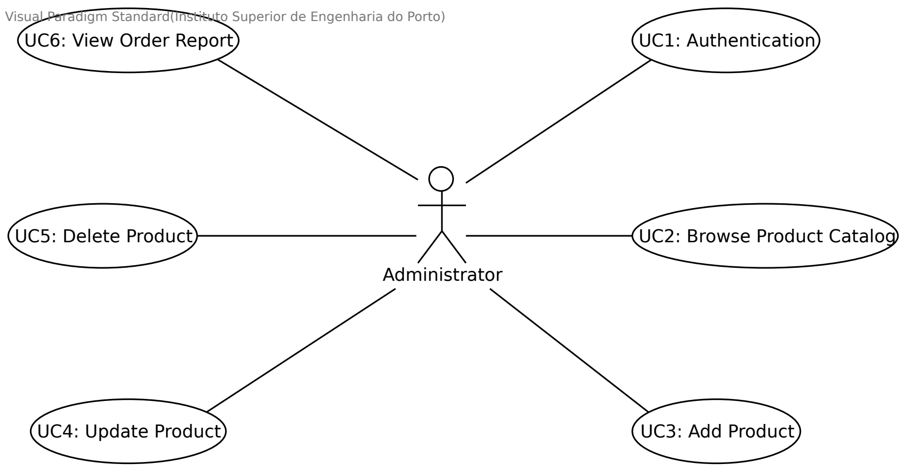


Figure 4.1: Use Case Diagram

Use Case 1: Authentication

As an Administrator, I want to be able to sign in by using my credentials to access the system's features.

Use Case 2: Browse Product Catalog

As an Administrator, I want to be able to browse the product catalog.

Use Case 3: Add Product

As an Administrator, I want to be able to add a product to the catalog.

Use Case 4: Update Product

As an Administrator, I want to be able to update an existing product.

Use Case 5: Delete Product

As an Administrator, I want to be able to delete a product from the catalog.

Use Case 6: View Billing Report

As an Administrator, I want to be able to view a billing report of the product sales made.

To allow a fair comparison between solutions, every proof-of-concept application will support the same set of functional requirements.

4.1.2 Domain Concepts

To provide a better understanding of the functional requirements previously enumerated in section 4.1.1, Figure 4.2 describes what concepts make up the system and how they interact through a Domain Model.

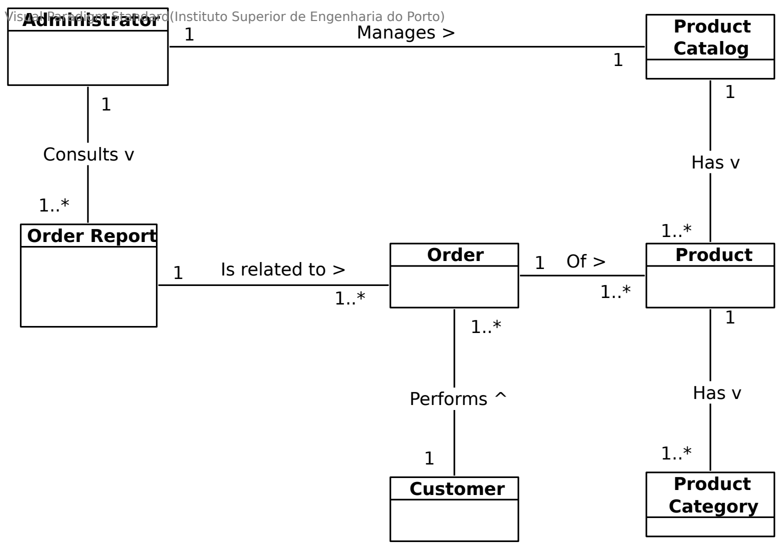


Figure 4.2: Domain Model

As formerly stated, the application's functionalities will revolve around a couple of administration features for a hypothetical ecommerce application. As such, some of the entities that make up the domain model, namely Customer and Order, will be pre-populated to mimic the usage of said ecommerce application.

Regarding the remaining concepts displayed, the main actor of the system will be an Administrator who is able to manage the catalog, which is in turn made up of products and their respective categories. The Administrator can also view the billing reports that provide an overview of the system's orders.

4.1.3 Non-Functional Requirements

Besides the functions a system should perform, oftentimes there are other constraints it must abide to. These are referred to as non-functional requirements.

The system's non-functional requirements are analyzed by means of the FURPS+ model which extends its predecessor, the FURPS model. The latter is composed by the following quality factors: functionality, usability, reliability, performance, supportability. The plus variant builds upon these factors by also bringing the specification of design, implementation, interface, and physical constraints. (Eeles, 2005)

Usability

The usability factor focuses on user interface requirements, concerning itself with characteristics such as aesthetics and consistency (Eeles, 2005).

Since a micro frontends-based architecture is composed by independent components with the possibility of each being developed with its specific technological stack, ensuring a consistent user interface can prove challenging. As such, the proof-of-concept applications should provide a seamless integration between their respective components to deliver an experience similar to a frontend monolith.

Reliability

The reliability factor tackles concerns surrounding the system's availability, accuracy and ability to recover from failure (Eeles, 2005).

The reliability requirements for this project fall in line with standard software development guidelines as the system should keep itself available and functional even when faced with an error by way of handling said error. In the event of a catastrophic failure, the system should be capable of creating new instances to ensure utter availability.

Performance

The performance factor focuses on characteristics like response and recovery time, throughput and startup and shutdown time (Eeles, 2005).

As detailed in section 2.4.2, one of the disadvantages Micro Frontends present revolves around the web application's dependencies and the size of the payloads requested by the browser when using them. This inherently affects the system's performance when considering what response times would be acceptable for its every-day users. As such, the system should mitigate this issue and achieve similar or better load times to what a frontend monolith provides.

Supportability

The supportability factor concerns itself with testability, adaptability, maintainability, compatibility, configurability, and scalability (Eeles, 2005).

For the proposed micro frontends architecture, it should be feasible for each of the system's components to be independently tested and maintained. Moreover, one should be able to replace a micro frontend without affecting the remaining components.

Design Constraints

As its name would indicate, a design constraint addresses restrictions surrounding the available options when designing a system (Eeles, 2005). For this project, the design constraints are as follows:

1. The system should be composed by multiple micro frontends, each with its dedicated business sub-domain.
2. Each of the system's micro frontends should have its respective microservice to address its needs.

Implementation Constraints

Implementation constraints limit the system's coding or construction (Eeles, 2005). This system's implementation constraints are mainly consequences of its developer's experience as well as the technology stack used by the company in which it was developed. They are as follows:

1. The micro frontends should be developed using Angular and React.
2. The micro services should be developed using .NET Core.

3. The applications should run in containers, more specifically, Docker containers.

4.2 Solution Design

This section details the solution design for both the monolithic and Micro Frontends-based approach.

4.2.1 Architecture

To better understand what changes between the proposed approach and the standard monolithic one, it is best to first introduce the latter by providing an overview of its design.

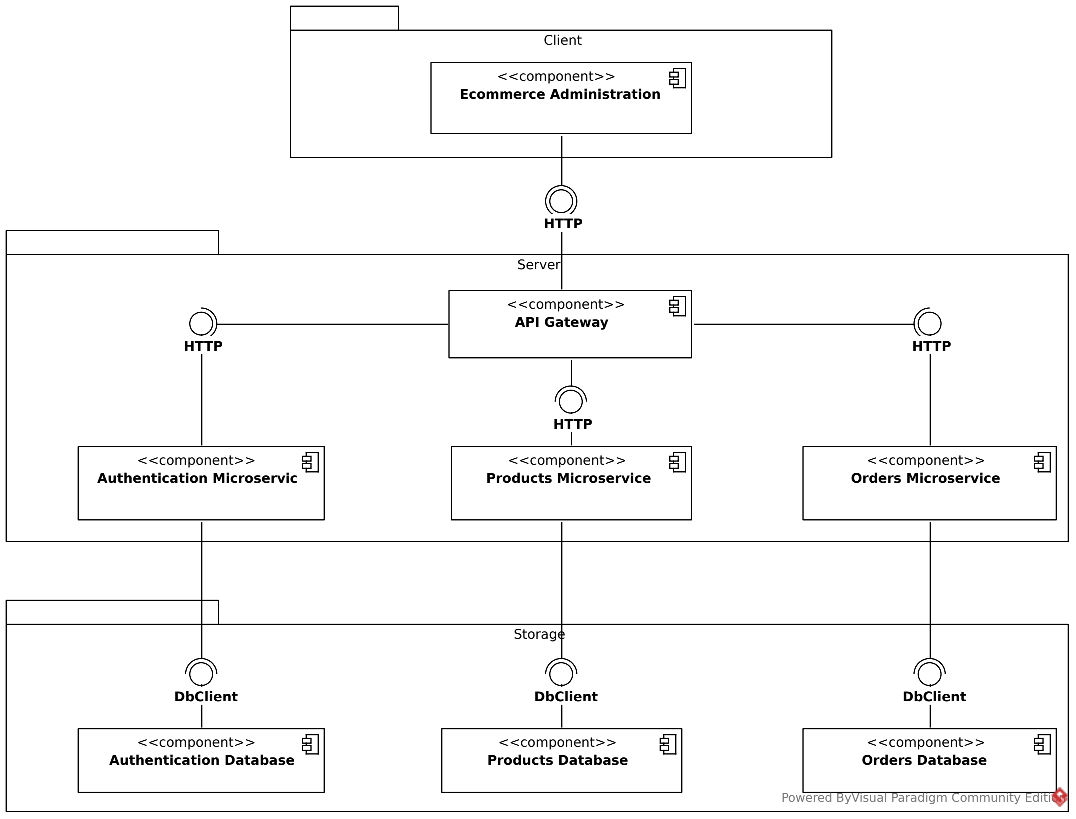


Figure 4.3: Monolithic Architecture - Component Diagram

As seen in Figure 4.3, only a single frontend component exists which consumes all the system's microservices through an API Gateway. This means the solution benefits from the advantages brought by a microservice-based backend but is unable to transpose them to its frontend, raising issues surrounding its scalability, deployment, among others which will be detailed later in one of the document's sections.

Now shifting the focus towards the alternative, Figure 4.4 presents the architectural proposal based on micro frontends.

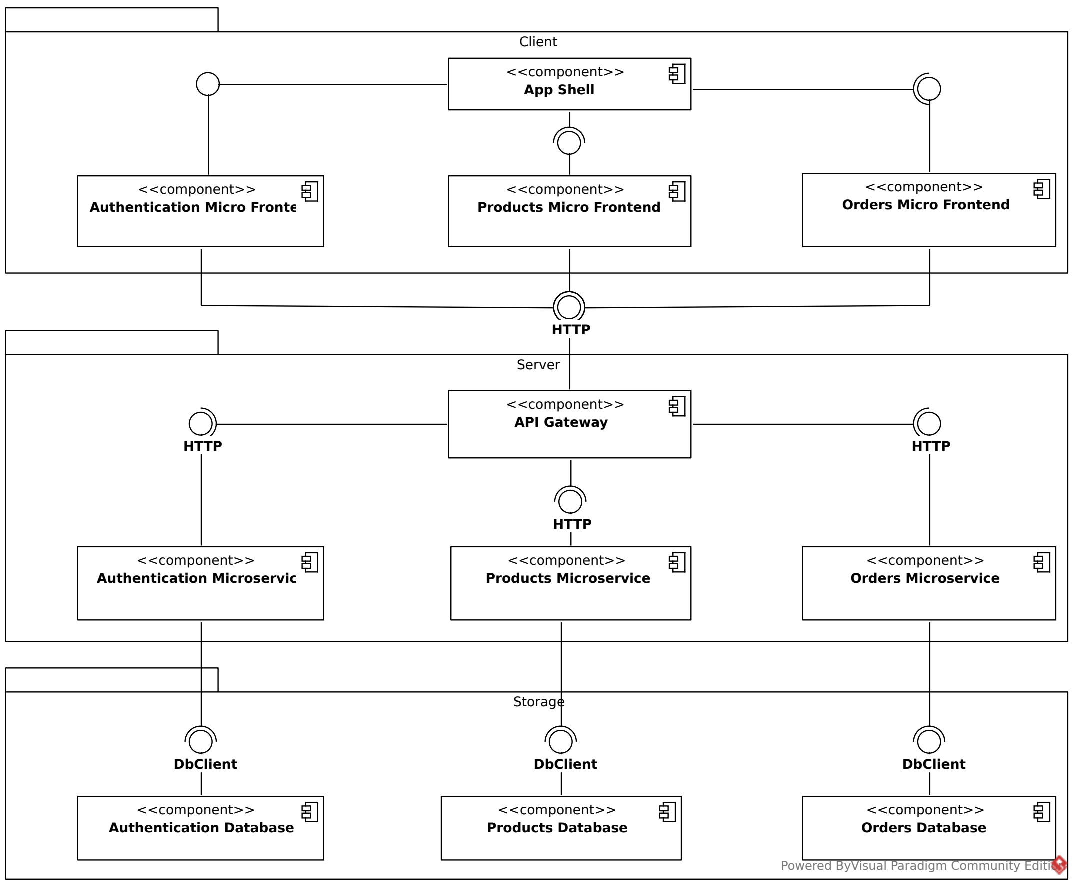


Figure 4.4: Micro Frontend Architecture - Component Diagram

As a result of the Value Analysis performed in section 3.2, the proof-of-concept application will be built by leveraging iFrames as components, orchestrating said components at runtime by using client-side composition.

