INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Development of an Angular Components Library to be used in Micro-Frontend Architecture

CARINA RAQUEL FERREIRA ALAS
Setembro de 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Development of an Angular Components Library to be used in Micro-Frontend Architecture

Cárina Alas

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Informatics Engineering, Specialisation Area of Games, Graphical and Interactive Systems

Supervisor: Dr. Paulo Baltarejo de Sousa

Porto, September 12, 2024

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, September 12, 2024

Cárina Alas

Abstract

The use of **Micro-frontend** architecture in the development of web applications has come to increase, as previously monolithic architectural structures are being replaced by a modular system composed of loosely coupled components. However, concerns regarding the consistency of user interface and user experience design arise since each micro-frontend acts as an independent service and implementation.

The integration of a **Design System** within a project based upon a micro-frontend architecture becomes crucial and the use of a **Components Library** to share components can, not only, be the solution to the user interface consistency, but also promote lack of code duplication and improve maintainability. The use of the **Atomic Design** methodology can assist the creation and documentation of the design system components, further enhancing a modular structure to the components that will be integrated in the system.

The solution entailed the development of an **Angular** web application within an architecture composed of a web API, several micro-frontends, a shell application and a components library that was integrated into the several frontend projects. Quality gate metrics and deployment frequency were used to qualify the project, that met expectations in regards to code duplication and maintainability. A user survey was used to gather data to evaluate the design consistency against hypothesis tests and a System Usability Scale test.

It could be concluded that a Components Library can be used in order to provide design consistency and cohesion throughout a web application, built using current industry standards.

Keywords: Micro-frontend, Design System, Components Library, Atomic Design, Angular

Resumo

O uso da arquitetura **Micro-frontend** no desenvolvimento de aplicações web tem aumentado, à medida que estruturas arquitetónicas anteriormente monolíticas vêm a ser substituídas por um sistema modular composto por componentes fracamente acoplados. No entanto, surgem preocupações quanto à consistência da interface gráfica e do *design* da experiência do utilizador, uma vez que cada micro-frontend atua como um serviço e possui uma implementação independente.

A integração de um **Design System** dentro de um projeto baseado em uma arquitetura micro-frontend torna-se crucial e o uso de uma **Biblioteca de Componentes** para compartilhar componentes pode, não só, ser a solução para a consistência da interface, mas também para promover a diminuição de duplicação de código e melhorar a capacidade de manutenção. A utilização da metodologia de **Design Atómico** pode auxiliar na criação e documentação dos componentes do *design system*, potencializando ainda mais uma estrutura modular aos componentes que serão integrados no sistema.

A solução implicou o desenvolvimento de uma aplicação web em **Angular**, numa arquitetura composta por uma *web API*, vários micro-frontends, uma aplicação *shell* e uma biblioteca de componentes que será integrada nos vários projetos de *frontend*. Métricas de qualidade e de frequência de *deployment* foram utilizadas para qualificar o projeto, que atendeu às expectativas em relação à duplicação de código e manutenibilidade. Um questionário respondido por usuários da aplicação foi usado para adquirir dados para avaliar a consistência do *design* em relação a testes de hipóteses e um teste de Escala de Usabilidade do Sistema.

Pode-se concluir que uma Biblioteca de Componentes pode ser utilizada para fornecer consistência e coesão de *design* numa aplicação web, construída usando os padrões atuais da indústria.

Acknowledgement

To my supervisor, professor Dr. Paulo Baltarejo de Sousa, who was always eager to help and review each step of the way in the writing of this dissertation.

To the ones who helped me SWitCH career paths and opened up my world to embrace software development, all of the professors that helped us through it, and to the people who made it all possible to endure - Group 2 from SWitCH 2018/19. Specially the ones who stayed in my life until today - André, Daniela, João, Maria and Miguel.

To the friends I've made along the way in this master's journey, for pushing me to deliver my best work, sometimes making me question my patience, but also bringing out the best in me. Special thanks to the *Famiglia Unicorn* group - André Gonçalves, André Morais, Daniel Dias, Francisco Dias, Luís Sousa, Miguel Cabeleira, Narciso Correia and Vitor Neto.

To all my mentors in my software development career so far, in the companies where I've been, for teaching me what being a real software engineer is about, and for trusting me.

To my biggest supporter, my best friend and soulmate in the whole world. Thank you Fábio, I could not have done this without you by my side.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

ACM	Amazon Certificate Manager.
API	Application Programming Interface.
AWS	Amazon Web Services.
BDD	Behavior-Driven Development.
CD	Continuous Delivery.
CDN	Content Delivery Network.
CI	Continuous Integration.
CLI	Command Line Interface.
CPU	Central Processing Unit.
CSF	Component Story Format.
CSS	Cascading Style Sheets.
DDD	Domain Driven Design.
DNS	Domain Name System.
DOM	Document Object Model.
E2E	End-to-end.
EC2	Elastic Compute Cloud.
ECR	Elastic Container Registry.
ECS	Elastic Container Service.
ER	Entity Relationship.
ES6	ECMAScript 6.
ESI	Edge Side Includes.
GDPR	General Data Protection Regulation.
HREF	Hypertext Reference.
HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.
HTTPS	Hypertext Transfer Protocol Secure.
IAM	Identity and Access Management.
IP	Internet Protocol.
JSX	JavaScript XML.
JWT	JSON Web Token.
MDX	Markdown/JSX.
MVC	Model-View-Controller.

NPS	Net Promoter Score.
OOP	Object Oriented Programming.
PoC	Proof of Concept.
REST	Representational State Transfer.
SCSS	Sassy Cascading Style Sheets.
SDLC	Software Development Life Cycle.
SFC	Single-File Component.
SPA	Single Page Application.
SSL	Secure Sockets Layer.
SSR	Server-Side Rendering Application.
SUS	System Usability Scale.
TLS	Transport Layer Security.
UI	User Interface.
URL	Uniform Resource Locator.
UX	User Experience.
VPC	Virtual Private Cloud.
XML	Extensible Markup Language.

Chapter 1

Introduction

This chapter exposes and contextualizes the problem that will be tackled in the dissertation while also interpreting it in an analytical and critical way, and additionally stating the research objectives, processes, methods and questions.

1.1 Initial Context

Frameworks for the development of the presentation (frontend) layer of web applications have evolved to provide a number of valid choices for creating robust, feature-packed applications. These options include Single Page Application (SPA), Server-Side Rendering Application (SSR), and the integration of static HyperText Markup Language (HTML) files to form a cohesive web page (Peltonen, Mezzalira, and Taibi 2021).

However, they still majorly rely on monolithic architectural structures. Once the presentation layer starts to grow, this often leads to hardships in terms of scalability of the application, since in most cases different teams work on the same frontend at the same time.

Today we are witnessing the rise of the use of micro-frontend architecture, in which the frontend is divided into individual and semi-independent micro-frontends, separating the business logic and creating services that interact with each other, similar to the concepts of microservices. This allows web applications to be divided into independently maintainable and deliverable components, each with their own business domain. In a company, this also allows for each team to be independently responsible for one micro-frontend (Pavlenko et al. 2020).

Nonetheless, micro-frontends come with certain disadvantages. These include the possibility of communication overhead if the system lacks careful design and adaptation for business growth, potential performance challenges when the incorporation of third-party components (services, or external entities) are not thoroughly considered, and a disrupted user experience when the governance of a design system is inadequately planned (Peltonen, Mezzalira, and Taibi 2021). The latter will be the main focus of this dissertation's problem.

1.2 Problem

A micro-frontend architecture may lead to inconsistencies on the User Interface (UI) design and User Experience (UX) of the application due to different teams developing different components on each micro-frontend. This can also be further enhanced due to the fact that nowadays applications are developed by teams consisting of a high number of elements

and, because of remote working practices, they can be at times separated physically making communication between team members less effective.

Effective UI/UX design fosters a higher user satisfaction, and in improved conversion rates and increased revenue resulting from a higher engagement in the application usage. Maintaining consistency is a fundamental principle in effective UI design, as it establishes a feeling of familiarity and predictability for users (Akram 2020). An efficient and unique design system is key in developing a brand secure identity.

UI/UX consistency is not only about aesthetics but also about optimizing the development process and ensuring a superior experience of the application. It fosters efficient code management, reduces errors, and ultimately contributes to the success of web application development by making it more maintainable and scalable, encouraging the reuse of code components, therefore reducing code duplication and effort.

When it comes to web development, available component frameworks can have limitations and expenses associated with them, primarily because web applications and companies often have unique branding and design requirements that cannot be easily replicated by using off-the-shelf libraries, which leads to a need of a distinctive and individual design system.

1.2.1 Interpretation

In micro-frontend setups, sharing code and information among different parts of the system is common. To enable code sharing, tools to publish public libraries such as npm (docs.npmjs 2023) are used, which ensures code reusability and improves development efficiency (Gómez 2023).

Pavlenko et al. 2020 exposes that shared component libraries can solve the problem of UI/UX consistency on micro-frontends. Said libraries should contain reusable UI elements. This allows each development team to combine components from the library to assemble their wanted interface. The library can include both simple components (icons, labels, buttons), and complex components, that contain shared logic and UI that can be consisting of a group of smaller elements.

Available shared design styles or frameworks (such as available NG-Bootstrap (ng-bootstrap 2023) or Material Design (material.angular.io 2023)) can also be used to maintain a consistent appearance and UI/UX experience, at a lower cost (Gómez 2023). However other studies have found that popular, free to use Angular Libraries projects don't present as the most performant and may be lacking key features (Cebrian 2017).

Looking at the problem from a critical perspective, it is evident that the integration of a component library within a micro-frontend architecture should be the main way to guarantee consistency, reduce code duplication and assure maintainability and stability of the different components of the system. At the same time, making sure that a correct design system is implemented in accordance to brand identity is essential for companies to compete in today's market, even if it comes with time and monetary expenses. That being said, the investment of developing an independent library could bring more advantages than disadvantages in that case.

1.3 Objectives

This project will carry out a research about the current state of the art when it comes to micro-frontend architecture, available frameworks, design systems and component libraries.

The main goal will be to develop a Proof of Concept (PoC) of a Components library using Angular Typescript, that can be used on a web application divided into several micro-frontends. The development of this PoC will have at its core the following goals:

- Accessibility: this pertains to making components readily accessible for developers to reuse. The goal is to create a design system that provides easy access to these components, ensuring that development teams can efficiently leverage and integrate them into various micro-frontends. This accessibility streamlines the development process and promotes consistency across different parts of the application;
- Reduced code duplication: Another primary aim is to eliminate code redundancy. By centralizing design elements and functionalities within the design system, developers can significantly reduce code duplication across different micro-frontends. This not only streamlines development but also minimizes maintenance efforts;
- Consistency: Achieving a high level of consistency in both UI and UX is the main objective of this project. The design system should provide guidelines and tools that foster a uniform look and feel across various micro-frontends, enhancing the overall user experience and brand identity.

Said library will be integrated within a micro-frontend architecture and implemented in each micro-frontend, that will be deployed into a single web application.

1.3.1 Research Process

The research process will be done by following Oates, Griffiths, and McLean 2022's model (fig. 1.1), more specifically the highlighted segments in red.

As seen in the model, this should start by a Literature review, then it will consist of defining a set of research questions and the main work of the dissertation will have the goal of finding answers to these questions. Experience and rationale are the motivators behind the interest in the project.

When it comes to the research questions, a main question comes in mind, as well as some sub questions:

- Main question (RQ1): In a micro-frontend architecture, how can we maintain consistent UI and UX design of the application, in a case where external libraries are discouraged, or the project should have a unique design?
- RQ2: In a micro-frontend architecture, how can we reduce code duplication, enhance the development process, and make the application more maintainable and scalable?
- RQ3: In a micro-frontend architecture, how can we integrate and deploy a components library package, in order to promote reusable components, reduced code duplication, consistency and compatibility?

These research questions will be addressed throughout the different chapters of this project and the main goal of the dissertation will be to answer them accordingly.

The research strategy will be “Design and Creation” since this strategy is used to define how the system should be (design), implementing the artefact (build), evaluate it, and draw conclusions from this process. This applies to this project since a PoC Components library that can be used on a web application divided into several micro-frontends will be designed and developed.

As for the evaluation metrics, data can be collected by "Questionnaires", in order to determine if the application is consistent in terms of UI and UX (using for example usability tests and user feedback). Data could also be collected using "Observational" methods, such as relying on code analysis to guarantee reduced code duplication.

"Quantitative" and analytical methods can be used to provide statistical evidence that the application is considered to be consistent in terms of design even if it’s made up of two or more micro-frontends. This can be done by analysing the answers to the questionnaires, through hypothesis tests for example, and the code analysis evidence. The interpretation of the user feedback through the surveys will also include a "Qualitative" data analysis, since a subjective interpretation could also take place.

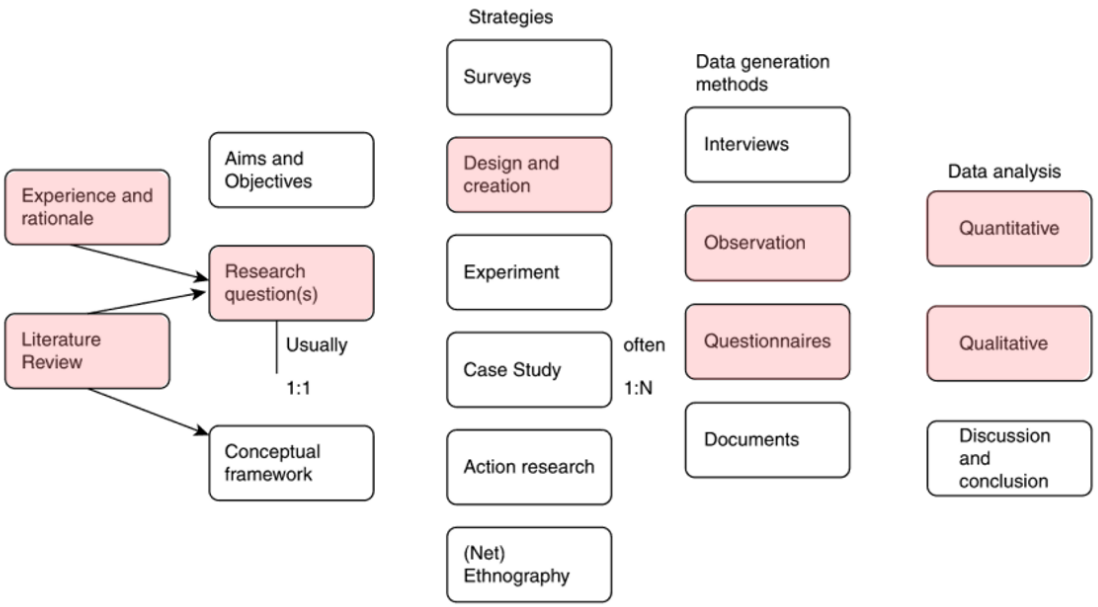


Figure 1.1: Oates, Griffiths, and McLean 2022’s model of Research process, with highlights of what will be followed in this project.

1.4 Structure

The following document presents a structure divided by chapters, each containing information regarding the objectives of this project:

- **Chapter 1 - Introduction**: in which the background and context for the project being conducted is presented, as well as the interpretation of the problem to solve and objectives;
- **Chapter 2 - State of the Art**: a state-of-the-art review following the recent developments in micro-frontends, Design Systems and Component Libraries;

- **Chapter 3 - Solution Planning**: schedule of activities planned to complete the project, including the planning of the design and implementation and planning for the experimentation and evaluation procedure;
- **Chapter 4 - Solution Design**: exposes the architectural design of the project, together with the UI design of the web application, in order to allow for the development of the design system;
- **Chapter 5 - Solution Implementation**: outlines the development of the project, including the implementation and project management, testing, deployment and documentation, as well as the maintenance plan;
- **Chapter 6 - Experimentation and Evaluation**: outlines the methodology used to test the hypotheses and validate the proposed solution, together with results and discussion;
- **Chapter 7 - Conclusions**: rounds up the document and mentions limitations and future work remarks.

Chapter 2

State of the Art

This chapter will gather a literature review of the current state of the art relevant to the problem at hand. It starts with exploring microservice architecture - as the base for micro-frontends. Then, with a thorough exploration of micro-frontend architecture, the problem is uncovered and an exploration regarding design systems and component libraries is made. A technology comparison will be done in order to see which will be the most suited for the development of the solution, and finally, some existing component libraries' implementation and design are described.

2.1 Microservice Architecture

In this section, a brief review of the history of microservices architecture is exposed, from monolithic applications, to backend microservices and micro-frontends.

2.1.1 Microservices

In the context of web development, a web application typically comprises three integral components: a client-side graphical UI coupled with its associated logic (commonly referred to as the frontend), a server-side application or computational engine (the backend), and the database where information is stored (Schäffer et al. 2019). In traditional monolithic web applications, these components often lack clear separation of responsibilities among the developers working on different sections, resulting in inter-dependencies that can hinder scalability and flexibility (see fig. 2.1 - full monolith).

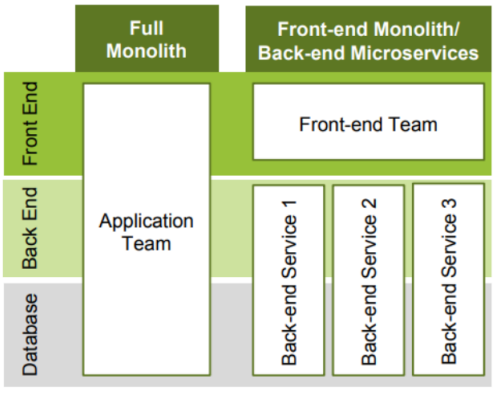


Figure 2.1: Responsibilities within the different architectures, full monolith (left) and backend microservices (right) (adapted from Schäffer et al. 2019).

As a response to the limitations of monolithic architectures, a growing trend is emerging in the direction of transitioning these systems into a more modular, microservices structure. This paradigm shift involves breaking down the monolithic application into smaller, more manageable services or modules that perform specific tasks, thus enhancing flexibility and scalability (see fig. 2.1 - backend microservices) (Geers 2023).

A microservice architecture then "*structures the application as a set of independently deployable, loosely coupled, components, a.k.a. services*" (Richardson 2019). These services are usually separated by business domain and are independently developed by the team/teams that are responsible for that business domain. In these cases, for each service there is a single source code repository, with its own deployment pipeline that enables the independent deployment of the service (Richardson 2019).

Plenty of big and large scale companies and websites (such as Netflix, Coca Cola, Amazon and eBay) have migrated from a monolithic architecture to a microservice architecture (Richardson 2019, Hannousse and Yahiouche 2021).

This new architectural design allowed for major scalability improvements, since it is no longer necessary for the whole system to scale if only one particular microservice has an increase on demand, as resources can be assigned accordingly to the needs of each individual microservice (Gitlab 2022). By being simpler to deploy they also provide a faster time-to-market, allowing for new features to be rolled out without affecting the other components in the application. Microservices are also attractive for engineers and teams since they tend to follow the latest recommended engineering practices, and allow for team optimization, autonomy and development flexibility.

However, a microservices architecture may bring some development challenges that are good to keep in mind when designing the architecture of an application. Some of these challenges may include, according to Gitlab 2022:

- *"Complexity: As the number of microservices increases, they can become more complex to manage and orchestrate"*;
- *"Data consistency: microservices work together within an extensive system while maintaining their own data. That means there will be common data across multiple services. Changes must be replicated across all consumer services whenever there are changes to such common data"*;
- *"Risk of communication failure: Unlike in-process function calls, calls to other services can fail over the network. That's a necessary trade-off with the resiliency and scalability gained from the microservices architecture"*.

According to Hannousse and Yahiouche 2021 microservices are also prone to security breaches that have increased threatening confidentiality, integrity and availability of these systems.

A specialized microservices architecture approach is the implementation with micro-frontends, also called composition or plugin approach (Schäffer et al. 2019).

2.1.2 Microservices and Micro-frontend Composition

The ongoing trend for frontend applications was to serve as a SPA existing on top of a microservice architecture, that eventually grows into a frontend monolith that becomes hard to maintain (Geers 2023). Since the concept used for backend microservices possesses

various advantages, it is to be expected that the same would be extended to be used to replace monolithic frontend applications (Prajwal, Parekh, and Shettar 2021).

Developers tend to integrate micro-frontend architecture into a composition of features or a composition of view (Geers 2023, Schäffer et al. 2019). In this composition, each team is responsible for one business area, and that team (or placeholder) will be responsible for developing features for that unique component, from the database to the backend and finally the frontend. A "*way to technically implement the composition of the view is to have the placeholders assembled by a webserver before the HTML page is delivered using (...) server-side rendering*" (Schäffer et al. 2019). Refer to fig. 2.2 (a) for a frontend monolith inserted into a microservice architecture, versus fig. 2.2 (b) in which a composition of view is displayed.

Nonetheless, this concept is far from new and shares similarities with the self-contained systems concept. In earlier iterations, similar strategies were also referred to as "Frontend Integration for Verticalized Systems" (Geers 2023).

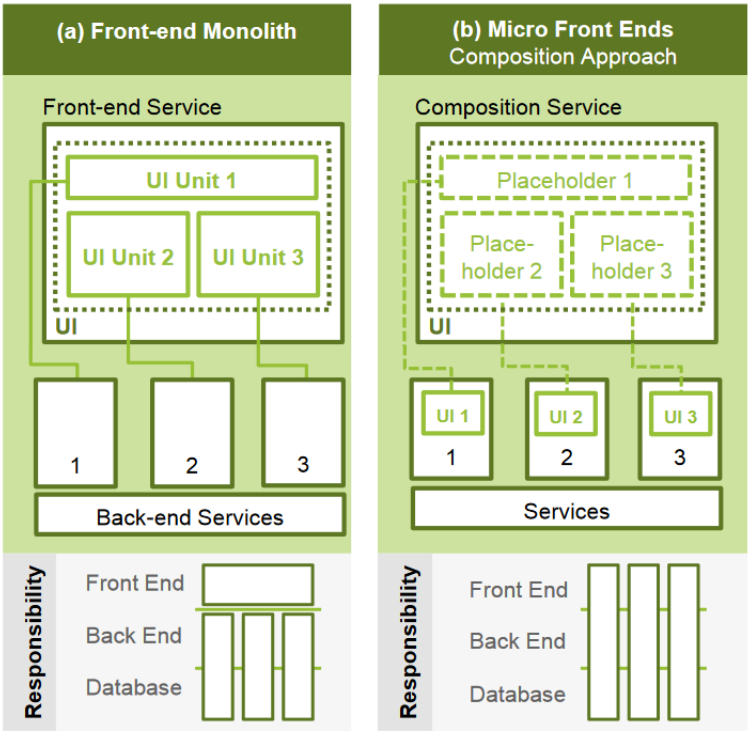


Figure 2.2: (a) Monolithic frontend; (b) Micro-Frontends within the composition approach (Schäffer et al. 2019).

Vertical organization in teams allows for the possibility of developing individual components using different technologies, providing more autonomy for the developers. The flexibility provided by this pattern increases by the fact that each service can be deployed independently, since a single change or added feature to one of the components does not entail a change in the other components (Schäffer et al. 2019). Teams also become multi-skilled with developers having to operate through a full-stack position with control over the overall area, which may increase motivation.

2.2 Micro-Frontend Architecture

This section will focus on the definition of micro-frontends and their integration, more specifically the types of composition, routing and communication.

2.2.1 Definition

Focusing merely on the micro-frontend part of the architecture, it is important to answer the question: what qualifies as a micro-frontend?

As it was previously hinted, a micro-frontend should represent an independent business domain, that is developed by one team responsible for its autonomous deliveries and development (Geers 2023, Mezzalira 2021). This description is connected to, and supported by the foundations described in Subsection 2.1.1 and Subsection 2.1.2.

However, from a technical point of view it isn't as simple as it may seem to define what should be considered a micro-frontend. There are two major ways to split a web application into components: the horizontal and the vertical split (Mezzalira 2021, p. 24).

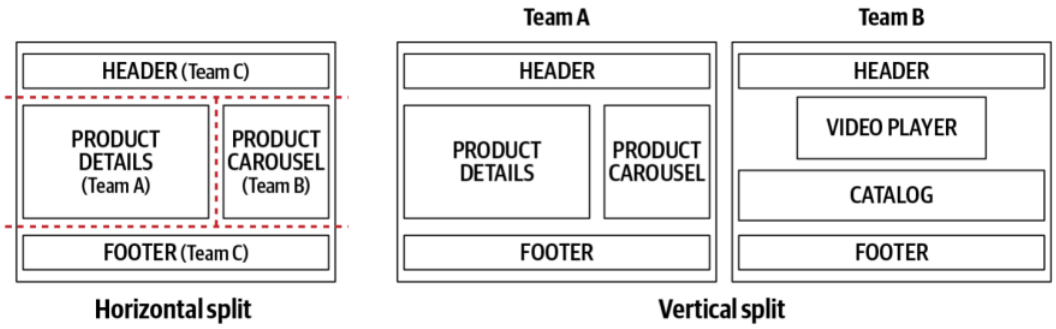


Figure 2.3: Horizontal versus a Vertical split of the view of a web page (Mezzalira 2021, p. 24).

In the horizontal split (fig. 2.3 - left), there can be a multiple number of micro-frontends in the same view. This means that various teams will need to coordinate in order to assemble a concise view since each one will be responsible for one micro-frontend. The main advantage of this approach is that it gives a big flexibility since the micro-frontends could be reused in other views. However, it requires a big level of organization and coordination between teams in order for the page to be coherent, and the project could end up divided into a big number of micro-frontends (Mezzalira 2021, p. 24).

As for the vertical split (fig. 2.3 - right), there is one micro-frontend per view. In this case each team is responsible for a view, which can be interpreted as a business domain. For example, in an e-commerce setting, one team can be responsible for the catalog experience, while another one is responsible for the checkout experience (Mezzalira 2021, p. 24). This way of defining a micro-frontend aligns with the microservice vertical composition described above, and can be linked to Domain Driven Design (DDD) that divides software responsibilities in accordance to bounded context of business logic.

It is important to note that a "view" may not include the header and the footer. If they change per view, then each team should be responsible for these components in their own separate views. However, if, like most cases, they are consistent throughout the views, the

header and footer will be loaded by the shell application that integrates the micro-frontends, or be separate micro-frontends of their own. So in this case a vertical view implies the content of the page, not including the header and footer. The concept of "shell application" will be relevant moving forward, specially on the micro-frontend composition discussion.

Defining a bounded context is essential for a correct division of responsibilities between vertical teams. Using a DDD model creates a strategic design that protects business concerns and provides the grounds for a bounding system to create a business-driven architecture (Vernon 2013, p. 9). This way, starting from the domain model of the application is essential to identify the bounded context, which guides the division of the development of the micro-frontends (Mezzalira 2021, p. 28).

In summary, a horizontal split is better suited for a static page without a lot of interaction (for example catalog or e-commerce websites), the contrary applies to the vertical split approach. A vertical split is likely to be better developed by a team that has a skill set inclined to a traditional client-side development, while horizontal split requires more effort in sync and organization between different teams (Mezzalira 2021, p. 29).

Like all technological developments, micro-frontend architecture also comes with some new concepts and challenges to keep in mind while implementing.

Pavlenko et al. 2020 describes some points:

- *"Orchestration - how to deal with loading applications when they are needed"*;
- *"Routing - how to manage routes in the application and decide which app to load"*;
- *"Isolation - how to bound to avoid collisions in the environment"*;
- *"Communication - how to provide applications the way they can communicate with each other"*;
- *"Dependency management - how to avoid one library to be loaded more than once"*;
- *"Consistency of UI/UX - how to manage common styles and make sure that the UX is consistent"*.

In the following subsections, these matters will be addressed. The latter point (consistency of UI/UX) is directly linked with the main research question of this dissertation, and will be further inspected in the next sections of this chapter.

2.2.2 Composition

There are three strategies for arranging a micro-frontend composition (fig. 2.4), a client-side, an edge-side and a server-side composition.

On the left side of the diagram in fig. 2.4, it is possible to see a client-side composition, where an application shell directly loads multiple micro-frontends in the client (normally the browser) from a Content Delivery Network (CDN) or from the origin server if they aren't cached yet at the CDN (Mezzalira 2021, p. 29). In the center, the final view is composed at the CDN level, pulling the micro-frontends from the origin and delivering the assembled result to the client. On the right, the composition occurs at the origin, where micro-frontends are combined into a view, cached at the CDN, and then served to the client (Mezzalira 2021, p. 29).

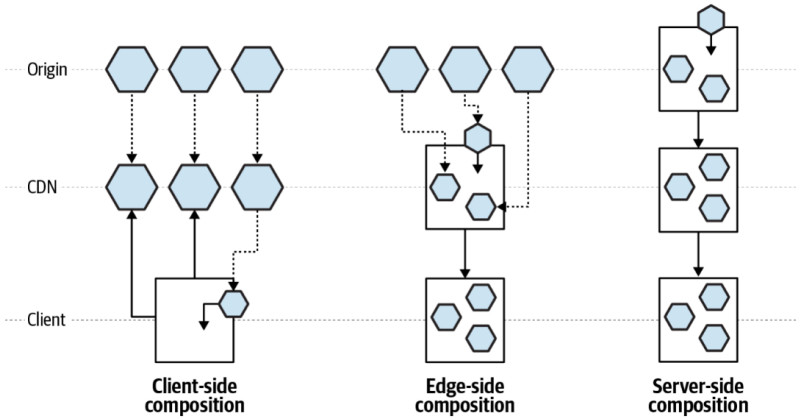


Figure 2.4: Micro-Frontends composition diagram (Mezzalira 2021, p. 29).

Client-Side Composition

The initial basis of a micro-frontend client-side composition is an application shell, that provides a skeleton for the web application that is responsible for loading all the other micro-frontends into itself dynamically. In the case of fig. 2.4, the application shell is the only component that is loaded directly in the client. In the context of a web application, this skeleton has the essential HTML, Cascading Style Sheets (CSS), and JavaScript required for the browser to render a rudimentary, functional application that typically encompasses components like a header, footer, navigation structure, and various placeholders (Goeleven 2021). Here, micro-frontends need to possess either a JavaScript or HTML file serving as their entry point. This allows the application shell to dynamically insert Document Object Model (DOM) nodes if it's an HTML file or initialize the JavaScript application when dealing with a JavaScript file (Mezzalira 2021, p. 30).

The shell can also be described as a container. Notably, there is no shared deployment or build between the container and the micro-frontends, resulting in a complete decoupling. Additionally, these components may employ different technologies and the container retains the authority to determine which version of the micro-frontend to deploy (Vasireddy 2023). Webpack Module Federation is the most widely used technology for this type of composition (Meraj 2023).

There is also the option of loading the micro-frontends into the shell application by using iframes or a technique called "client-side include" (which is a transclusion mechanism on the client side) (Mezzalira 2021, p. 30).

Edge-Side Composition

In the edge-side composition, the view is assembled at the edge layer, such as the CDN level. Micro-frontends are obtained from their origin source as they are aggregated at the CDN level to assemble the view. The resulting view is subsequently sent to the client. This strategy was implemented to facilitate the expansion of web infrastructure across multiple points of presence within the CDN network, enhancing scalability (Vasireddy 2023). An Extensible Markup Language (XML) markup language (Edge Side Includes (ESI)) is available to use by many CDN providers, and it "*allows a web infrastructure to be scaled in order to exploit the (...) points of presence around the world provided by a CDN network*" (Mezzalira 2021, p. 31).

Server-Side Composition

Server-side composition stays positioned between the browser and the application server. It leverages a CDN to store micro-frontends, subsequently offering them to the client during the build process or at compile time. At this stage, the server pieces together these micro-frontends to construct the final page (Mezzalira 2021, p. 31). The growing appeal of server-side composition can be attributed to its capacity to boost loading speed, reinforce application stability, and guarantee full page assembly before client engagement (Vasireddy 2023).

The use cases of the application should be analyzed prior to deciding to implement this composition, since if there is customization done to the page by the user, the scalability of the solution may be compromised. It should be best suited for pages that are highly cacheable (Mezzalira 2021, p. 31). Also Pavlenko et al. 2020 emphasises that "*the main drawback of this approach is that it is the old way when the HTML page is generated after each user's request and it vanishes all benefits that were introduced by Single Page Applications*".

2.2.3 Routing

It is important to note that the routing strategy used in micro-frontend architecture will be directly dependent on the type of composition chosen.

Starting by analyzing routing in the case of server-side compositions, the requests have to be routed at the origin since that's where the application logic resides (application servers). However this may come as a challenge in terms of scaling, since the retrieval of the micro-frontends needs to happen in each application server so that the composition page is served. One way to mitigate this is through a CDN, however it won't be reliable in the case of dynamic or personalized data (Mezzalira 2021, p. 31).

For the edge-side composition case, "*the routing is based on the page Uniform Resource Locator (URL), and the CDN serves the page requested by assembling the micro-frontends via transclusion at edge level*" (Mezzalira 2021, p. 32). Taking that into consideration, smart routing is not really an option in the case of edge-side composition.

In the client-side composition, the application shell will be responsible to load the micro-frontend as a SPA and should handle the routing logic. The shell will have the routing configuration and will load the correct micro-frontend according to it. The micro-frontends are also loaded following the user's state, for example if the user has yet to have logged in the application, the shell should redirect it to the landing page (login) or, if the user has already been authenticated, redirect to the authenticated area of the application. This strategy is the best when there is complex routing within the application (based on authentication or geolocalization for example), since the client relies on the URL to load the page (Mezzalira 2021, p. 32).

2.2.4 Communication

There are many different instances in which the micro-frontends and micro services within a system need to communicate. Communication between the frontend applications in the browser is called UI communication (fig. 2.5 (1)). The frontend-backend communication can be done through Hypertext Transfer Protocol (HTTP) requests using Representational State Transfer (REST), GraphQL, etc (fig. 2.5 (2)) and the communication between the

backend services are usually done through data replication mechanisms (feeds, messaging, streams, etc.) (fig. 2.5 (3)) (Geers 2020, p. 99).

Focusing on UI communication, micro-frontends inevitably end up needing to communicate with each other to notify other micro-frontends about user interaction.

This issue becomes more complex when facing a horizontal micro-frontend composition (in which several micro-frontends exist within one single view), or micro-frontends owned by different teams (Mezzalira 2021, p. 32). In this case, a simple URL link won't be enough (Geers 2020, p. 100).

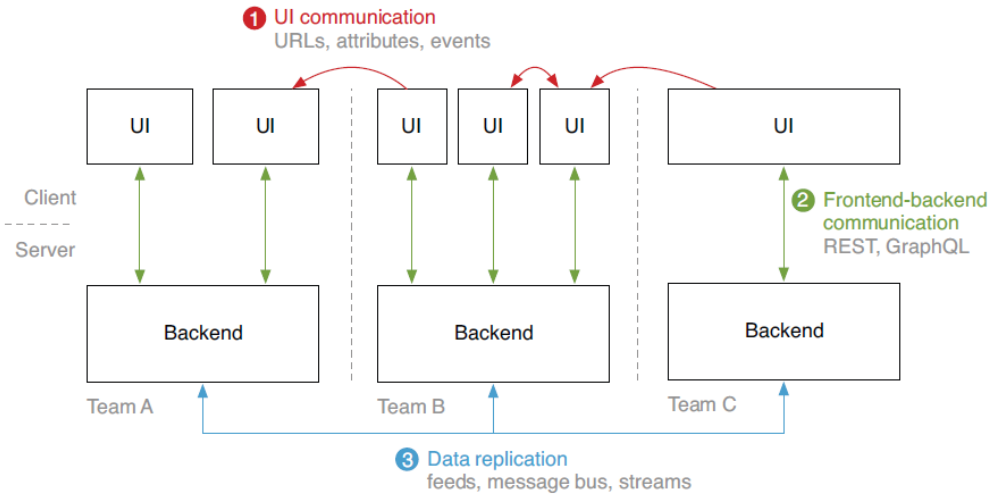


Figure 2.5: Different communication mechanisms in a micro-frontend / microservice architecture. UI communication (1). Each frontend fetching data from its own backend (2). Replication of data between the backends of the teams (3) (Geers 2020, p. 100).

However direct communication between micro-frontends becomes an anti-pattern, since each micro-frontend should be autonomous, independently deployable and unaware of other micro-frontends in the system. So, the best option in this case is using an eventbus, "*a mechanism that allows decoupled components to communicate with each other via events sent via a bus*" (Mezzalira 2021, p. 33). This way a micro-frontend can notify the event as the trigger (fig. 2.6 - component A), and another will listen and react to it (fig. 2.6 - component B and C).

Custom events can also be used to send custom data through a body. These custom events should be dispatched through an object that is available and shared through the micro-frontends, like the window object. This does not work in micro-frontends created by iframes since each iframe has its own window object.

In the case of vertical split, in which each page view is a different micro-frontend, the communication can become simpler, page-to-page communication through URLs. References such as identifiers can be transferred through the URL path or through query strings (Geers 2020, p. 100). It is important to keep in mind that query strings are not the most secure way to pass sensitive information like user identifiers, passwords, etc (Mezzalira 2021, p. 35). There is also the possibility of passing information, such as tokens, through a web

storage, local storage or cookies. The web storage becomes always accessible, as long as the micro-frontends exist in the same sub domain (Mezzalira 2021, p. 34).

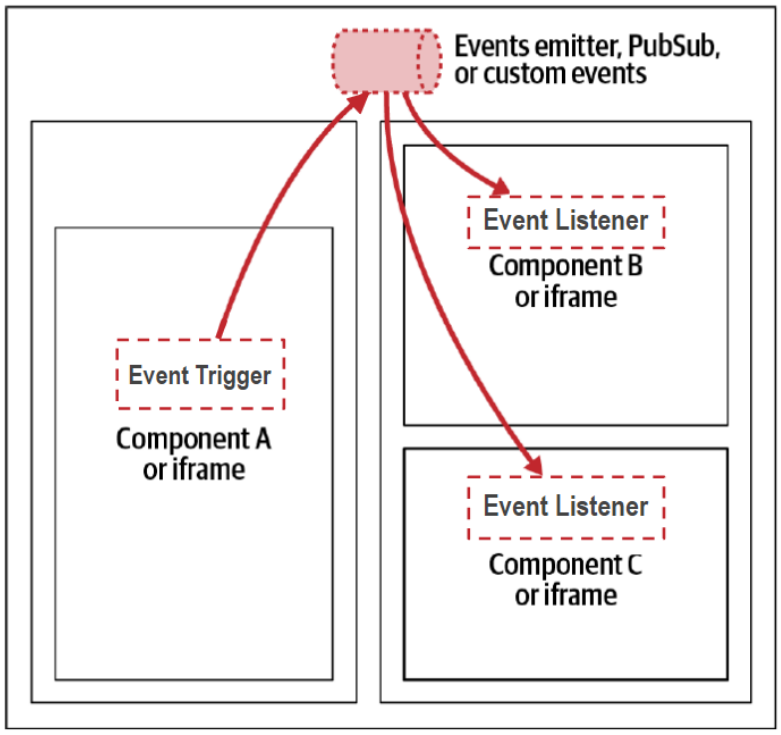


Figure 2.6: Event emitter and custom events diagram in micro-frontends in an horizontal split view (adapted from Mezzalira 2021, p. 33). Component A emits the event as the trigger, and components B and C are listeners.

