Madrid's Polytechnic University

Higher Technical School of Computer Systems Engineering

Master in Web Engineering

Master's final thesis

… SocialQL …

Author

Ramón Morcillo Cascales

Tutor

Jesús Bernal Bermúdez

July 2020

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my mother for the support and the advice she has given me.

To my tutor Jesus Bernal and the other master's teachers for all the knowledge and received throughout the course.

And finally, to Adrián Ferrera González and Uri Goldshtein for all the things I learned from them.

QUOTES

"Dude, sucking at sumthin' is the first step towards being sorta good at something."

Jake the Dog

"We are what we do to change what we are."

Eduardo Galeano

ABSTRACT

Nowadays the Software Development technologies have evolved are very different compared to the ones used in the last decades. This project aims to research and document this evolution of technologies and develop a Social Site Network with the latest and most innovative ones used by the main companies and developers actually.

This is why the SocialQL project consists of creating a Social Site Network with a web application where users can submit posts and interact between them using technologies such as TypeScript, GraphQL, React, Node.js among others.

Being a social platform, it has the characteristic features of this type of application like a user authentication system and system for uploading content by users and interact with the content from other users.

KEYWORDS

Social Site Network, Web application, TypeScript, GraphQL, Apollo, NodeJS, React, PostgreSQL

TABLE OF CONTENTS

Content

MOTIVATION & MAIN PURPOSE

The main motivation to develop SocialQL has been the **desire to learn about new modern software technologies**.

The lately personal researches about modern software libraries and frameworks like ReactJS and GraphQL made the purpose of this thesis to be the understanding how they work and how they are used to **develop modern web applications**.

The reason behind the decision of developing a social network has been to increase the amount of knowledge by not just studying and investigating about the libraries before mentioned but creating a project with them. The choice of the project to be a social site has been based on **the complexity that a social site has itself** and the difficulties that must be handled which will contribute in a deeper acquisition of knowledge.

Another side purpose is a personal one about using all this acquired knowledge in future side projects and to improve my skills on a professional way.

LIST OF FIGURES

INTRODUCTION

In recent decades, the way that web applications and services are developed has undergone unprecedented progress. To have a better understanding of the decision to use the technologies mentioned before it is needed to have a view of the current and past state of the web development. The next pages will go through the history of web development from the very beginning to the actual and future state.

1980 – 1990 Birth of the web

Before the Web could exist, the Internet had to be invented. This means the creation of networks and protocols for transmitting data, in particular the IP (Internet Protocol) and the TCP (Transmission Control Protocol). Then the HTTP, HTML, URL, and Browser would appear.

HTTP. It is the main protocol for the web (Hyper Text Transfer Protocol) which defines how a server should respond to client requests. Users can request data (GET, such as when accessing a website), send data (POST, such as when submitting a form), DELETE (to remove a resource), etc. All these operations are defined in the HTTP protocol.

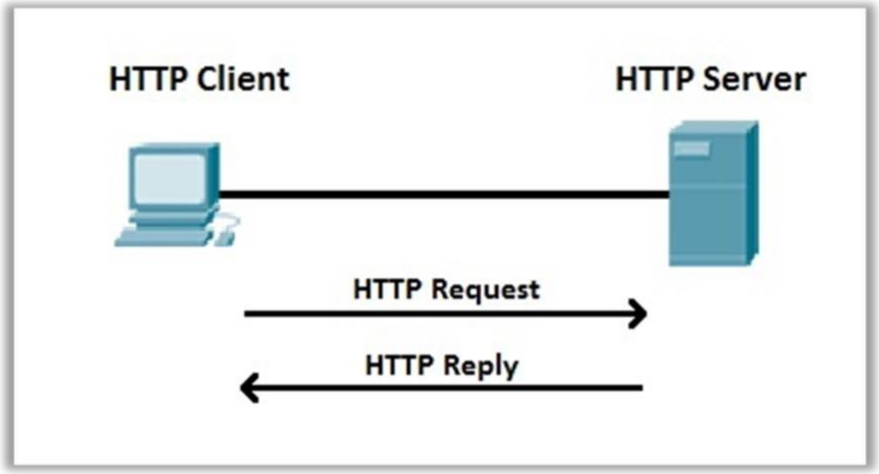


Figure 1 Http process explained.