The App Shell component is used as a container, being void of any responsibility aside from routing and displaying a navigation menu. This will reduce dependencies and ensure each micro frontend's independence while also mounting/unmounting them according to the user's needs to keep browser memory usage at a minimum. By making the App Shell devoid of business logic, the risk of a critical failure occurring is mitigated, which is of great importance given its job as the system's orchestrator.

Since all system calls must be properly authenticated, the Authentication micro frontend will make a sign-in screen readily available, allowing administrators to start a new session and access the solution's features. It is then possible to check and maintain the product list through the Product micro frontend and visualize reports through the Orders micro frontend which will be made available whenever the user navigates to the specific pages they are displayed in via the routing capabilities of the App Shell.

To access server data, each micro frontend will communicate with an API Gateway that is responsible for redirecting requests to their intended microservice, while also ensuring they are properly authenticated. The microservices will then make use of their respective databases to fetch whatever data is necessary.

By designing the system this way, we can ensure that a micro frontend focuses on its specific business sub-domain and does not conflict with any of the other existing ones. This aspect when combined with a microservice-based backend allows for each feature to be completely independent, providing better scalability. This independence results in also not having restrictions regarding which programming language or framework to use when developing.

Each micro frontend can also have their own continuous delivery pipeline, responsible for building, testing and deploying it.

4.2.2 Use Cases Realizations

This section presents the use cases depicted in section 4.1.1 by way of use cases realizations. Each use case will have a sequence diagram that displays its participating objects and respective interactions (Jacobson et al., 2011).

Authentication

For an administrator to have access to the system's features, he must first sign in into the application to become authenticated. Figure 4.5 represents the sequence diagram that describes this process.

The unauthenticated administrator begins by navigating to the Authentication Micro Frontend and filling out the sign in form with his respective credentials. From there, an authentication request is sent to the API Gateway which will forward it to the respective Authentication microservice instance. The user's credentials are validated and an access token is generated.

The bottom half of the sequence diagram displays how a resource request is handled when a token is present. The API Gateway will initially redirect the request towards the Authentication microservice to have the token verified and confirm whether the user is authorized to access the requested resource. Lack of authorization will result in an error message while a successful scenario will have the API Gateway continue the request by contacting the intended resource.

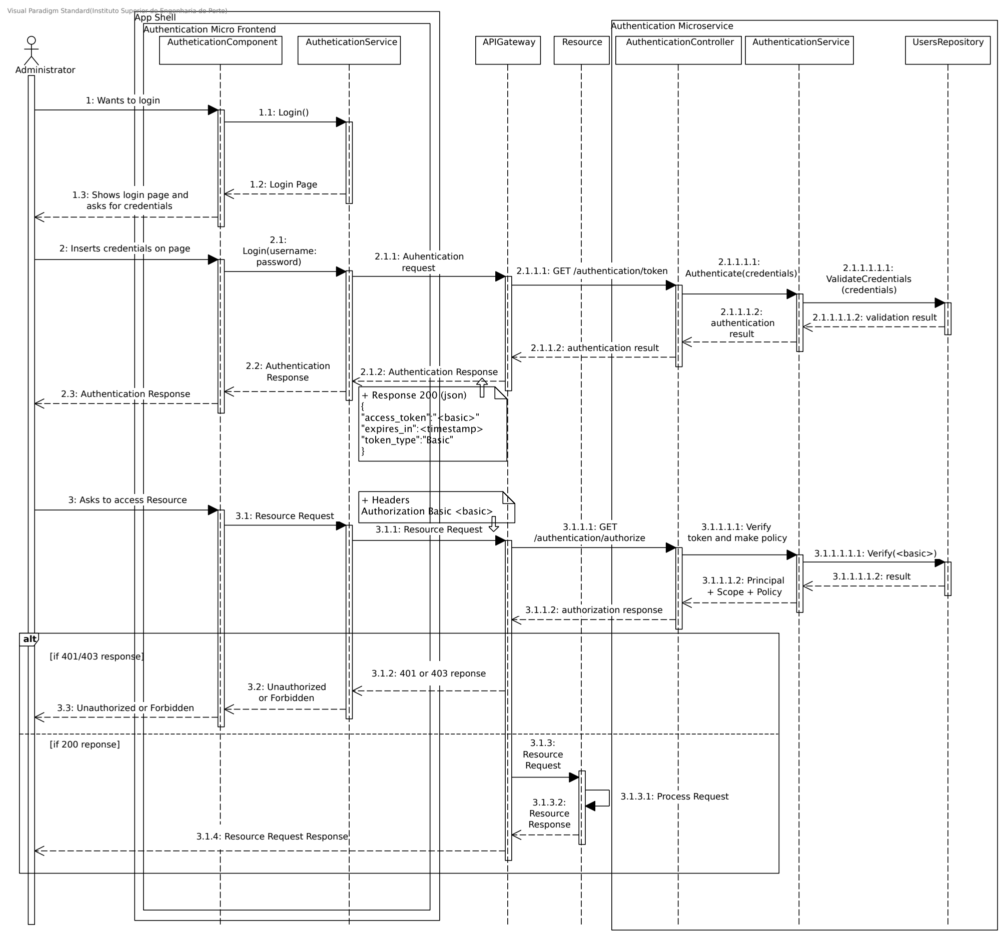

Visual Paradigm Standard(Instituto Superior de Engenharia do Porto)
App Shell
Authentication Micro Frontend
Authentication Microservice
Administrator
AutheticationComponent
AutheticationService
APIGateway
Resource
AuthenticationController
AuthenticationService
UsersRepository
1: Wants to login
1.1: Login()
1.2: Login Page
1.3: Shows login page and asks for credentials
2: Inserts credentials on page
2.1: Login(username: password)
2.1.1: Auhentication request
2.1.1.1: GET /authentication/token
2.1.1.1.1: Authenticate(credentials)
2.1.1.1.1.1: ValidateCredentials (credentials)
2.1.1.1.1.2: validation result
2.1.1.1.2: authentication result
2.1.1.2: authentication result
2.1.2: Authentication Response
2.2: Authentication Response
2.3: Authentication Response
+ Response 200 (json) { "access_token":"<basic>" "expires_in":<timestamp> "token_type":"Basic" }
3: Asks to access Resource
+ Headers Authorization Basic <basic>
3.1: Resource Request
3.1.1: Resource Request
3.1.1.1: GET /authentication/authorize
3.1.1.1.1: Verify token and make policy
3.1.1.1.1.1: Verify(<basic>)
3.1.1.1.1.2: result
3.1.1.1.2: Principal + Scope + Policy
3.1.1.2: authorization response
alt
[if 401/403 response]
3.1.2: 401 or 403 reponse
3.2: Unauthorized or Forbidden
3.3: Unauthorized or Forbidden
[if 200 reponse]
3.1.3: Resource Request
3.1.3.1: Process Request
3.1.3.2: Resource Response
3.1.4: Resource Request Response

Figure 4.5: Authentication Sequence Diagram

Create, Read, Update and Delete Product

After successfully signing in, the authenticated administrator can now browse and manage the system's product catalog by means of the Products micro frontend.

Considering that use cases number 2 through 5 consist in the four basic operations of persistent storage (create, read, update and delete or CRUD for short) and utilize the same Micro Frontend and microservice as a result of the business sub-domain they share, only a single sequence diagram will be presented.

As such, Figure 4.6 illustrates how the system's components interact to retrieve the product catalog and display it to the user.

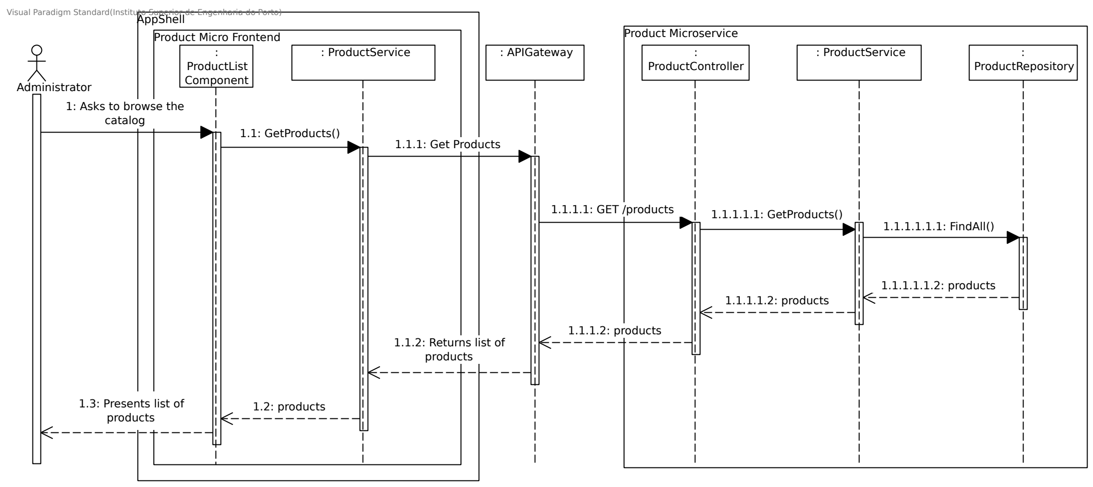


Figure 4.6: Browse Product Catalog Sequence Diagram

The use case is set off when the administrator first chooses to navigate towards the product catalog, which in turn causes the application shell to load the Products micro frontend.

From there, the component communicates with its dedicated service that triggers a request to the API Gateway. The gateway is then responsible for redirecting the request to its respective microservice while ensuring it is properly authenticated.

After making its way through the backend pipeline, a response is generated and falls all the way back to the original frontend component that finally displays it to the user.

View Order Report

The final use case consists in the visualization of a report of the system's orders.

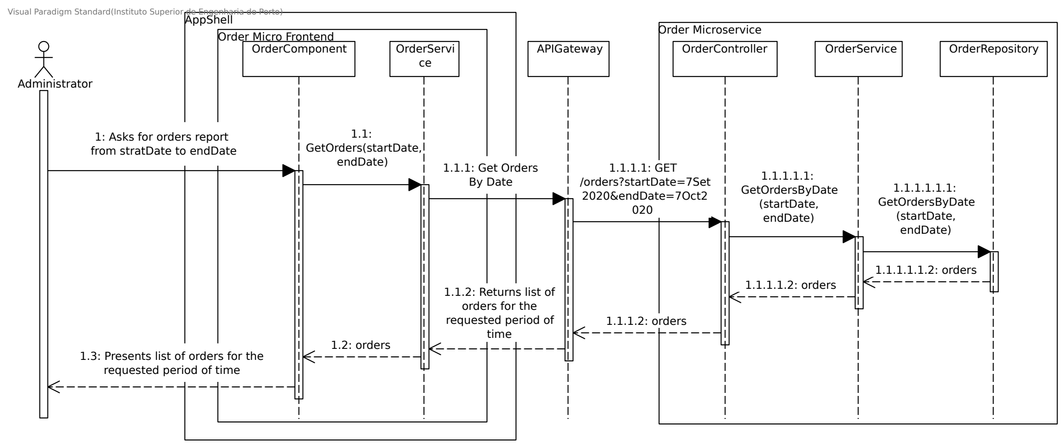

Visual Paradigm Standard(Instituto Superior de Engenharia do Porto)
AppShell
Order Micro Frontend
Administrator
OrderComponent
OrderService
APIGateway
Order Microservice
OrderController
OrderService
OrderRepository
1: Asks for orders report from stratDate to endDate
1.1: GetOrders(startDate, endDate)
1.1.1: Get Orders By Date
1.1.1.1: GET /orders?startDate=7Set 2020&endDate=7Oct2 020
1.1.1.1.1: GetOrdersByDate (startDate, endDate)
1.1.1.1.1.1: GetOrdersByDate (startDate, endDate)
1.1.1.1.1.2: orders
1.1.1.1.2: orders
1.1.1.2: orders
1.1.2: Returns list of orders for the requested period of time
1.2: orders
1.3: Presents list of orders for the requested period of time

Figure 4.7: View Order Report Sequence Diagram

As presented in Figure 4.7, the authenticated administrator starts by triggering the loading process for the Orders Micro Frontend by navigating to the reports page through the App Shell.