2.3 Design Systems

Within a micro-frontend application, it is essential to establish a common design system between teams (fig. 2.7) so that a consistent look and feel is established for the costumer. A design system, typically created by designers, should express how a website or web application visually communicates its message to users (Godbolt 2016, p. 27). When it comes to a web application, it is usually composed of simple elements such as buttons, input fields, typography, icons, or more complex components that will be used by each team on their respective micro-frontend. By using the same elements ("building blocks"), the teams guarantee a wise design consistency (Geers 2020, p. 12).

Design systems are gathering increased attention as a means to establish and maintain consistent design standards across expanding product portfolios. They offer systematic guidance to teams and have already become commonplace in many monolithic frontend architectures. Although they are occasionally referred to as pattern libraries or component libraries, these libraries typically represent only one facet of a comprehensive design system (Klimm, Noss, and Bente 2021).

In essence, design systems encompass interconnected patterns, principles, and guidelines that empower stakeholders to create products aligned with their intended purpose. Notably, micro-frontends aim to provide greater flexibility and agility in application development. On one hand, this underscores the importance of adopting a systematic design approach to

ensure cohesion in a product. On the other hand, the use of design systems introduces a form of coupling that could potentially give rise to anti-patterns in the context of micro-frontends, thereby elevating the demands on design systems within the realm of microarchitectures (Klimm, Noss, and Bente 2021).

Design systems function as programmatic embodiments of a website or web application's visual language, akin to spoken languages. This notion, as proposed by Godbolt 2016, emphasizes the structural nature of design systems and their role in facilitating consistent communication through web design.

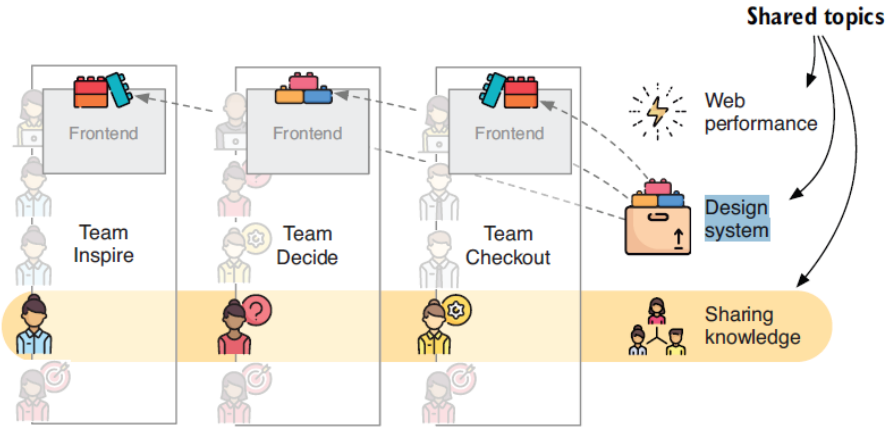


Figure 2.7: Design System as a shared topic between teams within a vertical Micro-Frontend architecture (Geers 2020, p. 11).

Furthermore, Couldwell 2019 highlighted the significance of branding within systematic design. Branding, serving as a form of distinction, allows to differentiate one competitor from another by establishing a unique, strong, and focused voice. From a critical standpoint, it's important to scrutinize how branding, often driven by commercial interests, can shape UX and potentially commodify design choices, potentially leading to a homogenization of web aesthetics. This perception prompts a broader consideration of how design systems navigate the balance between brand identity and the inclusivity of diverse voices in the digital landscape. In summary, the guidelines of a Brand identity define the assets, graphic and materials that uniquely represent a company. These guidelines can include "*logos, typography, colour palettes, messaging (such as mission statements and taglines), collateral (such as business card and PowerPoint templates)*" and more (Frost 2016, p. 24).

More style guides categories exist and should be kept in mind while creating a design system for a product, such as writing, voice and tone, code, design language, and UI patterns. Web style guides promote consistency and cohesion across a UI (Frost 2016, p. 23).

Ensuring a consistent style and UX is crucial when dealing with various standalone micro-frontends. This is achieved by setting up standardized UI/UX guidelines and making use of shared styling libraries or frameworks to preserve uniformity among micro-frontends. By employing Web Components along with CSS sandboxing through Shadow DOM, CSS rule conflicts are avoided. Additionally, adhering to naming conventions and leveraging CSS preprocessors like Sass helps prevent conflicts and guarantees a cohesive and harmonious appearance throughout the application (Gómez 2023).

There are many approaches to Design Systems. In the next subsections, Atomic Design and Component Libraries will be explored.

2.3.1 Atomic Design

The Atomic Design Principle is a methodology used to build web design systems. It promotes the design of smallest elements of a page (atoms), instead of designing entire pages at once. These atoms are the building blocks that can be used to build back together an entire web page (Godbolt 2016).

Atoms are the smallest element of the page that cannot be broken down any more, and they are the foundation of the websites, such as headings, list styles, images and videos and form elements. Atoms come together to form molecules, that in this case represent bigger elements composed of smaller ones, such as a search form, a media block or a navigation system. Molecules then come together to form a "*distinct section of an interface, like a blog article or comment*" (Godbolt 2016).

Templates are the final layer of Atomic Design, they represent the combination of interface elements to form a bigger element, like the navigation, content and footer elements, and enable the creation of pages. Pages are concrete implementations of templates that contain real representative content in the UI (Godbolt 2016, Frost 2016, p. 49).

In fig. 2.8, the many layers of the Atomic Design are evident and clear, in its application to the native mobile app Instagram. This methodology can then be applied to any web page or interface (Frost 2016, p. 60).

The advantages of atomic design are evident, since there is a clean separation between structure and content, that allows both the designers and developers to quickly shift between abstract and concrete. The interfaces are broken down into their atomic elements and those elements are what come together and form the final UX of the content page (Frost 2016, p. 52).

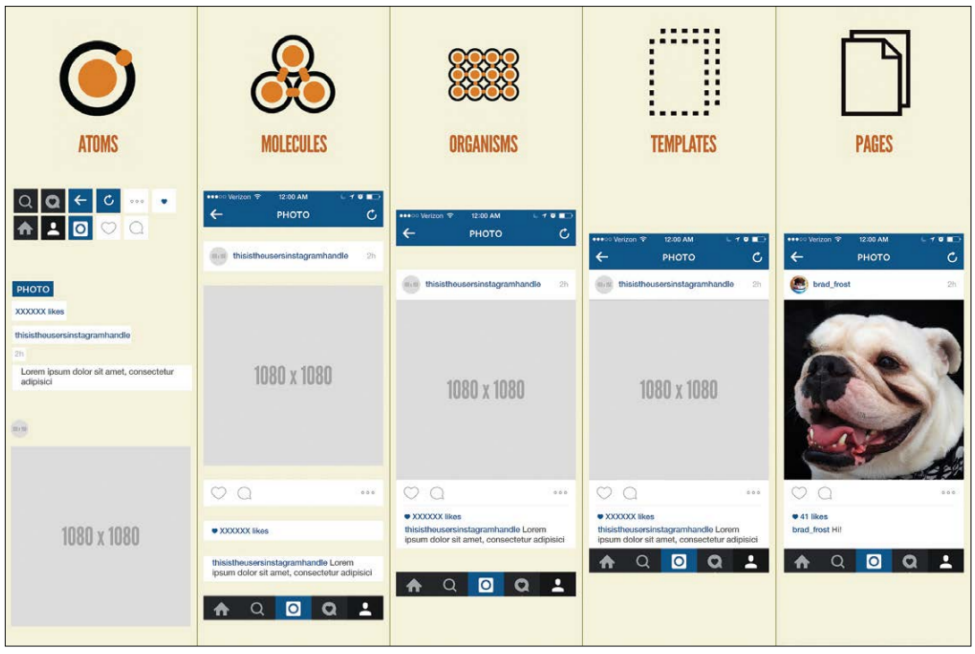


Figure 2.8: Atomic Design applied to the Instagram app (Frost 2016, p. 60).

Atomic design is a lot of times linked with pattern/component libraries and promotes their use in implementing a design system.

2.3.2 Component Libraries

In this subsection there will be a deeper analysis of current component libraries implementations and guidelines.

In order to obtain a UI/UX consistency in micro-frontends, shared component libraries that contain reusable UI elements/components is one of most regular solutions pointed out by the literature, although they are not easy to implement correctly (Jackson 2019, Gómez 2023).

Creating a library can include many benefits such as "*reduced effort through re-use of code and visual consistency*" (Jackson 2019). This same component library may act as a living documentation/styleguide, and can bridge the gap between designers and software developers (Jackson 2019).

Each team can then combine elements from the library in order to build the interface for their micro-frontend. This library can be consisted of both simple components like previously mentioned, and complex components (Pavlenko et al. 2020). Complex components usually include a large amount of UI logic, for example auto-completing, drop-down search field, and a sortable, filterable, paginated table. It is very important to ensure however, that the shared components only contain UI logic and possess no trace of business or domain logic (Jackson 2019), since that should be inherent to each micro-frontend/team.

There are several points to keep in mind when implementing a design system or a component library. One of the most common mistakes is to create too many components, too early. One can feel the urge to create all of the visuals that will integrate the interfaces of all micro-frontends, but without real life application and usage for them, it can "*result in a lot of churn in the early life of a component*" (Jackson 2019). Some components might even turn out to never be used by any micro-frontend (Pavlenko et al. 2020), so communication between teams is key.

A relevant issue is the way of integration of components in applications. Currently, there three ways to achieve this, using Web Components, using frameworks (Angular, React, etc.), and using pure JavaScript and CSS.

According to Pavlenko et al. 2020, "*Web Components are not well supported by browsers yet*". To make them functional, developers have the option of using polyfills, which are compact utility libraries that enable the integration of new JavaScript features in older language versions, however, it can result in larger content sizes and longer download times. Implementing libraries using frameworks typically mean that the components are limited to work only with these frameworks, which may be a drawback. By using pure JavaScript and CSS "*there is a higher possibility of collisions between different parts of applications, so, isolation issues should be considered during development*" (Pavlenko et al. 2020).

When it comes to building a components library, UXPin 2022 refers that "*an interface inventory or UI audit is a crucial first step*" after choosing the desired framework. This is specially relevant when performing a redesign or a refactor of an existing monolithic frontend. An audit often uncovers UI and programming inconsistencies engineers must address before building a component library. An interface inventory can also work as a resource

for advocating the component library to interested stakeholders. For this, Frost 2016 recommends using screenshots of every UI, and then proceeding to cut out each component. While tedious it will help organize the elements that are needed in the components library. Next, one should sort the components into categories which will form the foundation for the component library.

The next step is to develop the components, after the selection of the tools and framework. The Atomic Design approach (Frost 2016) can be used to develop components with defined responsibilities at a low level of complexity, and focused solely on UI logic (UXPin 2022, Jackson 2019).

As for testing, component testing is the best option, since a components library won't have the foundations for end-to-end testing - that should be done in each frontend application that implements the library. It is important to make sure that all the visual elements render correctly on all platforms, functionality operates as expected, and performance and accessibility meet expectations (Sukoinen 2020). Manual tests could be performed, but again it would make more sense to do so in each application that integrates the library.

The final stage is the documentation. If one chooses to develop the library using a JavaScript framework (React, Angular, Vue, etc.), Storybook can be used to and manage individual components (UXPin 2022). Storybook is a frontend workshop for building UI components and pages in isolation and can be used for testing, and documentation (Storybook 2023). Providing a clear documentation will guarantee a smoother implementation of the components in the design system. Documentation usually entails an overview, instructions, visual examples and code samples (Sukoinen 2020). It is essential that the documentation stays up to date with the updates made to the components.

In summary, component libraries promote consistency and cohesion throughout the experience, speed, and establish a more collaborative team's workflow, while promote a shared vocabulary through documentation. They also make cross-browser/device, performance, and accessibility testing easier and serve a foundation to modify, extend, and improve code (Frost 2016, p 65).

It is also possible to conclude that using an atomic design approach to a components library can be beneficial to create a hierarchical structure that supports a modular construction and a clear separation of concerns.

2.4 Technology

In this section, several frontend technologies based on a JavaScript framework will be compared and discussed, with the goal of accessing which would be better to develop a project with the integration of micro-frontends and a components library.

2.4.1 React

React is an open-source frontend JavaScript library, developed by Facebook in 2013, currently maintained by the open-source community and Facebook. It is used for developing UIs based on components (Pattakos 2023). It was created to resolve a specific set of challenges faces by the Facebook team at the time, however they were not only unique to Facebook. It was built to deal with displaying data in a UI, more specifically, to serve large-scale UIs (for Facebook and Instagram) for data that changes over time. React then served as a change in

the paradigm of the Model-View-Controller (MVC) pattern, removing the bottlenecks and lack of maintainability that come with it (Gackenheimer 2015, p 2).

React Native is a JavaScript library that supports mobile development for native Android and iOS applications, but both React and React Native support TypeScript development.

A small overview of the structure and state management in React projects will be presented in the following subsections.

Elements

React elements are the smallest building blocks of the React framework, and they describe what will appear on the UI. Different from browser DOM elements, "*React elements are plain objects, and are cheap to create. React (...) takes care of updating the DOM to match the React elements*" (reactjs.org 2023b).

React elements are immutable and can't be changed once they are created (neither can their children or attributes), since it represents the UI at a certain point in time. With this, the only way to update the UI is to create a new element, and render it through the root (`root.render()` method).

Components

Components are considered as independent and reusable pieces of code, that serve the same purpose as JavaScript functions, but work in isolation and return HTML. React components are made up of several React elements. React Components can either be Class components or Function components. In older versions of React applications, Class components were primarily used, however currently it is suggested to use Function components along with Hooks (added in React 16.8).

To define a React component class, it needs to extend `React.Component`, and implement the `render()` function. In the case of a Functional Component, it can be considered a React Component without the `render()` function, since everything that is defined in the function's body is the render function that returns JavaScript XML (JSX) in the end (reactjs.org 2023a).

State Management

Components usually require updates to the displayed content in response to user interactions. For example, entering information into a form updating the input field, selecting "next" on an image carousel switching to the next image, etc. In React, this specific memory management within components is referred to as "state". Local variables don't persist between renders, while state allows components to "remember" these types of information (react.dev 2023).

When React re-renders a component, it renders it from scratch and doesn't take into account any changes to local variables since they lack the ability to trigger re-renders. React remains unaware that a component needs updating with new data when changes occur to local variables. Achieving this requires two essential steps: retaining data between renders and triggering React to re-render the component with new data (react.dev 2023).

In Functional components, the `useState` Hook in React addresses these requirements by providing a state variable that retains data between renders, and a state setter function that

enables the update of the variable and signals React to re-render the component with the new data (react.dev 2023).

2.4.2 Vue.js

Vue is a progressive JavaScript framework for building UIs (Views, where its name comes from), that builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model.

According to Filipova 2016, Vue.js was initially developed as a tool for rapid prototyping, however currently it is used as a means to build complex scalable reactive web applications.The two core features of Vue include declarative rendering - "*Vue extends standard HTML with a template syntax that allows (..) to declaratively describe HTML output based on JavaScript state*", and reactivity - "*Vue automatically tracks JavaScript state changes and efficiently updates the DOM when changes happen*" (vuejs.org 2023).

Single-File Components

In most Vue projects, Vue components are authored using an HTML-like file format called Single-File Component (SFC) (also known as *.vue files). A Vue SFC, as the name suggests, allows the encapsulation of the component's logic (JavaScript), template (HTML), and styles (CSS) in a single file (vuejs.org 2023).

SFCs help improve code organization and maintainability by keeping related code in one place. They are also a defining feature of Vue and they are the recommended way to author Vue components in the use case of a needed build setup. In the scenarios where SFCs are not needed (in the case of simpler logic without a build step) Vue can still be used via plain JavaScript without a build step (vuejs.org 2023).

Vue's official documentation (vuejs.org 2023) notes that, even though many could agree that separate concerns are being put together in a single file, "*separation of concerns is not equal to the separation of file types*".

API Styles

Vue components can be developed in two distinct Application Programming Interface (API) styles: Options API and Composition API.

With the Options API, a component's logic is defined through an object of options like `data`, `methods`, and `mounted`. These options expose properties within functions using the `this` keyword, referring to the component instance. On the other hand, the Composition API are defined by the import of API functions to articulate a component's logic. In SFCs, the Composition API is commonly used with `<script setup>`, utilizing the `setup` attribute to streamline Vue's compile-time transforms. This approach minimizes boilerplate code, allowing direct utilization of imports and top-level variables or functions declared in `<script setup>` within the template (vuejs.org 2023).

The Options API is structured around the concept of a "component instance," aligning well with a class-based mental model, especially for developers familiar with Object Oriented Programming (OOP) languages. The Composition API focuses on declaring reactive state

variables within a function scope, composing state from multiple functions to handle complexity. While more free-form, it demands an understanding of Vue's reactivity for effective use (vuejs.org 2023).

In a production context, one should opt for Options API if there are no build tools or for low-complexity scenarios, such as progressive enhancement. As for Composition API, that should be used with SFCs for building full applications with Vue (vuejs.org 2023).

2.4.3 Angular

Angular is a free, open-sourced web application framework based on TypeScript, mainly used to develop SPAs. In 2016, Google launched Angular as a sequel to AngularJS, which was released in 2010. AngularJS used JavaScript for development, so the release of Angular was considered a drastic shift. Still, following its release, it caught the attention of the development community due to its features and because it was backed up by Google (Joshi 2023).

While the original framework, AngularJS, is considered a MVC, in Angular "*there's no strict association with MV*-patterns as it is also component-based*" (Pattakos 2023). Projects in Angular are structured into Modules, Components, and Services. Next there will be an overview of some of these concepts.

Components

According to the official Angular documentation (Angular.io 2023d), "*components are the fundamental building block for creating applications in Angular*".

Angular uses a component architecture with the aim of achieving peak organization, manageability, scalability and maintainability of the code. An Angular component is usually identified by the "component" suffix (e.g., `example-name.component.ts`) and is divided into a decorator and a Typescript class.

The decorator is used to define configurations for the selector, which defines the tag name of the component when used in an HTML template. It also has the HTML template that controls how the component will be rendered by the browser. HTML templates can be defined as an inline template within the TypeScript class, or in separate files with the `templateUrl` property. Like the HTML, component's styles can be declared in the same file as the TypeScript class, or in separate files with the `styleUrl` property. Components can optionally include a list of CSS styles that apply to that component's DOM.

The Typescript class is responsible for all the behaviour of the component (such as handling user input, managing state, defining methods, etc.).

Services

Services are used to share logic between components, allowing the developer to inject code into the components and managing it from a single source of truth. Angular services can be identified by the service suffix (e.g., `example-name.service.ts`) (Angular.io 2023d).

Services are compromised of a TypeScript decorator and a TypeScript class, very similar to the component definition. The TypeScript decorator is used for the definition of the configuration options for what parts of the application can access the service (by the `providedIn` property). For example, `providedIn: root` will allow a service to be accessed anywhere

within the application, and `providedIn:any` from outside the application (which could come in handy for micro-frontend dependency injection). The TypeScript class is responsible for the definition of the desired code that will be accessible when the service is injected (Angular.io 2023d).

Modules

Modules in Angular are used to configure the injector and the compiler and help organize related domain together. They "*consolidate components, directives, and pipes into cohesive blocks of functionality, each focused on a feature area, application business domain, workflow, or common collection of utilities*" (Angular.io 2023c).

An Angular Module is a class marked by the `@NgModule` decorator. This decorator takes a metadata object, which "*describes how to compile a component's template and how to create an injector at runtime (...) identifies the module's own components, directives, and pipes, making some of them public, through the exports property, so that external components can use them*" (Angular.io 2023c). `@NgModule` can also add service providers to the application dependency injectors.

Modules are the way an application is organized and allows the extension of it with capabilities from external libraries. Angular libraries are NgModules, such as `FormsModule`, `HttpClientModule`, and `RouterModule`. Many third-party component libraries are also available as NgModules (Angular.io 2023c). So Modules are essential in the exportation of Component Libraries.

Modules use the tag `NgModule` to declare which components, directives, and pipes belong to it. It can also export those components, directives, and pipes, making them public so that other module's component templates can use them. Other modules, directives and pipes can be imported to be used in that Module. Services are also provided so other application components can use them (Angular.io 2023c).

All Angular applications must have at least one module (also called the root module), since the application needs that module to be bootstrapped in order to launch. As the application grows, the root module can be refactored into smaller feature modules that represent collections of related functionality. These modules are then imported in the root module so they are correctly compiled. They can be loaded eagerly when the application starts or lazy loaded asynchronously by the router.(Angular.io 2023d).

2.4.4 Comparison

A simple difference between the three is that React is a UI library, Vue is a progressive framework, and Angular is a full-fledged front-end framework (Joshi 2023).

However, the main difference between Angular, React and Vue is the developer experience. As previously mentioned, Angular requires the use of TypeScript for a more effective programming. This can be advantageous for someone with a background on OOP languages (such as Java or C#), since TypeScript will be more familiar and conceals some "oddities" of the JavaScript language. On the other hand, Vue.js and React don't require TypeScript (but it is supported by both frameworks). They also tend to mix HTML, JavaScript, and CSS content together in a single file, when Angular encourages the separation of these files (Freeman 2020).

When it comes to popularity, following a StackOverflow survey performed in 2022, React was considered the favourite framework of 40.14% of developers, Angular with 22.96%, and Vue with 18.97% (Joshi 2023).

When comparing Angular vs React vs Vue with respect to statistics on their GitHub repositories, Vue has a bigger number of watchers, stars, and forks. However, the number of contributors for Vue are lower than Angular and React (see table 2.1). This can be explained due to the fact that Vue is driven by the open source community, "*whereas Angular and React have a significant share of Google and Facebook employees contributing to the repositories*" (Daityari 2019).

Table 2.1: Summarized comparison between Angular, React and Vue.js (Daityari 2019, Joshi 2023)

	Angular	React	Vue.js
Support	Google	Facebook	Community
Type	Framework	Library	Framework
Language	Typescript	JavaScript	JavaScript
Data Binding	Both	Unidirectional	Bidirectional
Popular Websites	Paypal, Samsung, Upwork	Netflix, Twitter, Amazon	Alibaba, Grammarly, GitLab
Github Watchers	3.1k	6.7k	6.3k
Github Stars	78.4k	180k	218k
Github Forks	20.6k	36.5k	35.7k
Github Contributors	1,500+	1,500+	400+

Due to its use of TypeScript, and also due to professional experience of the author in its use, Angular was the chosen technology to develop the project presented in this dissertation.

2.5 Angular Component Libraries

This section will dive into available, free to use, component libraries for the Angular framework, that could potentially be used to integrate into a micro-frontend architecture, or be used as reference for building one. The following component libraries were chosen due to their popularity, current use and relevance.

2.5.1 Angular Material

Angular Material is the official Angular UI component library developed by Google. It mainly focuses on implementing the application based on Google's Material Design, it is open-source and available on Github with an MIT License.

As of 2023, Angular Material has more than 23,000 GitHub stars, and in 2022 it had 1.1 million weekly NPM downloads (Arunodi 2022).

According to material.angular.io 2023, the main features of Angular Material are its versatility, the high quality and easy integration. Beyond the available set of pre-built components (for examples, see fig. 2.9), Angular Material also allows developers to create their own custom components based on the existent ones. With tools that facilitate the implementation of common interaction patterns, developers can customize the look and feel of their applications while maintaining adherence to the Material Design specification (material.angular.io 2023, Arunodi 2022). This demonstrates the versatility of this library and it can come in handy in creating an independent brand image as previously mentioned, in the case that the Material Design base matches the desired aesthetic or design of the application.

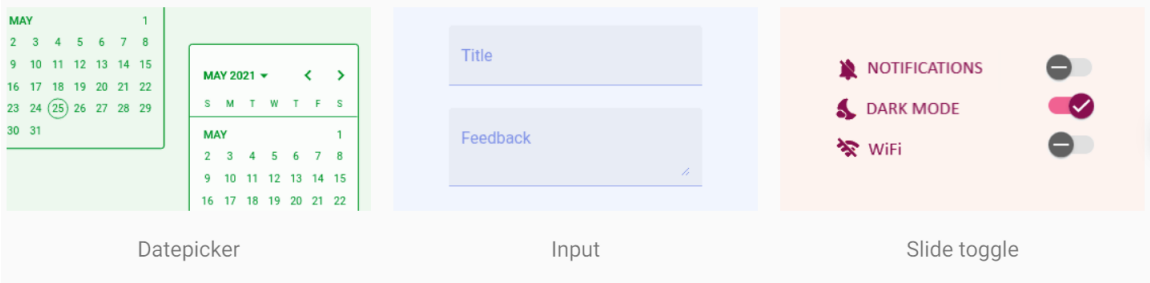


Figure 2.9: Angular Material base components for Datepicker, Input and Slider (material.angular.io 2023).

We can then assume that an angular component library can be built on top of Angular Material, which could decrease the development cycle, since the basis of the components are already built by this library. Also, since Angular Material is a product of the Angular team, it ensures a seamless integration experience for developers using Angular.

2.5.2 NG-Bootstrap

NG-Bootstrap is an open-source library built on top of Bootstrap CSS. Since it provides components and design patterns based on Bootstrap, it reduces the learning curve for new projects if the developer is already familiar with this library, making it easier to build Angular applications quickly and efficiently (Arunodi 2022).

It extends Bootstrap components as Angular directives, with two-way data binding and other Angular-specific features. This way, NG-Bootstrap allows the creation of "*responsive, mobile-friendly web applications that work seamlessly with Angular*" (Arunodi 2022).

In this library there are a variety of widgets available to use, including all Bootstrap widgets (ng-bootstrap 2023). Developers can use Bootstrap components like modals, carousels, progress bars, through components that are lightweight and responsive, and the official documentation (ng-bootstrap 2023) claims that the components are tested with almost 100% coverage. It also possesses a large community for support, with an open source repository on Github.

However, when it comes to component customization, even though some components are available in a custom form, for the most part one would need to import the original CSS of the chosen component from the library and use SCSS functions to override it (Ortico 2021). We can conclude that NG-Bootstrap isn't as extensible or customizable as Angular Material.

2.5.3 Clarity

Clarity Angular is a scalable and customizable design system built for Angular created by VMware. It is an open source project, and its goals is to join UX guidelines, HTML/CSS framework, and Angular components in a balanced experience for the user.

Clarity can be used for design, and its assets are available in Figma, a popular design tool that also documents components and foundations, allowing developers to export them directly into their project. Icons in SVG format and fonts are also available to download at its official site, and the documentation of the components is provided through Storybook. Clarity components are built using Web Components, this way it supports many different frameworks, including Angular, React, Vue, or event just plain JavaScript (clarity.design 2023, Rylan 2021).

The foundation of the Clarity design system is based around the concept of "cards", that act as containers to group related content in a structured and organized manner. Clarity offers card components for headers, footers and content sections, each easily adaptable and customizable with different styles and themes. These cards can be integrated with other available components from the library, such as modals, dropdowns and buttons in order to create coherent designs. The goal of the card-based design is to provide a flexible and modular system for designing interfaces (Ravoof 2023).

One of Clarity's strengths lies in an extensive set of form controls, that include input fields, select boxes, radio buttons, and more (fig. 2.10). Additionally, Clarity offers a suite of data visualizations, including bar charts, line charts, and pie charts (Ravoof 2023).

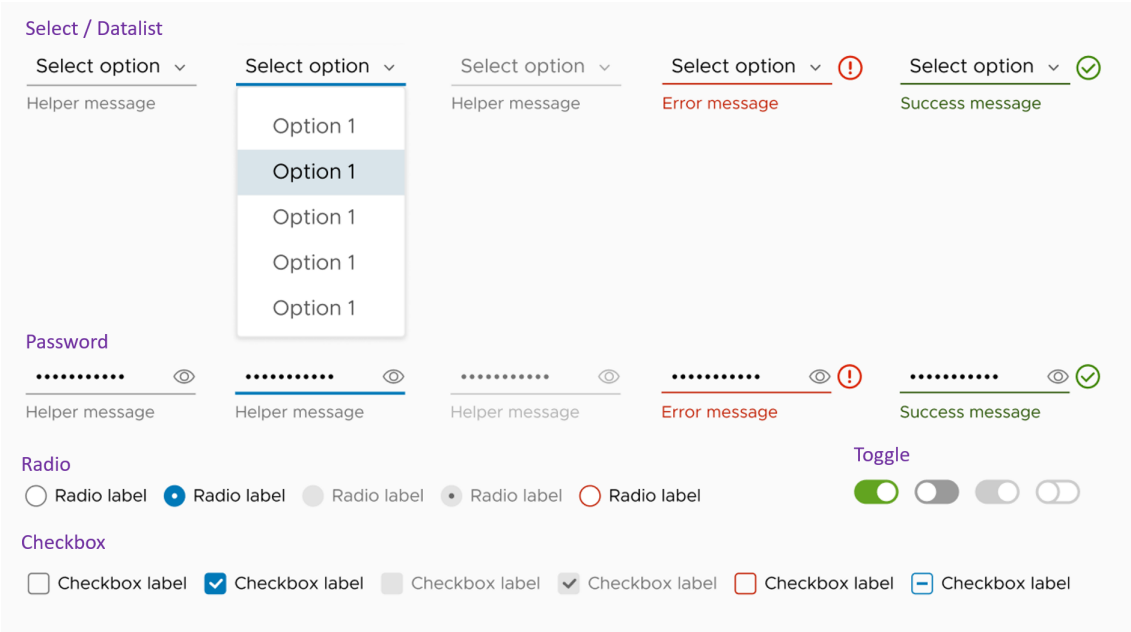


Figure 2.10: Clarity Design System Figma design for forms, displaying select boxes, radio buttons, password inputs, checkboxes and toggles (Figma 2023).

Clarity is not only a components library, but a full on design system that can used as basis for designing, implementing and documenting a design system into a micro-frontend architecture.

2.6 Summary

This chapter gathered a literature review of the state of the art relevant to the problem at hand. Following a comprehensive exploration of micro-frontend architecture, it not only provided insights on how these applications are structured, but the problem of UI/UX inconsistency was also unveiled and identified by the literature.

An investigation into Design Systems and Component Libraries was conducted, as the main reference of a possible solution to the the main question of this dissertation. The importance of Atomic Design is crucial for build structured, scalable and maintainable Design Systems.

Then, the first step of building a Component library was performed, which is to choose the development framework. For that, React Vue.js and Angular were compared and briefly analyzed. The chosen framework to further develop a solution was Angular due to its use of Typescript and the amount of component libraries available to use as a basis for this project.

Finally, some Angular components libraries and design systems were discussed and laid out, paving the way into the development of a similar solution, in a micro-frontend setting.

Chapter 3

Solution Planning

This chapter will discuss the schedule of activities planned to complete the project, describing each step of the research process and including planning for the design and implementation of the solution, as well as the planning for the evaluation procedure.

3.1 Work plan

The dissertation project has two main milestones: the delivery of the P2 project of the Preparation for the Dissertation (PREPD) subject, and the final delivery of the Dissertation.

The first delivery entails an explanation of the feasible objectives, research process, methods and questions used for the project, as well as an analytical, critical, and ethical interpretation of the problem to be solved. The dissertation writing process began shortly after the formalization step, focusing mainly on a state of the art study, from which the previous Chapter 2 was created. The next step would be the planning of the design and implementation of the solution and the planning for the evaluation procedure, that will be mentioned in this current chapter.

On the other hand, the delivery of the dissertation focuses on the actual design and implementation of the solution. This includes both architectural design and the design of the UI/UX graphics in order to identify the different components to be developed within the library. The implementation will include the solution's development, tests, deployment and documentation, and the project will be ending with the experimentation and evaluation stage. In this latter stage, the data will be collected and the results will be analyzed following the established methodology.

A Gantt chart (fig. 3.1) was created as a way to present the time management of the work that needs to get done in order to complete the project in the desired time frame. This graphic adds a visual representation of the work plan and the tasks, priorities and timelines of the project. It will serve as a guide to track progress and will be readjusted if needed throughout the process. A larger time window was given to the development of the solution, however it is important to note that the writing of the dissertation document is planned to be done in parallel with the development tasks.

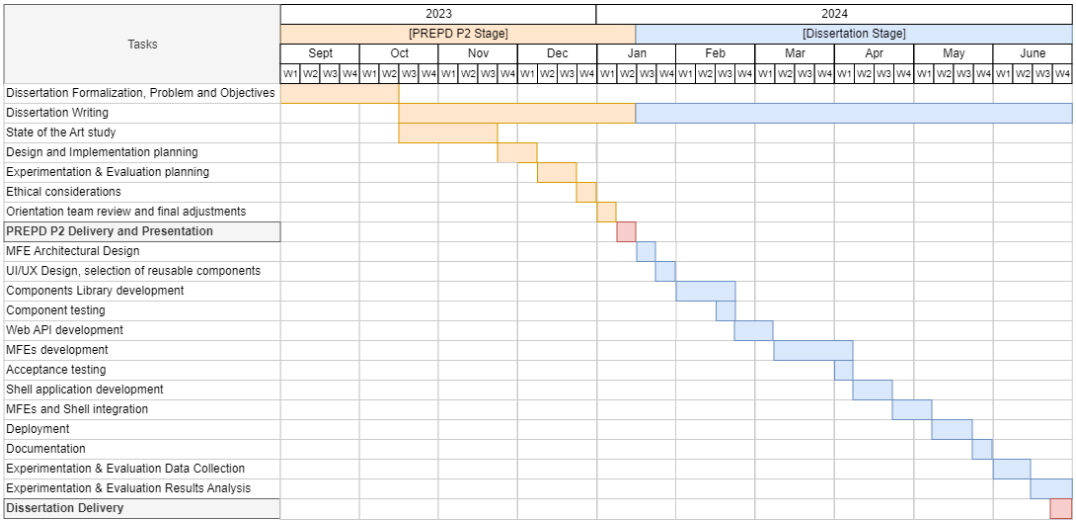


Figure 3.1: Gantt Chart of the project's timeline and tasks.

The initial planning went until June, since it accounted for the development of the solution in full-time. However, the delivery date had to be pushed forward due to lack of possibility to do so because of the working-student status of the author.

The planning for the tasks will be described in more detail in the following sub-sections of this chapter.

3.1.1 Project Management

The management and planning of the work needed for the implementation of each project depended on a Trello board (fig. 3.2) with different labels for each application to be developed. Cards that represent each task were added initially on the "to do" column, and passed onto the "on going" column while their development was at hand.

For each task of the Gantt chart (fig. 3.1) a label was created and each card in the Trello board was associated with it. For example, the dissertation writing task is reflected by the blue label, associated with different, smaller stories to help with organization.

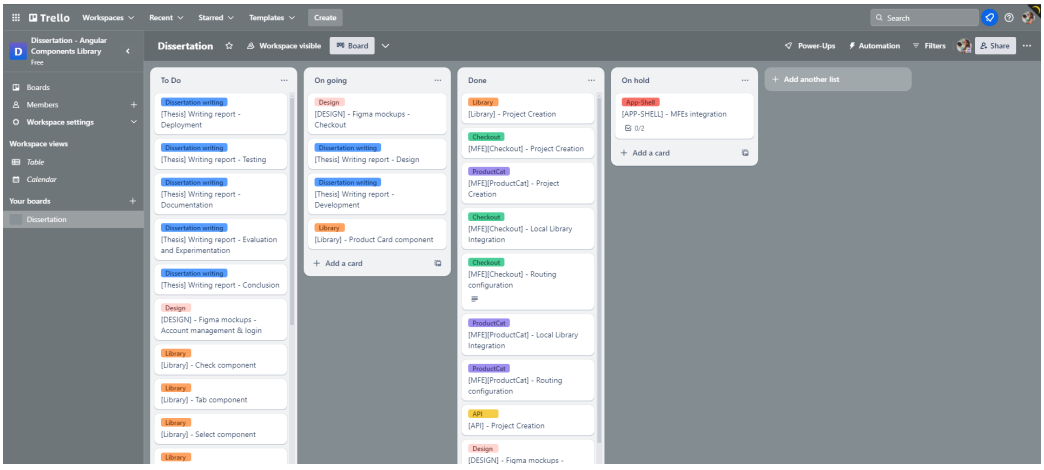


Figure 3.2: Trello board with the cards distributed through each step of the development process, represented by the columns.

In order for the card to be considered "done" and moved to that column, the task would need to be implemented, tested manually (locally) and automatic tests would have to be developed whenever it made sense - this would include unit or acceptance tests. If the development of the task had to be put aside because it depended on other developments, or other priorities arose, then it would go into the "on hold" column.

3.2 Design plan

A good software design defines the foundation of a software application and is essential for the creation of a structured solution. In this section, the plan for the architectural design and the UI design will be covered, describing the process and the tools that can be used in its arrangement. The design will be done during the Gantt chart's 'MFE Architectural Design' and 'UI/UX Design' tasks (fig. 3.1)

The main purpose will be to create a PoC of a solution that can be divided into micro-frontends following a clear business logic. For example, simulating an e-commerce website, the business areas are clear: the login page (see fig. 3.3 - a)), the products display pages (see fig. 3.3 - b)), the checkout page (see fig. 3.3 - c)), etc. All these views could be divided into different micro-frontend applications in a logical way.

It is important to note that fig. 3.3 does not represent the final design of the solution and its only purpose is to demonstrate the possible divide of the micro-frontends by business logic, in a vertical split applied to an e-commerce site example.