HTML. Born at the same time as the HTTP protocol. It a language (Hyper Text Markup Language) that describes the structure of a Web page with a series of elements that tell the browser how to display the content. Those elements are represented by **tags** which label pieces of content such as a "heading", "paragraph", "table", and so on.

Figure 2 Hypertext Markup Language.

URI. (Uniform Resource Identifier) Is another protocol defined, which details how resources should be named in order to be found. Some of the decisions made have long been regretted, like the fact that we still maintain the tedious http:// at the beginning of each URL. It is composed of the URL (Uniform Resource Locator) and the URN (Uniform Resource Name).

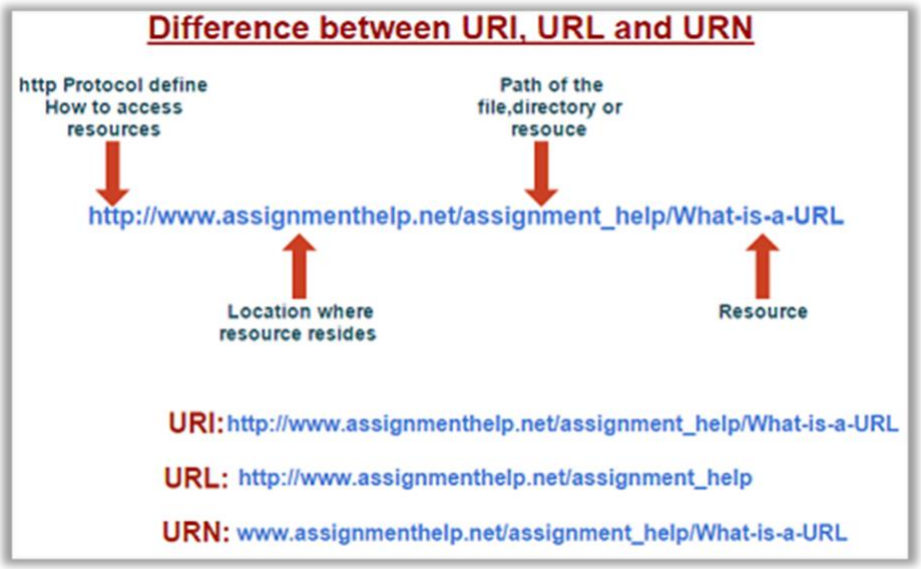


Figure 3 Difference between URI, URL and URN.

Web Browser. It is a software application for accessing the information on the World Wide Web. Appeared from the users' need of a tool to integrate all these protocols: manage URLs, communicate with servers, display HTML, etc.

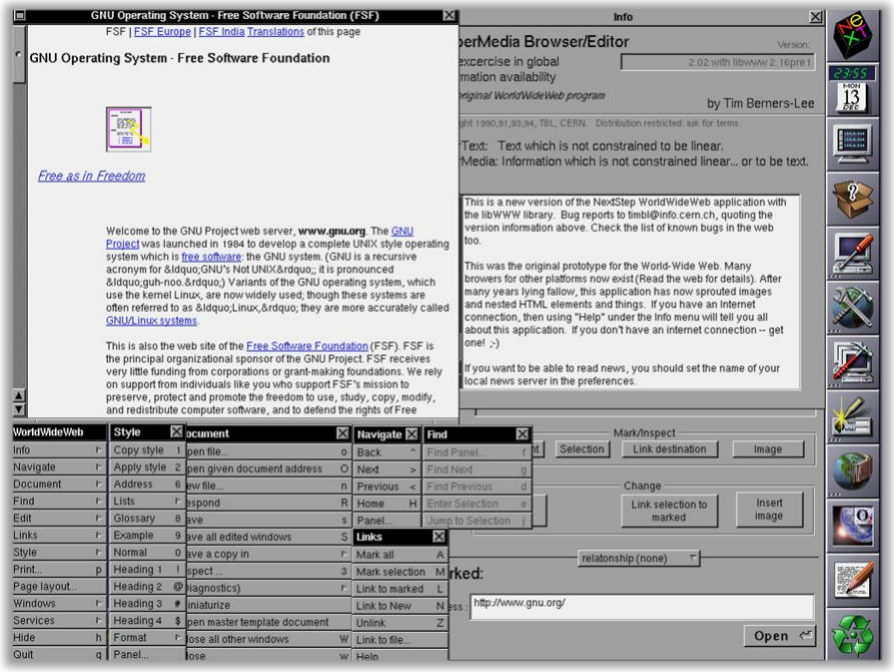


Figure 4 WorldWideWeb, the first web browser.