After loading, the component launches a request to the API Gateway that then redirects it to its intended destination, the Orders microservice. The information is then retrieved from the service's proprietary database and returned to the Micro Frontend that triggered the request which finally displays it.

4.3 Summary

In summary, the Solution Description chapter presents the Design for both systems that are up for comparison, monolithic and Micro Frontends.

An initial decision regarding the proof of concept's subject matter was made, resulting in the design of an ecommerce administration dashboard. Afterwards, an enumeration of the functional and non-functional requirements can be found in section 4.1.1 and section 4.1.3, respectively.

Regarding system architecture, component diagrams were presented for each solution in Figure 4.3 for the monolithic approach, and Figure 4.4 for the Micro Frontends.

Finally, section 4.2.2 presents the use cases realizations, providing a textual description and a sequence diagram for each existing one.

Chapter 5

Solution Implementation

This chapter details the actual implementation of both the proof of concept application and its monolithic counterpart. As a result of the Value Analysis performed in chapter 3.2, the Micro Frontend-based system leverages iframes as components to provide a clear separation of each business sub-domain.

5.1 Technology Stack

This section presents what technologies were used to build both the Monolithic solution and the Micro Frontends one. Since the goal of this dissertation is to assess the value of Micro Frontends while accepting microservices as a baseline for backend development, both solutions will make use of the same microservices, which will be initially presented.

5.1.1 Microservices

There are three separate microservices in the designed system to accommodate the use cases described in section 4.2.2, namely the Authentication, Products and Orders microservices.

Each was built using .NET Core by no specific reason other than the author's experience with said framework. It's a very capable framework that provides a fast and easy way of creating RESTful services which are very lightweight and commonly chosen to create APIs for standard web applications (Pautasso et al., 2008).

The Authentication microservice, as the name would indicate, is responsible for managing authentication requests and generating the access tokens that enable the administrator to use the system's resources.

Up next is the Products microservice which specializes in its homonymous business sub-domain, allowing authenticated administrators to view or manage the system's entries.

Last but not least, the Orders microservice is responsible for providing all attainable information surrounding the system's orders to the administrator.

5.1.2 Monolithic

The monolithic solution was built as a standard Single-Page Application (SPA), using Angular as its foundation.

Given that this dissertation focuses more on the architectural standpoint rather than what programming language is best for the development of web applications, the decision was solely based on the author's experience and proficiency with the Angular framework.

To ensure user interface consistency, an Angular user interface component library was used, more specifically, PrimeNG.

5.1.3 Micro Frontends

Even though all the Micro Frontends could be developed using the same frontend technology stack, a decision was made to diversify which frameworks were used to display one of the advantages this architectural approach provides, the ability to make case by case decisions per component.

In a real-world scenarios, companies might choose to have their teams work with one or multiple frameworks, but this decision proves that if a any of the technologies used becomes obsolete or a better, more viable one shows up, components can be individually replaced without affecting the overall solution.

Bearing this in mind, the following table 5.1 presents what framework was used for each Micro Frontend:

Table 5.1: Micro Frontend Technology Stack

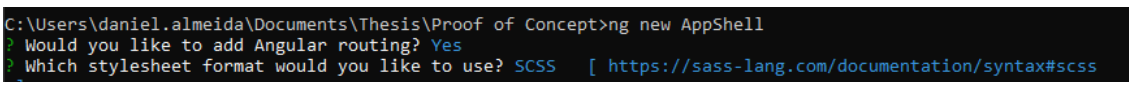
Micro Frontend	Framework
Authentication	Angular
Products	Angular
Orders	React

To try and address one of the concerns surrounding the usage of Micro Frontends in web applications, more specifically user interface consistency, the same user interface component library supplier was used in every component, Prime. It provides a broad collection of user interface elements for both Angular (PrimeNG) and React (PrimeReact) applications that are visually identical.

Finally, the App Shell component, which is responsible for orchestrating and loading/unloading the Micro Frontends was also built with Angular.

5.2 Monolithic Implementation

For the monolithic web application, firstly a new Angular project was created by means of the **ng new** command made available by its dedicated command-line interface (CLI):



```
C:\Users\daniel.almeida\Documents\Thesis\Proof of Concept>ng new AppShell
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS   [ https://sass-lang.com/documentation/syntax#scss
```

Figure 5.1: ng new command

As displayed in Figure 5.1, some of the supported parameters for this command consist in whether or not the application will use angular routing and what stylesheet format will be used. By choosing to use angular routing, a dedicated NgModule is created where one can configure the application's routes.

Modules are often used to help organize the app and keep things compartmentalized, in this case ensuring all routing information is kept in the same place.

For the stylesheet format, SCSS was chosen as it encourages the use of proper nesting rules and has a clearer syntax (Lorraine, 2011).

5.2.1 Initial Application Setup

After creating the project, a basic dashboard layout was created by initially generating cross-app components that represent its sidebar, navigation bar, and footer, displayed in Figure 5.2 and tagged with number 1, 2 and 3 respectively.

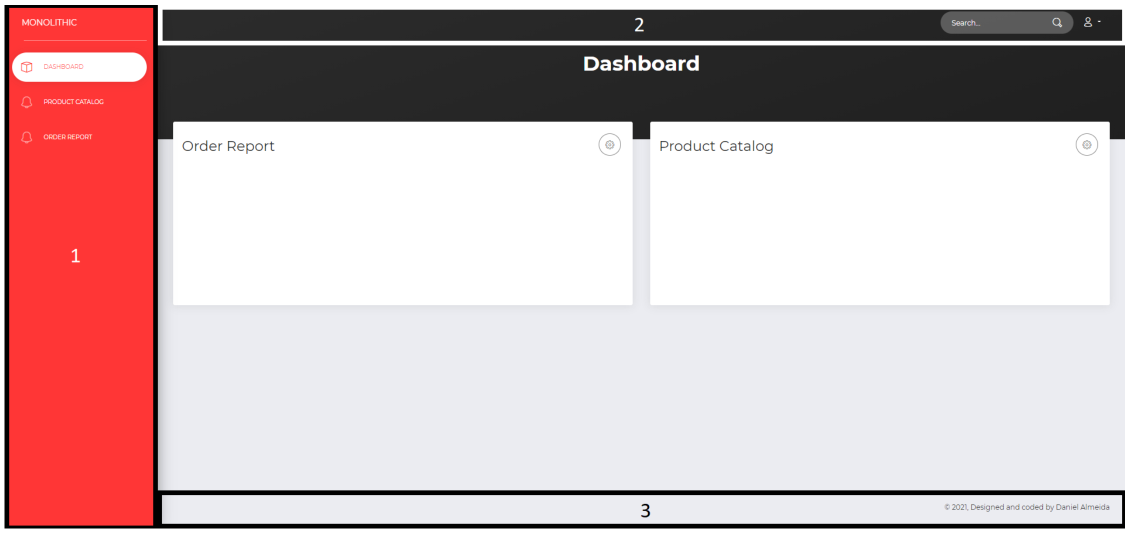


Figure 5.2: Dashboard Layout - Sidebar, Navigation bar and Footer

Now that the application's general layout is set up, it is time to create the necessary components to carry out its functionalities.

Since the application only supports simplistic functionalities, it's unnecessary to have small components to promote re-usability as they would serve no purpose. Instead, the component per page approach was chosen which, as the name indicates, consists in having a single Angular component that is responsible for the entirety of the page's layout/functionalities.

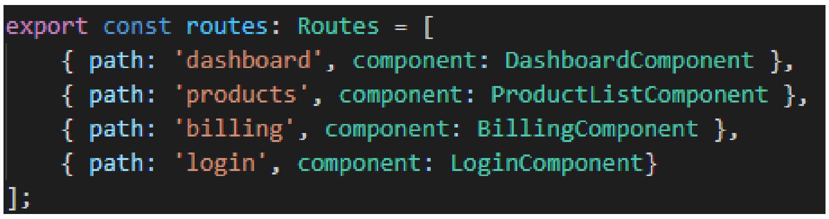
The following table 5.2 presents what components were created, each accompanied by a brief description of their respective use:

Table 5.2: Monolithic Component per Page

Component	Description
Dashboard	Act as a home page and display navigation cards for other pages
Product List	Display product catalog and allow an administrator to manage it
Billing	Display the system's orders and their contents
Sign In	Allow the administrator to sign in and access the system

5.2.2 Routing

With the components created, the next step is to configure the application's routing module so that each matches with a specific route when using the website:



```
export const routes: Routes = [
    { path: 'dashboard', component: DashboardComponent },
    { path: 'products', component: ProductListComponent },
    { path: 'billing', component: BillingComponent },
    { path: 'login', component: LoginComponent}
];
```

Figure 5.3: Route Configuration

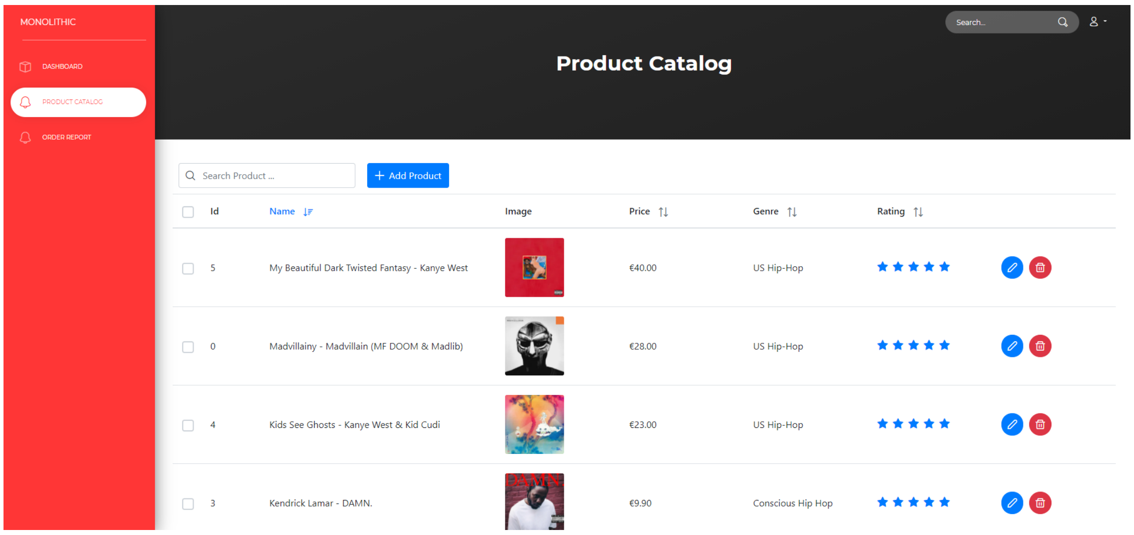
By setting up the configuration illustrated in Figure 5.3, the application is now functional from a navigation standpoint, only requiring the implementation of the components mentioned in Table 5.2 to be considered complete.

Given that the dashboard page only presents a couple of cards that facilitate user navigation, no additional detail will be provided on that component.

5.2.3 Product List

As stated in Table 5.2, the Product List component should allow the administrator to visualize and maintain the application's product catalog.

As such, Figure 5.4 displays the resulting layout whenever the user lands on this page.