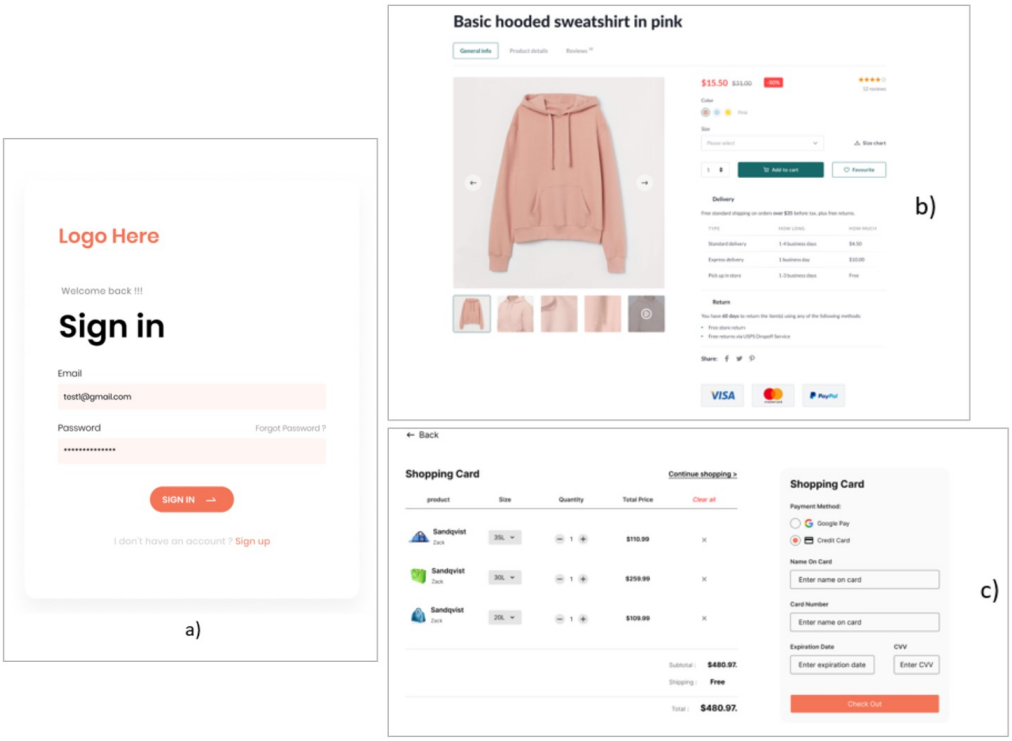


Figure 3.3: E-commerce pages example, from Figma, divided by business logic. a) - The login page view, that could be the Login micro-frontend (C. Figma 2023b); b) - The products display page, that could be part of the Product catalogue micro-frontend (C. Figma 2023c); c) - The Checkout page, that could be part of the Checkout micro-frontend (C. Figma 2023a)

Even though the architectural design and the UI design are intertwined and dependent on each other, they have different objectives that will be explained in full in the next subsections.

3.2.1 Architecture Design

In a micro-frontend system, the architecture and solution design will need to be thought thoroughly in order to make sure all the components are integrated in a scalable, maintainable way. This solution architecture will have to take into consideration the four separate types of projects present in this solution:

- Components library;
- Micro-frontends (at least two);
- Shell application to display the micro-frontends;
- Web API to provide data for the frontend projects.

The architectural design will not only keep in mind the development of the project, but also the integration and subsequent deployment of the parts.

Following the state of the art study performed in Chapter 2, important decisions have to be made in this stage, including the definition of the type of split (vertical or horizontal), composition, the routing and the communication of the micro-frontends in the architecture.

For the split, a vertical split was most supported by the literature and easier to divide according to DDD, so that will be the main split approach to evaluate in this stage. As for the composition, according to the studies referred in the state of the art, a client-side composition will entail an application shell that should be responsible to load the micro-frontend as an SPA and should handle the routing logic, making the routing straight forward - which is a plus for the client-side composition. A communication strategy between the micro-frontends will be evaluated according to the use cases in the chosen application, that means the UX requirement and the UI design will also play a part in the architectural design.

3.2.2 User Interface Design

The UI design stage should be done in parallel to the architectural design, since it will also be a pivotal point in the solution design. This is especially true in this case since the objective will be to create a design system that will be incorporated into a Components Library in a micro-frontend setting.

Since the goal of this project will be to create a coherent design and UX throughout the use of an application based on micro-frontends, a cohesive graphic design will be done for all the pages and views of the application.

The UI design will have two main tasks. The first one will be the creation of the visual identity, including the application name, logo, choice of color palette and typography. Then, the actual design layout can be created, following this visual identity, that will be crucial for maintaining the coherence in the UX.

Afterwards, the design will be split, following the atomic design approach, into smaller segments, allowing for the selection of the reusable components that will be developed inside the Components Library and integrated into the separate micro-frontends. This will

also help map out the components that are unique to each micro-frontend, and help separate the design responsibilities of each.

This design can be done using design tools such as Figma, which will also be used as documentation and reference for the development of the design system. Figma has plugins such as "Figma to Code" that can help developers to implement said components using CSS. Also, an interactive design can be created, showing the flows of the application using the mock-ups, that can be used for usability tests.

3.3 Implementation plan

The next step will be the implementation of the solution. This is planned to be the most time consuming part of the project, and the implementation will have to be divided into the four types of projects mentioned earlier.

Following the Gantt chart (fig. 3.1), the implementation stage will occur from the "Components Library development" all the way until the "Documentation" task.

The following sub-sections will explain the plan for the development, testing, deployment and documentation stages of the project.

3.3.1 Development, Testing and Deployment

The Components Library is planned to be the first project to implement. This will make it so the components are ready to be integrated once the implementation of the micro-frontends is done. Also, this enables the creation of documentation that will be support the Design System alongside the Figma files created for the components. This development will include component testing, with a testing framework supported by Angular, probably Cypress, to ensure and test the behaviour of the components. Since this is the main focus of the project, more time was dedicated to the development of this library, to be sure that it sets a good foundation for the next developments.

The Web API will need to begin to be developed, so it can provide data for the micro-frontends, this will include the project setup and the connection with a database.

The development of the micro-frontends will come next. This is the step that is planned to take the longest, since in reality it will be the development of three separate applications. Some acceptance testing is also included, to make sure that the components from the library are correctly integrated with the applications. At this stage, the integration of the Components Library and the Web API can be done locally, since the deployment is planned to come later.

The final project that needs to be developed is the shell application, that will integrate all the separate micro-frontends and load them into a since web experience. The shell application will have the components that are always displayed such as the header and the footer, and the rest will be mainly be routing and loading configurations. The micro-frontends and the Shell will be integrated, first locally, and then deployed during the deployment task.

In the deployment task, all projects will have to be integrated and deployed into a singular web application. This is planned for the end of the development tasks, but the deployment strategy will have to be defined during the architecture design.

3.3.2 Documentation

Some time was purposefully left for the documentation at the end of the development. This can include the documentation of the Design System and the Components Library, using for example Storybook or just the Figma files.

This planned time can also be used to adjust all of the development documentation in the dissertation writing, including necessary attachments, graphics, etc.

3.4 Experimentation and Evaluation plan

In this section, the planned tasks related to Experimentation and Evaluation of the solution are to be discussed.

3.4.1 Methodology Definition

For starters, a methodology for the Experimentation and Evaluation stage needs to be defined, this is to be done after the Design stage. Following the Research Process presented in Chapter 1 of this document, this stage is composed of the "data generation methods" and the "data analysis" steps, and the methodology should be chosen according to those.

In the Gantt chart (fig. 3.1), the tasks associated with the experimentation and evaluation are the "Experimentation & Evaluation Data Collection" and the "Experimentation & Evaluation Result Analysis".

Data Collection

For the "data generation methods", observation and questionnaires were assumed to be the best evaluation metrics for the project in hand.

As for the observational methods, like stated previously, code analysis can be used to test the code duplication and maintainability of the projects. However, it is important to note that the architecture design stage will have to take this into consideration and integration with a code analysis framework needs to happen, so the plan may need to be altered accordingly, depending on the chosen approach.

For the questionnaires, as said before, they could be used to determine if the application is consistent in terms of UI and UX, since that can be best determined by human interaction than by automated tests. Additionally, in this methodology definition stage, a System Usability Scale (SUS) (Brooke 1995) can be used to determine the usability of the created PoC.

The planned time for this data collection task is two weeks, including the creation of the questionnaire and the gathering of answers.

Data Analysis

The methodology for the data analysis stage will also be defined during the beginning stages of the project, and will take into consideration the "data analysis" step from the Research Process.

Quantitative and analytical methods were chosen to provide statistical evidence of the consistency of the UX and UI design of the application even if it's made up of two or more

micro-frontends. Hypothesis tests can be done to determine the answer to the research questions of this paper, by analyzing the results of the questionnaires.

Hypothesis testing is "*a statistical method used to determine if there is enough evidence in a sample data to draw conclusions about a population. It involves formulating two competing hypotheses, the null hypothesis (H0) and the alternative hypothesis (Ha), and then collecting data to assess the evidence*" (Biswal 2023). In this case, the null hypothesis would be that a Components Library within a micro-frontend architecture will maintain a consistent UI and UX design of the application, and the alternative hypothesis would be the opposite. The hypothesis tests can be performed following the below formula (3.1), in which $\overline{x}$ is the sample mean, $\mu 0$ is the population mean, σ is the standard deviation, and n is the sample size:

$$Z = (\overline{x} - \mu 0)/(\sigma/\sqrt{n}) \tag{3.1}$$

Code analysis could also be a reliable source of data analysis, since a framework such as Sonarqube can be integrated into the project and could provide metrics to support the hypothesis. These metrics could include code smells, bugs, and more importantly, code duplication and maintainability rating.

3.5 Summary

In conclusion, the dissertation project will follow a work plan with two main milestones — the delivery of the PREPD P2 project and the final dissertation. The Gantt Chart will serve as a basis for the work management through the dissertation development. The initial stages of the work plan involve the problem interpretation, state-of-the-art study, the solution and experimentation planning. The design plan is the next stage and will delve into the crucial aspects of software design, following micro-frontend architectural considerations, and a UI/UX design process using an Atomic Design approach.

The implementation plan entails the development, testing, deployment, and documentation stages, with a clear division of tasks for the different projects (Components Library, micro-frontends, and the Shell application). The experimentation and evaluation will need a well-defined methodology, and will incorporate data collection methods like code analysis and questionnaires, and data analysis utilizing quantitative and analytical approaches, such as hypothesis tests.

The comprehensive approach adopted in this dissertation project not only addresses the technical aspects but also places a strong emphasis on ethical considerations, that will be laid out in the next chapter.

Chapter 4

Solution Design

This chapter's goal is to describe the design process, starting with the description of the architectural design, and moving over to the UI design, with the design system information for the Components Library. This needs to be developed in accordance to the chosen solution, that will also be planned and described in this chapter.

4.1 Architectural Design

This section will delve into the architectural design, outlining the chosen PoC project, followed by the components diagram displaying the micro-frontends solution as well as the strategy for the backend service, and the Entity Relationship (ER) diagram displaying the main entities of the project.

4.1.1 Solution Description

The chosen project to display a micro-frontend system was an e-commerce solution, and the architectural design will focus on implementing projects aligning with a distinct business logic. For instance, envisioning an e-commerce platform, delineating business areas becomes evident: the login interface, product display pages, the checkout process, etc. Each of these functional components could be encapsulated into different micro-frontend applications, fostering a coherent and scalable architecture.

It's important to emphasize that the division of micro-frontends based on business logic is essential to follow DDD standards in software. An e-commerce solution will enable the creation of a project with clear boundaries of responsibilities. This approach allows for modular development, facilitating maintenance and scalability.

4.1.2 Solution Design

Following a micro-frontend system, the architectural design of the final solution aims to integrate all of the components into a single web experience.

In fig. 4.1 all of the components are displayed in a component diagram.

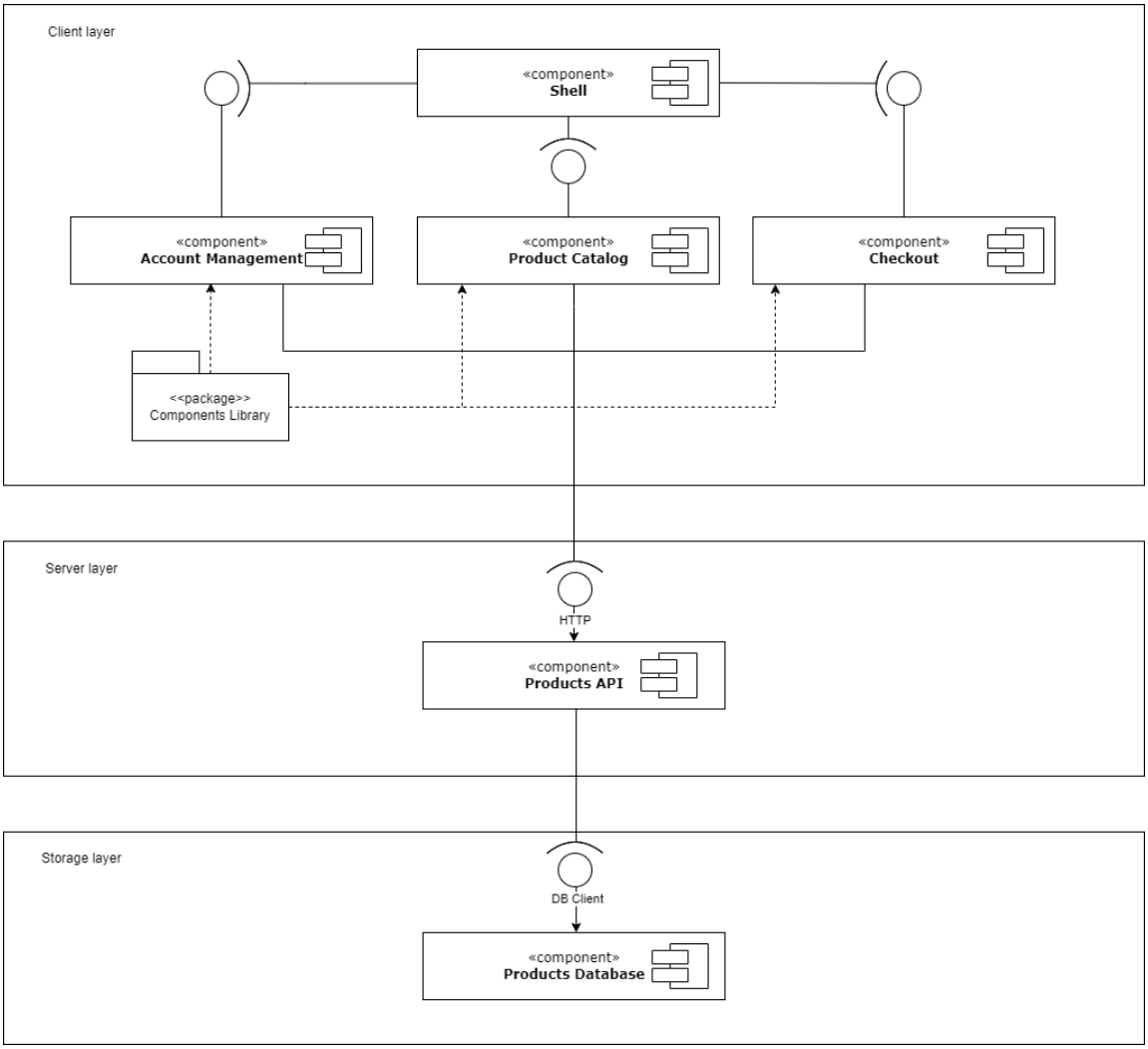


Figure 4.1: Architecture Design of the project - Component diagram, with the "Client", "Server" and "Storage" layers.

Since an application shell will be created, Client side composition will be followed (refer to Subsection 2.2.1). The micro-frontends will be created as SPAs and a Vertical split between them will be used to divide the different projects inside the web application (refer to Subsection 2.2.2). Each micro-frontend project will be responsible for several pages (or views), split by the business requirements and model.

The four frontend projects that will be created as Angular applications, visible in the Client layer (fig. 4.1) are the following:

- **Shell application** to display the micro-frontends, responsible for the routing and the header and footer of the application;
- **Account management micro-frontend** - this project will carry the responsibility of the authentication, authorization and management of the user's information through the Login, Register and Manage Account pages;
- **Product catalog micro-frontend** - as the name suggests, it's responsible for the Product listing and detail pages, as well as the main home page of the web application;

- **Checkout micro-frontend** - responsible for the products' Checkout and Payment process, the Shopping cart page and the Order management.

The **Components Library** project will be created inside an Angular workspace, as an Angular library that will be published as a public npm package, so it can be installed in each micro-frontend project.

Communication between the different micro-frontends can happen in the client layer, through URL parameters, browser storage, or window event listeners (Geers 2020, p. 99). However, the central point of communication between the micro-frontends will be done at the Server layer.

In the Server layer (fig. 4.1), an **API** will be created in order to manage the data through a backend service. This application will be developed using the latest supported version of ASP.NET Core (.NET 8.0), due to professional experience of the author in its use.

In a true micro-frontend ecosystem, each micro-frontend would have been linked with a respective microservice API, that would store and manage data relating to their correspondent business model, and then each microservice could communicate through messaging or HTTP requests to other APIs. For example, the Product Catalog micro-frontend could fetch and update data through a Product Catalog microservice, the Account Management micro-frontend could communicate with a Client User microservice, etc. However, for the smaller scale of this PoC and the intent being to focus on the frontend projects, a single backend service will be created, linked with a single database. For this, a minimal API (JeremyLikness 2022) can be created, since it can "build fully functioning REST endpoints with minimal code and configuration".

In the Storage layer (fig. 4.1), a database using MySQL will be created, to store the data for the web application. An ER diagram was designed (fig. 4.2), with the goal of modeling the database system (Bagui and Earp 2003, p. 29).

The ER diagram (fig. 4.2) was separated into structural aggregates, that each represent a business entity and their relationships, and subordinate structures (Shoval, Danoch, and Balabam 2004). In this case the aggregate roots would be: products, users and orders. Each of them are linked with a micro-frontend (Product Catalog, Account Management and Checkout respectively). Having this aggregation would make it easier to divide the API into different microservices in the future, that could each be integrated with the respective micro-frontends.

Value objects were defined, aligned with the DDD definition of being an immutable object that represents a specific value with no identity (Vernon 2013, p 219). For instance, the status of an order is represented by an immutable object composed of an enum with the possible order status, and is therefore an value object.

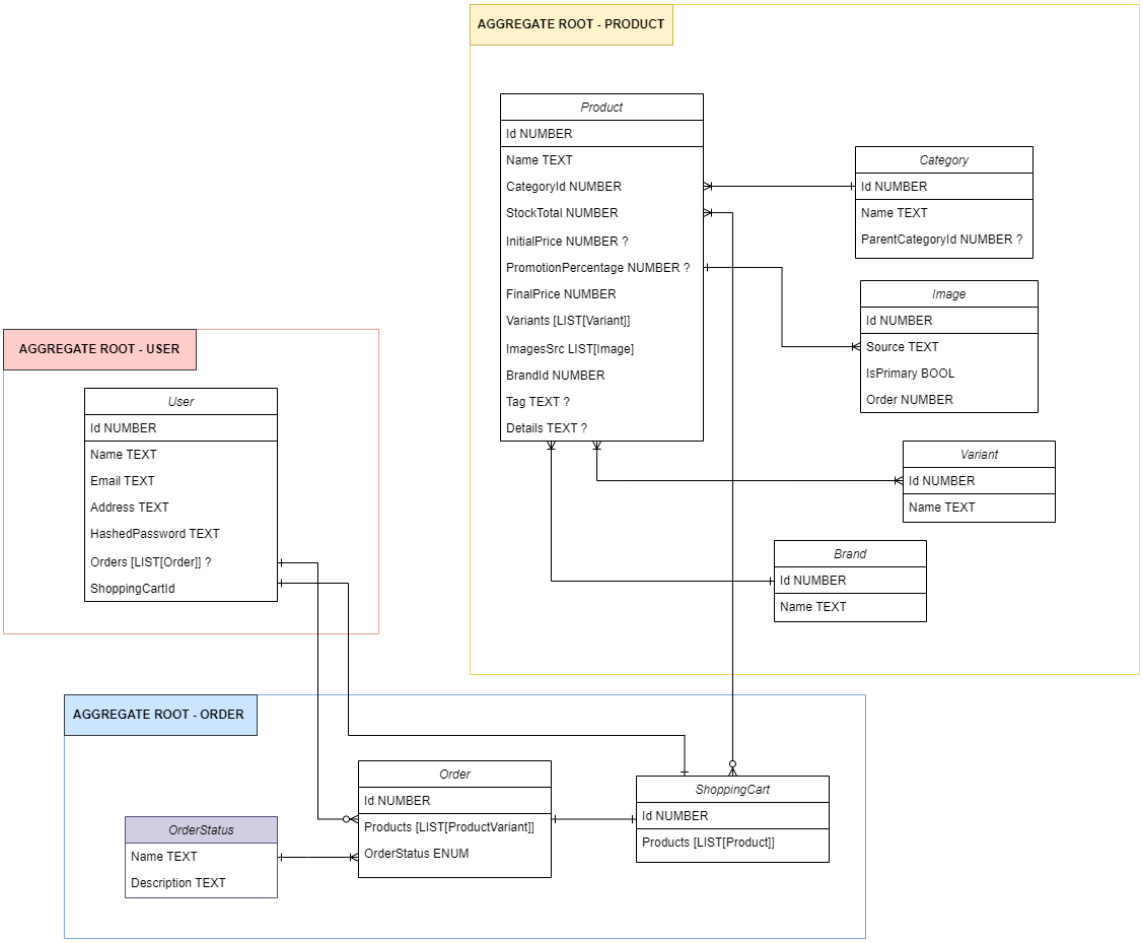


Figure 4.2: Entity Relationship Diagram, with the descriptions of the entity types.

As for the database schema (fig. 4.3), MySQL Workbench (Oracle 2024) was used to generate it through reverse engineering, after all the tables and keys were defined, following the business model of the application.

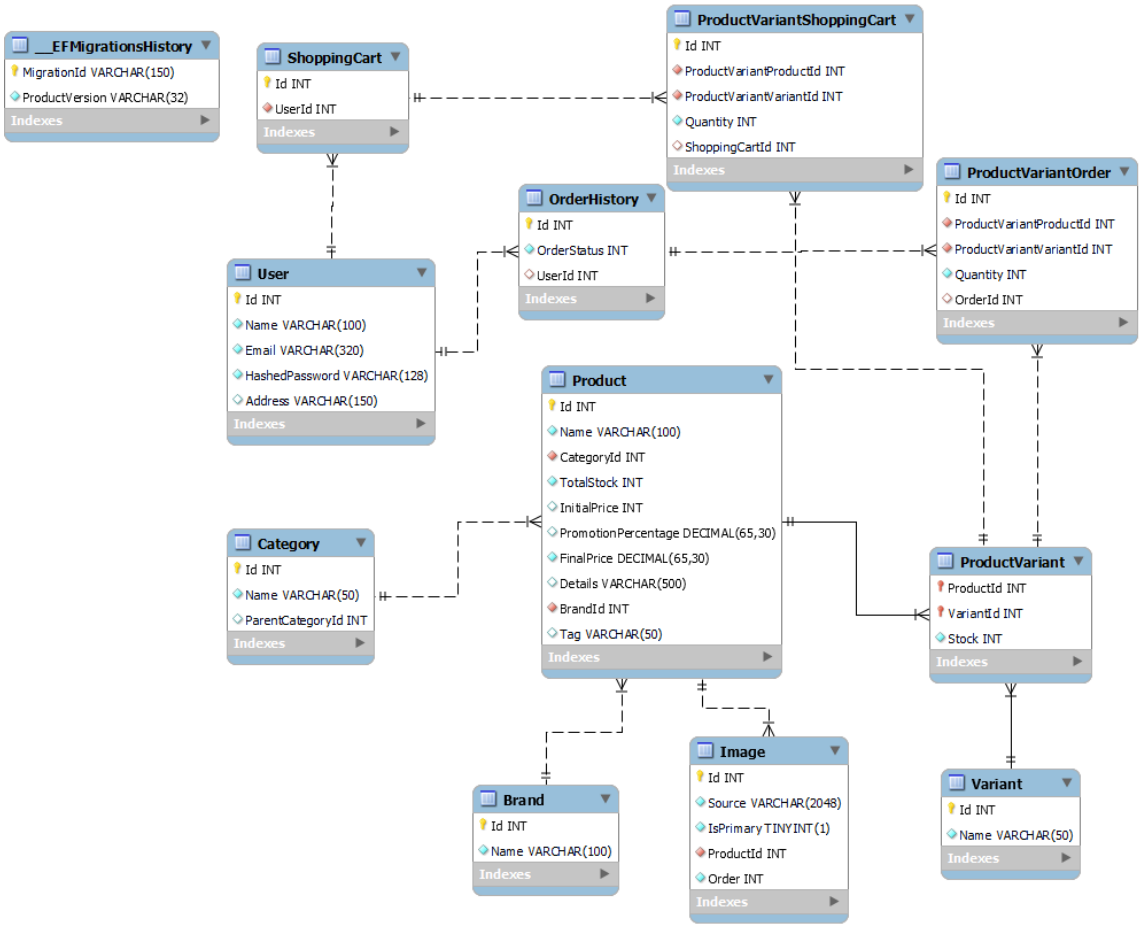


Figure 4.3: Database schema generated by reverse engineering.

4.2 UI Design

The design of the interface for the e-commerce solution was carried out in Figma, as it allows for integration of the Design System with the development of frontend code through the extension Figma for VS Code (help.figma.com 2024b). This allows for an easier integration of the design in the Software Development Life Cycle (SDLC) of the project and for greater accuracy in meeting the design requirements.

The first goal was to find the visual identity of the web application, so that the design could follow it and be clear and cohesive. Afterwards, the design of all of the pages was built and then the pages were broken down following the atomic design approach, in order to select the components that would be part of the design system.

4.2.1 Visual Identity

The PoC web application will provide an e-commerce experience, trying to emulate a high-end shopping brand imagined for the purpose of this dissertation, called Vinci.

The visual identity of this "brand" started out by creating a logotype (fig. 4.4) that defines the tone for the design of the web page experience - in this case, the logotype is simple and articulate, black and white with squared details, using the AmstelvarAlpha font, that is a

"serif typeface design with references to oldstyle and modern typeface designs" available by Google fonts (Funk 2024).

Figure 4.4: Logotype for the Vinci e-commerce web application.

Afterwards, the colour palette was chosen (fig. 4.5), in this case, colour ranges between dark grey (since black is typically too harsh) and white were the main focus, however coral colours were also used in the case of highlights and selections in order to capture the user's attention.

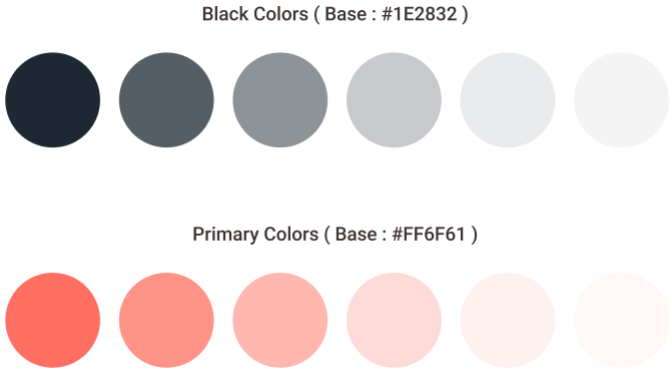


Figure 4.5: Color palette for the Vinci e-commerce web application.

As for the typography, the two fonts used in the project were Roboto (used for larger text sizes) and Open Sans (for the remaining text), both fonts are available to use in Angular projects in a straight forward way, and have similar open forms and friendly, open curves (Robertson 2024 and Matteson 2024).

4.2.2 Design System

The design for each page of the e-commerce web application Vinci was performed in Figma, and divided into the three subsequent business domains, that also correspond to the three micro-frontends: the product catalog (which involves the home page, product listing page and the product details page) - fig. 4.6; the checkout (which englobes the shopping cart and the checkout pages - fig. 4.7) and the account management (includes the login, register and profile edit pages - fig. 4.8).

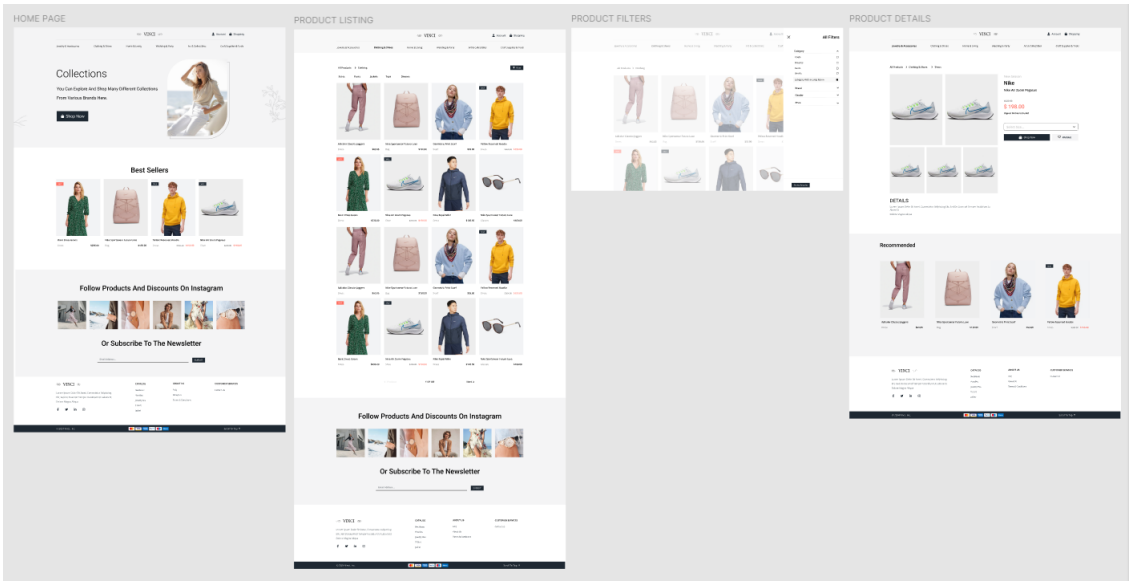


Figure 4.6: Figma design for the products pages.

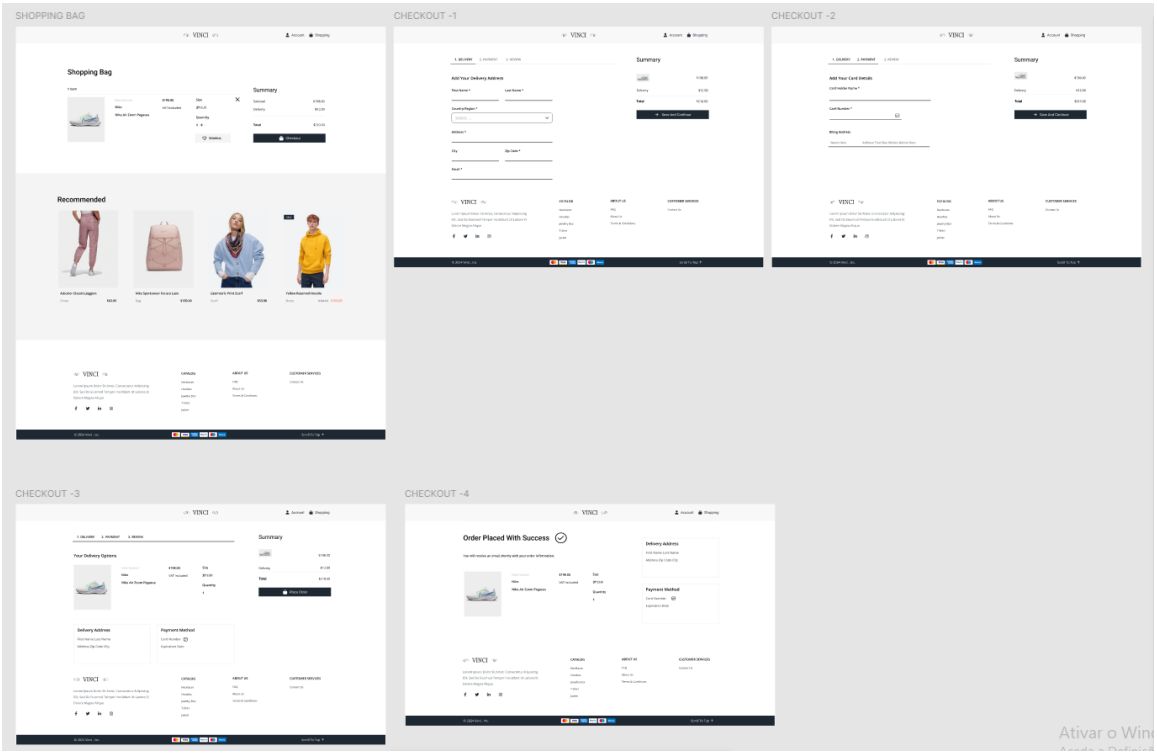


Figure 4.7: Figma design for the shopping bag and checkout pages.

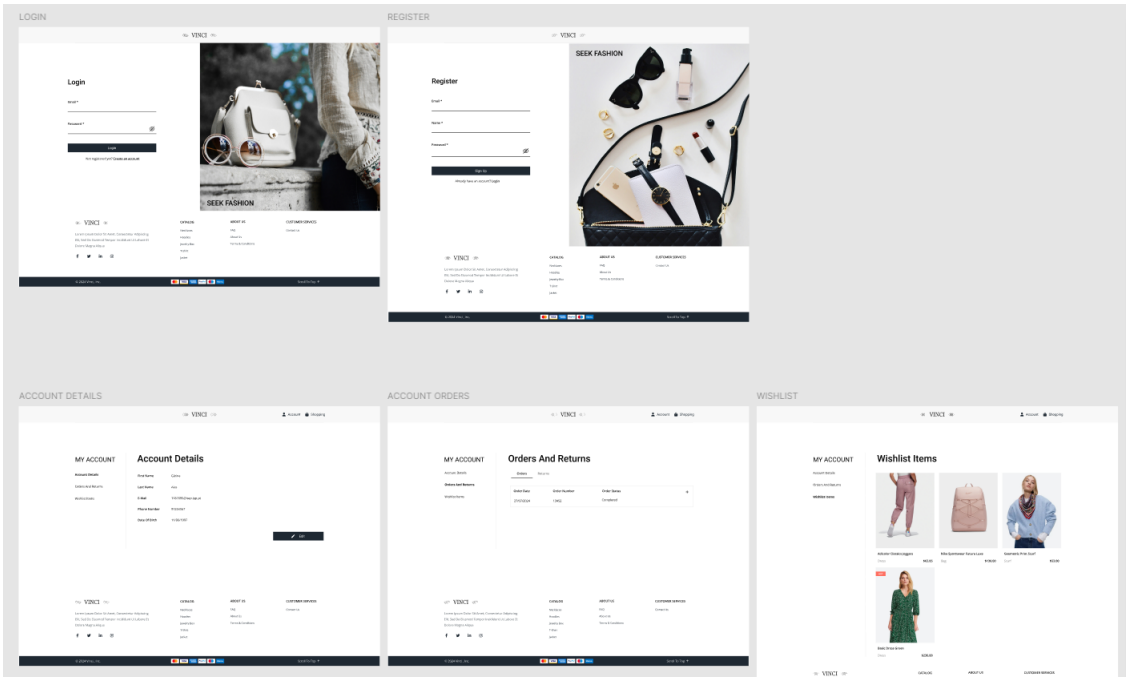


Figure 4.8: Figma design for the login, register and account details pages.

The design uses simple and contrasting elements following the visual identity colours. After designing each page, an Atomic Design approach was used to slowly build the Design System. Atomic elements were identified and turned into reusable components that would appear throughout the web application through different business boundaries. These atomic elements became available as components in Figma that could be reused in each page on the design (fig. 4.9). The objective was to choose the components that will become part of the Components Library and reused in each micro-frontend.

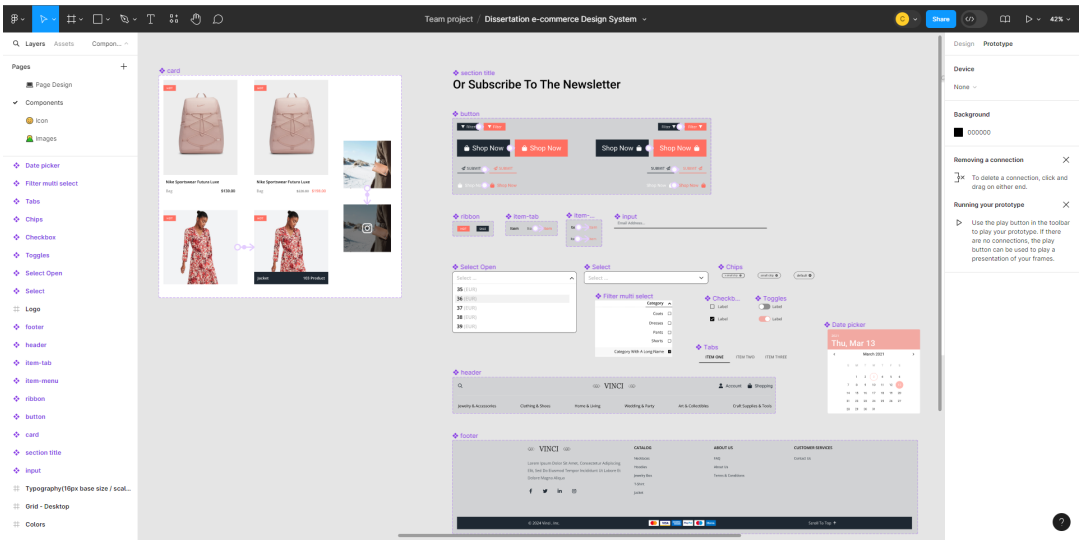


Figure 4.9: Figma design for the components, including the footer and header, created using atomic design.

The typography (fig. 4.10) was also added to the design system using text styles in Figma,

which allows the user "to define a set of properties that can be reused across team or organisations' files and projects" (help.figma.com 2024a). Same thing was done for the colour palette, where Colours Selection was used.