After developing all of this, Tim Berners-Lee published the first web site, which described the project itself, on 20 December 1990 in CERN, and this technology spread initially to other scientific communities. The site is still available nowadays.

World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
Pointers to the world's online information, subjects , W3 servers, etc.
Help
on the browser you are using
Software Products
A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)
Technical
Details of protocols, formats, program internals etc
Bibliography
Paper documentation on W3 and references.
People
A list of some people involved in the project.
History
A summary of the history of the project.
How can I help ?
If you would like to support the web..
Getting code
Getting the code by anonymous FTP , etc.

Figure 5 The first website.

1990 – 2000 The Static Web

After the first site, the web started getting more and more users, and not just scientific ones, many people surfed the web for the first time from their homes using Netscape or Internet Explorer with Windows 95 as their operating system.

We say the Web was static because its content did not change, the server sent the same HTML files to everyone which contained just text, links, and style: font, size, color, underlining, etc.

So, despite the high growth (1 million websites in 1996), this technology had limitations: the content was static and therefore it was difficult to show different

pages to different users (like when you log in to a service). Also, the style of the sites was poor, and people had to do tricks to enhance it.

Figure 6 Famous sites in the nineties.

2000 – 2010 The Dynamic & Interactive Web

Between 2000 and 2006 there was the explosion of the dynamic web, in which content often changed. A great example is Wikipedia, which appeared in this period, and allowed anyone to edit the pages. Mozilla Firefox appeared to compete with Internet Explorer, and Linux became popular among "human beings" thanks to Ubuntu.

Figure 7 Wikipedia and Firefox logos.

The web styling was separated from the content thanks to CSS (Cascading Style Sheets): with it, you can set colors, sizes, spacing, backgrounds, etc.

As for the server, dynamic sites were mostly based on the LAMP architecture: Linux as the Operative System, Apache as the web server, MySQL for the persistence, and PHP as the programming language. The flow started when a user (client) requested a dynamic page, the web server (Apache) would call a PHP file which communicated itself with MySQL to get the data and generate an HTML output that was handed to Apache, who served it to the client.

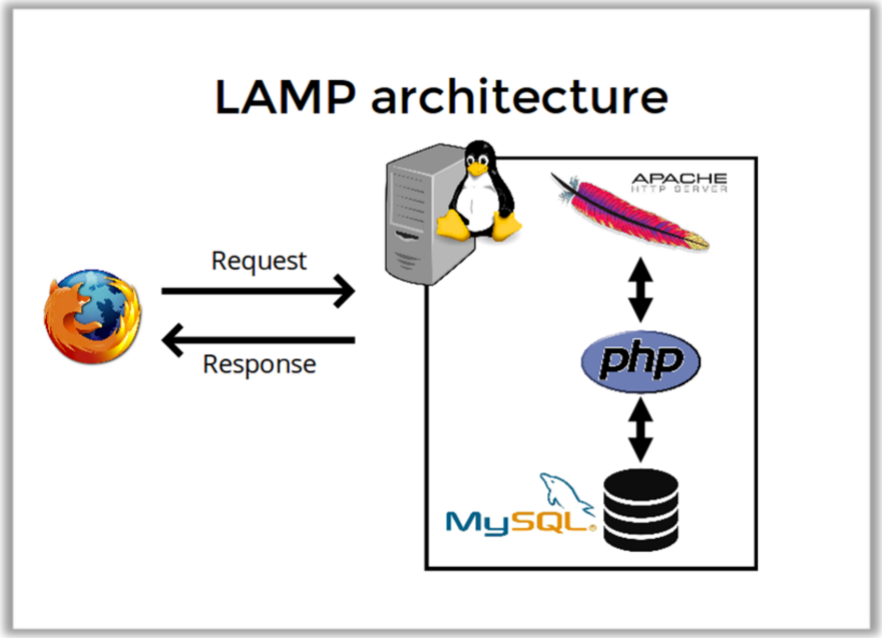


Figure 8 LAMP Architecture.