Id	Name	Image	Price	Genre	Rating
5	My Beautiful Dark Twisted Fantasy - Kanye West		€40.00	US Hip-Hop	★★★★★
0	Madvillainy - Madvillain (MF DOOM & Madlib)		€28.00	US Hip-Hop	★★★★★
4	Kids See Ghosts - Kanye West & Kid Cudi		€23.00	US Hip-Hop	★★★★★
3	Kendrick Lamar - DAMN.		€9.90	Conscious Hip Hop	★★★★★

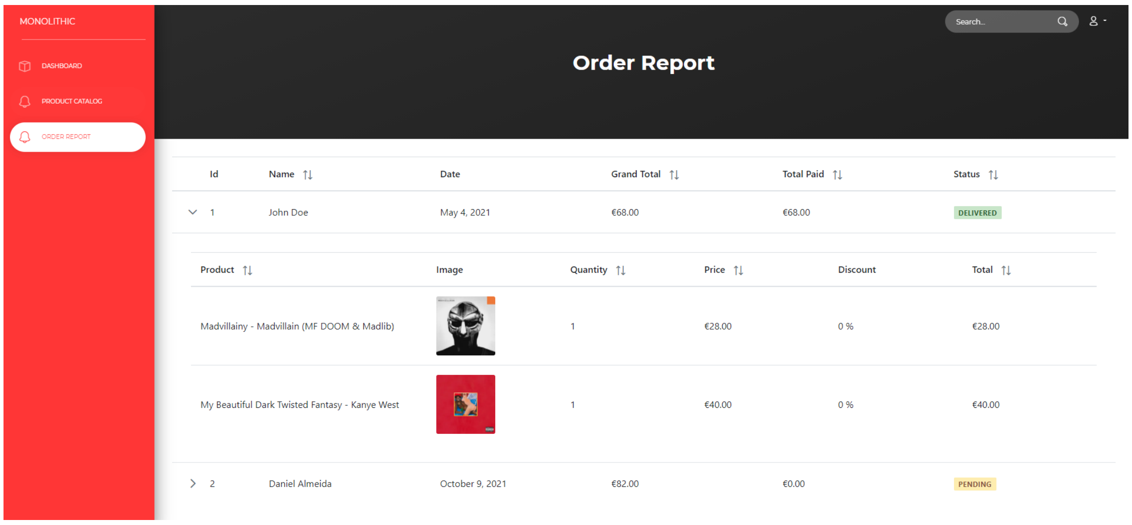
Figure 5.4: Product Catalog

The products are presented through a table which also contains action buttons that make it possible for the administrator to add, edit and remove entries.

5.2.4 Billing

In order for the Billing component to display the orders made in the system, it also leverages a table, in this case with the option to expand each order to get more details surrounding what products were ordered.

Figure 5.5 displays the final layout attained.



Order Report

	Id	Name	Date	Grand Total	Total Paid	Status
⌄	1	John Doe	May 4, 2021	€68.00	€68.00	DELIVERED

Product	Image	Quantity	Price	Discount	Total
Madvillainy - Madvillain (MF DOOM & Madlib)		1	€28.00	0 %	€28.00
My Beautiful Dark Twisted Fantasy - Kanye West		1	€40.00	0 %	€40.00

	Id	Name	Date	Grand Total	Total Paid	Status
›	2	Daniel Almeida	October 9, 2021	€82.00	€0.00	PENDING

Figure 5.5: Order Report

5.2.5 Sign In

The final page is the one responsible for letting the administrator sign in and access the system's features, the Sign In component.

It displays a small form that requests the user's username and password to then generate an access token. Since token storage is not a main point of concern given this dissertation's subject matter, it was opted to save the token using the browser's local storage. That same token is then used in posterior requests to system resources and will be validated by the Authentication Microservice.

5.3 Micro Frontends Implementation

Although the Micro Frontends developed for the solution that is presented in this dissertation were all developed by the author, an initial distribution of tasks into hypothetical teams was made to simulate a real world environment where this architectural approach is meant to be applied.

5.3.1 Team Organization

Table 5.3 presents the proposed team organization for the system's features:

Table 5.3: Team Organization

Team	Task Description
Core	Develop the App Shell component and the API Gateway
Product	Develop the Product Micro Frontend, Product Microservice and Product Database
Billing	Develop the Order Micro Frontend, Order Microservice and Order Database
Authentication	Develop the Authentication Micro Frontend, Authentication Microservice and Authentication Database

The reasoning behind the aforementioned task separation is to have vertical teams that address a specific business sub-domain. This provides a very independent development environment and minimizes outside interference.

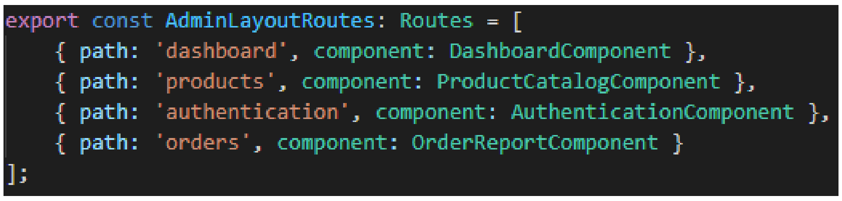
5.3.2 App Shell

Starting off with the App Shell Micro Frontend, as stated in section 5.1.3, this component was built with Angular. As such, in similar fashion to the monolithic web application, a new Angular project was created through its CLI by using the **ng new** command.

With the new and empty project now available, empty pages were created for each Micro Frontend through the **ng generate component** command, more specifically, a component for the Product Catalog, Order Report and Authentication.

A Dashboard component was also created simply to provide a landing page after logging in but its simplicity does not justify turning it into a separate Micro Frontend.

Now with the necessary pages created, the appropriate routing was put in place to allow the user to navigate inside the application.

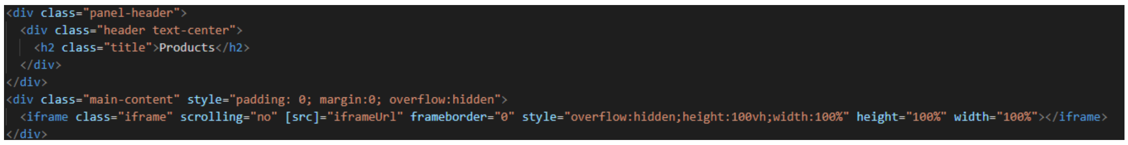


```
export const AdminLayoutRoutes: Routes = [
    { path: 'dashboard', component: DashboardComponent },
    { path: 'products', component: ProductCatalogComponent },
    { path: 'authentication', component: AuthenticationComponent },
    { path: 'orders', component: OrderReportComponent }
];
```

Figure 5.6: Routing Configuration - App Shell

The next step is to prepare the Angular components to display the Micro Frontends through iframes.

This process is almost identical for every page as each only requires a single iframe for the current solution. For demonstration purposes, the Product List will be used as an example.



```
<div class="panel-header">
  <div class="header text-center">
    <h2 class="title">Products</h2>
  </div>
</div>
<div class="main-content" style="padding: 0; margin:0; overflow:hidden">
  <iframe class="iframe" scrolling="no" [src]="iframeUrl" frameborder="0" style="overflow:hidden;height:100vh;width:100%" height="100%" width="100%"></iframe>
</div>
```

Figure 5.7: iFrame Preparation - App Shell

Figure 5.7 displays the HTML template for the Product List component. The initial **div** element is of little to no importance as it is a simple header for the page that displays its title. Below it lies the iframe that is used to embed the Micro Frontend and its encompassing div.

One of the concerns with the usage of iframes is their sizing, more specifically, their height. In past instances this would bring about issues but, thankfully, that is not longer the case as one can circumvent said issues with carefully placed CSS that make the iframe adjust to the rendered content.

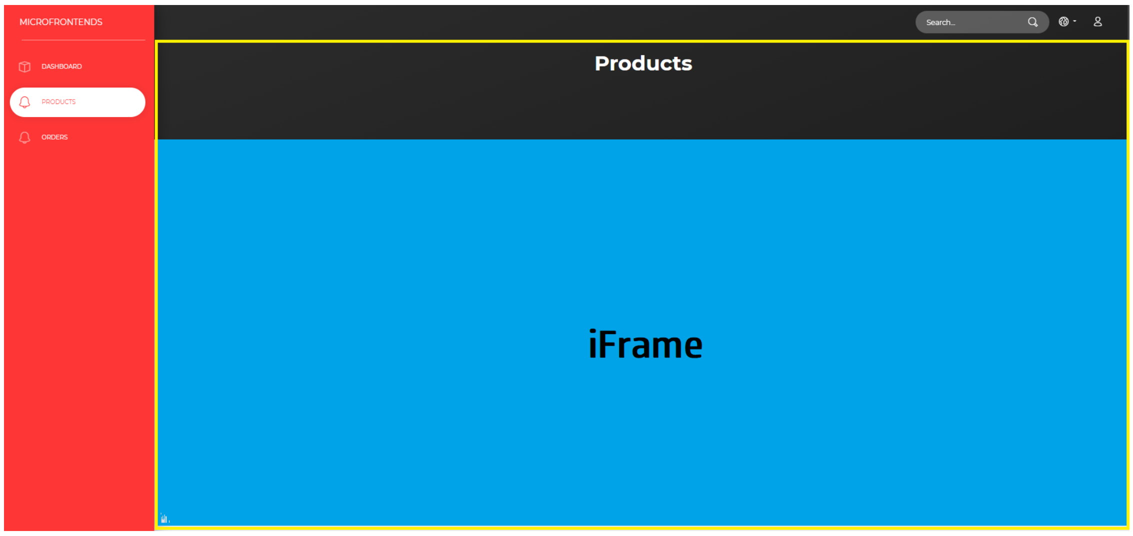


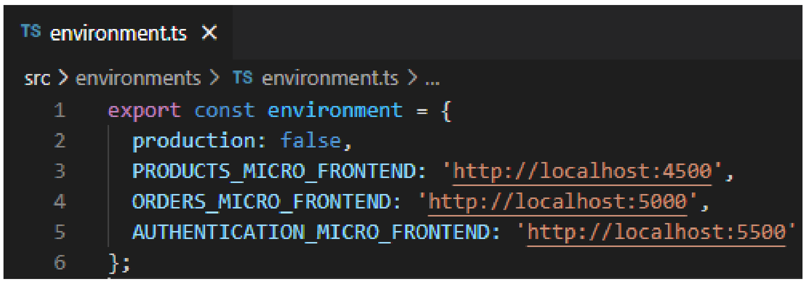
Figure 5.8: Product Catalog Layout - App Shell

The resulting page from the template implemented is illustrated in Figure 5.8 where the Product Catalog component is delimited by the yellow lines, while the iframe section is represented in blue.

This only leaves a single step to be performed in the App Shell to display the Micro Frontend's content, designate the iframe source.

This is easily achieved through Angular's environment configuration as it allows one to define different named build configurations per environment (e.g. development, production).

For this instance, two separate environment configurations were made, one for local testing and another to mimic a production stage. Figure 5.9 shows the local configuration:

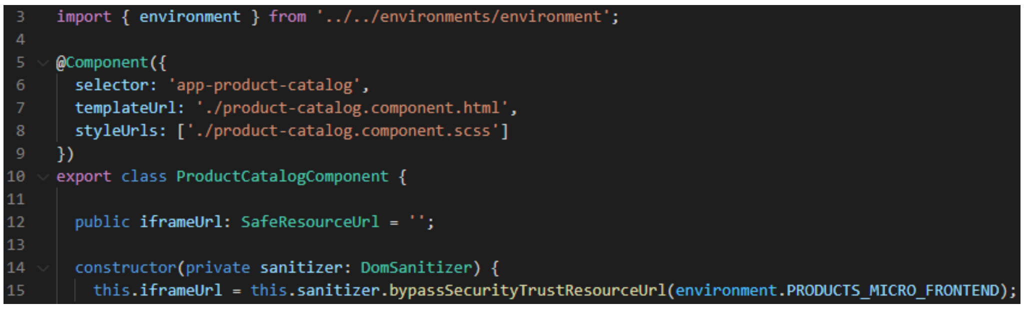


```
export const environment = {
  production: false,
  PRODUCTS_MICRO_FRONTEND: 'http://localhost:4500',
  ORDERS_MICRO_FRONTEND: 'http://localhost:5000',
  AUTHENTICATION_MICRO_FRONTEND: 'http://localhost:5500'
};
```

Figure 5.9: Environment Configuration

The production property indicates whether or not the given configuration is meant for the production environment. When set to **true**, said configuration is used when running the application with the **–prod** flag or with the **–configuration production** option.