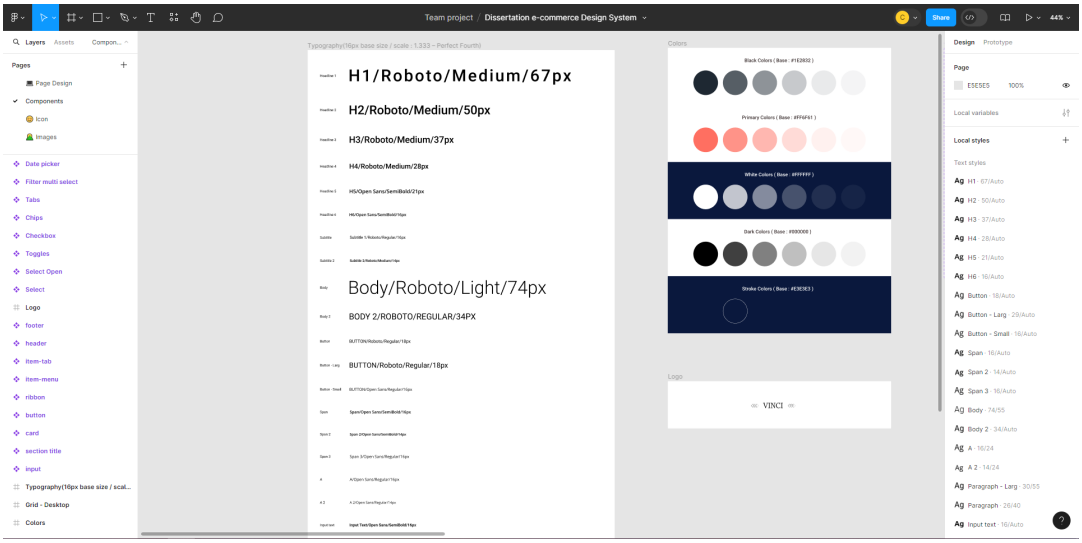


Figure 4.10: Figma text styles for the typography and colour selection for the colour palette.

The URL to the design can be accessed through the Appendix A of this dissertation, where the Figma pages are available for viewing.

4.3 Summary

The architectural design has set up the basis for the next chapter, which will enroll in the development of all of the projects needed to create the micro-frontend ecosystem. A Components Library, three micro-frontends and one app shell project will be created and part of the client (frontend) layer. A backend API with the responsibility of fetching and displaying data from a database will be implemented and deployed to be used throughout all of the micro-frontends.

The UI design of the web pages for the application was created in Figma, displaying all elements that compose the design system of the application, allowing for the selection of each of the components that will be available on the Components Library, as well as the visual identity (typography, colour palette) of the application. This will allow for a more efficient development of each component.

Chapter 5

Solution Implementation

This chapter will go over the development process of the web application, passing through the different projects that have been developed, including the Components Library, the micro-frontends as well as the application Shell and the Web API application. The goal is to describe the SDLC of the different projects integrated into a micro-frontend architecture. The Github repositories for all of the different projects can be accessed through Appendix B.

The testing, deployment and documentation stages will also be thoroughly described and showcased.

5.1 Development

The development of the six different projects (Components Library, three micro-frontends, the Shell application and the Web API) will be described thoroughly in the next sections, with implementation details.

The Angular projects were built using Angular and Angular Command Line Interface (CLI) 15.0 and the API using C# .NET 8.0 and Entity Framework for the MySQL integration.

5.1.1 Components Library

The first project to be implemented and the main focus of this dissertation was the Components Library, since it would be integrated in all of the micro-frontends.

The initial structure of the Components Library project was created mainly using Angular CLI commands, and a new project workspace named `ngx-isep-dissertation-libs` was created. After changing directory to the newly created workspace, the Components Library itself, called `ngx-isep-dissertation-components`, is generated inside the workspace. The prefix `ngx` is recommended when creating libraries, since it allows for developers to be aware that the library is made to be compatible with Angular (Angular.io 2022).

To generate a component that will be available to use through the library, the `ng generate` component command was used to create a new component called Button, inside the components portion of the library.

This component will be displayed as an example in this section to illustrate the development process of a component in this library.

The component was created inside the components folder with the four basic files, including the typescript component file, the unit tests (spec) file, the HTML and the Sassy Cascading

Style Sheets (SCSS) file. SCSS was chosen over CSS throughout the frontend projects since it allows for additional features and functionalities such as the use of variables, nesting, mixins, and other programming constructs (Verma 2023). A module file is also created, so the component can be grouped together with the respective services and models.

The library is supposed to accommodate to the Figma designs (refer to fig. 4.9), and therefore the components would need to follow the design vision. To make sure the components are customizable to all types of design available, (for example changing the button colour, size and icons) enums were created. These enums will serve as a contract. For example, a `ButtonColor` enum was created with the only colours available on the design for the Vinci web application - black, white and coral (following the colour palette visible in fig. 4.5). Same thing for the `ButtonSize` and types of `ButtonIcons` available.

This allows for customizable components, but at the same time follows the original design strictly, so that the overall UX in the application is cohesive and has less room for error. The possible configurations and combinations will be available on the Storybook documentation, which will be described in detail in a following section in this chapter - Section 5.2.

These designs are customizable through Angular `@Input` properties for the component. In the case of the Button component, it also has the label `@Input` (the button text), and the disabled boolean `@Input` (which disables the click event on the button).

The Button component HTML is as per the listing 5.1. The `ngClass` (line 4) is used to determine the colour and size of the button according to the inputs of the component. An `ngIf` is used to determine whether the icon is displayed or not, and if it is, the `ButtonIcon` enum is then mapped to the correspondent icon, taken from Figma's Iconify plugin (Figma 2024).

```
1  <div
2    role="button"
3    [attr.aria-disabled]="disabled ? true : null"
4    [ngClass]="{
5      'button-container': true,
6      white: color === white,
7      black: color === black,
8      coral: color === coral,
9      disabled: disabled === true,
10     medium: size === medium,
11     large: size === large,
12     small: size === small,
13   }"
14 >
15 <div *ngIf="icon">
16   <div class="button-icon" [innerHTML]="iconHtml" aria-label="icon"></div>
17 </div>
18   <div class="button-container-span">
19     <span>{{ label }}</span>
20   </div>
21 </div>
```

Listing 5.1: Button component HTML (HTML).

An additional file was added, called `exports.ts` and inside it, the component and module, and more files that are needed, such as the enums for the colour, size and icons of the button, are exported (listing 5.2).

```
export * from "./button.component"
export * from "./button.module"
export * from "./button-color.enum"
export * from "./button-size.enum"
export * from "./button-icon.enum"
```

Listing 5.2: Button component exports (Typescript).

In the root of the components folder, a components module is created to import and export the components of the library, and all the export files are brought together and exported as one in the file `public-api.ts` (listing 5.3). This will allow for the components to be available in the node modules of each application that installs the library package.

In total the library contains eight components that are exported, including:

- Button component;
- Checkbox component;
- Icon component;
- Input component;
- Multi-select component;
- Product tile component;
- Select component;
- Tabs component.

```
export * from './lib/ngx-isep-dissertation-components.service';
export * from './lib/ngx-isep-dissertation-components.component';
export * from './lib/ngx-isep-dissertation-components.module';
export * from './lib/button/exports';
export * from './lib/product-tile/exports';
export * from './lib/select/exports';
export * from './lib/multi-select/exports';
export * from './lib/checkbox/exports';
export * from './lib/icon/exports';
export * from './lib/tabs/exports';
export * from './lib/input/exports';
```

Listing 5.3: Public API Surface of ngx-isep-dissertation-Components Library (TypeScript).

The final folder structure is as follows in fig. 5.1. With the components all inside the components folder, other types of elements of the design system can be exported inside other folders, such as fonts, colours, etc. This allows for a greater folder organization.

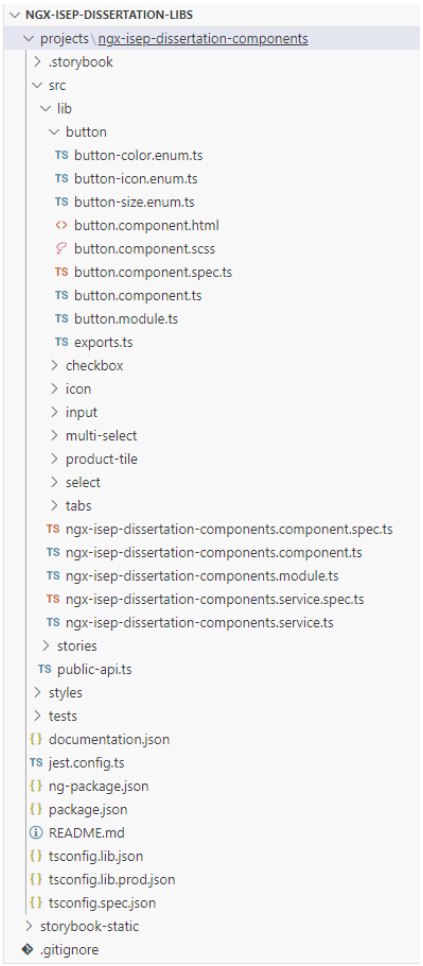


Figure 5.1: Folder structure of the Components Library project.

In order to develop and locally debug the Components Library, a local angular application can be created as a demo for the library to be integrated in, since the library is not a deployable application (J 2021). This way the library should be built locally using the `ng build` command with the `--watch` flag. This command will create a `dist` folder that will have the library available to be installed in other projects. The library can be installed using the path to the folder:

```
npm install "file://PATH_TO_FOLDER\dist\ngx-isep-dissertation-components"
```

This way the library can be developed and debugged locally, in real time, meaning that anytime a change is saved in the library project, it will appear on display in the project it has been installed, due to the `--watch` flag.

The components may be used in the micro-frontend applications, by importing the desired module to the component module. For example, to use the Button component, the `ButtonModule` is imported to the `ProductsHomePageModule` of the Product Catalog micro-frontend (listing 5.4 - line 9). More modules for the Components Library are imported (`ProductTileModule` for the Product Tile component in line 10, and `IconModule` for the Icon component in line 11).

```
import { ButtonModule, IconModule, ProductTileModule } from "ngx-isep-
    dissertation-components";
...

@NgModule({
    declarations: [ProductsHomePageComponent],
    imports: [
        ProductHomePageRoutingModule,
        CommonModule,
        ButtonModule,
        ProductTileModule,
        IconModule,
        CategoriesHeaderModule
    ],
    providers: [
        HttpClientModule,
    ],
})
export class ProductsHomePageModule { }
```

Listing 5.4: Product home page module importing the Components Library - Product Catalog micro-frontend (TypeScript).

Then the selector for the Button component `<isep-lib-button>` can be used on the component HTML, with the desired inputs. In listing 5.5 two different buttons are used, a black one with a label 'Shop Now' with an icon for a shopping bag that is disabled until variants are selected (lines 1-7), and a white one with a heart icon with the label 'Wishlist' (lines 9-14).

```
<isep-lib-button
    [label]="'Shop Now'"
    (click)="shopNow()"
    [color]="blackButton"
    [icon]="bagIcon"
    [disabled]="variantSelected === 0">
</isep-lib-button>

<isep-lib-button
    [label]="'Wishlist'"
    (click)="shopNow()"
    [color]="whiteButton"
    [icon]="heartIcon">
</isep-lib-button></div>
```

Listing 5.5: Example use of button component selector - Product Catalog micro-frontend (HTML).

This HTML will result in two buttons shown in fig. 5.2 (the different widths are related to additional CSS classes that can be applied, and the black one is enabled because a variant was selected).

Figure 5.2: Button component examples according to the previous HTML.

Npm Publish

Additionally, to allow the use of the library on other projects (other than locally), the package needed to be published using npm (docs.npmjs 2023).

A package version and name needs to be set in the `package.json` file, along with the author's name and repository so they are accessible on the npm package information. The version can be incremented manually by running `npm version major/minor/patch`, depending if the update is major, minor or patch.

A `README.md` file is also required and added with the information of the documentation (that will be further explained in Section 5.2 - Documentation). With the creation of the `dist` folder after running the `ng build` command, the library can be packed into a .tgz package using the command inside the `dist` library folder.

A package called `ngx-isep-dissertation-components-15.0.1.tgz` is created and it is ready to publish. '15.0.1' refers to the version of the package set in the `package.json` file: the 15 major corresponds with the Angular version compatibility, so that it's easily identifiable if the library will be compatible with the Angular project. Npm requires a login with two factor authentication to authorize the publishing of a library. After authenticating the package is published to npm using the command:

```
npm publish ngx-isep-dissertation-components-15.0.1.tgz
```

This way the package becomes available on the npm registry with information regarding the versions, code and usage of the package (fig. 5.3).

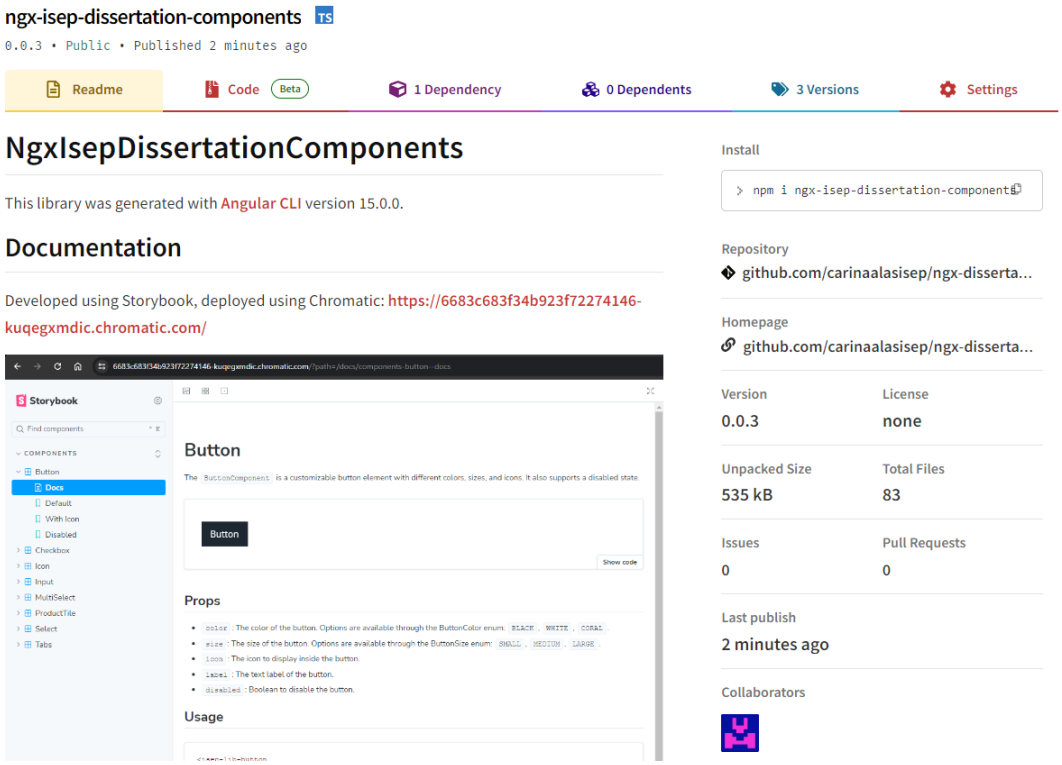


Figure 5.3: Npm library package for *ngx-isep-dissertation-components*.

The automation of the npm publishing process will be explained in Section 5.4 - Deployment.

5.1.2 Micro-frontends - Product Catalog

Each micro-frontend was built following the same structure. In this subsection, the `dissertation-product-cat-mfe` project (responsible for displaying the product catalog of the web application, from now on named Product Catalog) is described. The project was created using the `ng new` Angular CLI command that creates the structure of a new angular application (Angular.io 2023a).

According to the Figma design 4.6, the Product Catalog micro-frontend will be responsible for three views:

- The products home page;
- The products listing page;
- The product details page.

The Angular Router service (`RouterModule`) was used to manage navigation in the application through routing, to access these three views.

The main module for each individual micro-frontend that is created by default on an Angular project is usually called `AppModule`. This is the module that will be exposed in the bootstrap of the micro-frontend, for it to be loaded in the Shell application. The routing was handled thinking about the final routing in the application Shell - each micro-frontend's `AppModule` should manage the routing for the standalone micro-frontend, and the Shell will redirect to the same routing for the main application.

In the case of Product Catalog, the application will be accessible through the path `/products`. To make sure that the the application is always accessed through the `/products` route, an initial routing redirect is set up for that same route.

The module that will be loaded on the `/products` path, is the main module called `ProductsMainModule`, which will group all different modules of the application and manage their routing once inside the main route path. Its routing service (`ProductsMainRoutingModule`) will be responsible for loading the module that will be on the root of the `/products` path - the `ProductsHomePageModule` that declares the component of the products home page. It will also enable other routing paths inside it, such as the `ProductsListingModule`, accessible through `/products/listing`, which will show the products listing page (fig. 5.4).

Lazy loading of the different modules is done by a `loadChildren` method, which helps keep initial bundle sizes smaller, decreasing load times (Angular.io 2023b).

For the lower modules, the routing will be handled accordingly, inside each module. For each product detail page, the id of the product will define the routing path, through `products/listing/:id` (fig. 5.5).

The Components Library was installed after the library had been published to npm. It is used in every page of the Product Catalog micro-frontend, following the Figma design (fig. 4.6). For the home page, the Button, Icon and Product Tile components are used. The listing page uses the Button, Product Tile, Icon and the Multi-select components. The details page for each product uses the Button, Product Tile and Select components.

A service was created to integrate requests made to the Web API that supports the micro-frontend architecture, called `ProductService`. This service uses `HttpClientModule` to perform HTTP requests to the API endpoints to access the information needed to load on the pages. The API endpoints are accessed through the API URL that is configured in the `environment.ts` file, since it will vary depending on the environment the application is running (local or production).

On the loading of each page (making use of `ngOnInit`) the information needed from the Web API is loaded through calls to the `ProductService`. For example, on the loading of the `ProductListingComponent` information regarding the brands, categories and the products displayed on the page are loaded.

As it is visible on fig. 5.4, the categories are loaded onto the categories bar and the products are displayed in a grid, with a pagination feature. The categories and brands are also displayed on a drawer container (fig. 5.4 - right), with a set of filters that make use of the Multi Select component from the library. If the user selects a product, they will be redirected to the display page of that product (fig. 5.5).

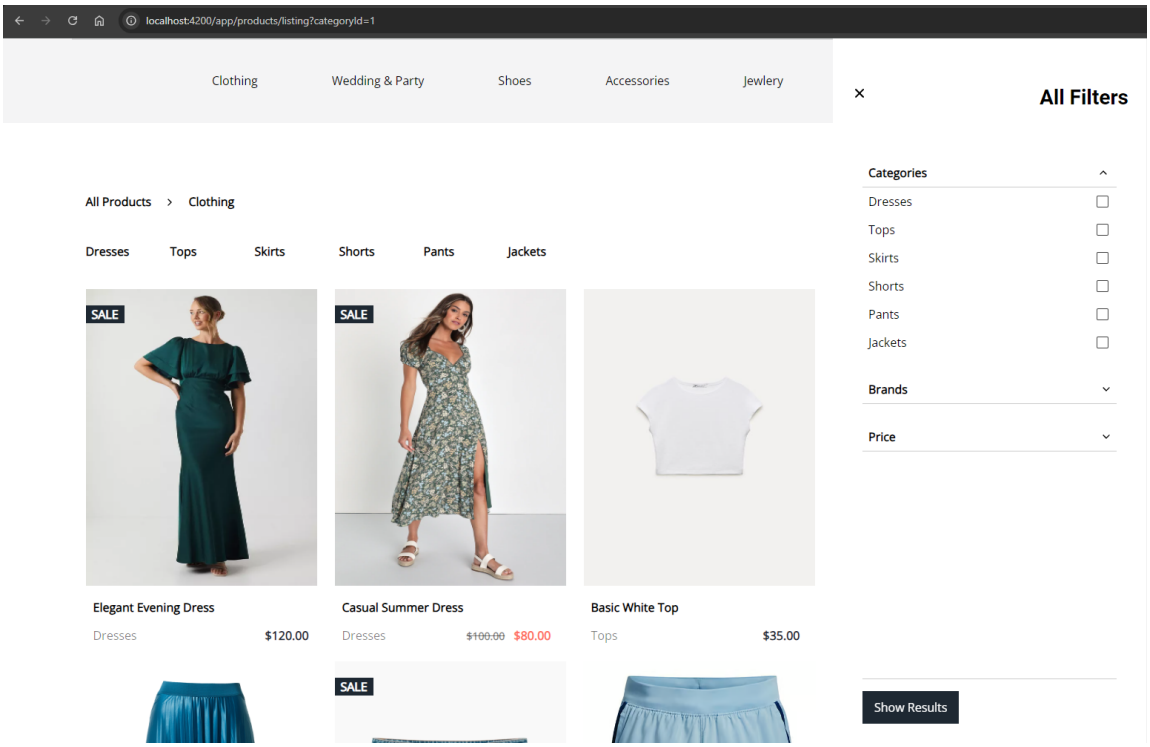


Figure 5.4: Product Listing Page loaded with service information, and the filters to apply the search.

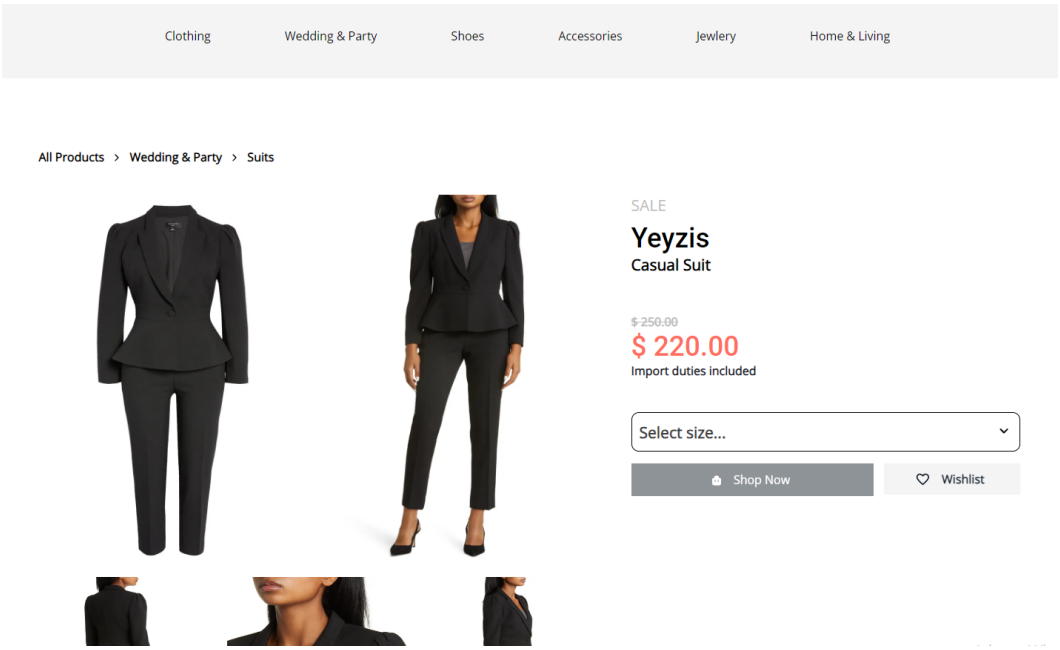


Figure 5.5: Product Details Page loaded with service information for the product id. The 'Shop Now' button is disabled since the variant (size) hasn't been selected yet.

5.1.3 Micro-frontends - Checkout

The `dissertation-checkout-mfe` project (Checkout) is responsible for the shopping cart, and order checkout on the Vinci application. All of the application setup was identical to what has been described in the previous subsection for the Product Catalog micro-frontend.

According to the Figma design (see fig. 4.7), the Checkout micro-frontend will display three main views:

- The shopping cart page;
- The order page, that has three states:
  - Delivery details page;
  - Card details page;
  - Review order details page;
- And finally the order placed with success page.

The routing inside the application was handled in a similar way to the Product Catalog micro-frontend. The `MainRoutingModule` sets `/checkout` as the prefix to all the routes inside this micro-frontend, and then the route `/checkout/shopping-cart` will use the user details to load the user's shopping cart with the correspondent products.

The products are added to the shopping cart in the Product Catalog using the Web API's POST shopping cart products endpoint, on the product details page after the user clicks the button 'Shop Now' (visible in fig. 5.5).

The products of the shopping cart are then loaded using a `ProductService` that performs HTTP requests to the Web API (similar to the one described previously for the Product

Catalog, since the Web API is the same - also configured by environment) - this is then displayed on the page (fig. 5.6). The components from the library present on the page and imported to the `ShoppingCartModule` are the Button, Product Tile and Icon components.

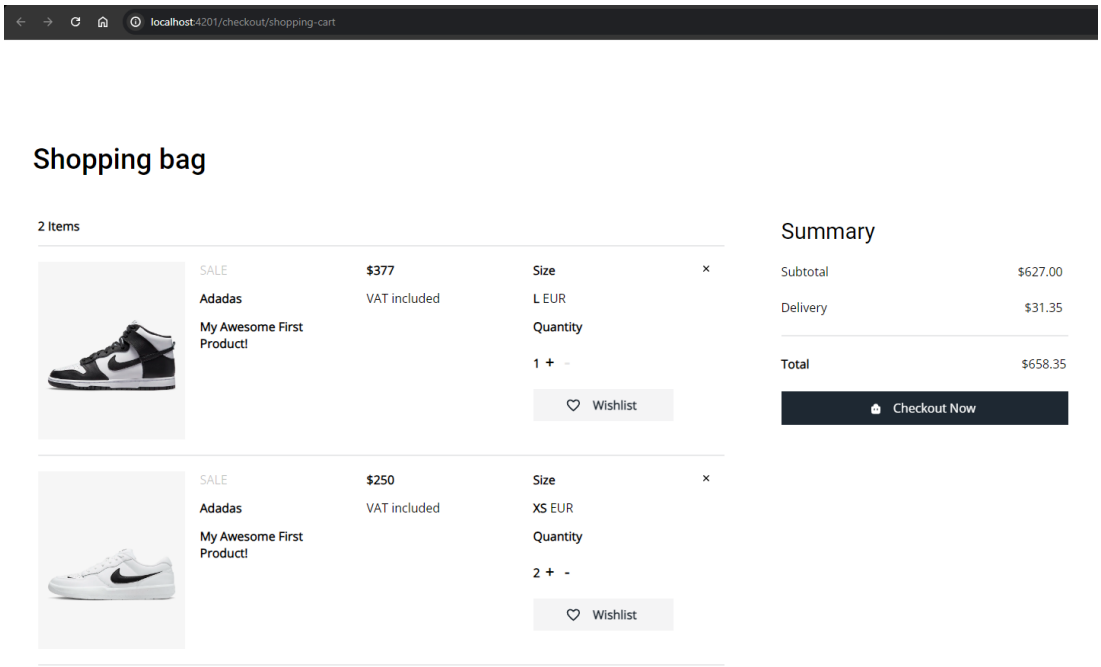


Figure 5.6: Shopping Cart Page loaded with service information.

By clicking 'Checkout now' the user's order is created with the shopping cart information. The page redirects to the order page, on the route `/checkout/order/:id` (the id of the order) (fig. 5.7). A form is presented asking for the delivery details of the products, using also components from the library, including the Input and Select components. These two components possess an "error" state, in which the border changes colour to red according to a flag. If the user tries to continue with the order without filling in every mandatory field, that flag is activated and the components turn red. The Tabs component is also used, to switch between the delivery, payment and review stages of the order process.

This user information asked in the form could be considered sensible information (name, address, credit card number, etc.), therefore it won't be stored in the database and will only exist on the static web page (same thing is applied to the following page where payment information is requested - it isn't actually used to perform any payment request and is just static).

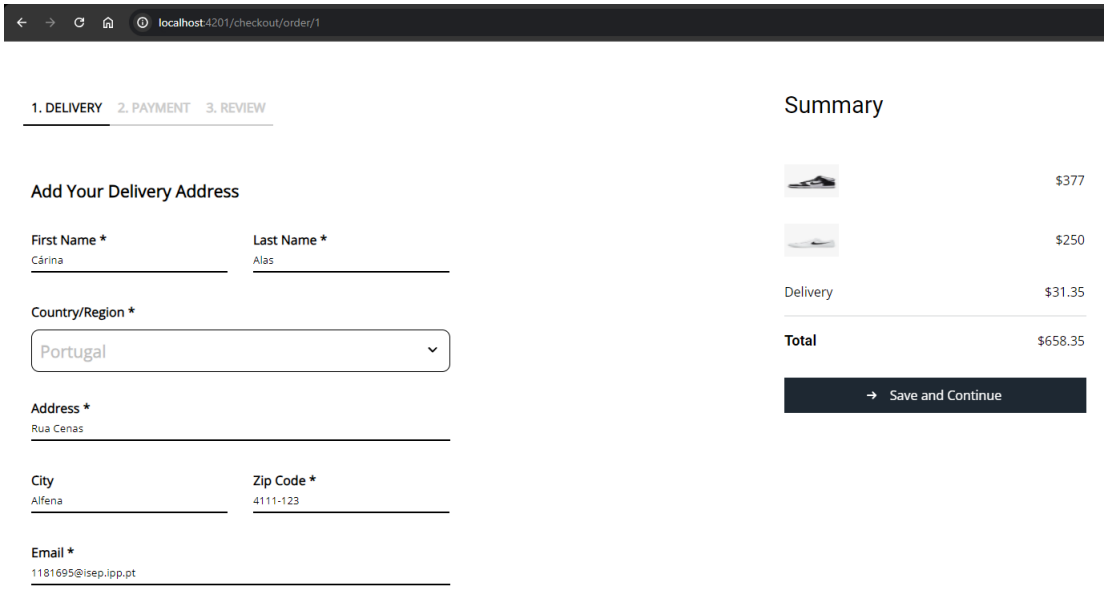


Figure 5.7: Order delivery details Page form filled in with first tab enabled and others disabled.

5.1.4 Micro-frontends - Account management

The `dissertation-account-mgnt-mfe` project (Account management) is responsible for the management of the state of the user when navigating through the Vinci web application.

This micro-frontend possesses a token service, where the user token is stored through the local browser storage (listing 5.6), so that it becomes available to all other micro-frontends, including the Shell application. The token is generated and managed by HTTP requests to the Web API, through the `ProductService`, in a similar way to the previous micro-frontends.

```
1 setToken(token: string): void {
2     localStorage.setItem('authToken', token);
3   }
```

Listing 5.6: Token service method to store the token in the local storage (Typescript).

This way, when routing to the shopping cart page, or the account details page, the Shell will check if the `authToken` storage item is stored with a valid token. If it isn't, the user is redirected to the login page since the user should be logged in in order to access that information.

The same happens in the Product catalog micro-frontend, in the product details page (fig. 5.5) - if the user tries to add the product to the shopping cart without being logged in, they will be redirected to the login page, but first a local storage item with the product id is set as `lastProductPageId`. Afterwards, the login component will check if that storage item is set, and if so, the user is redirected to the same product details page they were on before, otherwise, they are redirected to the home page.

In the login page, the user can also choose to register if they don't already have an account set up. After registering, they are required to login to access the web application again.

The views that are part of this micro-frontend are the following:

- User login page;
- User register page;
- Account management pages, divided into:
  - User details page;
  - Order details page;
  - Wishlist page;

It should be noted that the account management pages are mocked and static, meaning that the data cannot be updated, due to their implementation not being relevant for this PoC.

In the login and register pages, the Input component from the Components Library is used with a type "password", to hide the view of the text the user inputs. There is also an eye toggle icon that allows the user to view the password they have written (fig. 5.8).

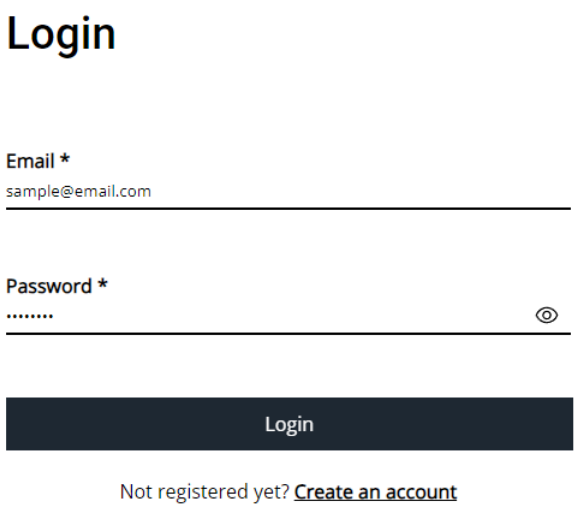


Figure 5.8: Login page form with input component of type "text" (email) and type "password" (password).

5.1.5 Micro-frontends - Webpack Module Integration

The package used for the module integration of the micro-frontends with the application Shell was Webpack Module Federation (Steyer 2020) which allows for client side composition integration with micro-frontends, using an application Shell. For each micro-frontend, the package `@angular-architects/module-federation` was installed using npm.

The main files on the micro-frontend side, needed to integrate with the Shell are:

- `webpack.config.ts` (listing 5.7) and `webpack.prod.config.ts` files;
- `src/bootstrap.ts` (listing 5.8) file;

- `AppModule` (listing 5.9) file.

The `webpack.config.ts` file is what enables the micro-frontend to be accessible for integration into another application. Within it, module exports are defined (listing 5.7 for Product Catalog example) and a `remoteEntry.js` file is specified (line 9) so that it is generated and available for the Shell application to integrate once the application is running. The Product Catalog application is set as a library of type `module` (line 7) since it is a micro-frontend (the alternative type, `shell`, will be set in the Shell application).

Additionally, the application will expose the `bootstrap.ts` file under the name `product-cat-mfe` (line 11) through the `remoteEntry.js`. The Shell application will need to know this name to load the micro-frontend correctly. The `shared` property (line 13) is configured to share all of the packages installed in the application's `package.json`, for the Shell application to load.

The `bootstrap.ts` file (listing 5.8) is responsible for exposing the `AppModule` (line 18). Subsequently, the `AppModule` (listing 5.9) is exported via the bootstrap process, transforming it into a web component that can be integrated with the Shell application.

The `DoBootstrap` hook (listing 5.9 - line 14) performs the manual bootstrapping of the application, using an `Injector` through the constructor (line 18). This injection is defined as '`product-cat-mfe`' via the `customElements` (line 27), that belongs to the `@angular-elements` package. This package creates a class encapsulating the functionality of the provided component—in this case, the `AppComponent`.

For the micro-frontends that possess static images present in the `assets` folder, an Angular `Pipe` had to be created in order for the images to be accessible by the Shell. This `Pipe` carried out the mapping of the path of the image within the micro-frontend project, to a Web Pack Public path (webpack.js 2024) accessible by all the applications within the micro-frontend Web Pack configuration.

```
1  module.exports = {
2
3    ...
4
5    plugins: [
6      new ModuleFederationPlugin({
7        library: { type: 'module' },
8        name: 'ProductCatUi',
9        filename: 'remoteEntry.js',
10       exposes: {
11         './product-cat-mfe': './src/bootstrap.ts',
12       },
13       shared: {
14         ...shareAll({
15           requiredVersion: 'auto',
16           singleton: true,
17         }),
18       },
19     }),
20     sharedMappings.getPlugin(),
21   ],
22 };
```

Listing 5.7: *webpack.config.js* module exports for the Product Catalog micro-frontend (JavaScript).

```
import { enableProdMode } from '@angular/core';
import { platformBrowser } from '@angular/platform-browser';
import { AppModule } from './app/app.module';

if (!(window as any).shell) {
  enableProdMode();
}

declare const require: any;
const ngVersion = require('../package.json').dependencies['@angular/core'];

(window as any).plattform = (window as any).plattform || {};
let platform = (window as any).plattform[ngVersion];
if (!platform) {
  platform = platformBrowser();
  (window as any).plattform[ngVersion] = platform;
}
platform.bootstrapModule(AppModule)
  .catch((err: any) => console.error(err));
```

Listing 5.8: *bootstrap.ts* - Product Catalog micro-frontend (TypeScript).

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserAnimationsModule,
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  bootstrap: [AppComponent],
  providers: [HttpClientModule, { provide: DOCUMENT, useValue: document }]
})
export class AppModule implements DoBootstrap {

  constructor(
    private router: Router,
    private injector: Injector) {
    this.router.initialNavigation();
  }

  ngDoBootstrap(): void {
    const ce = createCustomElement(AppComponent, {
      injector: this.injector,
    });

    customElements.define('product-cat-mfe', ce);
  }
}
```

Listing 5.9: *AppModule* snippet - Product Catalog micro-frontend (TypeScript).

5.1.6 Shell Application

The `dissertation-shell-app` (Shell) application's main responsibility is to integrate each micro-frontend module into a single web application experience. The configuration of a micro-frontend into a Shell application will be described in this subsection.

A new Angular project was created, similar to the previous micro-frontends, and named `dissertation-app-shell`. The `bootstrap.ts` file is then edited and the application type is defined as `shell` (listing 5.10 - line 7).

```
1 import { bootstrap } from "@angular-architects/module-federation-tools";
2 import { AppModule } from "./app/app.module";
3 import { environment } from "./environments/environment";
4
5 bootstrap(AppModule, {
6     production: environment.environment === "prod",
7     appType: 'shell'
8 })
```

Listing 5.10: bootstrap.ts file for the Shell application (TypeScript).

The Webpack Module Federation also requires the configuration of the `webpack.config.js` file with all the shared modules that the Shell needs to load from the micro-frontends, this also includes the `ngx-isep-dissertation-components` library.

The information regarding each micro-frontend was added to an `environment` file (listing 5.11), making it easily configurable depending on the environment the application is running (local or production). The `remoteEntry` property (lines 7 and 14) is the path to the `remoteEntry.js` file used to load the remote module onto the Shell application. The `exposedModule` property (line 9 and 16) is the name of the module that is exposed in each micro-frontend's `bootstrap.ts` file, and the `routePath` (line 10 and 17) is the routing inside the Shell in which each micro-frontend will be displayed.