There were three main options to add dynamism to the web at that time: JavaScript, Adobe Flash, and Java Applets.

JavaScript. Designed in 1995 by Brendan Eich is a programming language that conforms to the ECMAScript specification. JavaScript is high-level, often just-in-time compiled, and multi-paradigm. It has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions. It is a programming language that browsers understand and execute but the problem at that time was that it was very slow, had an unpopular syntax and that meant that very few people would take it seriously.

Adobe Flash. Originally developed by Future Wave Macromedia in 1996 and then by Adobe Systems. It is software used to create and execute multiple contents, including viewing multimedia, rich Internet applications, and streaming audio and video. It is run on a web browser as a browser plug-in or on supported mobile devices. Now it is deprecated, and Adobe announced that it would end support for Flash Player at the end of 2020 but at the time many pages like YouTube and browser games were made with Adobe Flash for many years.

Java Applets. Introduced in the first version of the Java language, which was released in 1995, Java applets were small applications written in the Java programming language. The user launched the Java applet from a web page, and the applet was then executed within a Java virtual machine (JVM) in a process separate from the web browser itself.

But there was also another term these years that which was gaining popularity: AJAX Asynchronous JavaScript and XML.

> *Ajax is a set of web development techniques using many web technologies on the client side to create asynchronous web applications. With Ajax, web applications can send and retrieve data from a server asynchronously without interfering with the display and behavior of the existing page. – Wikipedia*

This basically meant **updating a site's content without needing to refresh the page**.

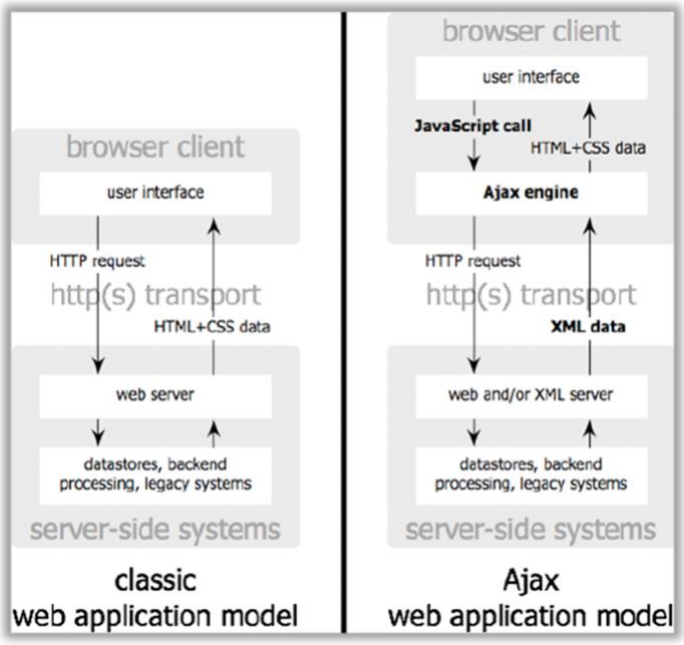


Figure 9 Ajax Model.

Google developed and offered services that integrated AJAX such as Gmail or Google Maps where you could receive and send emails without refreshing the page o browser the planet in the same way.

By that time, most web applications focused on the same: **CRUD**. Create, read, update, or delete data (articles from a blog, forum messages, comments...), manage users, display the content in different languages, process user-sent data from forms, etc. Therefore, a new code organization pattern became remarkably popular: **Model-View-Controller**.

Model: Database Server / Data Layer-persistence. Is responsible for managing the data of the application. It receives user input from the controller.

View: Web browser/client layer. Representation of the model in any kind of format.

Controller: Web Server / Business Logic Layer. The controller responds to the user input and performs interactions on the data model objects.