With our configuration put in place and dynamically changing according to the application's environment, it is now possible to use the Micro Frontend addresses it possesses as the iframe sources. To do so, one simply needs to import the default environment configuration and make use of its properties, as displayed in Figure 5.10:



```
import { environment } from '../../environments/environment';

@Component({
  selector: 'app-product-catalog',
  templateUrl: './product-catalog.component.html',
  styleUrls: ['./product-catalog.component.scss']
})
export class ProductCatalogComponent {

  public iframeUrl: SafeResourceUrl = '';

  constructor(private sanitizer: DomSanitizer) {
    this.iframeUrl = this.sanitizer.bypassSecurityTrustResourceUrl(environment.PRODUCTS_MICRO_FRONTEND);
```

Figure 5.10: Setup iFrame source from environment configuration

For testing purposes, one can create an empty angular project and locally run it in the port specified by the environment configuration (port 4500).

If both apps (App Shell and newly created Angular App) are up and running, it is possible to see the iframe content being displayed inside the App Shell, as demonstrated in Figure 5.11:

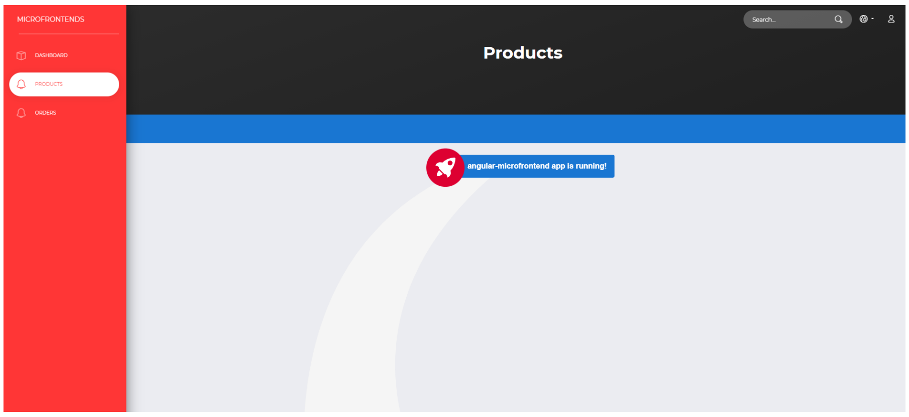


Figure 5.11: App Shell displaying Micro Frontend via iFrame

Now that the App Shell is functional and ready to display the iframes, changes to the actual Micro Frontend will be immediately reflected here without the need to make any changes to the App Shell.

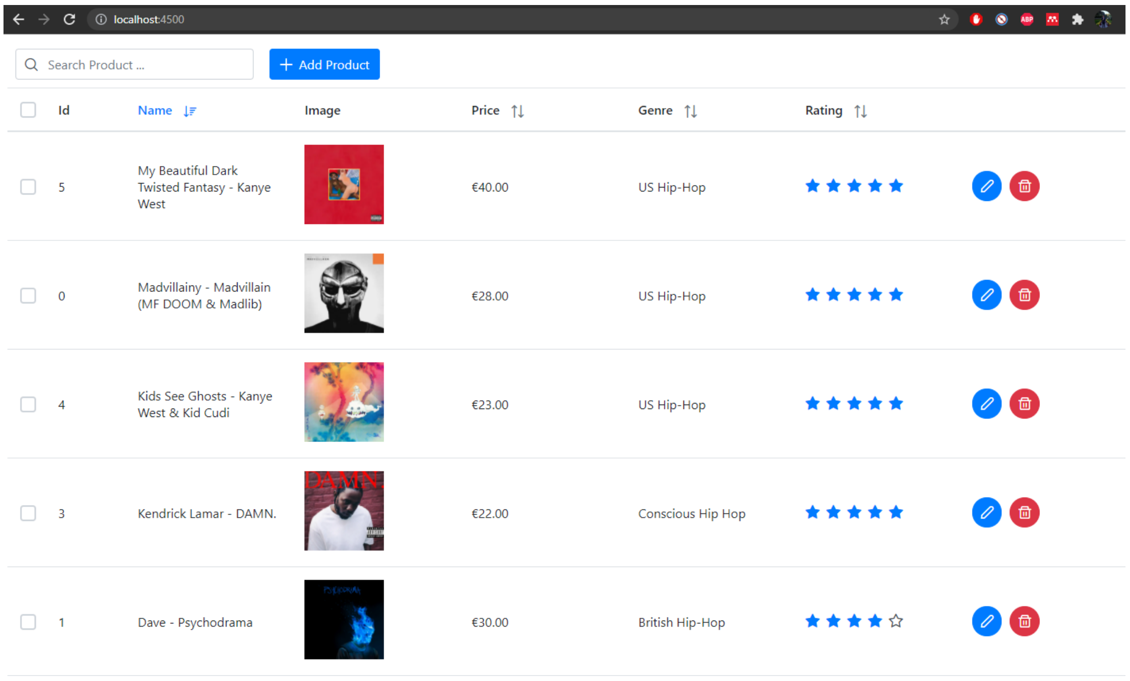
5.3.3 Product Micro Frontend

As stated in table 5.1, the Products Micro Frontend will be built using the Angular framework. Therefore, much like the App Shell and the monolithic web application, the starting point is to create a new Angular project.

After the initial setup, it's time to tackle the actual implementation of the Product Catalog and its features.

Since Micro Frontends are not meant to be identifiable to the user, the goal is to achieve an exact match to the layout presented in sub section 5.2.3 for the monolithic web application.

By using the same user interface component library, it's quite simple to replicate the exact same layout since the components both use are one and the same.


localhost:4500

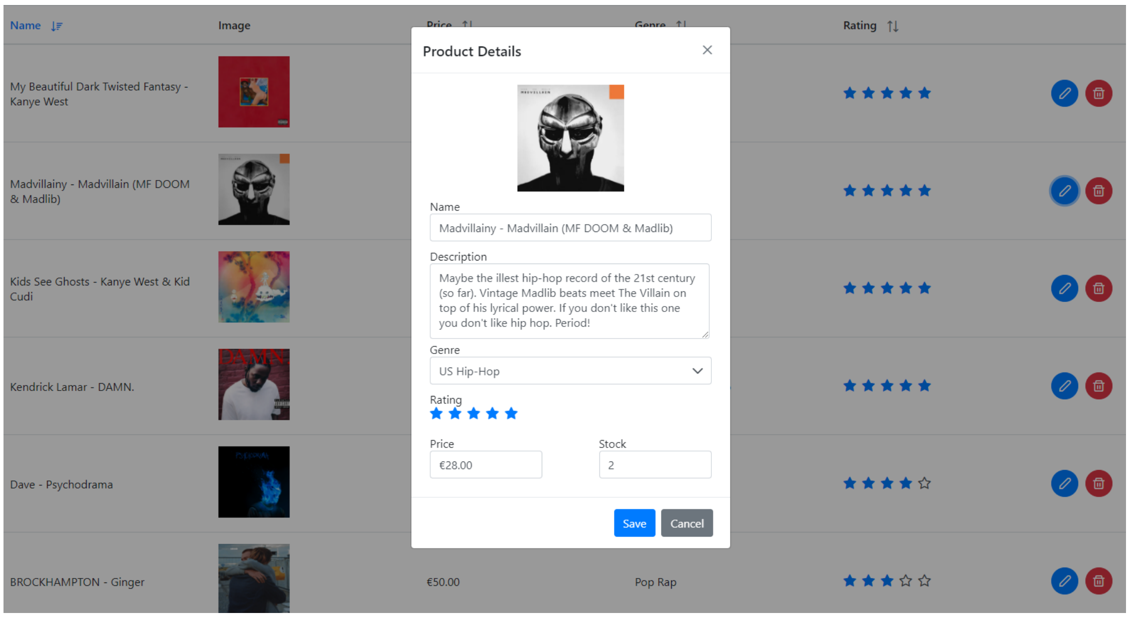
Search Product ... | + Add Product

	Id	Name	Image	Price	Genre	Rating
☐	5	My Beautiful Dark Twisted Fantasy - Kanye West		€40.00	US Hip-Hop	★★★★★
☐	0	Madvillainy - Madvillain (MF DOOM & Madlib)		€28.00	US Hip-Hop	★★★★★
☐	4	Kids See Ghosts - Kanye West & Kid Cudi		€23.00	US Hip-Hop	★★★★★
☐	3	Kendrick Lamar - DAMN.		€22.00	Conscious Hip Hop	★★★★★
☐	1	Dave - Psychodrama		€30.00	British Hip-Hop	★★★★☆

Figure 5.12: Product Micro Frontend

Figure 5.12 shows the final layout for viewing the Product Catalog.

Every action one can execute surrounding the Product entity is carried out through the action buttons presented in the table which trigger the following window to show up:


Name	Image	Price	Genre	Rating
My Beautiful Dark Twisted Fantasy - Kanye West				★★★★★
Madvillainy - Madvillain (MF DOOM & Madlib)				★★★★★
Kids See Ghosts - Kanye West & Kid Cudi				★★★★★
Kendrick Lamar - DAMN.				★★★★★
Dave - Psychodrama				★★★★☆
BROCKHAMPTON - Ginger		€50.00	Pop Rap	★★★☆☆

Product Details

Name: Madvillainy - Madvillain (MF DOOM & Madlib)

Description: Maybe the illest hip-hop record of the 21st century (so far). Vintage Madlib beats meet The Villain on top of his lyrical power. If you don't like this one you don't like hip hop. Period!

Genre: US Hip-Hop

Rating: ★★★★★

Price: €28.00

Stock: 2

Save | Cancel

Figure 5.13: Update Product - Product Micro Frontend

All of the Micro Frontends features were implemented as described in section 4.2.2, requiring no outside interventions from other modules of the system.

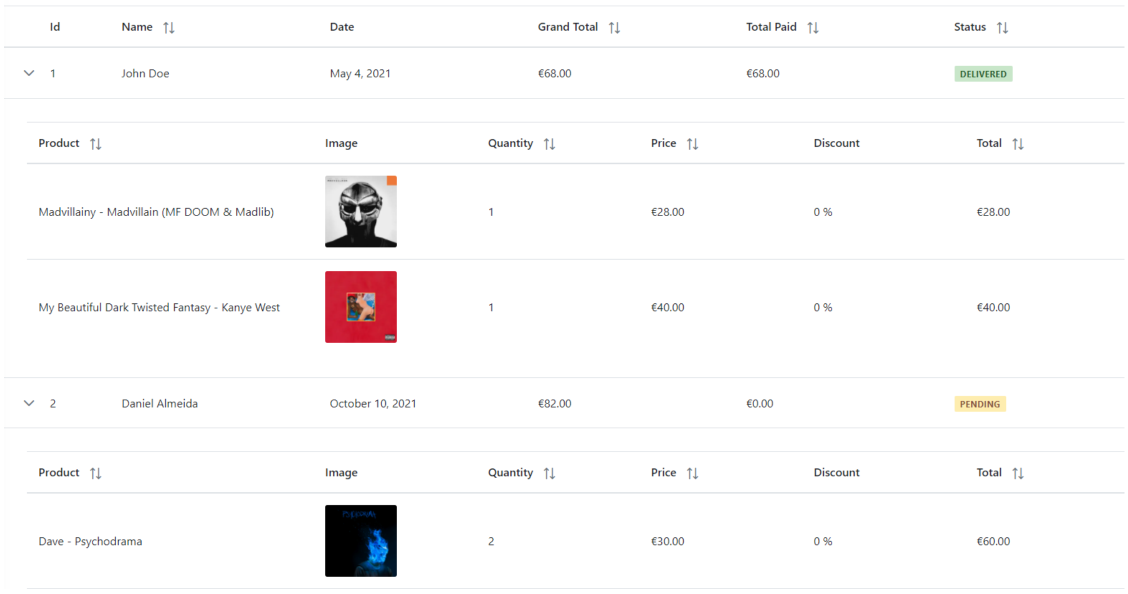
5.3.4 Orders Micro Frontend

As previously stated in section 5.1.3, the Orders Micro Frontend will be built with the React framework to demonstrate one the architecture's strengths, having a system made up of components implemented with different technology stacks.