```
1  import { Env } from "src/app/core/models/config.model";
2
3  export const environment: Env = {
4      environment: 'local',
5      productCatApiUrl: 'http://localhost:5187',
6      mfes: {
7          product: {
8              remoteEntry: 'http://localhost:4202/remoteEntry.js',
9              type: 'module',
10             exposedModule: './product-cat-mfe',
11             routePath: 'products',
12             elementName: 'product-cat-mfe',
13         },
14         checkout: {
15             remoteEntry: 'http://localhost:4201/remoteEntry.js',
16             type: 'module',
17             exposedModule: './checkout-mfe',
18             routePath: 'checkout',
19             elementName: 'checkout-mfe',
20         },
21         login: {
22             remoteEntry: 'http://localhost:4203/remoteEntry.js',
23             type: 'module',
24             exposedModule: './account-mgnt-mfe',
25             routePath: 'account',
26             elementName: 'account-mgnt-mfe',
27         }
28     }
29 }
```

Listing 5.11: Local environment configuration file for the Shell application (TypeScript).

In the `AppModule` of the Shell application, the base Hypertext Reference (HREF) route of the application was set as `/app`, so all routes inside of the Shell application will have that set as prefix. This is meant to ease the navigation inside the Shell, and make it easily identifiable that the micro-frontends are being accessed through the Shell application and not the standalone micro-frontend.

The main routing module of the Shell application is setup in a `RouterModule` (listing 5.12). The root path will fallback to the `/app` route, and the component that will always be loaded in that route is the `LayoutComponent` (line 11). This component is responsible for loading the header and the footer components at all times during the navigation inside of the Shell.

Any other path that doesn't match `/app` or any of its child routes, will land on the `NotFound Component`'s page (line 19), with a 404 message.

Inside the `children` property (listing 5.12 - line 12), are the configured routes to each micro-frontend with the information loaded from the `environment.ts` configuration file.

```
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3  import { NotFoundComponent } from './not-found/not-found.component';
4  import { MFERouting } from '../core/utils/create-mfe-route';
5  import { checkout, login, product } from '../core/models/mfes.constants';
6  import { LayoutComponent } from './layout/layout.component';
7
8  const routes: Routes = [
9    {
10     path: '',
11     component: LayoutComponent,
12     children: [
13       ...MFERouting.createRoute(product),
14       ...MFERouting.createRoute(checkout),
15       ...MFERouting.createRoute(login),
16     ]
17   },
18 {
19   path: '**',
20   component: NotFoundComponent,
21 }
22 ];
23
24 @NgModule({
25   imports: [RouterModule.forChild(routes)],
26   exports: [RouterModule],
27 })
28 export class MainRoutingModule { }
```

Listing 5.12: Main routing of the Shell application (TypeScript).

The `MFERouting` (listing 5.12 - lines 13 - 15) uses web components to load the remote modules of the micro-frontends, which allows for a technology agnostic integration (Meraj 2023). Let's say a new micro-frontend is developed in a different version of Angular, or even using other technologies such as React, Vue.js, etc. It is possible to integrate this new micro-frontend in this architecture, since the web component strategy applied to the Shell application would allow that.

The different properties of the micro-frontends set in the `environment.ts` file (listing 5.11) are loaded in the creating of the child route, namely the `routePath` (line 13) - which will be

used as the path in the application. This means that the Product Catalog micro-frontend will be loaded in the `/app/products` route, and that the Checkout micro-frontend will be loaded in the `/app/checkout` route (as per listing 5.11).

Each route will be loaded inside an `MfeWrapperComponent` with the data being the other properties set on the `environment.ts` file. The `MfeWrapperComponent` is a wrapper that uses the Webpack Module Federation's tools to set the options to load the remote module of the micro-frontend (`remoteEntry` file path, `exposedModule` name, etc.).

The micro-frontends will be loaded on each respective route, with the header and footer visible (fig. 5.9 - products page available on `app/products`, fig. 5.10 - shopping bag checkout page available on `app/checkout/shopping-cart`).

The header allows for navigation between the different micro-frontends inside of the Shell, by clicking on the Vinci logo, the user is redirected to the main page (products home page), and by clicking the shopping icon the user will be redirected to the shopping cart checkout page and see their shopping cart items.

The header is also at the center of the communication between the Shell and micro-frontends, since it is responsible for the display of the number of shopping cart items of the user. On the load of the application, if a user is logged in, a request is performed to the shopping-cart endpoint to retrieve the amount of products the user has already added. Each time a user adds a product to the shopping cart in the Product Catalog micro-frontend, an event is sent to notify the Shell's header that the number of items should be increased. Same thing happens in the Checkout micro-frontend, if the number of items is decreased or increased in the shopping bag checkout page (fig. 5.10) an event is sent to the Shell.

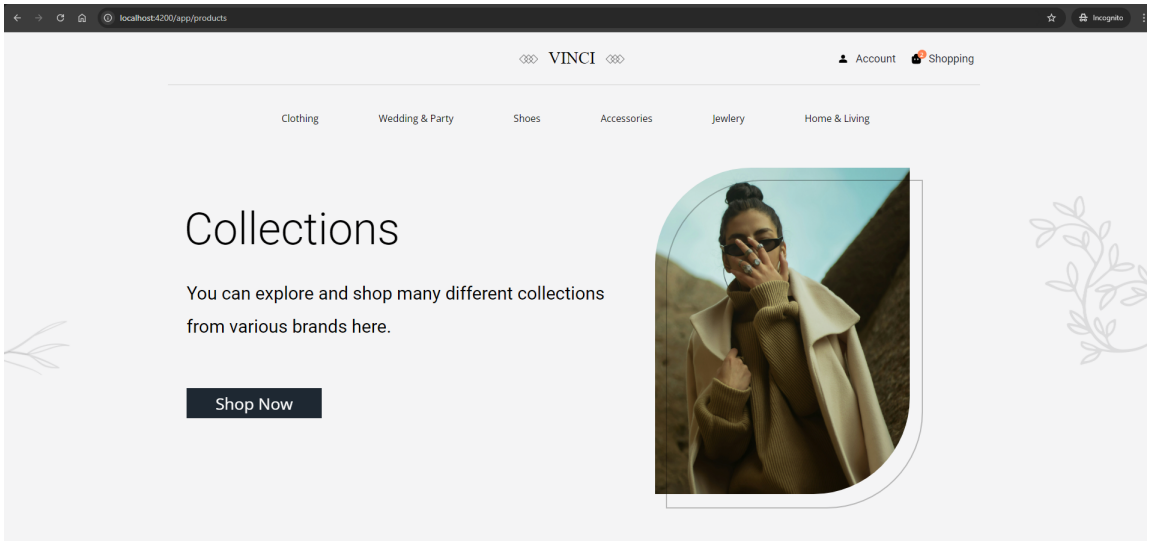


Figure 5.9: Products page of the Shell application, loading the Product Catalog micro-frontend with header.

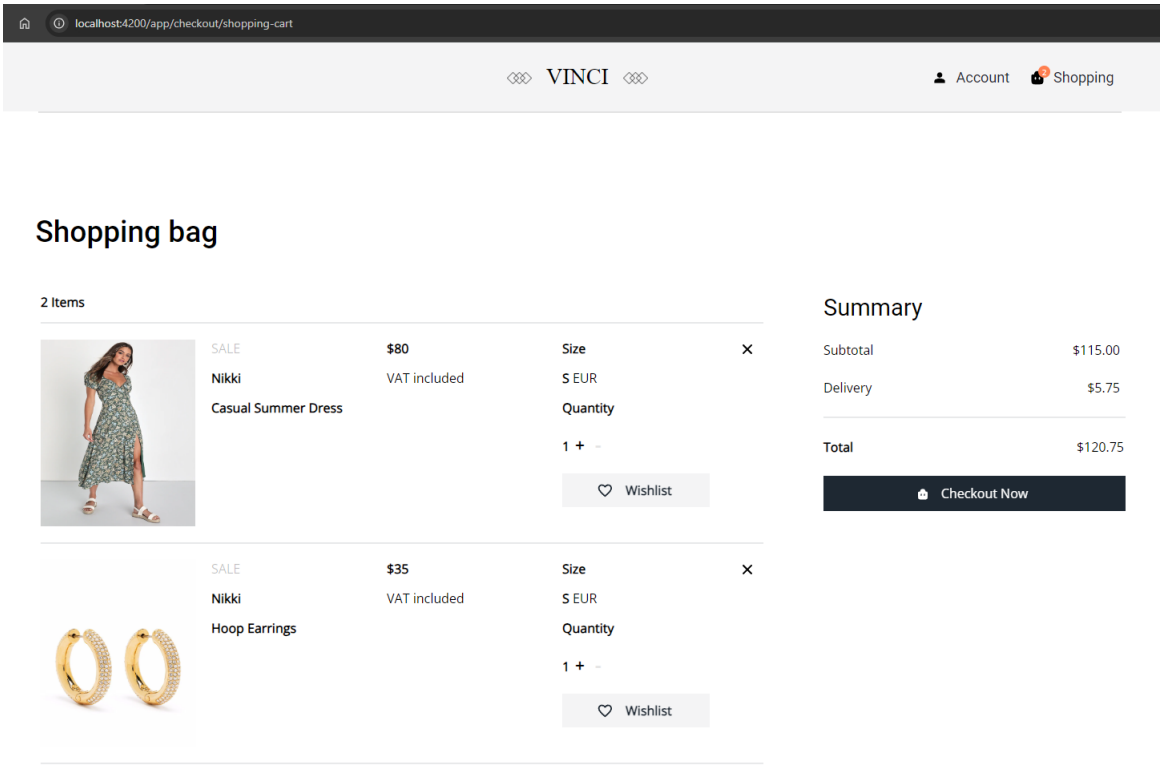


Figure 5.10: Shopping bag page of the Shell application, loading the Checkout micro-frontend with header.

As previously mentioned, the Shell is also responsible for managing the routing between the micro-frontends. In order to be redirected from the Product Catalog micro-frontend to the Checkout for instance, the navigation has to go through the Shell.

For that, an event named `mfeNavigate` is sent in the first micro-frontend, with the routing to the second (Product catalog to the Checkout in this case - listing 5.13)

```
goToCheckout() {
    const event = new CustomEvent('mfeNavigate', {
      detail: '/checkout/shopping-cart'
    });
    window.dispatchEvent(event);
  }
```

Listing 5.13: Micro-frontend redirection through events - Product Catalog micro-frontend (TypeScript).

In the Shell, the main `AppComponent` initializes a listener to this event, and whenever that is triggered, the Shell will perform the routing operation (listing 5.14). On the destruction of the component that event listener is removed.

```
ngOnInit(): void {
  window.addEventListener('mfeNavigate', this.handleNavigationEvent.bind
  (this) as EventListener);
}

handleNavigationEvent(event: CustomEvent): void {
  const url = event.detail;
  this.router.navigateByUrl(url);
}

ngOnDestroy(): void {
  window.removeEventListener('mfeNavigate', this.handleNavigationEvent.
  bind(this) as EventListener);
}
```

Listing 5.14: Shell listener and navigation (TypeScript).

5.1.7 Web API

The Product Catalog Web API is a backend service created to support the server layer of the micro-frontend architecture for the Vinci e-commerce platform. It is a minimal API, developed using .NET 8.0 following Microsoft's documentation (learn.microsoft 2024) and it possesses all the service logic to account for the e-commerce use cases. A minimal API allowed for brevity in code, minimal configuration and reduced overhead when creating a service (learn.microsoft 2024), which aligned with this PoC's scope.

The application layer of the Web API was responsible for the endpoints definition and the service logic. The endpoints that were created are as follows:

- Account endpoints:
  - **POST /account/login** - performs the login of an already registered user, and returns the valid token;
  - **POST /account** - registers a user;
- Brand endpoint:
  - **GET /brands** - retrieves all brands present in the database, to support Product Catalog's product listing page filter by brand feature;
- Category endpoints:
  - **GET /categories** - retrieves all categories, to display them in Product Catalog's product listing page;
  - **GET /categories/trees** - retrieves the category tree (parent categories grouped with their children);
  - **GET /categories/{categoryId}/tree** - retrieves the category tree of a specific category by id;
- Order endpoints:
  - **POST /orders** - creates an order using the user information and products they have added to their shopping cart (Checkout shopping bag page);

  - **GET /orders/{orderId}** - retrieves the order object with the products information (Checkout order page);
  - **PATCH /orders/{orderId}/status** - updates the order status (Checkout order page);
  - **POST /orders/{orderId}/payment** - performs payment operation and updates the order status to completed (Checkout order completed page);

- Product endpoints:
  - **GET /products** - paginated search endpoint for products, able to be filtered by category id, brand id, variant id, minimum price and maximum price. Page number and page size can also be set. It is used in the Product Catalog listing page;
  - **GET /products/{productId}** - retrieves product by product id, used in the Product Catalog details page;

- Shopping Cart endpoints:
  - **GET /shoppingcart** - retrieves the shopping cart information of the user by the user id (Checkout shopping cart page);
  - **PATCH /shoppingcart** - updates product information (quantity, variant, etc.) on the shopping cart (Checkout shopping cart page);
  - **POST /shoppingcart** - adds a product to the shopping cart (Product Catalog details page for the add to cart button);
  - **DELETE /shoppingcart** - removes a product from the shopping cart (Checkout shopping cart page);

- Variant endpoint:
  - **GET /variants** - retrieves all variants, for the filter by variant feature on Product Catalog listing page.

Entity Framework Core (learn.microsoft 2021) was used for the integration of the MySQL database with the API. A Repository layer was created on the project, where Entity Framework's fluent API (Svyryd 2023) was used to configure the database model through C# code, due to its practicality in this scope. Builders were used to define the tables, keys and properties of each database model. Scripts were generated and updated in the database using migrations (learn.microsoft 2023) that ran using a couple commands:

```
dotnet ef migrations add InitialScript

dotnet ef database update
```

On the application start up, the scripts are executed and the database schema will always stay up to date with what is configured in the API.

Model data seeding also followed the same strategy (Svyryd 2022) - the data was added using `EntityTypeConfiguration` classes and the `HasData` method (Svyryd 2023).

For the authentication process, JSON Web Token (JWT) configuration was used. In the register and login endpoints, the user passwords were stored in the database by hashing

the password using a cryptographic algorithm (in this case SHA-256) before saving it. The hashed password is then compared against the hashed version of the provided password during login, ensuring that the plain text passwords are never in risk of exposure.

When a user attempts to log in, the application verifies the password and, if correct, generates a JWT token using the user's email. This token serves as a secure way of authenticating the user in subsequent requests. The API is configured to require this token for accessing specific endpoints (for example the shopping cart and order endpoints, since a user must be logged in to perform these requests). If a request to a protected endpoint is made without a valid JWT token, the API responds with a 401 Unauthorized status.

The API was containerized and a Docker file was created for a better deploy strategy and future scalability. Since it has a direct dependency on MySQL, the configurations are set by environment variables set through the `launchSettings.json` file.

A swagger UI (fig. 5.11) was also configured, to serve as documentation.

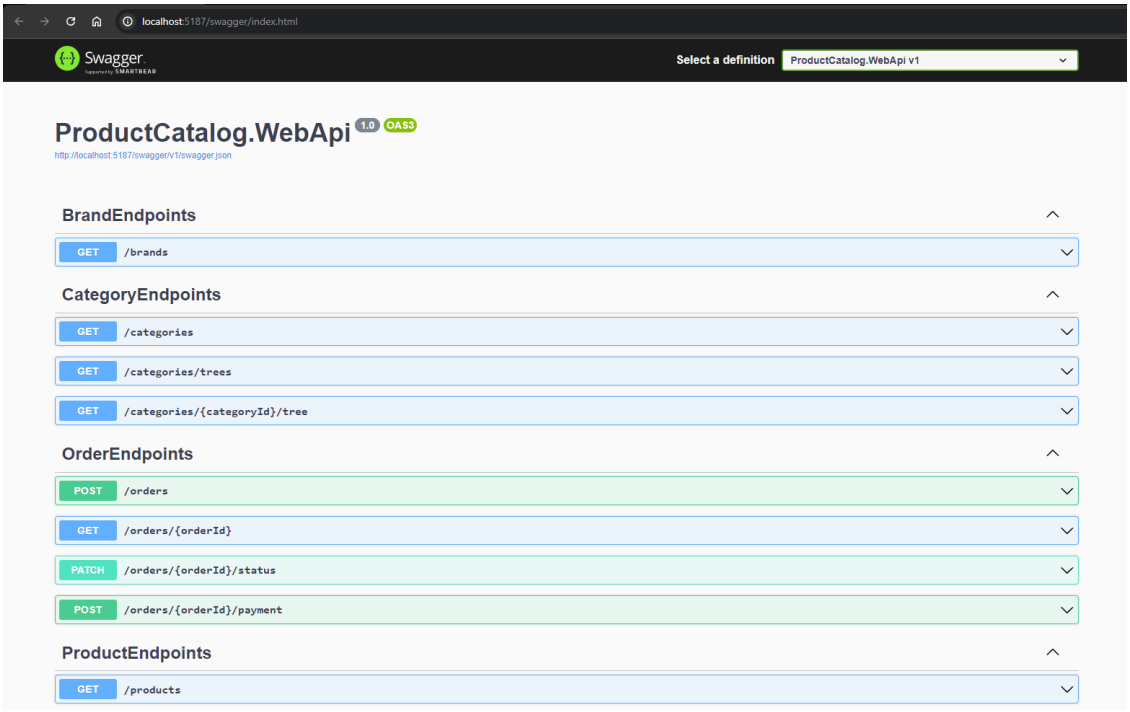


Figure 5.11: Product Catalog Web API - Swagger UI with the brand, category, order and product endpoints.

Database

A relational database was created using a MySQL docker image. MySQL Workbench (Oracle 2024) was the tool used to view the data schema locally, and rendering diagrams such as the model in fig. 4.3. Since the model accommodated for a relational database, MySQL was the chosen technology due to the author's professional experience in its use, its scalability and versatility for the use case of this PoC.

5.2 Documentation

This section will go over the documentation process for the Components Library.

As stated in the Subsection 2.3.2 Design System – Component Libraries, the documentation allows for a better understanding of how to implement the components of the design system into an Angular project. It must guarantee that visual examples, code samples and instructions of use are provided (Sukoinen 2020).

The framework chosen to develop this documentation was Storybook since it is a simple tool to configure and it possesses a wide range of features (Storybook 2023).

5.2.1 Storybook Installation

Storybook is an open source framework that provides a way to create demos (stories), test inputs, and outputs of components, without the need of an external project in which to integrate the library. It "is a standalone tool that runs alongside (an) app. It's a zero-config environment that works with any modern frontend framework" (Storybook 2023). It allows for the representation of different component states by modifying arguments, and for the inclusion of markdown documents (Storybook 2023).

For the installation of Storybook into the `ngx-dissertation-components-lib` project, `npx` was used. The project was automatically set up with the correct configurations and example files to implement the documentation of the library. In this case, stories and docs will be applied to the project.

5.2.2 Stories

Stories are a type of file that Storybook uses to bind the components of the project (in this case the Components Library) to its documents. In short, a story is a snapshot of the rendered state of a UI component, that possesses annotations and descriptions of a component and its behaviour and appearance correspondent to a set of arguments – this is a generic term for Angular components `@Input` properties (Storybook 2024e).

Stories are located at the root level of the project, inside a stories folder. They are defined using "the Component Story Format (CSF), an ECMAScript 6 (ES6) module-based standard that is easy to write and portable between tools" (Storybook 2024e).

In the story file, a default export that describes the component is defined, and other named exports with different behaviours of the component are also exported. In the next code samples for the `button.stories.ts` file (listing 5.15), the button component is used as a reference, even though this was done for every single component in the library

The `ButtonComponent` is imported, as are all the necessary modules to run the component, and the argument types for each input are defined so they are available on the Storybook page to configure. Stories are then exported with different names (for example Default (listing 5.15 - line 3), WithIcon (listing 5.15 - line 12), Disabled (listing 5.15 - line 22)) and these appear as options on the Storybook page visible to the user to interact with (view lateral options on the left, under Button Docs in fig. 5.12).

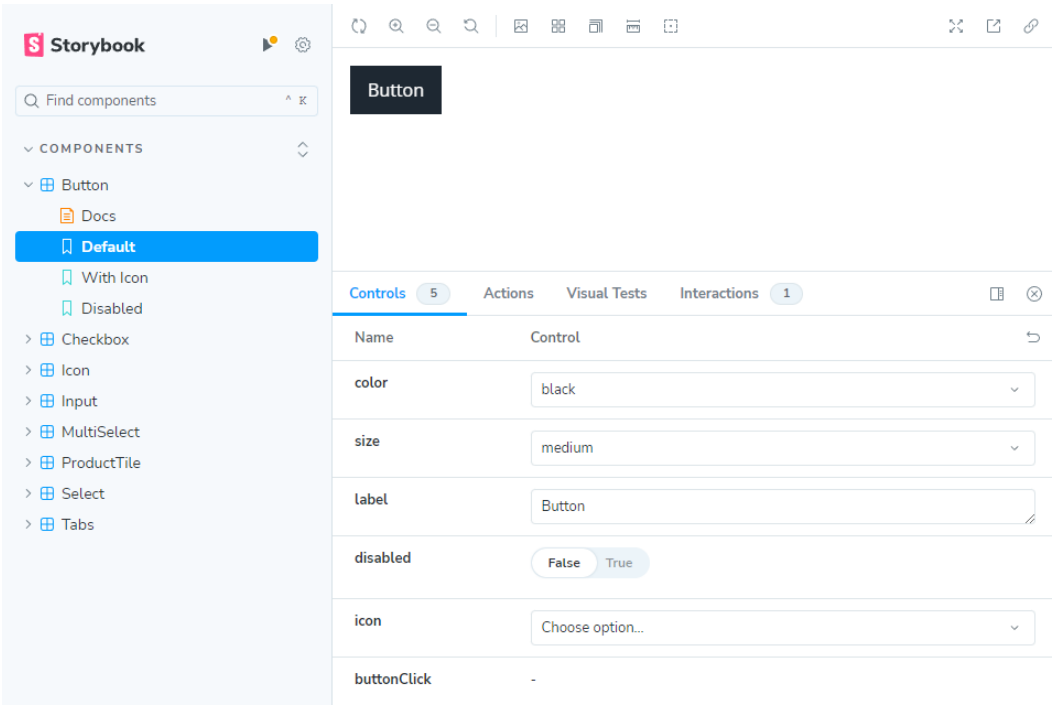


Figure 5.12: Storybook button component stories with configurable properties.

```
type Story = StoryObj<ButtonComponent>;

export const Default: Story = {
  args: {
    color: ButtonColor.BLACK,
    size: ButtonSize.MEDIUM,
    label: 'Button',
    disabled: false,
  }
};

export const WithIcon: Story = {
  args: {
    color: ButtonColor.CORAL,
    size: ButtonSize.LARGE,
    label: 'Button',
    icon: ButtonIcon.HEART,
    disabled: false,
  }
};

export const Disabled: Story = {
  args: {
    color: ButtonColor.WHITE,
    size: ButtonSize.SMALL,
    label: 'Button',
    disabled: true,
  }
};
```

Listing 5.15: Button stories documentation file (TypeScript).

5.2.3 Docs

Markdown/JSX (MDX) files can be used to enhance the documentation on Storybook, creating what is called docs. "MDX files mix Markdown and Javascript/JSX to create rich interactive documentation" (Storybook 2024c).

For the button component, a `button.mdx` file was created and placed on the same folder as the stories file (listing 5.16).

```
1  import { Canvas, Meta } from '@storybook/blocks';
2
3  import * as ButtonStories from './button.stories';
4
5  <Meta of={ButtonStories} />
6
7  # Button
8
9  The `ButtonComponent` is a customizable button element with different
       colors, sizes, and icons. It also supports a disabled state.
10
11 <Canvas of={ButtonStories.Default} />
12
13 ## Props
14
15 - `color`: The color of the button. Options are available through the
       ButtonColor enum: `BLACK`, `WHITE`, `CORAL`.
16 - `size`: The size of the button. Options are available through the
       ButtonSize enum: `SMALL`, `MEDIUM`, `LARGE`.
17 - `icon`: The icon to display inside the button.
18 - `label`: The text label of the button.
19 - `disabled`: Boolean to disable the button.
20
21 ## Usage
22
23 ```html
24 <isep-lib-button
25   [color]="ButtonColor.BLACK"
26   [size]="ButtonSize.MEDIUM"
27   [label]="'Button'"
28   [disabled]="false"
29 ></isep-lib-button>
30 ```
```

Listing 5.16: Excerpt of the button MDX documentation file (Markdown/JSX).

The MDX file has references to the stories file, through the Meta (line 5) and Canvas (line 11) blocks.

The Meta block is used to attach the custom MDX docs page alongside a component's list of stories (Storybook 2024d).

The Canvas block is a wrapper around a Story, featuring a toolbar that allows interaction with its content while automatically providing snippets of the source code (Storybook 2024a).

This will resolve in a "Docs" page available on the side bar, displaying the auto-generated documentation for components (fig. 5.13).

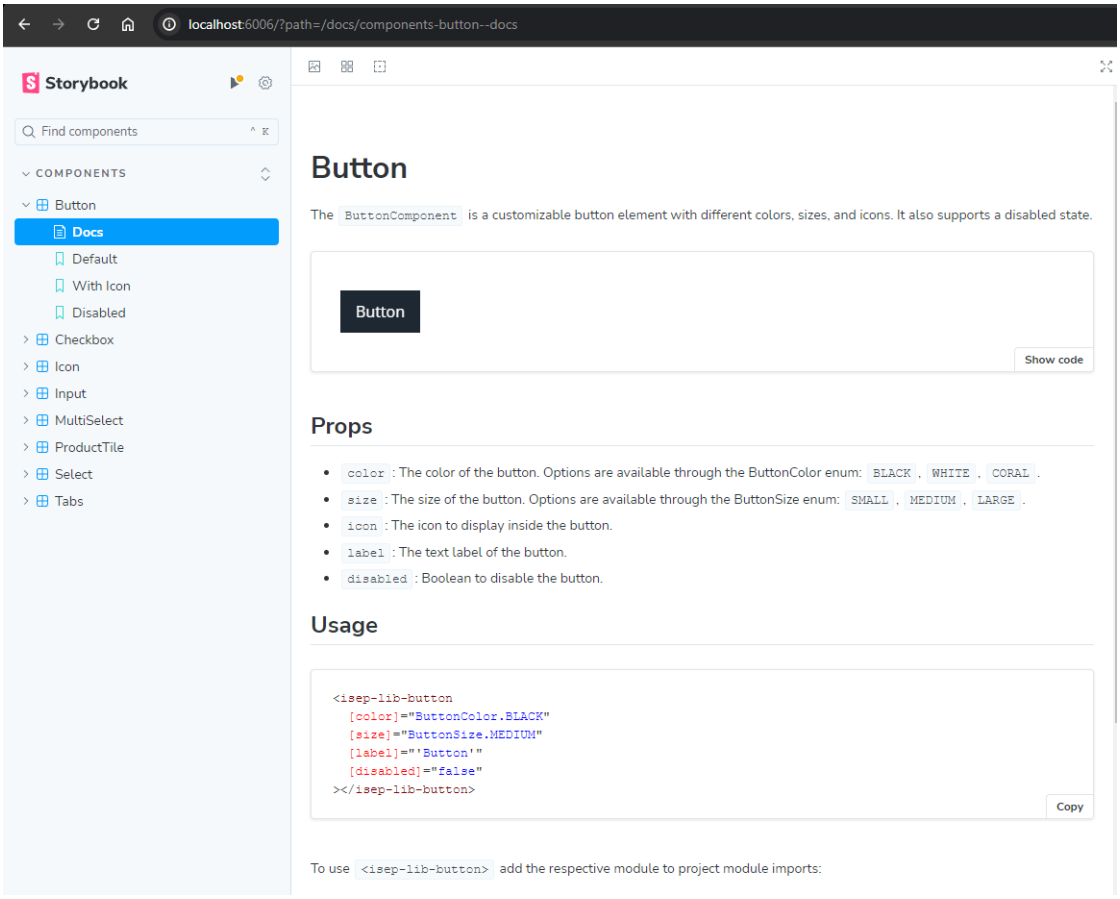


Figure 5.13: Storybook button component documentation.

5.3 Testing

The testing section will present the automated tests that were developed to assure the quality of the software of each application, including the Components Library. Component testing, unit tests and acceptance tests were implemented according to each

5.3.1 Components Library

For the Components Library project, Storybook's interaction tests (Storybook 2024b) were used for component testing.

Components don't just have the task of rendering the UI, they also possess behaviour, like fetching data and managing states (Storybook 2024b). These functional aspects can be tested on the components individually, to visualize errors on the Components Library before a new (broken) version is available to upgrade on the micro-frontend side.

Interaction tests are one of the ways functional aspects of UIs may be tested. By knowing the initial state of the component through stories, mentioned in the previous section, these tests can "simulate user behaviour such as clicks and form entries and check whether the UI and component state update correctly" through the browser (Storybook 2024b).

On the Story files the interaction tests can be setup inside the `play` function, alongside the creation of a story. In the following code snippet (listing 5.17) a story for the multi select component is created, and an interaction test is laid out.

```
1  const exampleCheckedOptions: MultiSelectModel[] = [
2    { id: 1, name: "Option 1", isSelected: true },
3    { id: 2, name: "Option 2", isSelected: false },
4    { id: 3, name: "Option 3", isSelected: true },
5  ];
6
7  export const PreSelectedOptions: Story = {
8    args: {
9      title: "Select Options",
10     options: exampleCheckedOptions,
11     errorLabel: false,
12   },
13   play: async ({ canvasElement }) => {
14     const canvas = within(canvasElement);
15
16     expect(canvas.getByText("Select Options")).toBeVisible();
17     fireEvent.click(canvas.getByText("Select Options"));
18
19     for (const option of exampleCheckedOptions) {
20       const optionElement = canvas.getByText(option.name);
21       expect(optionElement).toBeVisible();
22       if (option.isSelected) {
23         expect(
24           optionElement.parentElement?.querySelector('input[type="checkbox
     "]')
25         ).toBeChecked();
26       } else {
27         expect(
28           optionElement.parentElement?.querySelector('input[type="checkbox
     "]')
29         ).not.toBeChecked();
30       }
31     }
32   },
33 };
```

Listing 5.17: Excerpt of the multi select component stories file with interaction tests (TypeScript).

There are two goals for this test, firstly, making sure that all expected elements appear in the component, by clicking on the multi select component and exposing the list of options, and verifying that they are visible (listing 5.17 - lines 17 and 21). Secondly, asserting that the options that have a state as checked are in fact checked, and the ones that do not, are not checked (listing 5.17 - lines 22-29).

In the Storybook page, these tests can be viewed for each individual story (fig. 5.14).

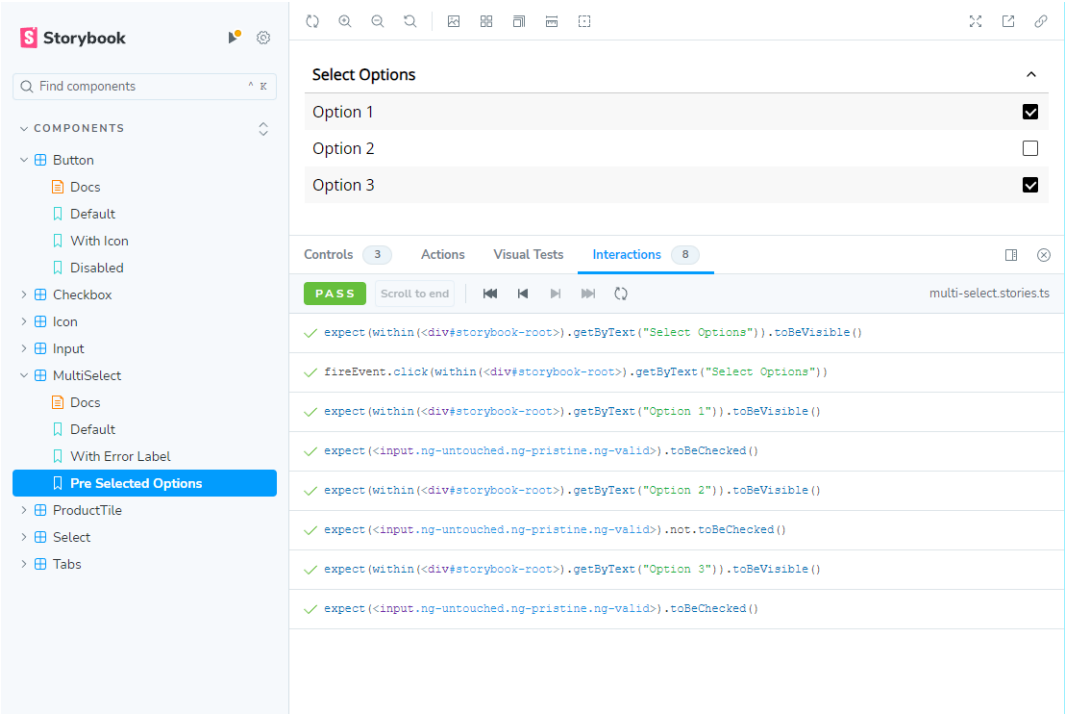


Figure 5.14: Storybook interaction tests run for the multi select component.

5.3.2 Micro-frontends - Unit tests

Unit testing allows the quality assurance of the smallest unit of work inside a project, guaranteeing that a piece of code works as expected (Osherove 2013, p. 21). A proper unit test should be automated, easy to implement, consistent, relevant and fully isolated and independent from other tests (Osherove 2013, p. 22).

For Javascript based projects, Jasmine can be used as a unit testing framework, including Angular applications (JasmineDocs 2024). In order for Jasmine tests to run, Karma - a test runner that provides code coverage reporting, and integration with Continuous Integration (CI)/Continuous Delivery (CD) tools (Angular.dev 2024) - needs to be configured in the project. For that, a configuration file named `karma.conf.js` is created at the root of the project, containing the information regarding frameworks (in this case Jasmine), coverage reports, plugins, etc. Another file, created in the src folder, called `test.ts` is added, since it is required by `karma.conf.js` and loads recursively all the spec and framework files present in the project.

Jasmine tests can then be run using the `ng test` command that has been configured in `angular.json`. The tests can also be run through the code editor, in this case Visual Studio Code, by installing the Karma Test Explorer (for Angular, Jasmine, and Mocha) extension (VisualStudioMarketplace 2024) (fig. 5.15).

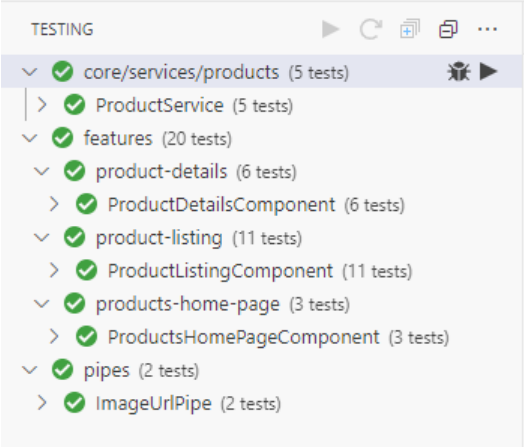


Figure 5.15: Karma Test Explorer tests - Product catalog micro-frontend example.

The unit tests were created for the services, components and the pipes of each micro-frontend. Jasmine allows for isolated unit tests since it provides a way to mock dependencies using spy objects, making use of Angular's dependency injection strategy with `TestBed`, the module to test can be injected with the mocked dependencies for the test cases. For this, the providers metadata property is injected with an array of the mocked services intended to allow for each test scenario (Angular.dev 2024). This way, the layers are isolated and can be tested individually.

5.3.3 Micro-frontends - Acceptance tests

Acceptance tests (or functional tests) are tests following requirements of business, usually provided by a client or a product representative, to determine if a system is working as expected (Miller and Collins 2001). They are the basis of Behavior-Driven Development (BDD), frequently used in microservices development, in which each microservice is responsible for their own acceptance tests (Rahman and Gao 2015) - therefore, in micro-frontends, each project should also be responsible for their own acceptance tests. These tests should be automated and run frequently, as to make sure the software quality and requirements are maintained.

When it comes to web applications based on Javascript, Cypress is a framework that can be used to develop acceptance tests. It allows for End-to-end (E2E) testing the applications directly in the browser, simulating user behaviour (docs.cypress.io 2024a). Playwright would also be a great option, however, according to Szahidewicz 2024, Playwright should be used when cross-browser testing, complex web features and performance-critical testing are the focus. Cypress is used mainly for JavaScript centric projects, real-time testing needs and simpler testing scenarios, which applied to this PoC.

A couple Cypress acceptance tests were developed in the Product Catalog project, as a demonstration for how acceptance tests should be used in a micro-frontend architecture.

Cypress can be configured in Angular projects by running the following commands on the root of the project:

```
npm install cypress --save-dev

npx cypress open
```

The first one will install the `cypress` package on the dev dependencies of the project, and the second one will open up the Cypress UI and allow for a project setup. After choosing E2E testing, a configuration file called `cypress.config.ts` will be created, where the `baseUrl` can be set with the same port as the local application, so that the tests are run on the correct URL.

A folder called `cypress` is also created on the root of the project, where the tests, fixtures and auxiliary commands will be implemented.

External dependencies such as service calls can be intercepted and mocked, so that the E2E tests are not affected by outside services, using the fixture files. This way the web application is completed isolated and different scenarios can be easily mocked (docs.cypress.io 2024b). The mocks of the service responses are JSON files, located inside the fixtures folder. Through the fixture function, the requests to the service that are done on the page load can be intercepted and mocked with the response JSON file, running before each test inside the test case (listing 5.18).

```
1  beforeEach(() => {
2    // Load the mock response from the JSON fixture file
3    cy.fixture('search_products.json').then((mockResponse) => {
4      // Intercept the request and return the mock response
5      cy.intercept('GET', 'http://localhost:5187/products?page=1&pageSize
     =4', {
6        statusCode: 200,
7        body: mockResponse
8      }).as('getProducts');
9    });
10 });
```

Listing 5.18: Cypress fixture request intercept for the search products endpoint (Typescript).

This request intercept is given an alias (`getProducts` in this case), that can be awaited on the test run, to make sure the request is correctly made before performing the asserts on the page. On the following test example (listing 5.19), asserts are performed to guarantee that the elements exist on the page. The elements can be retrieved by the id (line 10), by their class name (using a dot as a prefix on the class name - line 11 and 12) and by their selector (line 13). In this case, the test assures that six categories are loaded on the home page, the button name text and the section title text are correct, and four product tiles are displayed.

```
it('has all element of the page', () => {
    // Visit the page that triggers the request
    cy.visit('/products');

    // Wait for the intercepted request to complete
    cy.wait('@getProducts');
    cy.wait('@getCategories');

    // Assertions
    cy.get('[id="category"]').should('have.length', 6);
    cy.get('.button-container-span').first().should('have.text', 'Shop Now
    ');
    cy.get('.best-seller-text').should('have.text', 'Best Sellers');
    cy.get('isep-lib-product-tile').should('have.length', 4);
  });
```

Listing 5.19: Cypress scenario asserting the home page elements - Product catalog micro-frontend (Typescript).

Acceptance tests often involve replicating user actions, such as clicking and scrolling on a web page, and asserting the results of that action. In the following example (listing 5.20), the clicking of a button (line 17) on the home page triggers the navigation towards a different page (products listing page - as seen on line 20). In this new page, a request to the brands endpoint of the service is performed, therefore, a new intercept must be made and awaited before performing the assertions.