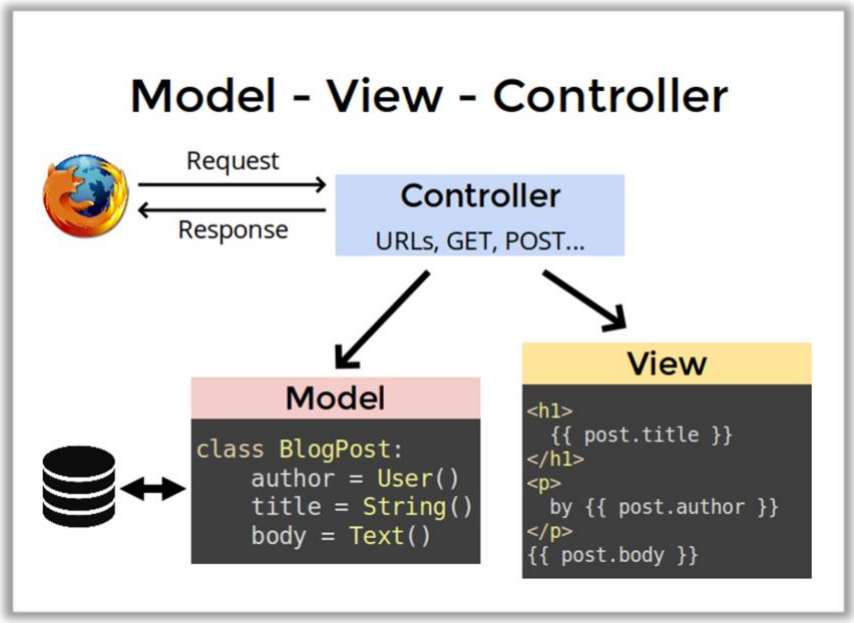


Figure 10 MVC Model-View-Controller.

A lot of frameworks were created following the Model-View-Controller pattern in a lot of programming languages providing a lot of tools commonly used in web applications. These frameworks allowed the developer to focus on its current project logic, and not on boring and repetitive tasks or configurations. Some of these frameworks are still used nowadays like Symphony (PHP), Django (python) Ruby on Rails (Ruby), Spring (Java). These frameworks added a new responsibility for the Controller: handling the initial HTTP request.

The Controller is now the entry point into the application, rather than the View. The responsibility of the View also changed: instead of presenting something to the user directly and handling input, its job was to assemble a bundle of HTML, JS, and CSS for the browser to render. The HTML/JS would contain logic like button click handlers that would dispatch an action back to the controller via an XMLHttpRequest. Notice that there is no significant presence of the MVC pattern within the browser. That would soon change with the advent of **Modern Web Frameworks**.

2010 – 2020 Modern Web Frameworks

We reached years, where smartphones started to be the main access to the web and pages, had to be designed to be responsive, with a **Mobile First** approach, and have a nice performance being fast and lightweight.

JavaScript was reborn, Google developed an open-source high-performance JavaScript engine for Chrome and Chromium-based browsers called **V8**. With a faster JavaScript and making use of new features like the **DOM API, ES6 and AJAX** companies began building more and more complicated web apps (sometimes called Single Page Applications- SPA).

Now the browser client has more responsibility with these Single Page Applications which include the logic (in JavaScript) for making HTTP requests against a set of resources served by "API Controllers", which usually respond with JSON:

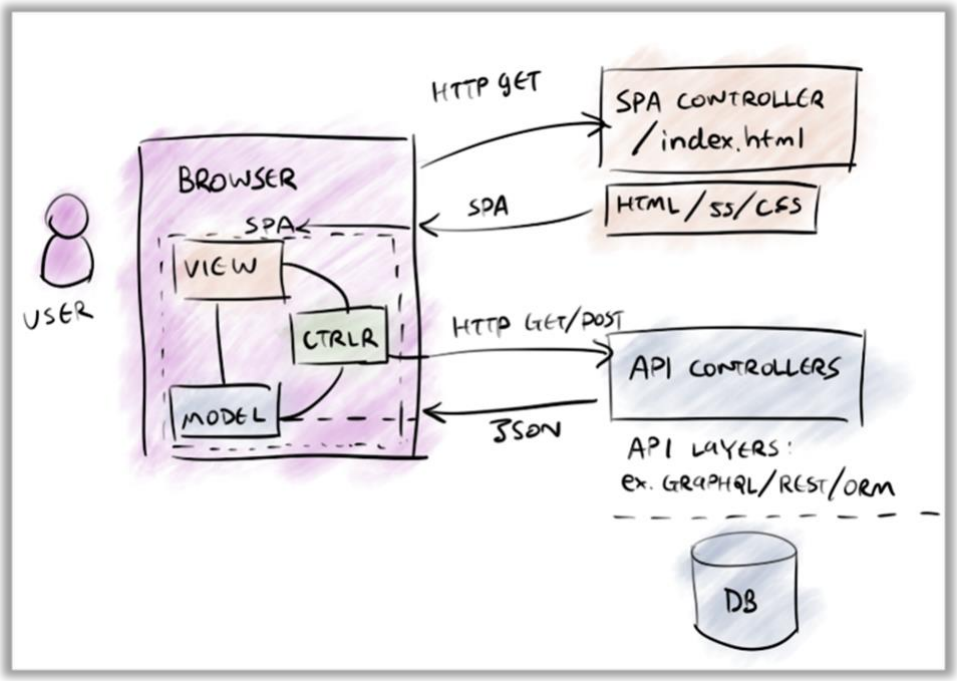


Figure 11 Modern Web Frameworks.

A lot of frameworks appeared and continue appearing today, the most popular ones at this moment would be: **React**, **Vue**, and **Angular**. I will delve into them later when I analyze the State of the Art.

These JavaScript improvements not only made a huge impact on the client-side but also on the server one. Thanks to the V8 engine, **Node.js** appeared, **a JavaScript runtime**

built on Chrome's V8 JavaScript engine. This way JavaScript started being used outside the browser on the server and desktop.

Speaking of the server-side a new software development approach appeared: **Microservices**. Old monolithic applications running in one machine were arranged as a collection of loosely coupled services that communicate with each other. These microservices and web service APIs that adhere to the REST architectural constraints are called RESTful APIs.

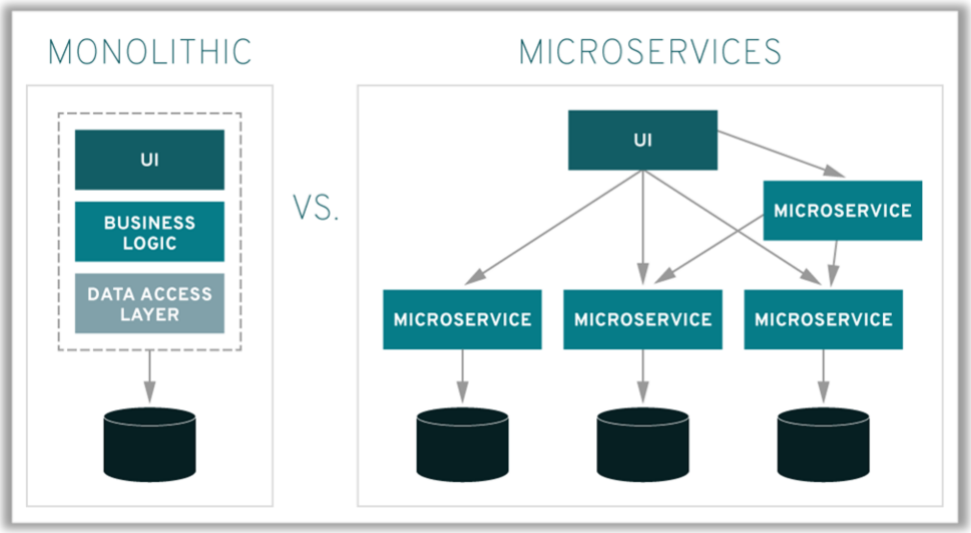


Figure 12 Monolithic vs Microservices.

HTTP-based RESTful APIs are defined with a base URI, such as *http://api.example.com/collection/* and standard HTTP methods (e.g., GET, POST, PUT, PATCH and DELETE). Using these methods, endpoints are created for the client to perform the requests.