With that in mind, to start off the development of our React Micro Frontend, a new project was created. To do so, the Create React App (CRA) tool was used since it is an officially supported way to create a React single-page application (McCurdy, 2020), only requiring a Node installation to do so.

To create the app, the tool's **npx create-react-app** command was used. This command also supports starting a new app from a template by means of the **–template** parameter. This option was used since the desired outcome was a TypeScript app. With the app now created, it's time to develop its features.

As with the Product Micro Frontend, the goal is to achieve an indistinguishable look from the monolithic app which, given the circumstances regarding the usage of a different framework (React instead of Angular), could become troublesome. Thankfully, by using the React counterpart (PrimeReact) of the user interface component library used in the Angular Micro Frontend (PrimeNg) the task becomes significantly simpler.



Id	Name ⇅	Date	Grand Total ⇅	Total Paid ⇅	Status ⇅
1	John Doe	May 4, 2021	€68.00	€68.00	DELIVERED

Product ⇅	Image	Quantity ⇅	Price ⇅	Discount	Total ⇅
Madvillainy - Madvillain (MF DOOM & Madlib)		1	€28.00	0 %	€28.00
My Beautiful Dark Twisted Fantasy - Kanye West		1	€40.00	0 %	€40.00

Id	Name ⇅	Date	Grand Total ⇅	Total Paid ⇅	Status ⇅
2	Daniel Almeida	October 10, 2021	€82.00	€0.00	PENDING

Product ⇅	Image	Quantity ⇅	Price ⇅	Discount	Total ⇅
Dave - Psychodrama		2	€30.00	0 %	€60.00

Figure 5.14: Order Micro Frontend

Figure 5.14 shows the Micro Frontend as a standalone app which ended up identical to the monolithic Orders page. As was the case with the Products Micro Frontend, this component and its respective microservice was independently developed from start to finish, totally lacking of interference from any of the hypothetical teams.

5.3.5 Authentication Micro Frontend

The last of the Micro Frontends for the proof of concept application is the Authentication component. The decision for this item was to create a new Angular project, with an identical setup to the Products Micro Frontend.

Layout-wise, this is the most straightforward component of the system since it consists of simple sign-in form made up of a username and password field.

The following figure 5.15 displays the Micro Frontend after implementation:

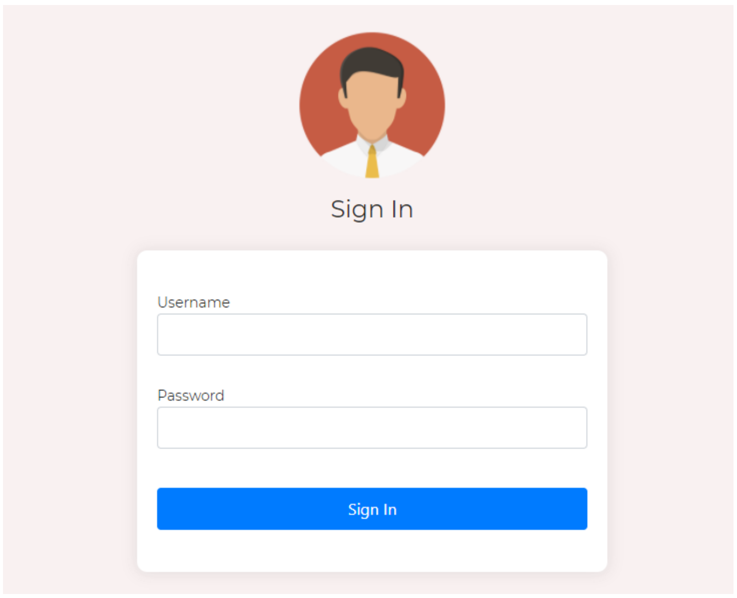


Figure 5.15: Authentication Micro Frontend

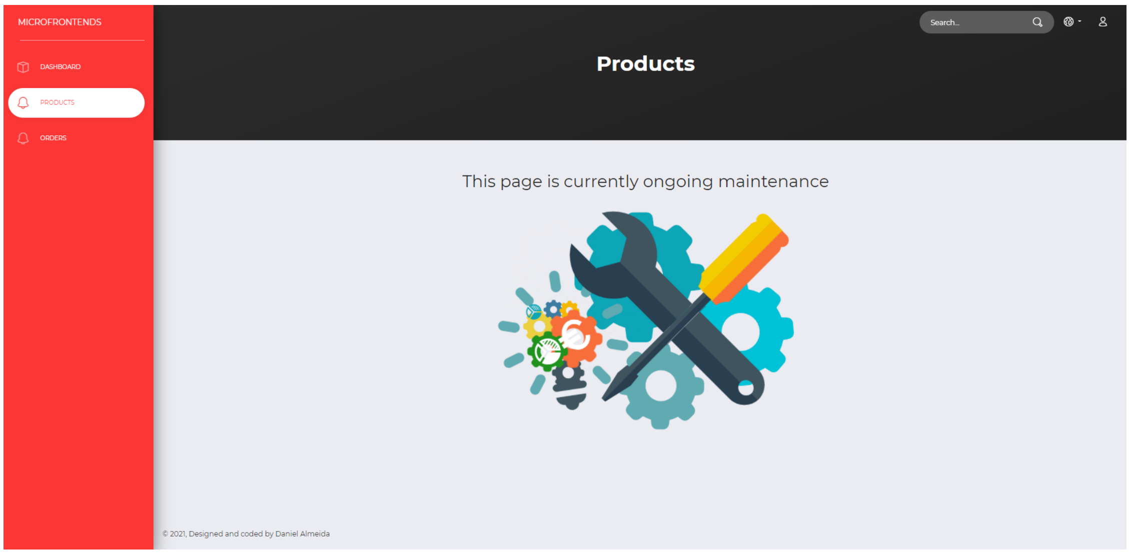
Given the nature of iframes and their isolation DOM wise, the token generated after a user sign in needs to be propagated to the App Shell component for use in future requests. To do so, the application makes use of the **Window** object, more specifically, its **postMessage()** method. It dispatches a **MessageEvent** from the Authentication Micro Frontend to the App Shell application, allowing posterior storage by the latter.

5.3.6 Error Handling

Considering that the solution is no longer a traditional frontend monolith, but instead a system made up by several pieces (the Micro Frontends), what happens when one of the system's components can't be loaded? After all, connectivity issues may cause the iframe request to falter or the respective Micro Frontend could be down for maintenance.

To address this issue, the App Shell possesses a validation request that is triggered whenever one of the Micro Frontends is requested to load. It consists of a simple Ajax request which enables one to bind success and error handlers, allowing for proper management of the situation per the response given.

As stated before, there are two instances where an iframe may fail to load, one of which is expected and caused by the development team, maintenance. In this scenario, the application must display a warning for the end user.



This page is currently ongoing maintenance

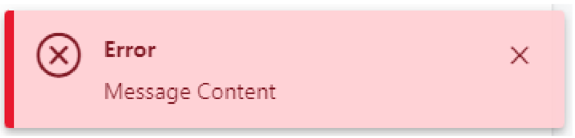
© 2021, Designed and coded by Daniel Almeida

Figure 5.16: Maintenance Warning

Figure 5.16 presents one example for a warning message, triggered by leveraging the aforementioned Ajax request, more specifically when it fails.

Aside from availability issues, bugs are always a possibility in any system. Application's often display the appropriate error messages in the same fashion for the entirety of their systems, be it through a small notification center, a message toast in one of the screen's corners, among others.

Since every iframe has its own DOM, one can either have each Micro Frontend display their own errors their own way or, have a messaging mechanism that enables cross component communication, more specifically between the App Shell and its Micro Frontends in this instance. For the proof of concept, the latter was chosen as it keeps the user experience consistent all the way through the system by showing eventual errors in a similar fashion.


Error

Message Content

Figure 5.17: Error Messages - App Shell

Figure 5.17 displays the message toast component that will be used to convey the error messages to the user, which will appear at the top-right corner of the web browser when triggered.

5.4 Summary

The first section of the Solution Implementation chapter focuses on the technology stack used while implementing the solution for both its backend (section 5.1.1) and frontend components (section 5.1.2 for the monolithic approach and section 5.1.3 for the Micro Frontends one).

Afterwards, a step by step description surrounding the setup and implementation of the monolithic solution is presented which can be used as a comparison reference for the proof of concept application regarding the final product attained.

Section 5.3 then follows the same explanatory process for the Micro Frontends solution, albeit with more detail, as it also describes how orchestration by the App Shell was managed, along with each component's implementation.

Finally, section 5.3.6 presents how error handling was carried out.

Chapter 6

Evaluation

As the name would indicate, the evaluation chapter concerns the description of the process through which the solution is to be evaluated and analyzed. Firstly, research hypothesis should be identified, followed by a description of the evaluation methodology and its indicators.

6.1 Hypothesis

A research hypothesis is a clear, quantifiable, and testable predictive statement in regard to the possible outcome of a scientific research study that's based on a specific property of a population (e.g. differences between groups assumed from a particular variable or association between multiple variables). Research hypotheses specification is one of the most important steps when planning a scientific quantitative research study (Kalaian & Kasim, 2008).

The main goal of this project is to assess the validity and applicability of a Micro Frontend architecture in conjunction with the standardized microservices backend in the context of web application development, as opposed to the traditional monolithic frontend. Moreover, the project's final architectural structure should align with the adoption of agile development methodologies to permit incremental and iterative software design.

6.2 Evaluation Indicators

One of the basic principles of scientific research is that results must be replicable (Castillo, 2013). In order for a hypothesis to be accepted, one must be able to test it. As means of either corroborating or refuting the proposed hypothesis, evaluation indicators are required, each of which functions as a pre-defined metric that will be examined in the solution.

In the presented context, the evaluation indicators are as follows:

- Development and Deployment Isolation – evaluate the architectures isolation when it comes to component development and deployment pipeline. Can teams function independently?
- Reliability and Testability – evaluate how error prone the architecture is and whether it mitigates the existence of a single point of failure. Moreover, ease of testability should also be considered.
- Reusability – assess whether or not each micro frontend is reusable in a timely manner with a small to none added technical debt.

- User Experience – determine whether user experience is consistent throughout the multiple micro frontends that compose the web application.
- Performance – evaluate the proof of concept's performance based on response times to user input, initial loads, and overall application requests.
- Development Cost – assess the development costs of a web application from design and orchestration all the way through actual implementation.

6.3 Assessment Methodology

To finalize the evaluation process of a solution, it is important to determine what set of activities will be carried out to reach a conclusion of the aforementioned indicators towards the proposition, assessing its validity.

Simulation-based evaluations rely on high level implementations of some or all of the components in the software architecture which are then used to evaluate desired indicators (Mårtensson, 2006). Prototype applications can be executed in the intended context of the completed system, providing a great comparison to "real-life" web apps.

As such, a proof of concept web application was developed enabling an end-to-end assessment from the design process, all the way through to the actual implementation. Moreover, this evaluation method will also serve as an indicator on whether the architecture is suited for agile development methodologies.

6.4 Results

This section presents the results attained through the analysis of the proof of concept application in regards to the evaluation indicators enumerated in section 6.2.

6.4.1 Development and Deployment Isolation

By utilizing the Micro Frontends proof of concept application developed and its monolithic counterpart, it is now possible to check how both architectures compare regarding their development and deployment processes.

The Micro Frontends-based system is, by definition, composed by several separate codebases, albeit much smaller than the single codebase for the monolithic solution. This allowed for careful separation of each component according to a specific business sub-domain, more specifically, Authentication, Products and Orders.

As a result, the development process for a Micro Frontend was completely independent, as each hypothetical team focused solely on its application without worrying about the remaining ones. This enabled a case by case decision making process, illustrated by the freedom of choice regarding what technology stack to use.

This independence continues over from app development to its deployment, where each Micro Frontend has its own pipeline, responsible for testing and deploying it.

On the other hand, the monolithic solution aggregates the entirety of the solution's frontend in a single codebase which has to be deployed as a whole. This limits the system from a continuous deployment standpoint, as every single change requires building and deploying the entire thing. The Micro Frontends proof of concept contemplates per-component deployments and employs a maintenance check that warns the user in case one of its components is down for any particular reason. This way, the system can remain available while one of its parts is updated.