```
it('changes page', () => {
    cy.fixture('brands.json').then((mockResponse) => {
      // Intercept the request and return the mock response
      cy.intercept('GET', 'http://localhost:5187/brands', {
        statusCode: 200,
        body: mockResponse
      }).as('getBrands');
    });

    // Visit the page that triggers the request
    cy.visit('/products');

    // Wait for the intercepted request to complete
    cy.wait('@getProducts');
    cy.wait('@getCategories');

    cy.get('.button-container-span').first().click();
    cy.wait('@getBrands');

    cy.url().should('include', '/products/listing');
  });
```

Listing 5.20: Cypress scenario asserting navigation between pages - Product catalog micro-frontend (Typescript).

In the Cypress UI, the tests can be viewed running and updating as changes are made to the files (fig. 5.16).

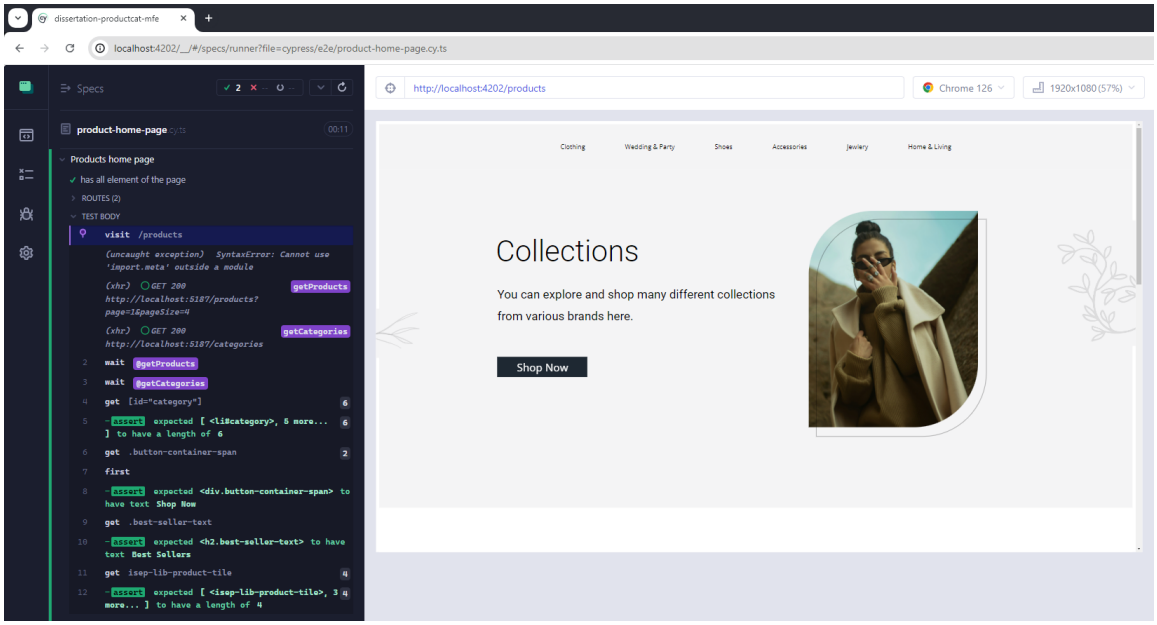


Figure 5.16: Cypress test - Product catalog micro-frontend home page scenarios.

5.4 Deployment

This section will go over the deployment of the static web applications for the micro-frontend architecture, including the documentation that supports the design system of the Components Library. It will also go over the deployment of the Web API and the database production setup.

CI/CD practices were taken into account, using Github's actions to configure pipelines and deploy the Components Library.

5.4.1 Components Library Documentation

In order to proceed with the deployment of the documentation, so that it is available for public viewing, the Storybook portion of the project must be exported as a static web application. This is a feature already enabled by Storybook, by running the following command:

```
npm run build-storybook
```

This command will "output a static Storybook in the storybook-static directory, which can then be deployed to any static site hosting service" (Storybook 2024f). For publishing the storybook documentation, Chromatic – a free publishing service created by the Storybook team – was used (Storybook 2024f). The process began by installing the chromatic package in the project by using *npm*.

Then, since the project was already available on Github, Chromatic could automatically deploy the storybook documentation through the Github repository link, associating it with a project token.

Afterwards, the documentation is available at an URL provided by Chromatic and the published Storybook is ready to be shared, available in Appendix C (fig. 5.17).

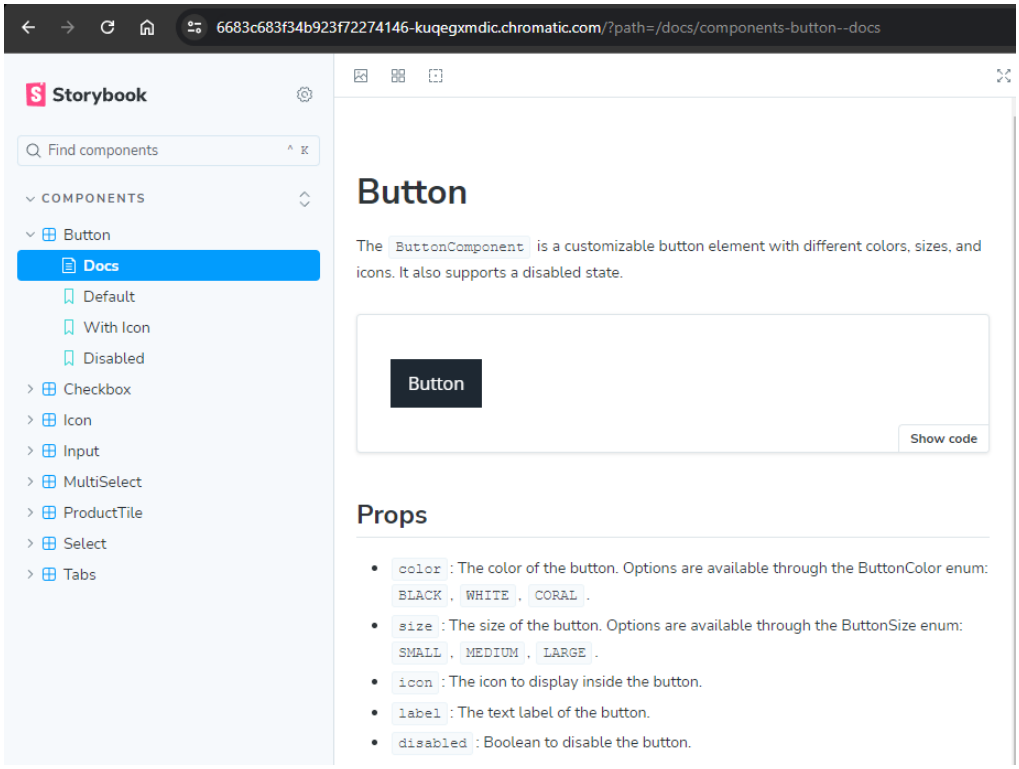


Figure 5.17: Final documentation deployed via Chromatic.

Chromatic also allows for a view of each deployment build and the run of the tests (fig. 5.18) as well as a view of the library components available on the documentation (fig. 5.19).

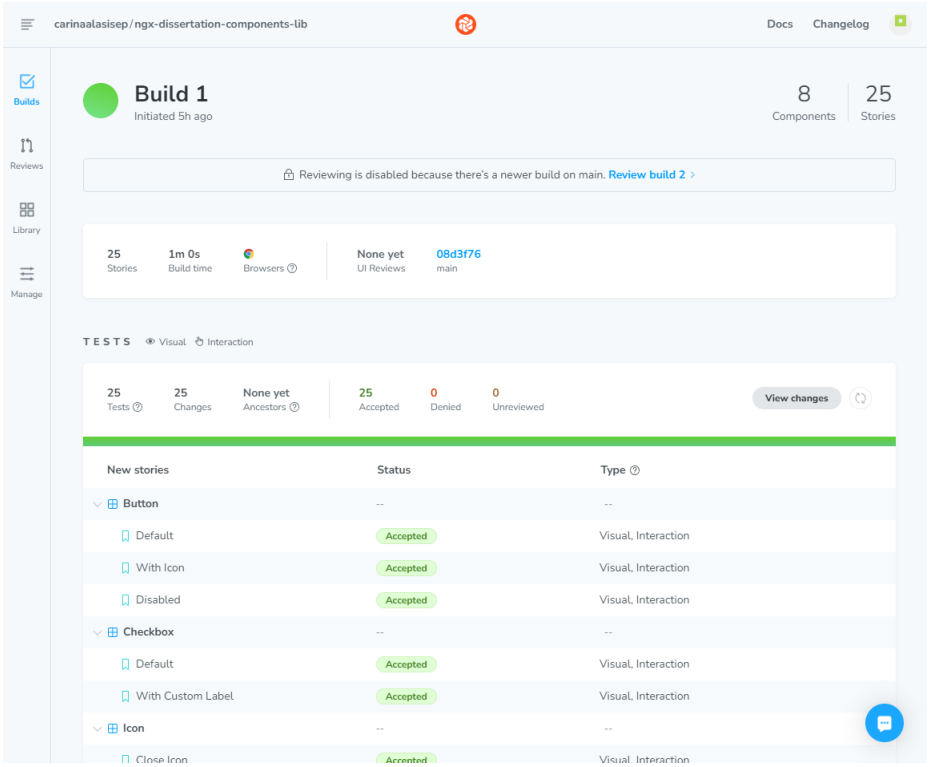


Figure 5.18: Chromatic view of the pipeline build and test run.

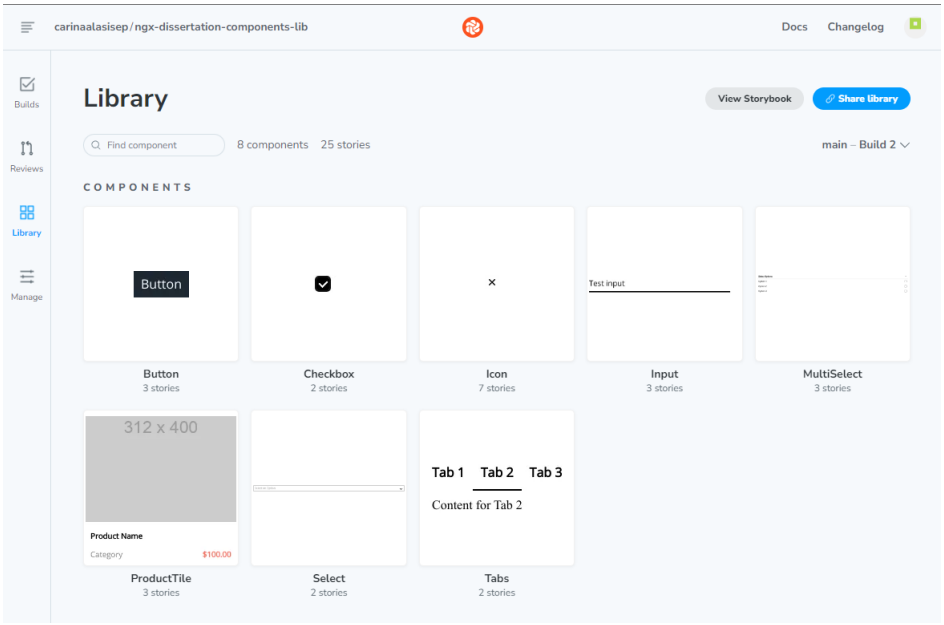


Figure 5.19: Chromatic view of the components right after build run.

A CI/CD strategy for the deployment of the Storybook documentation was created, updating it at each push in the main branch, using Github Actions (Chromatic 2024).

For that, a pipeline configuration YAML file was created in the `.github/workflows` folder. Since sensible information such as the project token needs to be available in the YAML file, Github secrets (fig. 5.20) were used and configured in the repository (GitHub 2024).

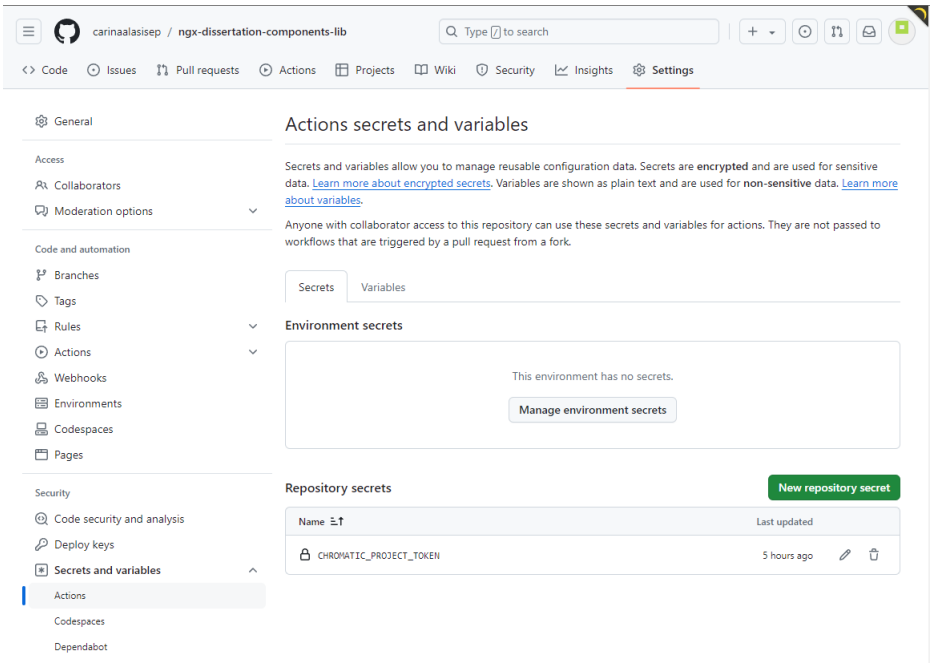


Figure 5.20: Github actions secrets configuration.

The GitHub Actions pipeline job (fig. 5.21) automates the process of running visual regression tests using Chromatic, and the deployment of Storybook documentation into Chromatic. The workflow is triggered by a push event, meaning each time a commit is pushed on the main branch on the repository, it will trigger a pipeline build.

The pipeline consists of several steps: it checks out the repository's code, sets up a Node.js environment, installs project dependencies, and finally runs Chromatic using a secure project token stored in GitHub Secrets (fig. 5.20), deploying this build to the platform.

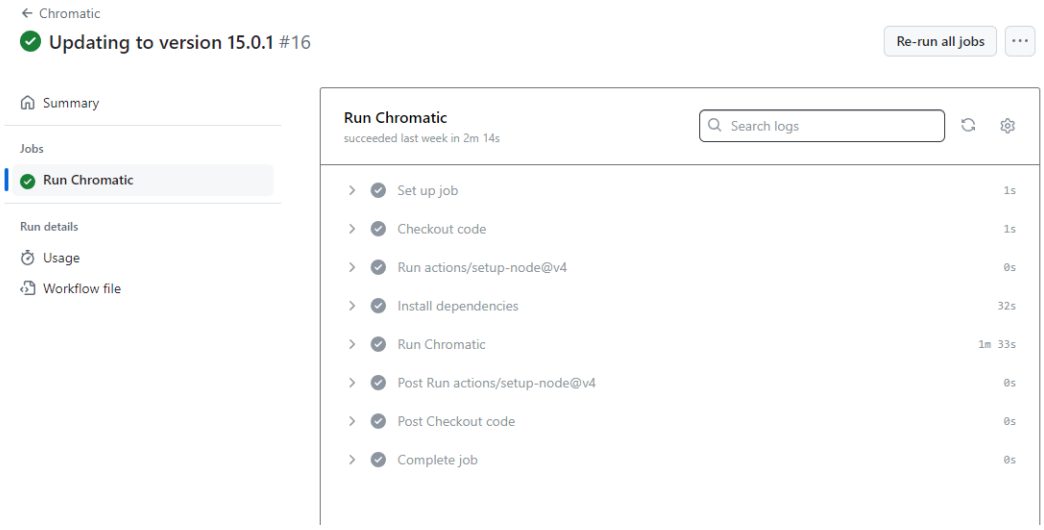


Figure 5.21: Github action - Chromatic pipeline job steps.

If the GitHub build is successful, in Chromatic another build will be triggered and run with all the interaction tests (fig. 5.22 - each with a run similar to fig. 5.18). The build will also be notified via e-mail to the author of the commit (fig. 5.23).

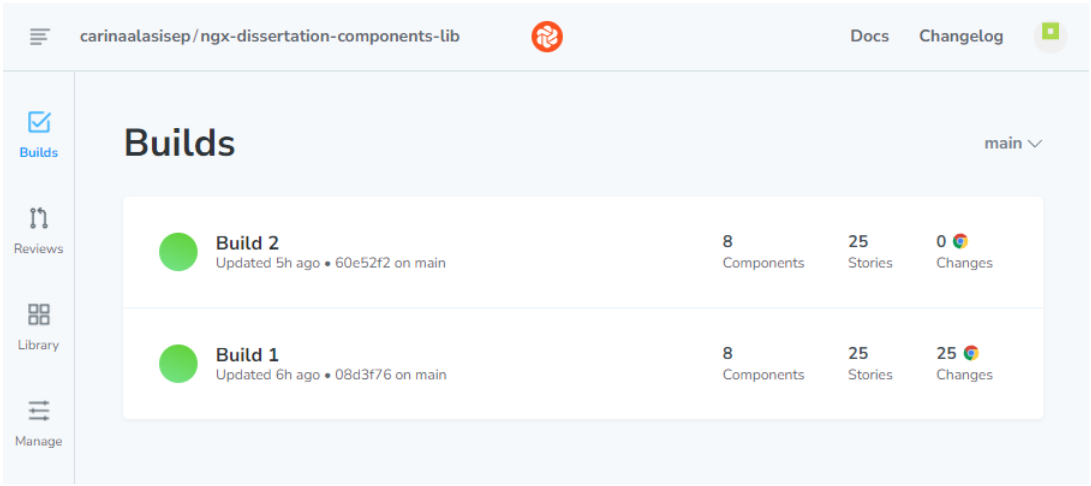


Figure 5.22: Chromatic pipelines triggered by Github actions.

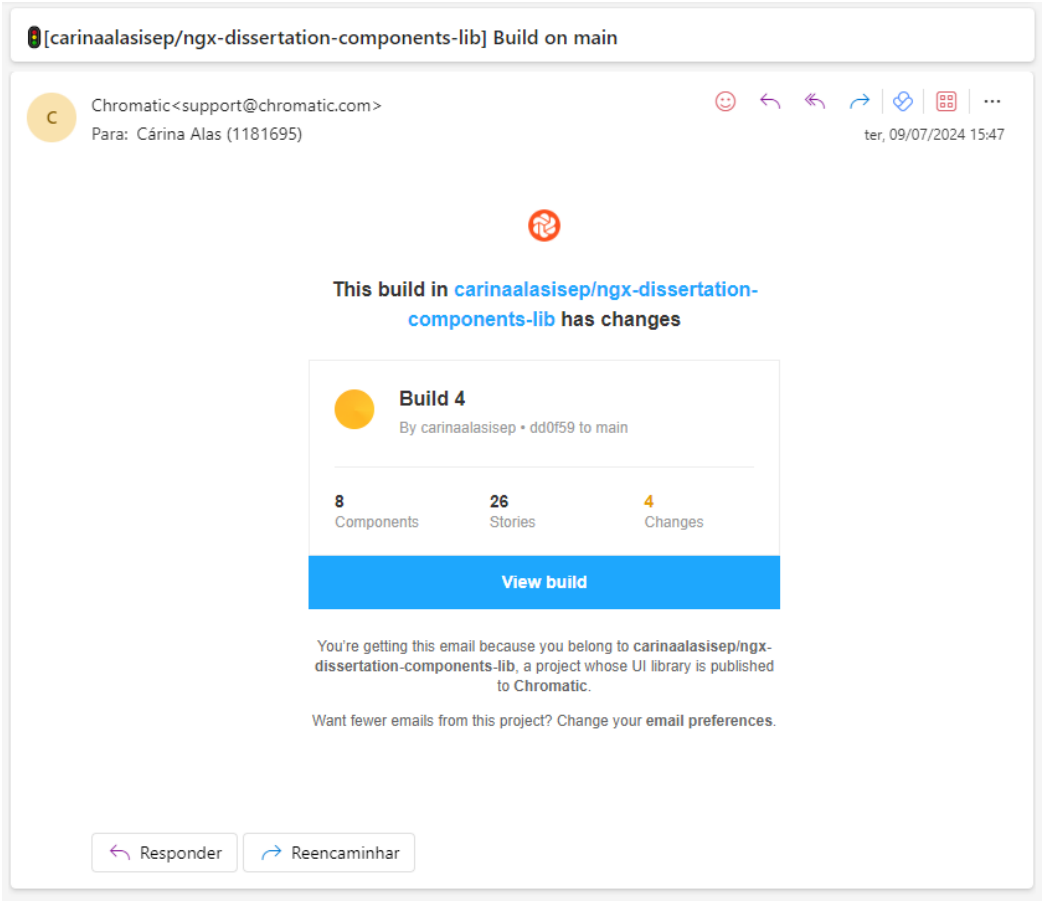


Figure 5.23: Chromatic pipeline build e-mail notification after push.

5.4.2 Components Library NPM Publish

The automation of the publishing of the `ngx-isep-dissertation-components` library to npm was also done using Github Actions (Chromatic 2024).

A new workflow named "Publish Angular Package" was created, with the goal of automatizing the publishing of the npm package whenever a new version is updated and sent to the main branch. This pipeline job (fig. 5.24) is only triggered after the previous workflow - Chromatic deploy - has run successfully. This setup ensures that visual regression tests, handled by the Chromatic workflow, pass before attempting to publish the package, adding a layer of quality control.

The commands that were stated in the previous Subsection 5.1.1 - Npm Publish, are then automated and run - the first steps being the installation of Angular CLI, npm dependencies and building the Angular project.

Afterwards, a new step checks if the package version already exists, by comparing the version of the library currently set in the `package.json` file and the version existent in the npm registry under that same library name. If the version is already in npm, then the package won't be published. If the version is newer, it will pack and publish the package, using an npm auth token that has been set using Github Secrets (GitHub 2024).

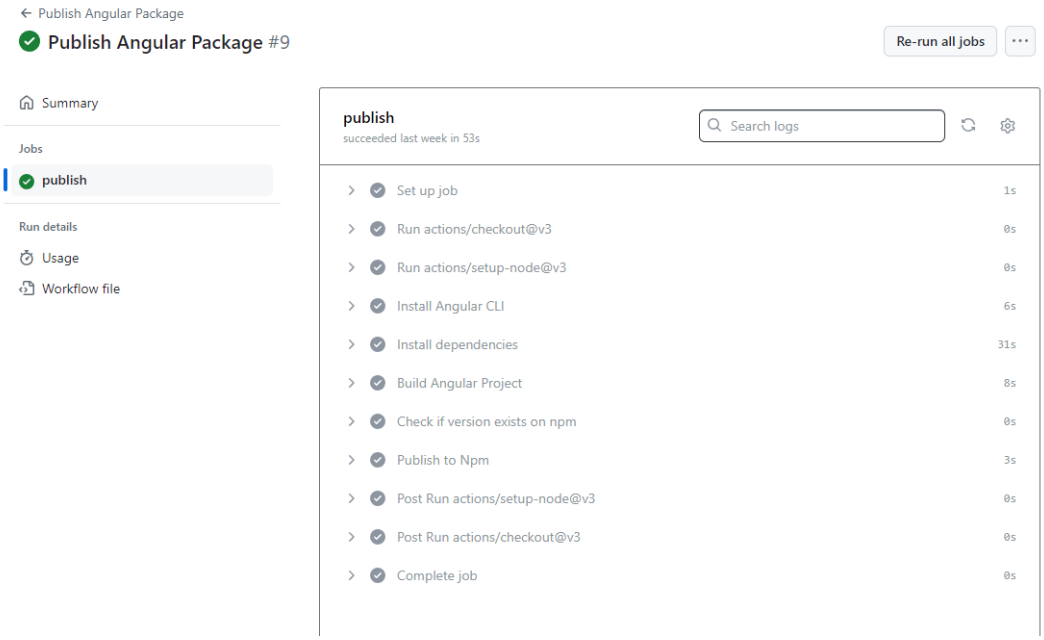


Figure 5.24: Github action - Publish npm pipeline job steps.

5.4.3 Web API

Since the Web API was containerized, the application could be easily deployable through Amazon Web Services (AWS) Elastic Container Service (ECS). ECS allows the deployment of applications by pushing container images, without installing or scaling infrastructure (docs.aws.amazon.com 2024d). In the first quarter of 2024, AWS was the main vendor in the cloud infrastructure market (Vailshery 2024) making it a reliable source. These were the reasons why AWS was chosen as the cloud platform for the deployment of the Web API.

Firstly, an AWS account was setup, and an Identity and Access Management (IAM) user was created, so that only that user can access the container repositories and images in Elastic Container Registry (ECR) (docs.aws.amazon.com 2024a). Then, an Amazon ECR (docs.aws.amazon.com 2024c) repository was created, and after locally logging in to the IAM user account, the docker image of the Web API was built, named "latest" and pushed to the repository using the IAM credentials.

Afterwards, a new cluster named `product-catalog-api-cluster` and two services inside that cluster, named `product-catalog-api-service` (made with the image of the Web API in the ECR repository) and `product-catalog-db-service` (made with a public image of MySQL), were created in ECS. The services were set as Fargate services - this type of service is a simple serverless compute engine that works with just the application containers. The other option would be to use Elastic Compute Cloud (EC2), however that would entail creating a virtual machine, which was not necessary for this PoC.

Inside each service, task definitions were set up, referencing the Web API and MySQL images present in the ECR repository. These tasks definitions include environment variables configurations, port mappings, Central Processing Unit (CPU) and memory allocations, etc. The Web API data base server variable needed to point to the private Internet Protocol (IP) of the MySQL task definition, so that the two services could be connected.

The infrastructure is set inside an AWS Virtual Private Cloud (VPC). A load balancer responsible for managing traffic to the application was set up. To allow the `product-catalog-api-service` to be available in the internet through HTTP, a security group was associated with the load balancer, with rules enabling all inbound traffic on the load balancer listener port (80), being the only public entry point to this cloud deployment. The load balancer has a target group associated, that routes the requests to individual registered targets, that correspond, in this case, to the individual task instances of the `product-catalog-api-service` (docs.aws.amazon.com 2024b). A different security group was created for the services, enabling only traffic from the load balancer and between the internal services. This way, if there was a need to scale up the API service instances, due to load concerns, the load balancer would automatically distribute the traffic to the several instances (through the target group).

For the Domain Name System (DNS) to be reachable through Hypertext Transfer Protocol Secure (HTTPS) requests, that are required by the browser, a domain name had to be registered, in this case by the name of `dissertation-product-cat-api.online`, in the NameCheap platform.

After the DNS records were propagated globally, a new request for a public certificate was made in Amazon Certificate Manager (ACM) and the DNS name had to be added and associated with the newly generated certificate. In the application load balancer, a new listener for HTTPS was added, in port 443, for the same target group. The Secure Sockets Layer (SSL)/Transport Layer Security (TLS) was set with the ACM certificate. The load balancer security group also had to be updated to allow inbound HTTPS requests. In the NameCheap platform, the DNS was configured to redirect the requests from the domain to the public AWS load balancer DNS.

The load balancer is then available through the DNS (URL link available in Appendix C).

This architecture is summarized in fig. 5.25 - the diagram showcases a scenario with two instances of the `product-catalog-api-service`.

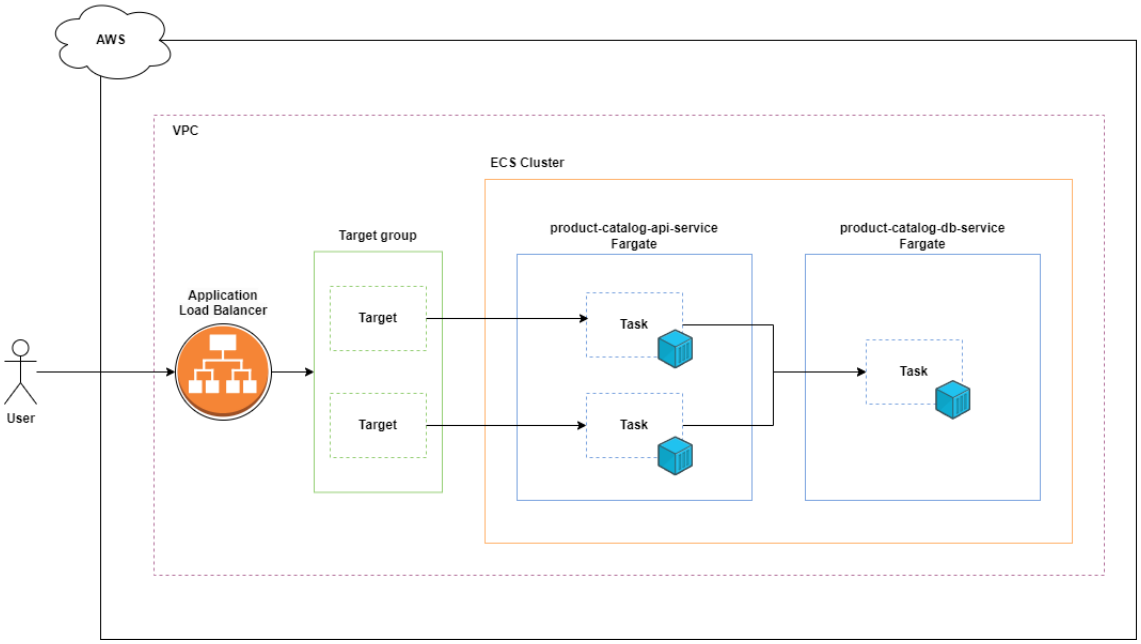


Figure 5.25: Product catalog Web API deployment architecture diagram.

5.4.4 Micro-frontends

For the micro-frontends deployment, Vercel, a serverless cloud platform created by Next.js, was used (vercel.com 2024).

Vercel was chosen by its simplicity and ease of use on deploying Angular applications. After setting up an account and linking it to a GitHub account, developers can deploy their Angular applications simply by selecting the repository they choose to deploy, and by each commit pushed to the main branch, the application will continue to update automatically.

For each micro-frontend project (Product Catalog, Checkout and Account Management) the `environment.prod.ts` file was updated, so that the product catalog Web API URL was set with the correct public domain. Afterwards, each one of the projects were deployed in Vercel, configuring the build command to match the production configurations.

After the three micro-frontends were deployed, the Shell project had to be updated with the production environment variables as well. The `remoteEntry.js` file path had to be changed from localhost to each of the micro-frontend's new domain URL. Also, on the production build configuration in Vercel, the base HREF also had to be set up to be `/app`, in order for the application routing to work as expected in a deployed production environment. The URL for the final deployed web application (fig. 5.26) is accessible in Appendix C.

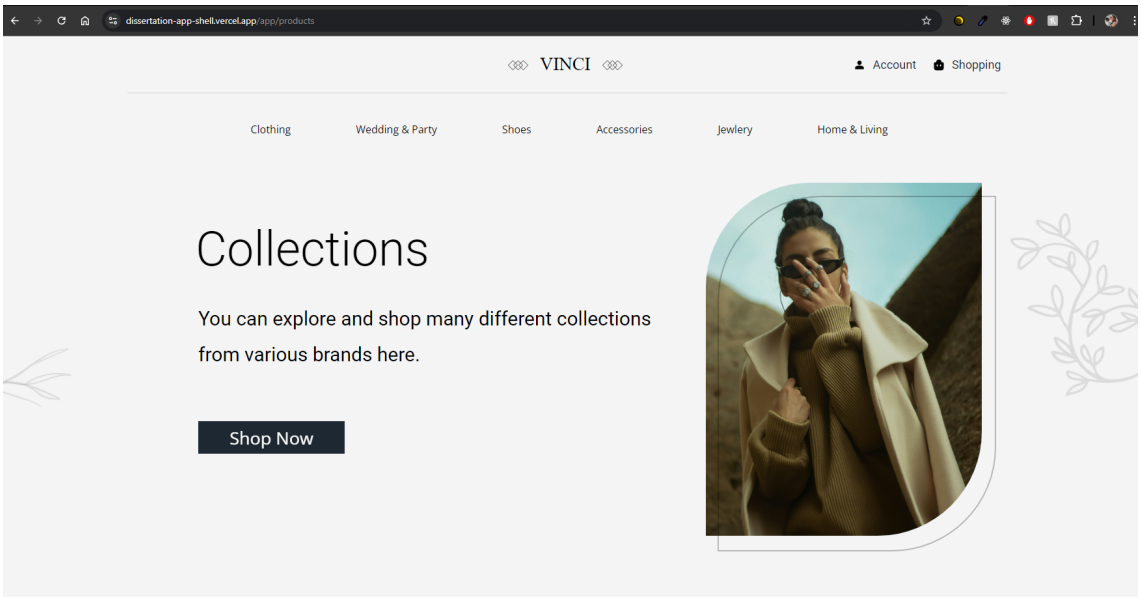


Figure 5.26: Shell application integrated with micro-frontends deployed in production environment using Vercel.

5.5 Summary

In the Solution Implementation chapter, the primary objective was to emulate a real-world SDLC by developing a comprehensive PoC. This PoC was designed to replicate the scenarios and dynamics of a real-life corporate environment, where multiple teams are concurrently developing different micro-frontends. These micro-frontends collaboratively contribute to a cohesive single web application experience. At the center of this implementation was the use of a custom Components Library, which had a goal of ensuring consistency and integration across the various development teams.

The Components Library was used in each micro-frontend implementation, with the objective of creating a seamless UI experience for the user. A web application for an e-commerce platform called Vinci was developed with functional flows, inside a complex architecture ranging from the frontend client layer, to the Web API in the server layer and the database layer.

A thorough documentation for the Components Library was implemented, using Storybook, with examples of usage and customizable properties, to align the implementation methodology and reduce errors and changes within the development in the case of different teams using the library.

The components from the library were tested with interaction testing, also within Storybook, to validate their individual functionality and ensure they met expected behaviour standards. Additionally, the integration of these components into the web application was guaranteed with acceptance tests on the micro-frontend side, verifying that they fulfilled the necessary business requirements.

The whole solution was then deployed and made available publicly, starting with the publishing of the Components Library, that was integrated in a CI/CD solution with the integration of Github Actions, and Chromatic to deploy the documentation and run the tests before the publishing.

The Web API developed for this PoC was deployed using AWS and a containerized approach through ECS. The web application was also available for public viewing using Vercel after the deployment of all the micro-frontend projects.

Chapter 6

Experimentation and Evaluation

This chapter will outline the experimentation and evaluation process undertaken to explore the effectiveness of a components library in addressing the challenges inherent in a micro-frontend architecture.

The developed PoC will be used as a basis for the experimentation and research. The primary goal is to investigate if the developed Components Library can impact UI/UX consistency, code duplication, maintainability, and scalability across the micro-frontends present in the developed solution.

Through a combination of quantitative and qualitative methods, a comprehensive assessment of the Components Library's role in improving the overall architecture will be evaluated.

6.1 Purpose

The purpose of this chapter is centered around answering the three core research questions (RQs), introduced in Chapter 1, that guided the study up until now:

- **Main RQ - RQ1**: How can consistent UI and UX design be maintained in a micro-frontend architecture, particularly when external libraries are discouraged, or the project requires a unique design?
- **RQ2**: How can we reduce code duplication, enhance the development process, and improve the maintainability and scalability of applications within a micro-frontend architecture?
- **RQ3**: How can a components library package be effectively integrated and deployed to promote reusable components, reduce code duplication, and ensure consistency and compatibility across micro-frontends?

These questions aim to assess the Components Library's ability to address the above challenges, and the objective is to ultimately determine its value as a solution within a micro-frontend environment.

6.2 Methodology

In order to achieve the desired goal and answer the research questions, a mixed-methods approach that integrates both quantitative and qualitative analysis techniques was chosen, and will be defined in greater detail in this section.

6.2.1 Quantitative Analysis - RQ2 and RQ3

As for the quantitative analysis, automated code analysis tools will be used to measure key metrics such as code duplication and maintainability. These metrics can provide objective data and are directly correlated in answering RQ2.

Sonarqube (sonarsource 2024) was the chosen tool for the quality gate analysis. Other code quality tools were considered, such as DeepSource (deepsource 2024) and CodeScene (codescene 2024). However, Sonarqube was chosen since it is "is one of the most adopted code analysis tool in the context of CI environments (...) by more than 85,000 organizations" (Marcilio et al. 2019). The author also has previous experience in its use, and it can be easily set up locally, with a future integration with CI/CD being possible since it is a greatly documented tool. For this PoC, the static code analysis will be performed locally.

Sonarqube performs the analysis and rate the quality gate of the code, comparing it with code quality and performance guidelines. Duplication is measured by the amount of duplicated blocks of code, duplicated files and duplicated lines (docs.sonarsource 2024). Assuring that the duplication is within the desired limits will guarantee that the Components Library was efficient in lowering this metric. Maintainability is measured by the amount of code smells and technical debt. The maintainability rate is given by the time will take to resolve all those issues, in relation to the time already invested in the development of the project (docs.sonarsource 2024). This metric is essential to guarantee the quality of the projects as a whole.

When it comes to RQ3, the integration and deployment of the Components Library can be measured by deployment frequency (Codacy 2022), and comparing that to the published versions of the library on npm, to assure that the chosen deployment strategy was effective.

6.2.2 Survey Analysis - RQ1

Following a qualitative analysis approach, user testing and feedback questionnaires will be conducted to evaluate the consistency and quality of the UI/UX across different micro-frontends. Since design is subjective, and user experience can be inherently personal, feedback from users can be valuable to verify a cohesive design language through a components library. Open ended questions were included in the questionnaire in order to get feedback and constructive criticism that could result in suggestions for the improvement of the experience and design cohesion.

Quantitative methods will also be applied to the survey's answers. A SUS (Brooke 1995) will be part of the questionnaire, in order to evaluate the Vinci web application experience as whole. The SUS is composed of a 10 item questionnaire with five response options, ranging from 1 to 5, (where 1 is "Strongly Disagree" and 5 is "Strongly Agree") (Sauro 2011). The base questions are as follows:

1) I think that I would like to use this system frequently;

2) I found the system unnecessarily complex;

3) I thought the system was easy to use;

4) I think that I would need the support of a technical person to be able to use this system;

5) I found the various functions in this system were well integrated;

6) I thought there was too much inconsistency in this system;

7) I would imagine that most people would learn to use this system very quickly;

8) I found the system very cumbersome to use;

9) I felt very confident using the system;

10) I needed to learn a lot of things before I could get going with this system.

These questions may be altered to better reflect the characteristics of the system being evaluated. In this case, the questions were adapted to better inquire the research questions themes.

Other sets of questions, more focused on the UI components and the user experience and overall design, will also be included. The questions will vary into three different inputs: yes and no answers, ratings from 1 to 5 (for example, rating on the ease of use of certain functionality), and open-field answers. These questions will be analysed using hypothesis tests and SUS, as described in Section 3.4.1.

The survey will be carried out in Google Forms, so that the participants can do it remotely while accessing the deployed Vinci web application. The participants chosen for this test must be over 18 years of age, have technology dexterity and no accessibility impairments. The participants will remain anonymous, and the survey won't require the use of an identifier, such as an email address, as to comply with European General Data Protection Regulation (GDPR) laws.

The Google Forms survey is accessible in Appendix D of this project.

By combining quantitative and qualitative methods, a well-rounded evaluation is set to take place, that not only quantifies the Components Library's impact but also captures the subjective experiences of users interacting with the micro-frontends.

6.3 Results Analysis

As stated above, this section will cover the results of the quantitative and qualitative analyses of the PoC. The results of these analyses will inform the discussion on the viability of the Components Library as a solution for the identified challenges in a micro-frontend architecture, answering the research questions.

6.3.1 Quantitative Analysis

After Sonarqube was installed and set up locally, each of the frontend projects (Product Catalog, Checkout, Account Management micro-frontends and the application Shell) were configured with a `sonar-project.properties` file, to connect them with the local code analysis.

For the Product Catalog project (fig. 6.1), it is possible to see that the maintainability score was an A, the highest score, which means that the maintainability cost would be lower or equal to 5% of the time that has already gone into the application development (docs.sonarsource 2024). The code duplication was given a 0.0%, assuring that there were no duplicated blocks of code within the project.

Figure 6.1: Sonarqube quality gate analysis - Product catalog micro-frontend.

For the Checkout project, the duplication was of 1.5%, also within the acceptable less than 3% mark (as defined by the default Sonarqube metrics), with a green quality gate analysis (fig. 6.2).

However, the duplicated block appeared to be related to the navigation redirects between the Checkout micro-frontend and the products and login page, which had to be done on two different components (the shopping cart page and the checkout page). Therefore, the duplication was not related to HTML components.

Figure 6.2: Sonarqube quality gate analysis - Checkout micro-frontend.

For the Account Management micro-frontend and the application Shell projects, the code duplication analysis resulted in 0%, just like the Product Catalog project.

The projects received a green quality gate, with code coverage above 80%, the recommended by Sonarqube, and acceptable code quality scores.

With green-lit duplication and maintainability metrics, one can conclude that the Components Library was efficient in reducing code duplication, and improving the maintainability within a micro-frontend architecture, answering **RQ2**.

As for the deployment and package publishing strategy, the following table 6.1 indicates how many versions of the library were published to npm between 14/07/2024 and 11/08/2024, comparing them to the commits to the main branch on the GitHub repository of the Components Library.

Table 6.1: Commits vs Components Library Versions

Date time	Number of Commits	Number of Versions
2024/07/14	0	0
2024/07/25	2	2
2024/07/26	6	1
2024/07/28	2	1
2024/08/03	1	1
2024/08/11	1	1

The data was turned into a graph with the two plot data sets (fig. 6.3) to facilitate data visualization. As it is possible to view, since the first published version on 25/07/2024, at least one version of the library was published for each day where commits to main occurred. Not every commit to main resulted in a version increase, since not every change had that need.

With a high deployment frequency (in this case push to main) following the release of at least a published version per day of, the integration of the publishing versions and the development of the library are bound to be highly integrated, answering **RQ3**.

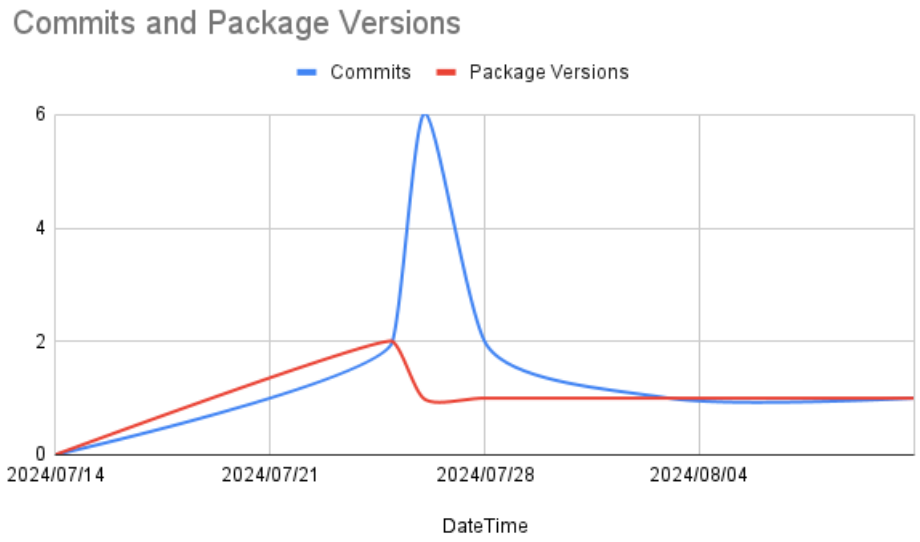


Figure 6.3: Github commits vs Published package versions.

6.3.2 Survey - Quantitative Analysis

The Google Forms survey had the contribution of 23 participants, all complying with the pre-required conditions mentioned previously (being over 18, having technological dexterity and no physical impairments). All of the answers for the survey can be accessed on Appendix E.

The first part of the survey was focused on evaluating the functionalities of the Vinci web application, as well as scoring the design cohesion of the pages from 1-5. In fig. 6.4, the answers for the design cohesion of the products, shopping cart, login and checkout pages are laid out.

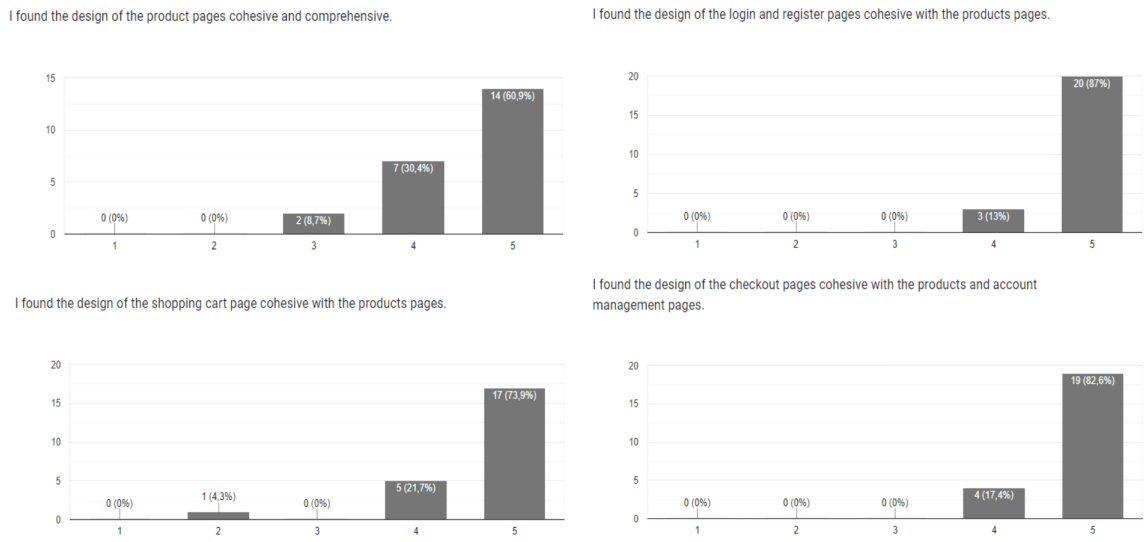


Figure 6.4: Histograms for the design cohesion scores of the products, shopping cart, login and checkout pages.

Another histogram was made, with the average of all the answers, in order to analyse if the data had a normal distribution. This would help in identifying the hypothesis test best suited for the data.

As is visible in fig. 6.5 the average scores of the design cohesion data was is not in a normal distribution shape (bell shape). In this case, a non-parametric statistical test should be used.

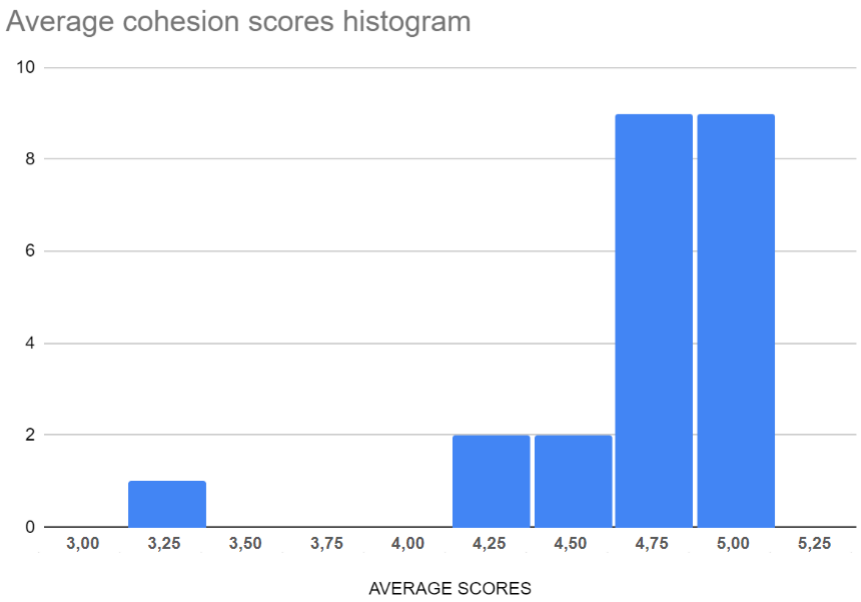


Figure 6.5: Histogram for the average of the design cohesion scores.

The non-parametric statistical test chosen was a one-sample Wilcoxon test, to determine if the sample data is statistically close to a predetermined value of the acceptable rating for the design cohesion. In this case, the hypothesis that were set in place for this tests were:

- Null Hypothesis (H_0): The median rating is equal to 4.75, indicating the design is cohesive;
- Alternative Hypothesis (H_1): The median rating is not equal to 4.75, indicating the design may not be cohesive.

The significance level (also called p-value) for this test was 0.05, following the recommended value for most studies (Kwak 2023), to determine if the difference in data is statistically significant or not statistically significant. At the end of the test, if the p-value is less than 0.05, the null hypothesis is rejected and the alternative is accepted, since in that case the median rate would be significantly different from 4.75. If the p-value is greater than 0.05, it is judged as "not significant" - therefore the null hypothesis is accepted, and the median score could be considered not significantly different from 4.75.

The calculations were performed in Microsoft Excel, using a sample spreadsheet from PeterStatistics 2020 with the required calculations - available in Appendix F.

The first step is calculating the absolute differences between each observed score and the hypothesized median (excluding any differences of zero). Next, these absolute differences are ranked, and the signs (positive or negative) of the original differences are determined. The sum of the ranks for positive differences is then calculated to obtain the test statistic W. The number of non-zero differences (n) is counted, and the unadjusted variance (VAR) is calculated (PeterStatistics 2020).

The number of tied ranks for each unique rank is then determined, and these values are cubed and adjusted by subtracting one from each and then summing the results. This sum is divided by 48 to obtain an adjustment factor, which is used to calculate the adjusted variance (VAR^*). The standard error (SE^*) is derived from the adjusted variance, and the test statistic W is adjusted using this standard error to calculate W^* . Finally, the significance of W^* (also called p-value) is determined by a normal distribution, since, according to DATAtab 2024, if the sample is bigger than 20 (in this case there are 23 answers) then normal distribution of W^* values can be assumed.

The final p-value was 0.9206, which is much higher than the typical significance level (0.05). Since the p-value is greater than 0.05, the null hypothesis is accepted. There is not enough evidence to suggest that the median rating of design cohesion differs from the baseline value of 4.75. In other words, the test concludes that the design is cohesive with a median rating close to 4.75.

The second part of the survey had a list of questions meant to be used in a SUS score analysis. For that, the users had to answer the 10 questions, similar to the ones laid out in the previous section, ranking each of them from 1 to 5, based on their level of agreement, the responses can be found on table 6.2.

Table 6.2: System Usability Scale - survey responses

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
User 1	5	1	5	1	5	1	5	1	5	1
User 2	4	1	5	1	5	1	4	1	4	1
User 3	5	1	5	1	5	1	5	1	5	1
User 4	1	1	5	1	4	1	5	1	4	1
User 5	5	1	4	1	5	1	5	1	5	1
User 6	5	1	5	1	5	1	5	1	5	1
User 7	5	1	4	1	5	1	5	1	5	1
User 8	3	3	3	1	3	3	2	2	2	1
User 9	5	1	5	1	5	1	5	1	5	1
User 10	5	1	5	1	5	1	5	1	5	1
User 11	5	1	4	1	5	1	4	1	5	1
User 12	5	1	5	1	5	1	5	1	5	1
User 13	4	1	5	1	5	1	4	1	5	1
User 14	4	1	4	5	5	1	4	1	5	1
User 15	5	1	5	1	5	1	5	1	5	1
User 16	5	1	5	1	5	1	5	5	5	1
User 17	4	1	5	3	4	1	4	1	1	1
User 18	5	1	5	1	5	1	5	1	5	1
User 19	4	1	5	1	5	1	4	1	4	1
User 20	4	1	5	1	5	1	4	1	4	1
User 21	4	1	5	2	5	2	4	1	5	2
User 22	4	1	5	1	5	1	5	1	4	1
User 23	4	2	4	1	5	1	5	1	5	1

To obtain the final SUS score, the following calculations (table 6.3) had to be done:

- For each of the odd numbered questions, 1 must be subtracted from the score;
- For each of the even numbered questions, their value must be subtracted from 5;
- These new values should be added up for the total score, which will have a maximum of 40. This must then be multiplied by 2.5, so a 1-100 score is present;
- After doing this for each of the 23 answers, the average of the total scores is calculated.

Questions that are odd will generate a positive response (the ideal response is 5), but questions that are even, generate a negative response (the ideal response in this case is 1) - and that's the reason why their scores have to be inverted (Sauro 2011).

Table 6.3: System Usability Scale - survey responses converted, with average final score

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Total**	**1-100**
User 1	4	4	4	4	4	4	4	4	4	4	40	100
User 2	3	4	4	4	4	4	3	4	3	4	37	92,5
User 3	4	4	4	4	4	4	4	4	4	4	40	100
User 4	0	4	4	4	3	4	4	4	3	4	34	85
User 5	4	4	3	4	4	4	4	4	4	4	39	97,5
User 6	4	4	4	4	4	4	4	4	4	4	40	100
User 7	4	4	3	4	4	4	4	4	4	4	39	97,5
User 8	2	2	2	4	2	2	1	3	1	4	23	57,5
User 9	4	4	4	4	4	4	4	4	4	4	40	100
User 10	4	4	4	4	4	4	4	4	4	4	40	100
User 11	4	4	3	4	4	4	3	4	4	4	38	95
User 12	4	4	4	4	4	4	4	4	4	4	40	100
User 13	3	4	4	4	4	4	3	4	4	4	38	95
User 14	3	4	3	0	4	4	3	4	4	4	33	82,5
User 15	4	4	4	4	4	4	4	4	4	4	40	100
User 16	4	4	4	4	4	4	4	0	4	4	36	90
User 17	3	4	4	2	3	4	3	4	0	4	31	77,5
User 18	4	4	4	4	4	4	4	4	4	4	40	100
User 19	3	4	4	4	4	4	3	4	3	4	37	92,5
User 20	3	4	4	4	4	4	3	4	3	4	37	92,5
User 21	3	4	4	3	4	3	3	4	4	3	35	87,5
User 22	3	4	4	4	4	4	4	4	3	4	38	95
User 23	3	3	3	4	4	4	4	4	4	4	37	92,5
											Average	**92,61**

After performing the calculations, visible in table 6.3, the final average score was 92,61. This score can be interpreted in a few different ways, as can be seen on fig. 6.6.

Firstly, by a grade - "grades range from A, which indicates superior performance, to F (for failing performance), with C indicating average" (Sauro 2018). In this case, Vinci had a grade of A.

Next, the adjectives found to be used to describe the systems evaluated by the SUS score, with Vinci being in the "Excellent" category.

As for the acceptability, Vinci lies on the "Acceptable" portion. Additionally, for the Net Promoter Score (NPS) interpretation, it "designates three classes of recommenders based on their responses to the 11-point (0 to 10) likelihood to recommend question. Promoters score 9 and 10; passives, 7 and 8; and detractors, 6 and below" (Sauro 2018). For Vinci, the "Promoter" NPS was aligned.

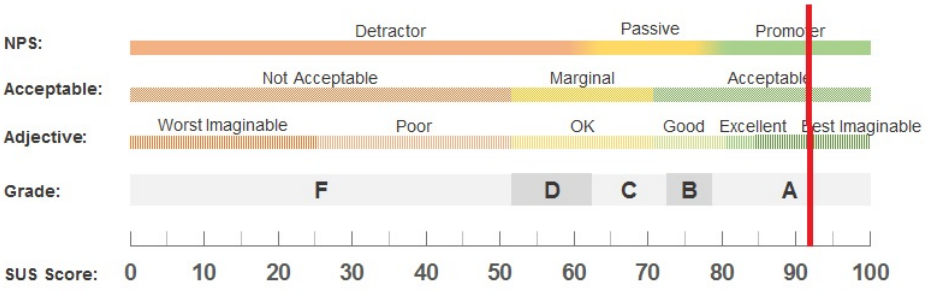


Figure 6.6: SUS Raw Score - with grade, adjective, acceptability, and NPS category (adapted from Sauro 2018)

Even though this score is out of 100, it is not a percentage. The average of the SUS is actually 68 (as can be seen on fig. 6.7) - and the curve is a cumulative distribution function for the normal distribution, so the 92,61 score is closer to a percentile score of 100%.

With this we can confirm that the usability and the user experience of the Vinci web application had an excellent score following SUS.

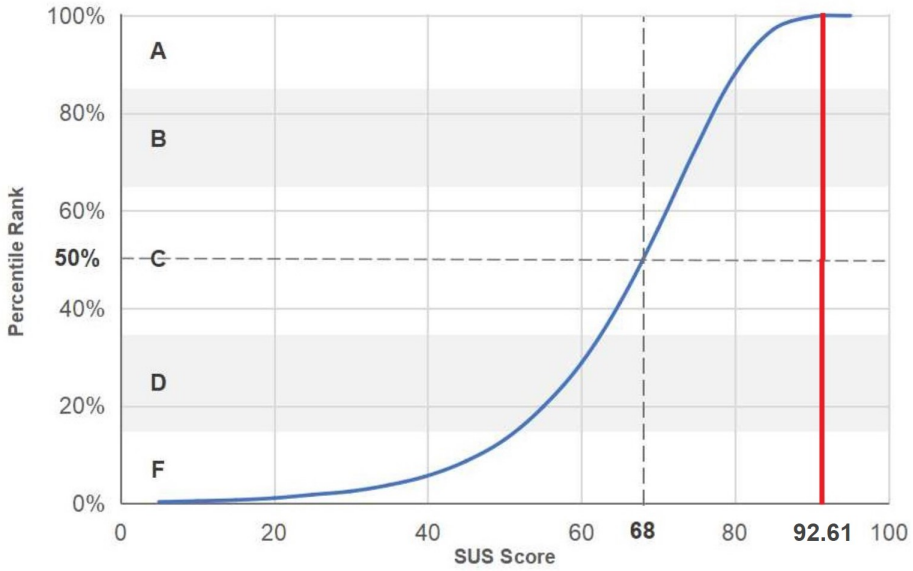


Figure 6.7: SUS Raw Score percentile (adapted from Sauro 2018)

For this survey analysis, it can be concluded that, according to the Wilcoxon t-test, the design of the different micro-frontend pages is cohesive, relying on a components library. Additionally, according to the SUS score, the conclusion is that the usability of the Vinci web application's design was above average. This answers **RQ1**, sustaining the hypothesis that a custom components library can maintain a coherent UI and UX design in a micro-frontend solution.

6.3.3 Survey - Qualitative Analysis

The final part of the survey had an open-ended question, where users could leave their feedback on the web application and their suggestions for improvement in terms of design cohesion. The feedback is largely positive, with several users praising its cohesive design, intuitive layout, and modern aesthetic. The design was considered consistent across pages, the navigation was considered easy and the alignment with current market standards make it user-friendly and accessible. Users appreciated the minimalist and elegant design, which allows for a focus on products and enhances the shopping experience. The responsiveness of the site was also noted as a strong point.

However, a few areas for improvement were highlighted. One common suggestion related to user experience on the components was the need for better interactivity, such as making all clickable elements more intuitive, since the cursor icon was not displaying consistently on clickable areas. The other comments were regarding e-commerce features that were not completely implemented on this PoC.

6.4 Summary

The purpose of this chapter was to answer the three research questions that were introduced as the main focus of this project, by analysing data according to quantitative and qualitative methods.

Quantitative methods such as the static code analysis and deployment frequency were used to answer RQ2 and RQ3 respectively. The created solution englobing the custom Components Library was sustained by the results as a good way to achieve the desired duplicated code percentage and maintainability score throughout the micro-frontend architecture - answering RQ2. The developed CD strategy was successful in integrating the custom library with the micro-frontends, answering RQ3.

A questionnaire was used to obtain data to support the design cohesion and usability of the developed solution. Then quantitative methods such as the hypothesis test confirmed that the design was believed to be cohesive with a score significantly close to 4.75 out of a maximum of 5. A SUS analysis was also done and confirmed that the usability of the Vinci web application was well above average. The user feedback was also mostly positive, with reinforcement on the overall aesthetic and design cohesion, making the qualitative analysis successful. This answers the main research question, RQ1, and concludes that the Components Library was efficiently used in the PoC solution in order to make the UI/UX design cohesive.

Chapter 7

Conclusions

This chapter will summarize all of the accomplished goals and achieved objectives throughout this project, as well as some challenges and limitations, that lead into some planned future work, ending with a summary.

7.1 Achieved Objectives

The main goal of this dissertation project was to answer the three research questions presented in the introduction.

As for the first research question (RQ1), the goal was to discover how to maintain consistent UI and UX design of the application, in a case where external libraries are discouraged, or the project should have a unique design. For this, following the results of the survey analysis in the previous Chapter - 6, both the SUS calculation score and the Wilcoxon test proved that a cohesive design was created in a micro-frontend solution, using a components library based on a design system.

The second research question (RQ2) was related to finding a way to reduce code duplication, enhance the development process, and make the application more maintainable and scalable. This was answered by using static code analysis to prove that the developed solution had low code duplication and high maintainability.

The third research question (RQ3), related to the deployment and integration strategy of a components library package, was answered by creating a CI/CD approach to the publishing of the components library, that happened every time a push to main with an upgrade to the version of the package occurred.

As for the project itself, all of its steps were carefully planned and discussed, with a strong basis on the state of the art and the problem at hand. Micro-frontend architecture disadvantages were laid out at the core of this said problem. By creating a project from scratch, with four main web components and a web API based on industry standards and guidelines, the developed PoC stands as a faithful representation of an up-to-date micro-frontend system.

A design system was created with a focus on modern UI design guidelines, following an atomic design approach, in order to support the development of the Components Library and micro-frontends.

The developed Components Library was also sustained with a detailed documentation, which includes playroom for testing, style documentation, incorporation and usage rules.

CI/CD methods were taken into consideration and set in place for the Components Library, the center of this project. The documentation and new versions were deployed through pipelines in order to create a reliable strategy of integration with Angular applications, in this case with the micro-frontends of this PoC.

Quality assurance was provided in the solutions in form of different types of testing throughout the different components in this PoC, with the projects achieving high percentages of coverage, and tests also being part of the CI/CD validation of the Components Library.

All of the layers (client, server and storage) were developed with the intention of the deployment of the final functional project, that was reviewed by participants on its design and consistency throughout the web application experience. Both qualitative and quantitative methods of data collection were used, as planned, with the project achieving high grades and qualifications in both, proving that the chosen solution was the answer to the questions imposed by the state of the art. The PoC met the standards and metrics of modern software engineering.

7.1.1 Challenges and Limitations

Though the planning took into consideration 6 months of development, the PoC took longer than expected, due to the author's working student status.

The latest version of Angular was not used, since most of the documentation regarding micro-frontends as web components using Webpack module were found to be in version 15.0 - which was the one used in this project. This version is however still commercially supported, and the library should be available to be used at least up to two major versions prior to the version of the application that consumes it.

Since the focus of this PoC was not specifically on the features of the Vinci web application, but on the Components Library and overall architectural strategy, not a lot of time was dedicated to the specific features of each micro-frontend implementation (about one month for all). The application could have better functionality - something that will be mentioned in the next section.

7.2 Future Work