GraphQL introduction

In 2015 **GraphQL** entered the game. GraphQL is a query language for APIs developed and open-sourced by Facebook to speed the request process. While RESTful APIs had been a popular way to expose data from a server, instead of having multiple endpoints that return fixed data structures, GraphQL just has **a single endpoint** and it is the client's job to specify what data needs from it.

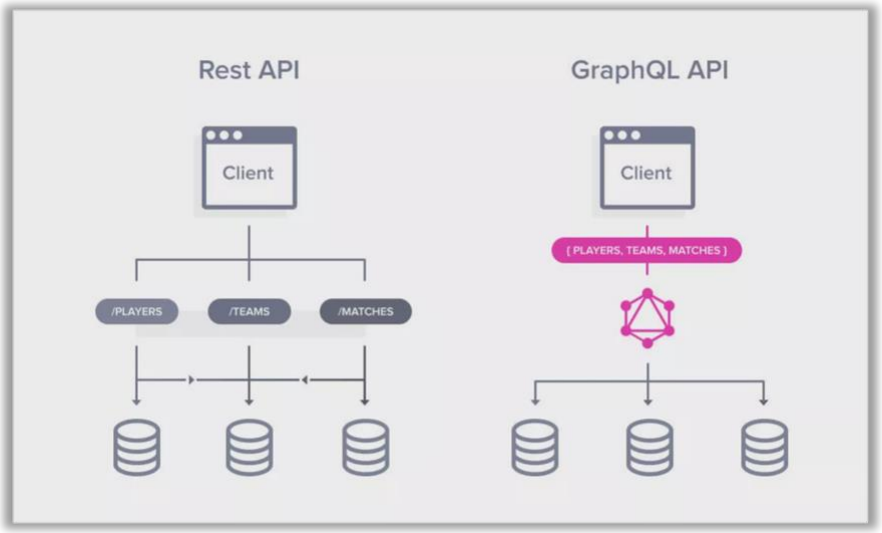


Figure 13 Rest API vs GraphQL API.

When the REST was developed, client applications were relatively simple, and the development pace was not like today's. REST thus was a good fit for many applications but the API landscape has radically changed over the last couple of years by mainly three equally important factors that have been challenging the way APIs are designed:

- Efficient data loading for **increased mobile usage**.

Facebook developed GraphQL focusing on low-powered devices and slow internet connections. GraphQL minimizes the amount of data that needs to be transferred over the network which majorly improves applications operating under these conditions.

- Huge **variety of frontend frameworks** and platforms.

Nowadays the landscape of frontend frameworks and platforms is so wide that it makes it difficult to build and maintain an API that would fit the requirements of all. With GraphQL, each client can precisely access the data it needs without having to specify endpoints for each one of the requests it has to perform.

- **Fast development & need to adapt** to new features.

Continuous deployment and integration are now the standards for many companies where rapid iterations and frequent product updates are indispensable. With REST APIs, the way data is exposed by the server often needs to be modified to account for specific requirements and design changes on the client-side which makes it difficult to follow fast development practices and product iterations.

The fact that GraphQL can be used everywhere a client communicates with an API has made the **community behind it to grow rapidly**. Today, GraphQL is used in production by lots of different companies such as GitHub, Twitter, Yelp, and Shopify - to name a few. In fact, companies like Netflix or Coursera were working on similar ideas. Coursera technology allowed a client to specify its data requirements and Netflix even open-sourced their solution called *Falcor*.

TypeScript introduction

Recently another language is gaining a lot of attention: **TypeScript**, or **TS** (TypeScript official abbreviation) which is nothing more than **modern JavaScript** with the addition of **static type system**. It is a **compiled language** that's default compilation target is JavaScript. TypeScript is an open-source and started as a project by Microsoft in 2012.

Static typing in TypeScript makes it possible to know the type of the variables at the compile time. JavaScript, on the other hand, is an interpreted language with a **dynamic type system**, and while it has its advantages like being **not having to specify your types** in your code directly, static typing brings better error-proneness, and usually much finer IDE support and tooling which greatly improve your coding experience.