It should be mentioned that cross-team communication is still encouraged and necessary in the project's genesis as the App Shell component requires some basic information, namely hosting data, to orchestrate and setup the solution.

6.4.2 Reliability and Testability

System reliability is a common goal for most solutions, typically measured by assessing how error prone it is, where said errors can occur and what impact they have in the overall system.

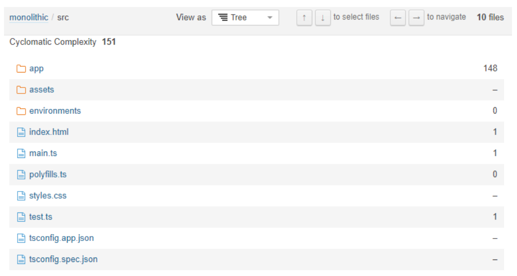
Given the nature of a monolithic application, a critical error that failed to be considered and dealt with is bound to affect the entire application. This makes the frontend application a single point of failure, as the aforementioned scenarios make the system unusable.

On the other hand, the Micro Frontend proof of concept reduces this risk, although it is not able to completely extinguish it. Since each component is its own application that is then displayed through an iframe and, given an iframe's innate isolation capabilities, critical failures are not propagated to the App Shell. This means that when an uncontrolled error occurs in one of the Micro Frontends, only said element will fail, keeping the rest of system still operational.

The reason why it was previously mentioned that this does not completely remove the single point of failure issue, is due to the App Shell. As the solution's orchestrator, responsible for loading/unloading each component, critical failures in the App Shell will still cause the entire system to go down. The risk is, however, somewhat reduced, as the App Shell is devoid of any responsibility other than routing and displaying system messages (e.g. errors).

As far as testability goes, the metric used to evaluate this indicator was the cyclomatic complexity of the applications. It is a quantitative measure of the number of linearly independent paths through a program's source code (Watson & McCabe, 1996), often associated with the number of tests cases required for full test coverage.

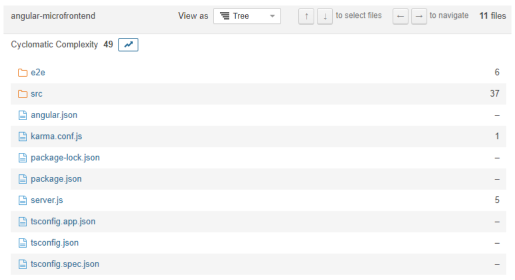
To obtain these values for both the monolithic codebase and the Micro Frontends, SonarQube was used.



Cyclomatic Complexity 151

app	148
assets	–
environments	0
index.html	1
main.ts	1
polyfills.ts	0
styles.css	–
test.ts	1
tsconfig.app.json	–
tsconfig.spec.json	–

Figure 6.1: Monolithic Cyclomatic Complexity



Cyclomatic Complexity 49

e2e	6
src	37
angular.json	–
karma.conf.js	1
package-lock.json	–
package.json	–
server.js	5
tsconfig.app.json	–
tsconfig.json	–
tsconfig.spec.json	–

Figure 6.2: Products Micro Frontend Cyclomatic Complexity

Figure 6.1 and Figure 6.2 present the cyclomatic complexity values for the monolithic (163) and Product Micro Frontend (49). The remaining Micro Frontends for Authentication and Orders scored 52 and 38, respectively.

These results show that, by splitting the solution into smaller components, the resulting components become easier to test as each focuses on their specific business sub-domain and supports fewer features. Although this leads to the expected conclusion that Micro Frontends are easier to test than a monolithic application given since each can be tested in isolation, one issue arises.

Now that the application dynamically loads content, trying to test the system as whole and simulate real-world usage can become more complicated, especially when considering an iframe's isolation properties. To circumvent this problem, tools like Cypress or Selenium can be used, as they are agnostic to the project's implementation details.

6.4.3 Reusability

When assessing the reusability aspect of both developed solutions (monolithic and micro frontends), one focuses on whether any of the developments could be used for other systems without the need to pick and choose specific blocks of code or any given feature, but instead, have a given component exported to a separate solution and becoming fully functional with minimal configuration changes.

While the monolithic solution presents no value in this matter as it is composed by a single codebase, the same cannot be said for the Micro Frontends proof of concept.

A valid example for a component that could easily be reused in other applications is the Authentication one, as it is a standard feature most systems require. By cloning both projects that make up the Micro Frontend and microservice and then performing any required configuration changes (e.g. what authentication provider is used), one can have a fully functional authentication flow in a timely manner.

This indicator would be most valuable for companies that specialize in the development of a given type of application (e.g. ecommerce apps), as the chances of having reusable Micro Frontends would inherently be higher.

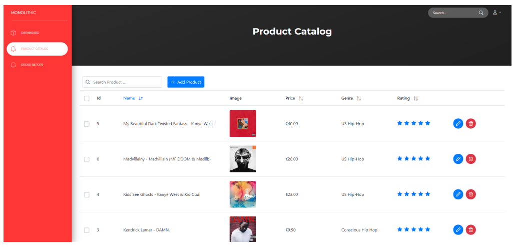
6.4.4 User Experience

Although a Micro Frontends architectural approach splits the application into smaller pieces, the resulting product should still have the look and feel of a traditional web application to the end user.

By having teams tackle separate components and using different technology stacks, it's always a concern that the final integration between them is not seamless, making them identifiable when using the system.

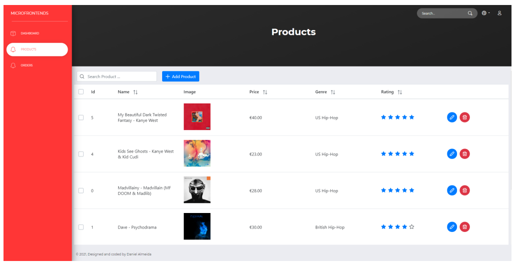
To mitigate this issue and provide a compatible user experience throughout the entirety of the system, the same user interface component library was used as it supported both React and Angular projects. The same result could be achieved by having styling guidelines put in place for all teams to follow before starting the development stage.

By comparing the monolithic and Micro Frontends applications, it is possible to visualize how identical the final products are.



Id	Name	Image	Price	Genre	Rating
5	My Beautiful Dark Twisted Fantasy - Kanye West		€40.00	US Hip-Hop	★★★★★
0	Madvillainy - Madvillain (MF DOOM & Madlib)		€28.00	US Hip-Hop	★★★★★
4	Kids See Ghosts - Kanye West & Kid Cudi		€23.00	US Hip-Hop	★★★★★
3	Kendrick Lamar - DAMN.		€9.90	Conscious Hip Hop	★★★★★

Figure 6.3: Monolithic Product Catalog



Id	Name	Image	Price	Genre	Rating
5	My Beautiful Dark Twisted Fantasy - Kanye West		€40.00	US Hip-Hop	★★★★★
4	Kids See Ghosts - Kanye West & Kid Cudi		€23.00	US Hip-Hop	★★★★★
0	Madvillainy - Madvillain (MF DOOM & Madlib)		€28.00	US Hip-Hop	★★★★★
1	Dave - Psychodrama		€30.00	British Hip-Hop	★★★★☆

© 2021, Designed and coded by Daniel Almeida

Figure 6.4: Micro Frontend Product Catalog

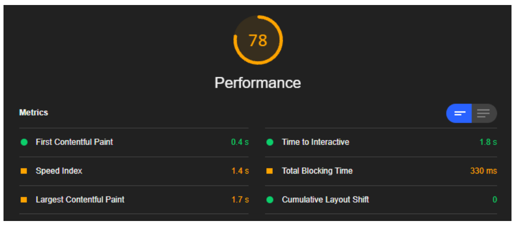
Figure 6.3 and Figure 6.4 display the product catalog when viewed in the monolithic and Micro Frontend solutions, respectively. The final products are nearly identical and the actual usage of the page does not require any additional actions by the user, being completely the same flow-wise.

By implementing an event-based messaging mechanism between the iframes and the App Shell, both solutions also present user messages/notifications in the same fashion.

6.4.5 Performance

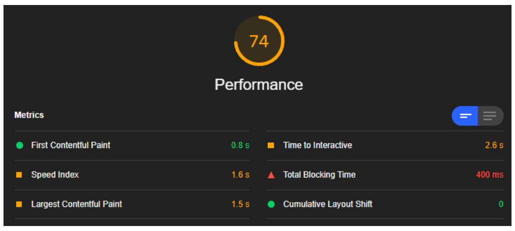
In respects to performance, Google Lighthouse was used to measure both the solutions' quality. It is an open-source tool that can be run against any web page, auditing performance and optimization (Developers, 2021).

It is important to note that reports generated with the tool made no use of the browser's cache in order to simulate a user's initial landing on one of the pages, albeit already authenticated.



```
Metrics
● First Contentful Paint     0.4 s    ● Time to Interactive        1.8 s
■ Speed Index                1.4 s    ■ Total Blocking Time       330 ms
■ Largest Contentful Paint   1.7 s    ● Cumulative Layout Shift      0
```

Figure 6.5: Monolithic Lighthouse Performance Report



```
Metrics
● First Contentful Paint     0.8 s    ■ Time to Interactive        2.6 s
■ Speed Index                1.6 s    ▲ Total Blocking Time       400 ms
■ Largest Contentful Paint   1.5 s    ● Cumulative Layout Shift      0
```

Figure 6.6: Micro Frontend Lighthouse Performance Report

Figure 6.5 and Figure 6.6 respectively show the performance scores for the monolithic and Micro Frontend solutions. As the results show, there is a slight drop-off in the overall score for the Micro Frontend proof of concept, mostly caused by **Time to Interactive** metric which, as per the official documentation, dictates the amount of time it takes for the page to become fully interactive (Developers, 2021).

This difference in results is easily explained by the additional requests required to load the respective Micro Frontends when landing on a page, as opposed to having everything a singular application, as is the case with the monolithic solution.

Regarding payload size, the monolithic solution did a total of 22 requests with a combined transfer size of 945.7 kilobytes (kB), while the Micro Frontends proof of concept app had 30 total requests with a transfer size of 1129.0 kB. This equates to a 19.38% increase.

By having each component be a standalone application that has their own dependencies, it was expected for the modular approach to have larger payloads requested by the browser, some of which may even be duplicates, as is the case for the styling sheets used. While projects get bigger, dependencies must be carefully managed to avoid performance issues with each Micro Frontend introduced.

It is important to note that employing an external user interface component library brought about a lot of unwanted style sheets, most ending up being unused in the application. The usage of proprietary styles would have been more appropriate for this dissertation's project.

6.4.6 Development Cost

When evaluating development costs associated with a given project, it is important to consider both its design and implementation, as well as future maintenance and developments.

Starting off with the design and implementation, this is a stage where development costs were higher for the Micro Frontends proof of concept when compared to the monolithic one. This is a direct consequence of the architectural approach, as the existence of **n** amount of components means there are **n** amount of frontends to design, setup and implement. There are also more orchestration and governance concerns given the need for a container application, namely the App Shell in this instance, to put together the entire system as a whole.

While this seems to declare a clear win for the monolithic solution as far as development costs go, there is still the matter of future maintenance and developments. In this aspect, the tables turn in favor of the Micro Frontends architectural approach, as tasks such as bug fixing become simpler when executed over a smaller, more granular codebases.

Futures additions to the solution can now also become new Micro Frontends to be designed and implemented, as opposed to having an ever-increasing codebase that increases in complexity as time goes by and the solution grows.

These conclusions show that, development costs-wise, the decision to go with Micro Frontends or a standard frontend monolith will vary according to future prospects and how far the project is expected to grow.

6.5 Summmary

To properly evaluate the research hypotheses proposed in the beginning of the current chapter, a case study was developed and assessed according to the evaluation indicators listed in section 6.2.

The first hypothesis regarded the validity and applicability of Micro Frontend-based architectures, when composed with their corresponding microservices, as an alternative to monolithic frontend development practices.

Upon evaluation, it is possible to conclude that, in regards to the aforementioned hypothesis, all the evaluation indicators were either partially or completely fulfilled. The solution was able to provide a similar and consistent user experience to the one provided by its monolithic counterpart while exceeding it in other matters, namely reliability, reusability and testability (with a small caveat regarding real-world testing). However, it did slightly lag behind it in terms of performance even though their testing scores were similar.