An upgrade on Angular version to the latest (at this time 18.0) could be best suited for longer support, this however would have impact on how the micro-frontend solution has been set up.

In terms of the SDLC for the project, some methodologies were done only in some projects, such as the acceptance tests. An improvement on the quantity (and quality) of tests for all micro-frontends can be a future improvement.

Same thing can be said regarding the CI/CD strategy, which had a main focus on the Components Library project. On the micro-frontends, Vercel was used for automatic deployment on pushes to the main branch, however the web API was manually updated and deployed in AWS whenever needed. CI/CD practices can take place in these other projects, including code quality verification with Sonarqube on the pipelines.

A number of possible improvements were commented at on the Vinci web application survey, where participants mentioned that there could be better interactivity, such as making all

clickable elements more intuitive and ensuring that filters automatically update without needing to press a "Show Results" button in the products page. Some users noted some functionality issues such as some lack of input validations. The web application was also missing a responsive web design, which meant that it did not adapt to different device layouts. This could be a big improvement.

However, as mentioned previously, this was due to time constraints and for the main focus of the work being put towards the architecture and Components Library solution.

Additionally, some pages were just mockups of future features, such as the account management pages, which left the Account Management micro-frontend only responsible for the login and registration. The edit profile, view order history and wishlist features can be a good addition to the project. With the increment of these pages, more components could be added to the library, which can be an on going process.

When it comes to the Components Library itself, the creation of a marketplace to disseminate the design system packages is also an interesting possible future work.

In summary, the feedback gathered from the Vinci web application survey reinforced the importance of maintaining a cohesive and intuitive design throughout the user experience. The application was praised for its consistent and intuitive design, which aligns with the project's goal of developing a cohesive user interface that is easy to navigate and use. This is a direct reflection of the project's focus on reducing code duplication, ensuring maintainability, and implementing a structured deployment strategy, all of which were central objectives of the dissertation.

Future work should focus on addressing the specific usability concerns raised by users. These improvements would not only refine the existing application but also expand the Components Library, further solidifying the project's architectural foundation.

Finally, despite the challenges and limitations found throughout the development of this project, the research questions were answered, sustained by meaningful data. This PoC was able to prove that the disadvantages regarding the design inconsistency of micro-frontend architecture described in the state of the art can be overcome with the development of a Components Library.

Bibliography

Akram, Shereen (Jan. 15, 2020). "The importance of UI/UX in software development for business growth". In: *MSCS Survey Reports*. url: https://ir.iba.edu.pk/survey-reports-mscs/38.

Angular.dev (2024). *Angular - testing*. url: https://angular.dev/guide/testing (visited on 07/09/2024).

Angular.io (Feb. 2022). *Angular - Creating libraries*. url: https://angular.io/guide/creating-libraries (visited on 04/30/2024).

– (Aug. 30, 2023a). *Angular - Create a new project*. url: https://angular.io/tutorial/tour-of-heroes/toh-pt0 (visited on 04/30/2024).

– (Oct. 24, 2023b). *Angular - Lazy-loading feature modules*. url: https://angular.io/guide/lazy-loading-ngmodules (visited on 04/30/2024).

– (Feb. 28, 2023c). *Angular - NgModules*. url: https://angular.io/guide/ngmodules (visited on 11/19/2023).

– (Aug. 15, 2023d). *Angular - What is Angular?* url: https://angular.io/guide/what-is-angular (visited on 11/19/2023).

Arunodi, Nipuni (May 6, 2022). *Top 10 Angular Component Libraries*. Section: Angular. url: https://www.syncfusion.com/blogs/post/top-angular-component-libraries.aspx (visited on 11/29/2023).

Bagui, Sikha and Richard Earp (June 27, 2003). *Database Design Using Entity-Relationship Diagrams*. Google-Books-ID: KZKylX9oH8IC. CRC Press. 263 pp. isbn: 978-0-203-48605-4.

Biswal, Avijeet (Nov. 10, 2023). *What is Hypothesis Testing in Statistics? Types and Examples | Simplilearn*. Simplilearn.com. url: https://www.simplilearn.com/tutorials/statistics-tutorial/hypothesis-testing-in-statistics (visited on 12/18/2023).

ng-bootstrap (2023). *Angular powered Bootstrap*. ng-bootstrap. url: https://ng-bootstrap.github.io (visited on 12/01/2023).

Brooke, John (Nov. 1995). "SUS: A quick and dirty usability scale". In: *Usability Eval. Ind.* 189.

Cebrian, Michael C (May 1, 2017). "Angular Component Library Comparison". In.

Chromatic (2024). *Automate Chromatic with GitHub Actions • Chromatic docs*. url: https://www.chromatic.com/docs/ (visited on 07/02/2024).

clarity.design (2023). *Clarity Design System*. Clarity Design System. url: https://clarity.design/ (visited on 12/01/2023).

Codacy (Mar. 21, 2022). *How to measure Deployment frequency?* url: https://blog.codacy.com/how-to-measure-deployment-frequency (visited on 08/14/2024).

codescene (2024). *Next generation code analysis | CodeScene*. url: https://codescene.com (visited on 09/09/2024).

Couldwell, Andrew (Oct. 16, 2019). *Laying the Foundations: A book about design systems*. Google-Books-ID: GInVDwAAQBAJ. Owl Studios. 268 pp.

Daityari, Shaumik (Jan. 10, 2019). *Angular vs React vs Vue: Which Framework to Choose in 2023*. CodeinWP. Running Time: 484. url: https://www.codeinwp.com/blog/angular-vs-vue-vs-react/ (visited on 11/18/2023).

DATAtab (2024). *DATAtab: Online Statistics Calculator*. url: https://datatab.net/tutorial/wilcoxon-test (visited on 08/25/2024).

deepsource (2024). *DeepSource: The Code Health Platform*. url: https://deepsource.com/ (visited on 09/09/2024).

docs.aws.amazon.com (2024a). *IAM users - AWS Identity and Access Management*. url: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html (visited on 08/11/2024).

– (2024b). *Target groups for your Application Load Balancers - Elastic Load Balancing*. url: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-target-groups.html (visited on 08/11/2024).

– (2024c). *What is Amazon Elastic Container Registry? - Amazon ECR*. url: https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html (visited on 08/11/2024).

– (2024d). *What is Amazon Elastic Container Service? - Amazon Elastic Container Service*. url: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html (visited on 08/11/2024).

docs.cypress.io (Apr. 11, 2024a). *Comprehensive Cypress Test Automation Guide | Cypress Documentation*. url: https://docs.cypress.io/guides/overview/why-cypress (visited on 07/10/2024).

– (Mar. 28, 2024b). *intercept | Cypress Documentation*. url: https://docs.cypress.io/api/commands/intercept (visited on 07/10/2024).

docs.npmjs (Oct. 23, 2023). *About npm | npm Docs*. docs.npmjs. url: https://docs.npmjs.com/about-npm (visited on 12/28/2023).

docs.sonarsource (2024). *Metric definitions*. url: https://docs.sonarsource.com/sonarqube/8.9/user-guide/metric-definitions/.

Figma (2023). *Clarity Design System Figma*. Figma. Section: Design. url: https://www.figma.com/file/DjLHTKDvXKklPi8SL3TGqF/Clarity-UI-Library-light-5.0.0-Community (visited on 12/01/2023).

– (July 3, 2024). *Iconify*. Figma. url: https://www.figma.com/community/plugin/735098390272716381/iconify (visited on 07/11/2024).

Figma, Community (2023a). *Figma - E-commerce Checkout Page*. Figma. Section: Design. url: https://www.figma.com/file/6pCs5wLyTNA63JYGRooQEE/E-commerce-Checkout-Page-Community (visited on 12/16/2023).

– (2023b). *Figma - E-commerce Login UI*. Figma. Section: Design. url: https://www.figma.com/file/SG5JcWKa2u8WQO2XhU1NOM/E-commerce-Login-UI-Community (visited on 12/16/2023).

– (2023c). *Figma - E-commerce Product Detail Page*. Figma. Section: Design. url: https://www.figma.com/file/S2uiH6g7WoYlNKEL8euBpx/Ecommerce-Product-Detail-Page-Community (visited on 12/16/2023).

Filipova, Olga (Dec. 13, 2016). *Learning Vue.js 2*. Google-Books-ID: nszcDgAAQBAJ. Packt Publishing Ltd. 323 pp. isbn: 978-1-78646-113-1.

Freeman, Adam (June 12, 2020). *Pro Angular 9: Build Powerful and Dynamic Web Apps*. Google-Books-ID: ji3rDwAAQBAJ. Apress. 791 pp. isbn: 978-1-4842-5998-6.

Frost, Brad (Dec. 5, 2016). *Atomic Design*. Google-Books-ID: 1e92vgAACAAJ. Brad Frost Web. 193 pp. isbn: 978-0-9982966-0-9.

Funk, Ezekiel (2024). *AmstelvarAlpha Font Family*. AmstelvarAlpha Font Family. url: https://www.cufonfonts.com/font/amstelvaralpha (visited on 04/23/2024).

Gackenheimer, Cory (Sept. 11, 2015). *Introduction to React*. Google-Books-ID: NZCKCgAAQBAJ. Apress. 141 pp. isbn: 978-1-4842-1245-5.

Geers, Michael (Oct. 13, 2020). *Micro Frontends in Action*. Google-Books-ID: FFD9DwAAQBAJ. Simon and Schuster. 294 pp. isbn: 978-1-61729-687-1.

– (2023). *Micro Frontends - extending the microservice idea to frontend development*. Micro Frontends. url: https://micro-frontends.org/ (visited on 10/15/2023).

GitHub (2024). *Using secrets in GitHub Actions*. GitHub Docs. url: https://docs.github.com/en/actions/security-guides/using-secrets-in-github-actions (visited on 07/02/2024).

Gitlab, Team (Sept. 29, 2022). *What are the benefits of a microservices architecture?* GitLab. url: https://about.gitlab.com/blog/2022/09/29/what-are-the-benefits-of-a-microservices-architecture/ (visited on 10/21/2023).

Godbolt, M. (2016). *Frontend Architecture for Design Systems: A Modern Blueprint for Scalable and Sustainable Websites*. O'Reilly Media. isbn: 978-1-4919-2673-4. url: https://books.google.pt/books?id=acV4CwAAQBAJ.

Goeleven, Yves (Feb. 5, 2021). *App Shell Pattern*. Goeleven. url: https://www.goeleven.com/blog/app-shell/ (visited on 10/22/2023).

Gómez, Oliver Carvajal (July 13, 2023). *Micro Frontends Unleashed: Better Performance, Scalability and User Experience. | LinkedIn*. url: https://www.linkedin.com/pulse/micro-frontends-unleashed-better-performance-user-carvajal-g%C3%B3mez/ (visited on 10/16/2023).

Hannousse, Abdelhakim and Salima Yahiouche (Aug. 2021). "Securing microservices and microservice architectures: A systematic mapping study". In: *Computer Science Review* 41, p. 100415. issn: 15740137. doi: 10.1016/j.cosrev.2021.100415. url: https://linkinghub.elsevier.com/retrieve/pii/S1574013721000551 (visited on 10/21/2023).

help.figma.com (2024a). *Create and apply text styles*. Figma Learn - Help Center. url: https://help.figma.com/hc/en-us/articles/360039957034-Create-and-apply-text-styles (visited on 04/28/2024).

– (2024b). *Figma for VS Code*. Figma Learn - Help Center. url: https://help.figma.com/hc/en-us/articles/15023121296151-Figma-for-VS-Code (visited on 04/21/2024).

J, Justin (Sept. 17, 2021). *Create An Angular Library And Consume it locally (with debugging)*. DEV Community. url: https://dev.to/jsanddotnet/create-an-angular-library-and-consume-it-locally-with-debugging-cma (visited on 04/28/2024).

Jackson, Cam (June 19, 2019). *Micro Frontends*. martinfowler.com. url: https://martinfowler.com/articles/micro-frontends.html (visited on 11/01/2023).

JasmineDocs (2024). *Jasmine Documentation*. url: https://jasmine.github.io/index.html (visited on 07/09/2024).

JeremyLikness (Dec. 2, 2022). *Minimal APIs overview*. url: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/overview?view=aspnetcore-8.0 (visited on 04/07/2024).

Joshi, Mohit (May 11, 2023). *Angular vs React vs Vue: Core Differences*. BrowserStack. url: https://browserstack.wpengine.com/guide/angular-vs-react-vs-vue/ (visited on 11/18/2023).

Klimm, Marvin Christian, Christina Noss, and Stefan Bente (Feb. 1, 2021). "Design Systems for Micro Frontends". In: url: https://epb.bibl.th-koeln.de/frontdoor/index/index/docId/1666.

Kwak, Sanggyu (May 2023). "Are Only p-Values Less Than 0.05 Significant? A p-Value Greater Than 0.05 Is Also Significant!" In: *Journal of Lipid and Atherosclerosis* 12.2, pp. 89–95. issn: 2287-2892. doi: 10.12997/jla.2023.12.2.89. url: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10232224/ (visited on 08/25/2024).

learn.microsoft (May 25, 2021). *Overview of Entity Framework Core - EF Core*. url: https://learn.microsoft.com/en-us/ef/core/ (visited on 07/12/2024).

– (Jan. 12, 2023). *Migrations Overview - EF Core*. url: https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/ (visited on 07/12/2024).

– (July 1, 2024). *Tutorial: Create a minimal API with ASP.NET Core*. url: https://learn.microsoft.com/en-us/aspnet/core/tutorials/min-web-api?view=aspnetcore-8.0 (visited on 07/12/2024).

Marcilio, Diego et al. (May 2019). "Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube". In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). Montreal, QC, Canada: IEEE, pp. 209–219. isbn: 978-1-72811-519-1. doi: 10.1109/ICPC.2019.00040. url: https://ieeexplore.ieee.org/document/8813272/ (visited on 09/09/2024).

material.angular.io, Documentation (2023). *Angular Material*. Angular Material. url: https://material.angular.io/ (visited on 10/15/2023).

Matteson, Steve (2024). *Open Sans*. Google Fonts. url: https://fonts.google.com/specimen/Open+Sans (visited on 04/28/2024).

Meraj, Md (July 11, 2023). *Mastering Micro Frontends: A Web-Component Adventure and the Lessons Learned*. Medium. url: https://medium.com/@mmeraj/mastering-micro-frontends-a-web-component-adventure-and-the-lessons-learned-e2584a67dc1f (visited on 07/10/2024).

Mezzalira, Luca (Nov. 17, 2021). *Building Micro-Frontends*. Google-Books-ID: NDpPEAAAQBAJ. "O'Reilly Media, Inc." 337 pp. isbn: 978-1-4920-8296-5.

Miller, Roy W and Christopher T Collins (2001). "Software Developer RoleModel Software, Inc. 342 Raleigh Street Holly Springs, NC 27540 USA +1 919 557 6352". In: *XP Universe Conference*.

Oates, Briony J., Marie Griffiths, and Rachel McLean (Jan. 12, 2022). *Researching Information Systems and Computing*. Google-Books-ID: tN5XEAAAQBAJ. SAGE. 372 pp. isbn: 978-1-5297-8493-0.

Oracle (2024). *MySQL :: MySQL Workbench*. url: https://www.mysql.com/products/workbench/ (visited on 04/28/2024).

Ortico, Jack (July 16, 2021). *How to customize Bootstrap styles and variables when using ng-bootstrap*. Medium. url: https://medium.com/@jackortico/how-to-customize-bootstrap-styles-and-variables-when-using-ng-bootstrap-9df293ae903d (visited on 12/01/2023).

Osherove, Roy (Nov. 24, 2013). *The Art of Unit Testing: with examples in C#*. Google-Books-ID: tTkzEAAAQBAJ. Simon and Schuster. 459 pp. isbn: 978-1-63835-305-8.

Pattakos, Aris (Oct. 26, 2023). *Angular vs React vs Vue: Which Framework Is Better? 2023*. aThemes. url: https://athemes.com/guides/angular-vs-react-vs-vue/ (visited on 11/18/2023).

Pavlenko, Andrey et al. (May 31, 2020). "Micro-frontends: application of microservices to web front-ends". In: *Journal of Internet Services and Information Security* 10.2, pp. 49–66. doi: 10.22667/JISIS.2020.05.31.049. url: https://doi.org/10.22667/JISIS.2020.05.31.049 (visited on 10/14/2023).

Peltonen, Severi, Luca Mezzalira, and Davide Taibi (Aug. 2021). "Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review". In: *Information and Software Technology* 136, p. 106571. issn: 09505849. doi: 10.1016/j.infsof.2021.106571. url: https://linkinghub.elsevier.com/retrieve/pii/S0950584921000549 (visited on 10/14/2023).

PeterStatistics (2020). *Peter's Statistics Crash Course*. url: https://peterstatistics.com/CrashCourse/2-SingleVar/Ordinal/Ordinal-2a-Test.html (visited on 08/25/2024).

Prajwal, Y R, Jainil Viren Parekh, and Dr Rajashree Shettar (2021). "A Brief Review of Micro-frontends". In: 02.8.

Rahman, Mazedur and Jerry Gao (Mar. 2015). "A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development". In: *2015 IEEE Symposium on Service-Oriented System Engineering*. 2015 IEEE Symposium on Service-Oriented System Engineering, pp. 321–325. doi: 10.1109/SOSE.2015.55. url: https://ieeexplore.ieee.org/abstract/document/7133548 (visited on 07/10/2024).

Ravoof, Salman (Mar. 15, 2023). *9 Nifty Angular Component Libraries to Jump-Start Development*. Kinsta®. url: https://kinsta.com/blog/angular-component-libraries/ (visited on 12/01/2023).

react.dev (Jan. 1, 2023). *State: A Component's Memory – React*. url: https://react.dev/learn/state-a-components-memory (visited on 11/26/2023).

reactjs.org (Jan. 1, 2023a). *React.Component – React*. url: https://legacy.reactjs.org/docs/react-component.html (visited on 11/26/2023).

– (Jan. 1, 2023b). *Rendering Elements – React*. url: https://legacy.reactjs.org/docs/rendering-elements.html (visited on 11/26/2023).

Richardson, Chris (2019). *Microservices Pattern: Microservice Architecture pattern*. microservices.io. url: http://microservices.io/patterns/microservices.html (visited on 10/21/2023).

Robertson, Christian (2024). *Roboto*. Google Fonts. url: https://fonts.google.com/specimen/Roboto (visited on 04/28/2024).

Rylan, Cory (Sept. 30, 2021). *Building forms with Angular and Clarity Design - Angular 17 | 16*. url: https://coryrylan.com/blog/building-forms-with-angular-and-clarity-design (visited on 12/01/2023).

Sauro, Jeff (Jan. 3, 2011). *Measuring Usability with the System Usability Scale (SUS) – MeasuringU*. url: https://measuringu.com/sus/ (visited on 08/14/2024).

– (Sept. 19, 2018). *5 Ways to Interpret a SUS Score – MeasuringU*. url: https://measuringu.com/interpret-sus-score/ (visited on 08/21/2024).

Schäffer, Eike et al. (2019). "Microservice-based architecture for engineering tools enabling a collaborative multi-user configuration of robot-based automation solutions". In: *Procedia CIRP* 86, pp. 86–91. issn: 22128271. doi: 10.1016/j.procir.2020.01.017. url: https://linkinghub.elsevier.com/retrieve/pii/S2212827120300172 (visited on 10/15/2023).

Shoval, Peretz, Revital Danoch, and Mira Balabam (Nov. 2004). "Hierarchical entity-relationship diagrams: the model, method of creation and experimental evaluation". In: *Requirements Engineering* 9.4, pp. 217–228. issn: 0947-3602, 1432-010X. doi: 10.1007/s00766-004-0201-9. url: http://link.springer.com/10.1007/s00766-004-0201-9 (visited on 04/07/2024).

sonarsource (2024). *Code Quality Tool & Secure Analysis with SonarQube*. url: https://www.sonarsource.com/products/sonarqube/ (visited on 09/09/2024).

Steyer, Manfred (Apr. 26, 2020). *The Microfrontend Revolution: Module Federation with Angular*. ANGULARarchitects. url: https://www.angulararchitects.io/blog/the-microfrontend-revolution-part-2-module-federation-with-angular/ (visited on 04/30/2024).

Storybook (Nov. 5, 2023). *Storybook: Frontend workshop for UI development*. url: https://storybook.js.org (visited on 11/05/2023).

– (2024a). *Storybook - Canvas*. Storybook. url: https://storybook.js.org/docs/api/doc-blocks/doc-block-canvas (visited on 07/02/2024).

– (2024b). *Storybook - Interaction tests*. Storybook. url: https://storybook.js.org/docs/writing-tests/interaction-testing (visited on 07/02/2024).

– (2024c). *Storybook - MDX*. Storybook. url: https://storybook.js.org/docs/writing-docs/mdx (visited on 07/02/2024).

– (2024d). *Storybook - Meta*. Storybook. url: https://storybook.js.org/docs/api/doc-blocks/doc-block-meta (visited on 07/02/2024).

– (2024e). *Storybook - Writing Stories*. Storybook. url: https://storybook.js.org (visited on 07/02/2024).

– (2024f). *Storybook Tutorials - Deploy*. url: https://storybook.js.org/tutorials/intro-to-storybook/angular/en/deploy/ (visited on 07/02/2024).

Sukoinen, Mikael (Nov. 12, 2020). *Building and maintaining the component library of a design system | Vaadin*. url: https://vaadin.com/blog/building-and-maintaining-the-component-library-of-a-design-system (visited on 11/05/2023).

Svyryd, Andriy (May 11, 2022). *Data Seeding - EF Core*. url: https://learn.microsoft.com/en-us/ef/core/modeling/data-seeding (visited on 07/12/2024).

– (Mar. 28, 2023). *Creating and Configuring a Model - EF Core*. url: https://learn.microsoft.com/en-us/ef/core/modeling/ (visited on 07/12/2024).

Szahidewicz, Dominik (July 12, 2024). *Playwright vs Cypress: A Comprehensive Comparison*. BugBug.io. url: https://bugbug.io/blog/test-automation-tools/playwright-vs-cypress/ (visited on 09/12/2024).

UXPin (Dec. 13, 2022). *Building a Component Library – A Step-by-Step Guide*. Studio by UXPin. url: https://www.uxpin.com/studio/blog/building-component-library-guide/ (visited on 11/05/2023).

Vailshery, Lionel Sujay (May 21, 2024). *Global cloud infrastructure market share 2024*. Statista. url: https://www.statista.com/statistics/967365/worldwide-cloud-infrastructure-services-market-share-vendor/ (visited on 08/11/2024).

Vasireddy, Karthik (July 25, 2023). *Redefining architecture of developments with micro frontends*. url: https://www.technoidentity.com/insights/architectural-approach-to-large-scale-web-applications-with-micro-frontends/ (visited on 10/22/2023).

vercel.com (Aug. 13, 2024). *Get started with Vercel*. url: https://vercel.com/docs/getting-started-with-vercel (visited on 08/13/2024).

Verma, Riya (Jan. 19, 2023). *SCSS Vs CSS - Key Difference Between CSS and SCSS*. Scaler Topics. url: https://www.scaler.com/topics/difference-between-css-and-scss/ (visited on 04/24/2024).

Vernon, Vaughn (Feb. 6, 2013). *Implementing Domain-Driven Design*. Google-Books-ID: X7DpD5g3VP8C. Addison-Wesley. 656 pp. isbn: 978-0-13-303988-7.

VisualStudioMarketplace (2024). *Karma Test Explorer (for Angular, Jasmine, and Mocha) - Visual Studio Marketplace*. url: https://marketplace.visualstudio.com/items?itemName=lucono.karma-test-explorer (visited on 07/09/2024).

vuejs.org (Jan. 1, 2023). *Introduction | Vue.js*. url: https://vuejs.org/guide/introduction.html (visited on 11/26/2023).

webpack.js (2024). *Public Path*. webpack. url: https://webpack.js.org/guides/public-path/ (visited on 08/11/2024).

Appendix A

Figma Design

User Interface Figma Design - https://www.figma.com/file/JEdVeVvHMlE9LuYChTwW0T/Dissertation-e-commerce-Design-System?type=design&node-id=0-1&mode=design&t=OScrV9ng0T8OJhK8-0

Appendix B

Github Repositories

B.1 Components Library

- Angular Library ngx-dissertation-components-lib
  - `https://github.com/carinaalasisep/ngx-dissertation-components-lib`

B.2 Micro-frontends

- Product catalog micro-frontend
  - `https://github.com/carinaalasisep/dissertation-productcat-mfe`
- Checkout micro-frontend
  - `https://github.com/carinaalasisep/dissertation-checkout-mfe`
- App Shell
  - `https://github.com/carinaalasisep/dissertation-app-shell`

B.3 Web API

- Product web API
  - `https://github.com/carinaalasisep/dissertation-api`

Appendix C

Deployed applications

C.1 Components Library

- Documentation
  - `https://6683c683f34b923f72274146-wxbstdhwkx.chromatic.com`
- NPM registry
  - `https://www.npmjs.com/package/ngx-isep-dissertation-components`

C.2 Web API

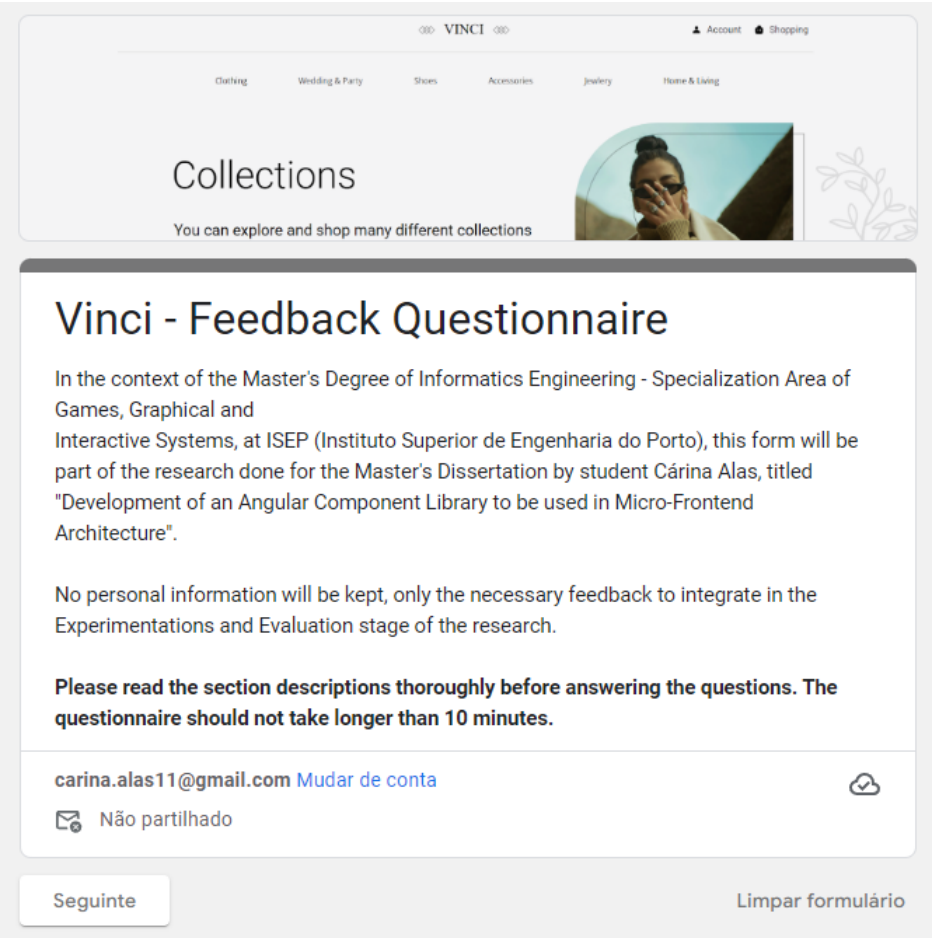
- `www.dissertation-product-cat-api.online`

C.3 Web application

- `https://dissertation-app-shell.vercel.app/app/products`

Appendix D

Feedback Questionnaire



Personal Information

This section is only meant to assure that you are a viable candidate to be a part of this research.

Name

A sua resposta

Are you over the age of 18? *

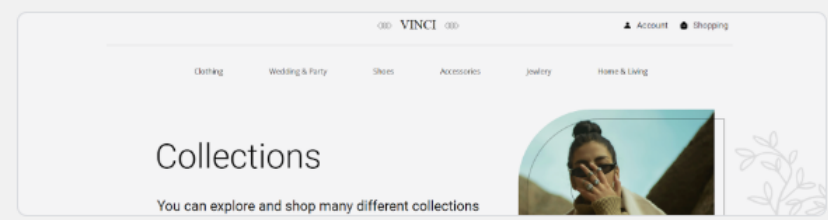
- ◯ Yes
- ◯ No

Are you used to working on a computer? *

- ◯ Yes
- ◯ No

Do you have any motor disabilities? *

- ◯ Yes
- ◯ No



Vinci - Feedback Questionnaire

carina.alas11@gmail.com Mudar de conta

Não partilhado

FEEDBACK QUESTIONNAIRE

Vinci is a web application developed to be a proof of concept of an e-commerce experience.

It is recommended that you test it using a screen of size **1920x1080** with the browser window maximized, since the application is not responsive for other resolutions.

The objective will be to evaluate the user interface and user experience of the application's design and functionality. When navigating through the application, take note of any issues or feedback you'd like to give for the end of the survey. Please follow along testing the application while you answer the survey.

Please be mindful that not all features are enabled, for instance, the account details pages and the wishlist feature.

You can access the Vinci platform at: https://dissertation-app-shell.vercel.app/app/products and head on to the next section of the survey.

PRODUCTS PAGE

After opening the link, you should be at the products home page, view it, and then click on "Shop Now".

You will be at the products listing page, you can filter the products by clicking on one category on the top of the page, or using the filters option on the right.

Were you able to navigate to the products listing page through the home page? *

- ◯ Yes
- ◯ No

Were you able to filter the products by selecting a category on the top bar? *

- ◯ Yes
- ◯ No

Were you able to filter the products by the filter side bar? *

- ◯ Yes
- ◯ No

Please rate what you feel in regards to this statement: *

I found the design of the product pages cohesive and comprehensive.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

REGISTER & LOGIN

Click on the 'Account' icon at the header of the application. You should be redirected to the login page.

Were you redirected to the login page? *

- ◯ Yes
- ◯ No

If you try to login with a new, non-registered email address, can you login? *

- ◯ Yes
- ◯ No

After registering with a new account, could you login with that same information? *

- ◯ Yes
- ◯ No

After logging in, were you redirected to the products home page, or the product details page of a product you opened previously? *

- ◯ Yes
- ◯ No

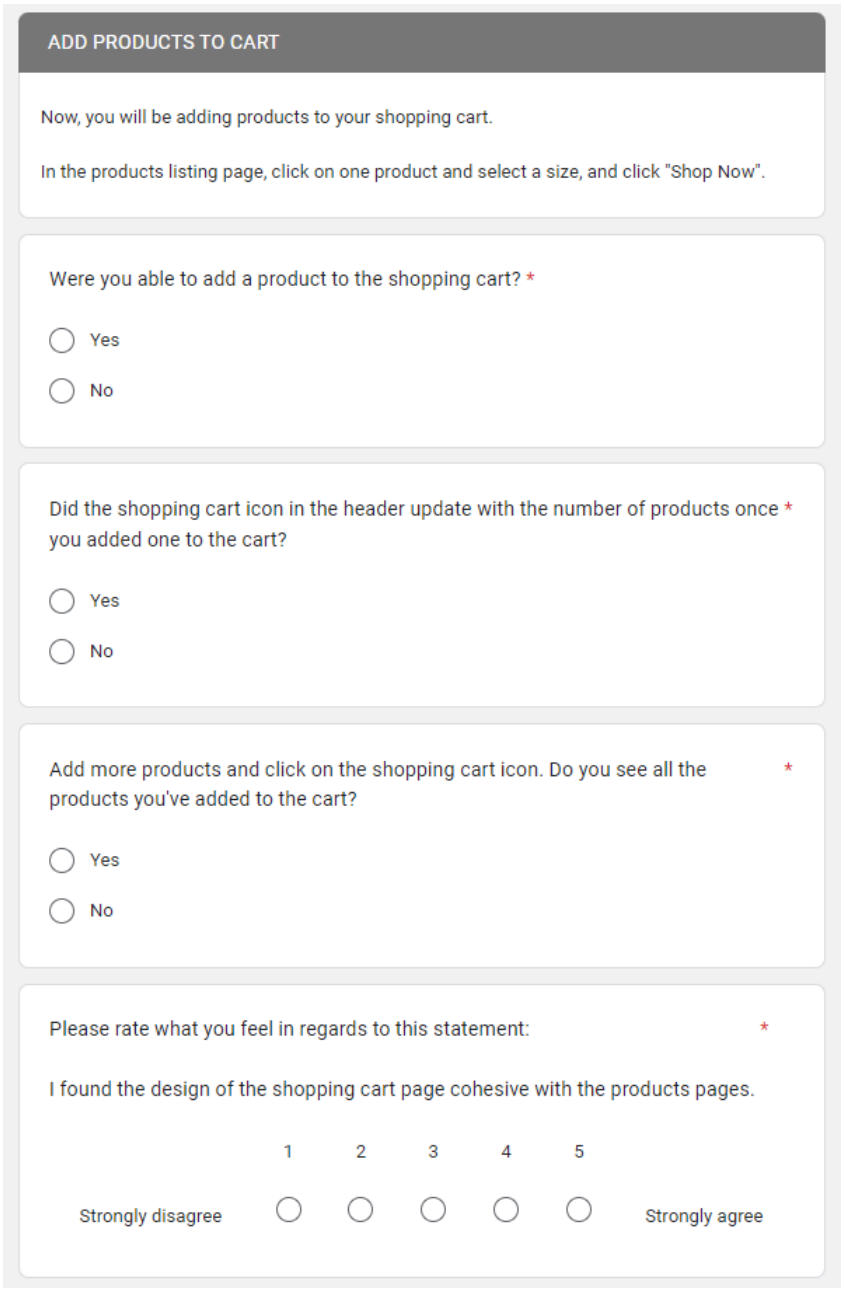
Click on the Account icon on the header of the page. You will be redirected to a mockup page of the Account Details. Could you navigate through them? *

- ◯ Yes
- ◯ No

Please rate what you feel in regards to this statement:

I found the design of the login and register pages cohesive with the products pages.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree



ADD PRODUCTS TO CART

Now, you will be adding products to your shopping cart.

In the products listing page, click on one product and select a size, and click "Shop Now".

Were you able to add a product to the shopping cart? *

- ◯ Yes
- ◯ No

Did the shopping cart icon in the header update with the number of products once you added one to the cart? *

- ◯ Yes
- ◯ No

Add more products and click on the shopping cart icon. Do you see all the products you've added to the cart? *

- ◯ Yes
- ◯ No

Please rate what you feel in regards to this statement: *

I found the design of the shopping cart page cohesive with the products pages.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

CHECKOUT

After adding the products to the shopping cart, you are now ready to proceed to checkout. Click the shopping cart icon again and click "Checkout".

You should be asked to fill in a checkout form with 3 steps. No personal information will be stored in the data base, and you can use random numbers for the "credit card" input. At the end, you can place the order, obviously free of charge.

Were you able to fill the form, and navigate through the "Delivery", "Payment" and "Review" tabs? *

- ◯ Yes
- ◯ No

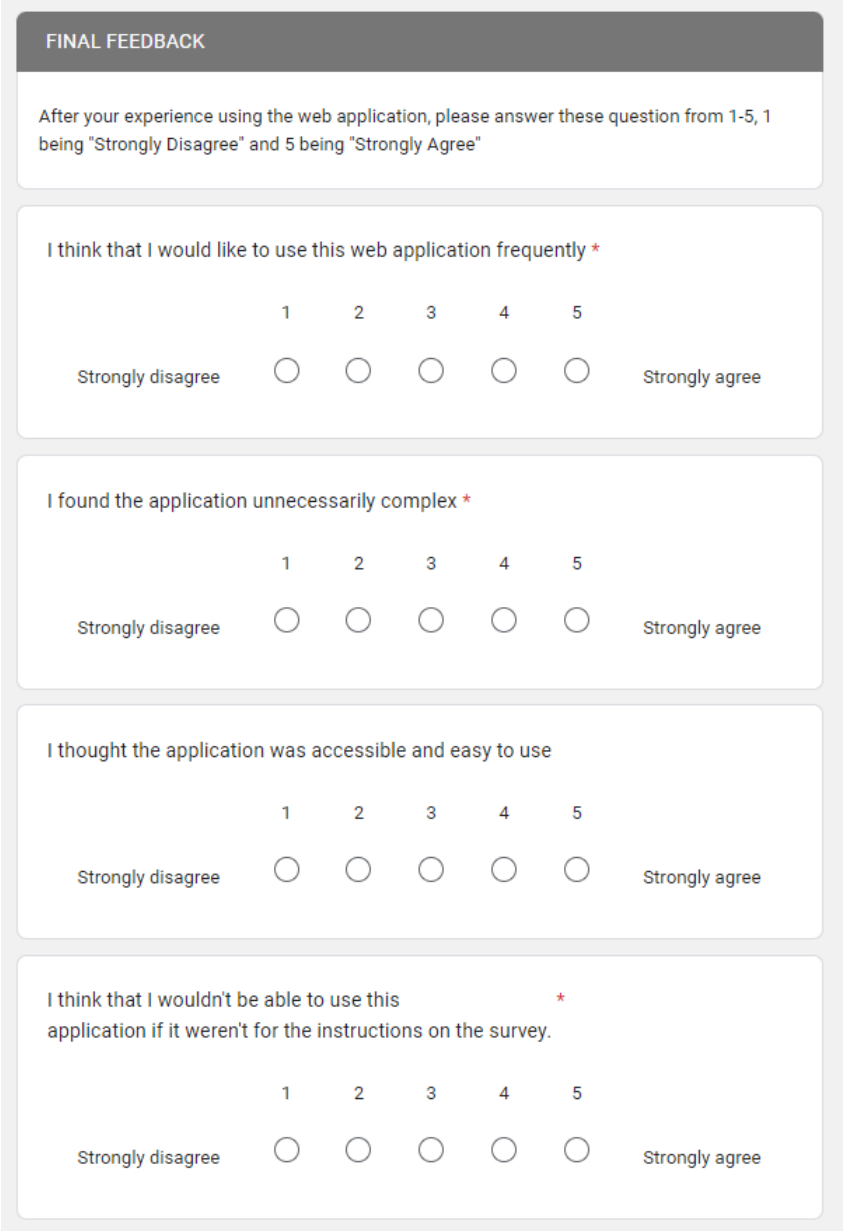
Were you able to complete the order?

- ◯ Yes
- ◯ No

Please rate what you feel in regards to this statement:

I found the design of the checkout pages cohesive with the products and account management pages.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree



FINAL FEEDBACK

After your experience using the web application, please answer these question from 1-5, 1 being "Strongly Disagree" and 5 being "Strongly Agree"

I think that I would like to use this web application frequently *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I found the application unnecessarily complex *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I thought the application was accessible and easy to use

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I think that I wouldn't be able to use this application if it weren't for the instructions on the survey. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I found the various pages and features in this application were well integrated *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I thought there was too much inconsistency in this application's design *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I would imagine that most people would use this application very easily *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I found the application very inconvenient to use *

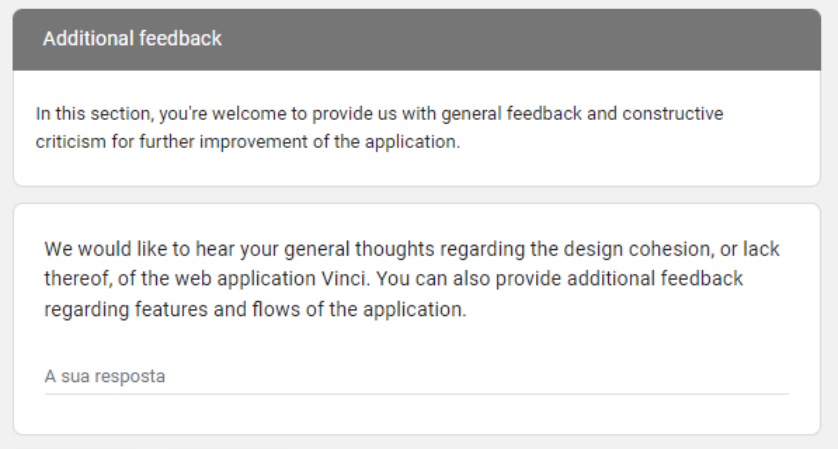
	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I felt confident using the application *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I needed to learn a lot of things before I could use this application *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree


Additional feedback
In this section, you're welcome to provide us with general feedback and constructive criticism for further improvement of the application.
We would like to hear your general thoughts regarding the design cohesion, or lack thereof, of the web application Vinci. You can also provide additional feedback regarding features and flows of the application.
A sua resposta

Appendix E

Feedback Questionnaire Answers

Carimbo de data/hora	Name	Are you over the age of 1	Are you used to working	Do you have any motor c	Were you able to navigat	Were you able to filter th	Were you able to filter th	Please rate what you fee I found the design of the
2024/08/16 18:08:18		Yes	Yes	No	Yes	Yes	Yes	5
2024/08/17 22:49:24		Yes	Yes	No	Yes	Yes	Yes	5
2024/08/17 23:53:40		Yes	Yes	No	Yes	Yes	Yes	5
2024/08/18 07:28:42	André Morais	Yes	Yes	No	Yes	Yes	Yes	4
2024/08/18 10:36:36	Maria Meireles	Yes	Yes	No	Yes	Yes	Yes	4
2024/08/18 17:23:33	André Leal	Yes	Yes	No	Yes	Yes	Yes	4
2024/08/18 21:20:02	Julia Dias	Yes	Yes	No	Yes	Yes	Yes	4
2024/08/18 21:38:02	Miguel Martins	Yes	Yes	No	Yes	Yes	Yes	3
2024/08/18 22:57:19	Bruno Lopes	Yes	Yes	No	Yes	Yes	Yes	5
2024/08/19 16:10:48	Vítor Neto	Yes	Yes	No	Yes	Yes	Yes	5
2024/08/19 17:20:41	Luís Sousa	Yes	Yes	No	Yes	Yes	Yes	5
2024/08/19 20:29:05	Daniela	Yes	Yes	No	Yes	Yes	Yes	5
2024/08/19 20:29:38	André	Yes	Yes	No	Yes	Yes	Yes	5
2024/08/19 21:19:19	Ruben Lopes	Yes	Yes	No	Yes	Yes	Yes	4
2024/08/19 23:27:02	Sara	Yes	Yes	No	Yes	Yes	Yes	3
2024/08/19 23:55:55	Miguel Oliveira	Yes	Yes	No	Yes	Yes	Yes	5
2024/08/20 10:28:20	André Oliveira Gonçalves	Yes	Yes	No	Yes	Yes	Yes	5
2024/08/21 00:50:32	Sofia Canelas	Yes	Yes	No	Yes	Yes	Yes	4
2024/08/21 13:35:36	João Pereira	Yes	Yes	No	Yes	Yes	No	5
2024/08/21 18:19:20		Yes	Yes	No	Yes	Yes	Yes	5
2024/08/21 18:25:28		Yes	Yes	No	Yes	Yes	Yes	5
2024/08/21 18:27:14		Yes	Yes	No	Yes	Yes	Yes	5

If you try to login with a n	After registering with a ne	After logging in, were you	Click on the Account icor	Please rate what you fee I found the design of the	Were you able to add a p	Did the shopping cart ico	Add more products and c	Please rate what you fee I found the design of the
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	4
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	No	Yes	4	Yes	Yes	Yes	2
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	4
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	4
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	5	Yes	Yes	Yes	5
No	Yes	Yes	Yes	4	Yes	Yes	Yes	4
No	Yes	Yes	Yes	4	Yes	Yes	Yes	5

		Please rate what you fee						
Were you able to fill the f	Were you able to comple	I found the design of the	I think that I would like to	I found the application un	I thought the application	I think that I wouldn't be a application if it weren't fo	I found the various pages	I thought there was too m
Yes	Yes	5	5	1	5	1	5	1
Yes	Yes	5	4	1	5	1	5	1
Yes	Yes	5	5	1	5	1	5	1
Yes	Yes	4	1	1	5	1	4	1
Yes	Yes	5	5	1	4	1	5	1
Yes	Yes	5	5	1	5	1	5	1
Yes	Yes	5	5	1	4	1	5	1
Yes	Yes	4	3	3	3	1	3	3
Yes	Yes	5	5	1	5	1	5	1
Yes	Yes	5	5	1	5	1	5	1
Yes	Yes	5	5	1	4	1	5	1
Yes	Yes	5	5	1	5	1	5	1
Yes	Yes	5	4	1	5	1	5	1
Yes	Yes	5	4	1	4	5	5	1
Yes	Yes	5	5	1	5	1	5	1
Yes	Yes	5	5	1	5	1	5	1
Yes	Yes	4	4	1	5	3	4	1
Yes	Yes	5	5	1	5	1	5	1
Yes	Yes	5	4	1	5	1	5	1
Yes	Yes	5	4	1	5	1	5	1
Yes	Yes	4	4	1	5	2	5	2
Yes	Yes	5	4	1	5	1	5	1

AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM
I would imagine that mos	I found the application ve	I felt confident using the	I needed to learn a lot of	We would like to hear your general thoughts regarding the design cohesion, or lack thereof, of the web application Vinci. You can also provide additional feedback regarding featur						
5	1	5	1	Overall the application presents itself in a very cohese structure and modern design, making it very simple and intuitive to use. The responsiveness was also great which is a big p						
4	1	4	1	I feel like the design was cohesive throughout the pages, the account management pages were mockups but I could still get the idea behind them. The data loaded fast and the ap						
5	1	5	1	The cursor hand icon could be displayed in all the clickable area and not only in the name of the product; it could be useful to identify the brand in the listing page; when the categc						
5	1	4	1	In the payment review page, the card icon is placed on top of my card's number. In the product page, when clicking on Wish List button, it will add the product to the cart. The register form allowed me to create an account too insecure (andre).						
5	1	5	1	When I click on a single product from a list, it opens the single product page but the page stays focused on the position it was on the list (or previous page). Example: If I click on a						
5	1	5	1							
5	1	5	1	Some elements like the login form were not 100% accessible for use in keyboard by not submitting the form automatically when pressing enter.						
2	2	2	1	Era fixe os filtros laterais fazerem update sozinhos em vez de eu ter de carregar show results.						
5	1	5	1	having the best sellers shown on the homepage was a nice touch. all pages appeared to stay on brand and to be consistent. the call to action 'shop now' is well positioned. overall a very easy to use website for an online clothing shop.						
5	1	5	1	The application felt very cohesive in its design, felt very natural and easy to learn.						
4	1	5	1	The design was well chosen. It's simple, elegant, accessible and a good colour palette. Also, the web application layout, design and structure is similliar to market competitors whic						
5	1	5	1	It has an elegant and minimalist design. It makes it easy to focus on the products and what we are looking for. It is easy to navigate. Sale items stand out in the listing. Filters work						
4	1	5	1	Only found some inconsistency with the some of the displayed values, ie, delivery fees, but Im guessing the values are still not refined						
4	1	5	1	The Vinci web application offers a cohesive and user-friendly design that enhances the shopping experience. The intuitive layout, consistent design, and easy navigation make bro						
5	1	5	1	I couldn't find the "show results" in the filter, seems to be bugging on the first time opening (if you go back to homepage and try to filter again a category, it doesn't show unless you And I think the "shop now" button should be instead "add to cart", because usually "shop now(buy now?)" redirects to checkout instead of adding to the cart - so just some pointers						
5	5	5	1							
4	1	1	1	Eu acredito que aplicação está muito bem feita em termos de design e programação, mas tenho que ser sincero, eu não intendo porque ao fazer o checkout é preciso submeter o						
5	1	5	1	The application is easy to use (and easy to remember how to use) and its design is very cohesive throughout the whole navigation experience.						
4	1	4	1							
4	1	4	1							
4	1	5	2							
5	1	4	1							

Appendix F

One Sample Wilcoxon T-Test Excel Sheet

Average Design cohesion score	di	ri	si	tf	tf3 - tf	
5	0,25	6	1	11	1320	
5	0,25	6	1			
5	0,25	6	1			
4,5	0,25	6	-1			
4,5	0,25	6	-1			
4,75	ignore					
4,75	ignore					
3,25	1,5	14	-1	1	0	
5	0,25	6	1			
5	0,25	6	1			
4,75	ignore					
5	0,25	6	1			
4,75	ignore					
4,75	ignore					
4,25	0,5	12,5	-1	2	6	
5	0,25	6	1			
4,75	ignore					
4,75	ignore					
5	0,25	6	1			
5	0,25	6	1			
4,25	0,5	12,5	-1			
4,75	ignore					
4,75	ignore					

One-Sample Wilcoxon Signed-Rank Test

example

Hypothesized median:

4,75

Step 1 Determine the absolute difference between each score and the hypothesized median (removing 0 differences)

See column B B2: =IF(OR(A2="";A2=M6);"ignore";ABS(A2-M6)) $d_i = |x_i - \theta|$

Step 2 Rank these differences

See column C C2: =IF(B2="ignore";"";RANK.AVG(B2;B:B;1))

Step 3 Determine the sign for each difference (not the absolute)

See column D D2: =IF(C2="";"";SIGN(A2-M6))

Step 4 Add up all the ranks of scores that had a positive difference

=SUMIF(D:D;1;C:C)

W 54 51 $W = \sum_{i=1, s_i=1}^{n} r_i$

Step 5 Determine the number of nonzero differences

n 14 =COUNT(B:B)

Step 6 Determine the unadjusted variance $VAR = \frac{n \times (n+1) \times (2n+1)}{24}$

VAR 253,75 =J24*(J24+1)*(2*J24+1)/24

Step 7 Determine the number of tied ranks for each unique rank

See column E E2: =IF(C2="";"";COUNTIF(C:C;C2))

Step 8 Cube each of the values from the previous step and subtract it once

See column F F2: =IF(E2="";"";E2^3-E2)

Step 9 Determine the adjustment by summing all the results of the previous step, and divide by 48

adj 27,625 =SUM(F:F)/48 $A = \frac{\sum_{f=1}^{m}(t_f^3 - t_f)}{48}$

Step 10 Determine the adjusted variance

VAR* 226,125 =J28-J39 $VAR_* = VAR - A$

Step 11 Determine the standard error

	SE*	15,03745324	=SQRT(J43)	$SE_* = \sqrt{VAR_*}$
Step 12	Determine D			
	D	52,5	=J24*(J24+1)/4	$D = \frac{n \times (n+1)}{4}$
Step 13	Determine the adjusted test statistic			
	W*	0,099750933(	=(J20-J51)/J47	$W_* = \frac{W - D}{SE_*}$
Step 14	Determine the significance			
	appr. Sig.	0,9205421	=2*(1-NORMSDIST(ABS(J55)))	

Cohesion Scores

	Q1	Q2	Q3	Q4
User1	5	5	5	5
User2	5	5	5	5
User3	5	5	5	5
User4	4	5	5	4
User5	4	5	4	5
User6	4	5	5	5
User7	4	5	5	5
User8	3	4	2	4
User9	5	5	5	5
User10	5	5	5	5
User11	5	5	4	5
User12	5	5	5	5
User13	5	5	4	5
User14	4	5	5	5
User15	3	5	4	5
User16	5	5	5	5
User17	5	5	5	4
User18	4	5	5	5
User19	5	5	5	5
User20	5	5	5	5
User21	5	4	4	4
User22	5	4	5	5
User23	4	5	5	5