Finally, the development costs evaluation revealed that both approaches have their pros and cons, making them a better fit for different scenarios: monolithic frontend for smaller solutions with fewer growth prospects and Micro Frontends for highly scalable systems.

The second hypothesis assessed the architecture's merit when it comes to adopting agile development methodologies. The development and deployment isolation evaluation indicator successfully tests this hypothesis as it revealed that, by having a system decomposed into smaller, more manageable pieces that can be independently developed and deployed, it becomes easier to manage an incremental process to develop, deliver and test specific features.

Chapter 7

Conclusion

While architectural styles like microservices have revolutionized the way backend development is perceived, frontend developers appear to still have a proclivity towards building their apps through a singular codebase, a monolithic application. This approach to web application development has several flaws which are described in section 2.2.1.

The main objective of this dissertation was to assess the validity and worth of Micro Frontends as an alternative to the traditional frontend monolith. The concept was simple, take what was learned with the usage of microservices and apply it to frontend development. Section 2.4 introduces the concept of Micro Frontends while also presenting its theoretical advantages and disadvantages.

With a better understanding of the concept and what it provides, research was performed to evaluate how to employ this architectural style. The research conducted illustrated that multiple approaches, both technology and composition wise, can be perceived as applicable and provide respective advantages and disadvantages (section 2.5).

By means of a value analysis and, with the aid of a decision analysis method (namely TOPSIS), the alternatives were compared in order to arrive at a conclusion on what approach would be best suited for the proof of concept application (section 3). The resulting decision was to leverage iframes with client-side composition to build the solution.

Afterwards the Micro Frontends system was designed alongside an equivalent monolithic application, both tackling the same use cases and utilizing the same microservices. Chapter 4 thoroughly details this process by presenting the requirements, both functional and non-functional, detailing how the components interact and how the respective system architectures were plotted out. A Micro Frontend was created per business sub-domain, resulting in three separate components: Authentication, Products and Orders.

After that, the solutions were implemented according to their design to try and meet all their requirements.

7.1 Achieved Requirements

As far as what requirements were actually achieved by the final solution, starting off with the functional ones, all of them were fulfilled. This was the expected outcome since simple features were chosen due to the higher degree of importance of the non-functional requirements that better illustrate how the architecture betters the system.

In that regard and kicking off the with the usability factor, it is safe to conclude that said requirement was fulfilled as the system provides a nigh identical experience to the monolithic application, even though it used separate components with different technology stacks.

Now moving towards reliability, this requirement was partially fulfilled given that the system does handle errors in a compartmentalized way, making it so an untreated error in one of the Micro Frontends does not cause the entire solution to crash. There is, however, an issue with catastrophic failures in the App Shell component, as they would cause the whole system to become unavailable.

Performance-wise, the proof of concept did meet the requirement, partially due to the size of the application. Like the evaluation performed in section 6.4.5 entails, the Micro Frontends solution scored similar results to the monolithic one in the performance analysis, while also illustrating that the former did utilize more network calls to attain its required dependencies, hence why the requirement can be considered to have been met in this specific scenario while raising some concerns for bigger, more complex projects.

The proposed design constraints were fulfilled, as the system was composed by multiple Micro Frontend/microservice combinations, each tackling a specific business sub-domain. This also made it possible to fulfill the supportability requirement and individually test and maintain said components.

Lastly, the implementation constraints were respected, given that the microservices were developed using the suggested technology and then ran in Docker containers, while the Micro Frontends varied from using Angular and React as their frameworks.

7.2 Contributions

The developed system and its analysis act as a contribution towards the Micro Frontend architecture subject matter, while focusing primarily on runtime integration (more specifically, client-side composition) from an orchestration standpoint and leveraging iframes to dynamically load components.

The Micro Frontends proof of concept solution developed demonstrates how a system like this can be built while resorting to the usage of Angular and React frontend applications, providing a step by step description of the process.

A monolithic counterpart of the aforementioned solution was also implemented to help facilitate any required comparisons between the design (section 4) and implementation (section 5) processes, as well as evaluating the final products (section 6).

7.3 Challenges and Limitations

Despite being able to meet its requirements, some challenges and limitations still presented themselves during this case study.

As Micro Frontends are a relatively new concept in the grand scheme of things, the scarce amount of scientific articles that analyse this topic definitely cripples the research aspect of this dissertation. While there is an abundance of information surrounding the topic when resorting to popular search engines, it is mostly a mere introduction to the subject matter accompanied by some of the assumptions that it entails.

Another limitation that presented itself came from the inability to develop the proof of concept application in a real-world scenario with multiple development teams, having to have the author emulate this environment by himself. This, in addition to the complexity of the developed applications, does impair the results obtained in its evaluation.

7.4 Future Work

Given the vast array of possible solutions one could follow to successfully implement a Micro Frontends architecture, it would be ideal to develop other proof of concept applications using different methods for a better comparison between the different techniques.

As the concept matures, new alternatives are surely to follow, which would make this architecture more robust, easier to achieve and maintain. These should also be targeted for future work.

To tackle one of the limitations encountered while developing, it would also be a great addition to the architecture's merit to have a proof of concept designed and implement in a real-world environment, with multiple teams working on it.

Bibliography

ASIAA, IDIA, & NRAO. (2019). Client-server architecture. https://carta.readthedocs.io/en/1.0/introduction.html

bit.dev. (2021). Micro-frontends architecture is defined into smaller and simpler units. https://docs.bit.dev/docs/workflows/microfrontends#:~:text=Micro%5C%2Dfrontends%5C%20architecture%5C%20is%5C%20defined,into%5C%20smaller%5C%20and%5C%20simpler%5C%20units.

Castillo, M. (2013). The scientific method: A need for something better? *American Journal of Neuroradiology September 2013*. http://www.ajnr.org/content/34/9/1669

Developers, G. (2021). Tools for web developers - lighthouse. https://developers.google.com/web/tools/lighthouse

Eeles, P. (2005). Capturing architectural requirements. https://web.archive.org/web/20201112020231/http://www.ibm.com/developerworks/rational/library/4706.html#N100A7

Fulton, R., & Vandermolen, R. (2014). Airborne electronic hardware design assurance: A practitioner's guide to rtca/do-254. https://books.google.pt/books?id=ZQMvDwAAQBAJ&pg=PA89&redir_esc=y#v=onepage&q&f=false

Geers, M. (2020). Micro frontends in action. https://livebook.manning.com/book/micro-frontends-in-action/meap-version-3/v-3/

Gnatyk, R. (2018). Should you use a monolithic architecture? *Microservices vs Monolith: which architecture is the best choice for your business?* https://overcoded.dev/posts/Arch-7

Jackson, C. (2019a). Micro frontends. *https://martinfowler.com/*. https://martinfowler.com/articles/micro-frontends.html?utm_source=arador.com

Jackson, C. (2019b). Micro frontends. *martinfowler.com/*. https://martinfowler.com/articles/micro-frontends.html?utm_source=arador.comIntegrationApproaches

Jacobson, I., Spence, I., & Bittner, K. (2011). Use-case 2.0 - the guide to succeeding with use cases. https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/use-case_2_0_jan11.pdf

Joseph, R. (2015). Single page application and canvas drawing. https://www.researchgate.net/publication/272194242_Single_Page_Application_and_Canvas_Drawing

Kalaian, S. A., & Kasim, R. M. (2008). Research hypothesis. in p. j. lavrakas (ed.) *Encyclopedia of survey research methods (pp. 732-733). SAGE Publications, Inc.* http://sk.sagepub.com/reference/survey/n472.xml

Koen, P. A., M.Ajamian, G., Boyce, S., Clamen, A., Fisher, E., Fountoulakis, S., Johnson, A., Puri, P., & Seibert, R. (2002). 1 fuzzy front end: Effective methods, tools, and techniques. *The PDMA ToolBook for New Product Development*. https://media.wiley.com/product_data/excerpt/13/04712061/0471206113.pdf

Lorraine. (2011). Sass vs. scss: Which syntax is better? https://thesassway.com/sass-vs-scss-which-syntax-is-better/

Mamczur, P., Gajda, M., Kajtoch, T., Toporek, W., Wysoczański, A., Swoboda, J., Habarta, M., & Głowiński, K. (2020). State of frontend 2020. *The Software House*. https://tsh.io/state-of-frontend/

Mårtensson, F. (2006). Software architecture quality evaluation. https://www.diva-portal.org/smash/get/diva2:838153/FULLTEXT01.pdf

McCurdy, N. (2020). Create react app. https://create-react-app.dev/docs/getting-started/

Meder, S., Antonov, V., & Chang, J. (2017). Driving user growth with performance improvements. *Pinterest Engineering Blog*. https://medium.com/pinterest-engineering/driving-user-growth-with-performance-improvements-cfc50dafadd7

Media, C. (2020). Micro front-ends is the latest web development trend. https://www.cygnismedia.com/blog/micro-frontends-development

Mezzalira, L. (2020). Micro-frontends in context. *https://increment.com/*. https://increment.com/frontend/micro-frontends-in-context/

Miller, J., & Osmani, A. (2020). Rendering on the web. *Web Fundamentals*. https://developers.google.com/web/updates/2019/02/rendering-on-the-web

Mozilla. (2020). <iframe>: The inline frame element. *MDN Web Docs*. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

Neoteric. (2016). Single-page application vs. multiple-page application. *NeotericEU*. https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

Osterwalder, A., & Pigneur, Y. (2006). Business model generation. *Business Model Generation: A handbook for visionaries, game changers, and challengers, 2010*. https://books.google.pt/books?hl=en&lr=&id=UzuTAwAAQBAJ&oi=fnd&pg=PA9&dq=A.+Osterwalder+e+Y.+Pigneur,+Business+Model+Generation:+A+handbook+for+visionaries,+game+changers,+and+challengers,+2010.&ots=yYCRxeK6Xx&sig=X0fS6EbetcDxqreQYn18PmqSoVQ&redir_esc=y#v=onepage&q&f=false

Pautasso, C., Zimmermann, O., & Leymann, F. (2008). Restful web services vs. big web services: Making the right architectural decision. http://www.jopera.org/docs/publications/2008/restws

Pupius, D. (2013). Beyond pushstate — building single page applications. *Writing by Dan Pupius*. https://writing.pupius.co.uk/beyond-pushstate-building-single-page-applications-4353246f4480

Rahim, R., Supiyandi, S., Siahaan, A. P. U., Listyorini, T., Utomo, A. P., Triyanto, W. A., Irawan, Y., Aisyah, S., Khairani, M., Sundari, S., & Khairunnisa, K. (2018). Topsis method application for decision support system in internal control for selecting best employees. https://iopscience.iop.org/article/10.1088/1742-6596/1028/1/012052/pdf

Rich, N., & Holweg, M. (2000). Value analysis, value engineering. *INNOREGIO project*. https://www.urenio.org/tools/en/value_analysis.pdf

Richardson, C. (2018). Pattern: Microservice architecture. *Microservice Architecture*. https://microservices.io/patterns/microservices.html

Shah, K. (2020). Speed and responsiveness. *Blog thirdrocktechkno.com*. https://www.thirdrocktechkno.com/blog/single-page-apps-vs-multi-page-apps-what-to-choose-for-web-development/

Spoonhower, D. (2020). Microservices architecture. *Microservices Architecture: When and How To Move To Microservices*. https://lightstep.com/blog/microservices-architecture-when-and-how-to-move-to-microservices/

Tsimelzon, M., Weihl, B., Chung, J., Frantz, D., Basso, J., Newton, C., Hale, M., Jacobs, L., & O'Connell, C. (2001). Esi language specification 1.0. *W3C Note 04 August 2001*. https://www.w3.org/TR/esi-lang/

Ulaga, W., & Eggert, A. (2006). Relationship value and relationship quality: Broadening the nomological network of business-to-business relationships. *European Journal of Marketing*. https://www.emerald.com/insight/content/doi/10.1108/03090560610648075/full/html

Watson, A. H., & McCabe, T. J. (1996). Structured testing: A testing methodology using the cyclomatic complexity metric. http://www.mccabe.com/pdf/mccabe-nist235r.pdf

What are web components? (2019). *webcomponents.org*. https://www.webcomponents.org/introduction

WHATWG. (2021). Integration with the javascript agent formalism. *HTML Living Standard*. https://html.spec.whatwg.org/multipage/webappapis.htmlintegration-with-the-javascript-module-system