VIABILITY STUDY

Before starting the development of the project, it has been important an analysis of the objectives of it to know if these are persistent or necessary, in other words, know if it is feasible to carry out the project. For this, the **SWOT** analysis methodology will be used, and a Risks analysis will be performed too to be aware of the ones that would be faced along with the project and design a management plan in case of facing them.

SWOT analysis

SWOT analysis consists of studying the situation of a project, analyzing the **internal** characteristics (Strengths and Weaknesses) along with the **external** ones (Opportunities and Threats). These analyses are placed in a square matrix for a better view of all the characteristics.

The name is an acronym for the four examined parameters:

- **S**trengths: Internal project characteristics that give it an advantage over others.
- **W**eaknesses Internal project characteristics that place the project at a disadvantage relative to others.
- **O**pportunities: External elements in the environment that the project could exploit to its advantage.
- **T**hreats: External elements in the environment that could cause trouble for the project.

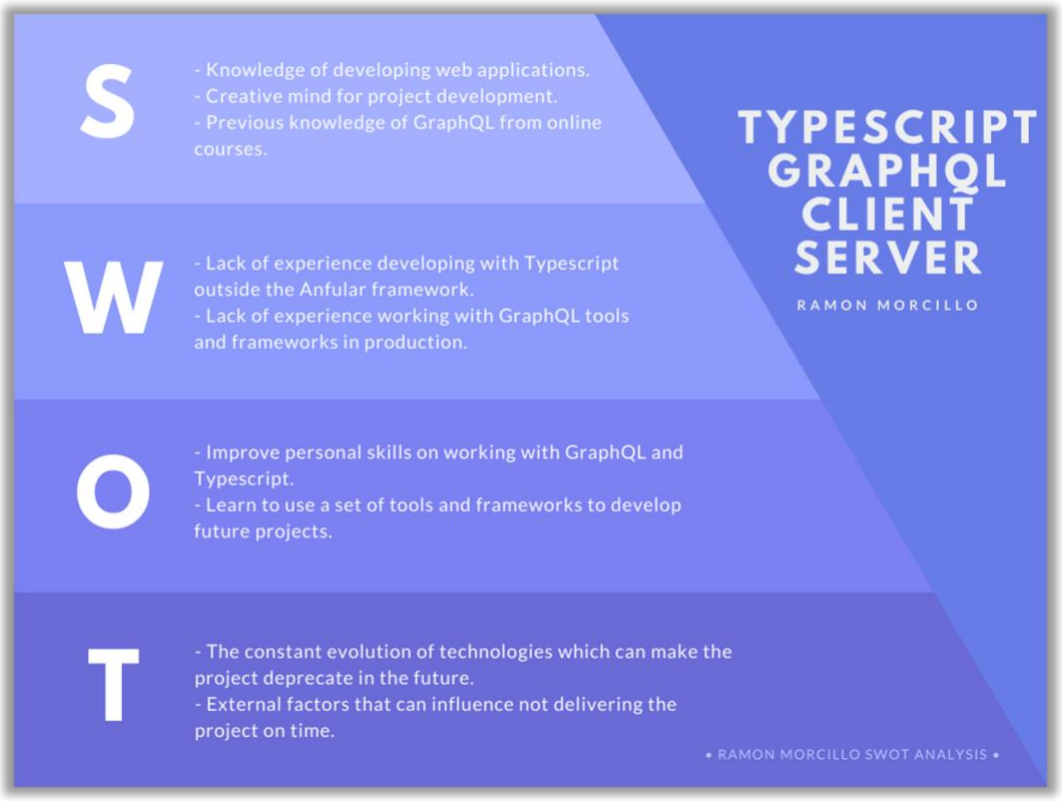


Figure 14 SWOT Analysis.

First, there are the strengths, our strong advantages. The first is the **knowledge on how to develop web applications** thanks to what has been learned over the years in the career as a software engineer. The assumption is done that this knowledge will improve during the development of the project.

Another of the main strengths is the consideration of **being a creative person**, with an open and imaginative mind. This quality can help in times of taking hard choices and speed some creative process like the design of the interfaces.

The last strength to highlight would be the fact of having a **previous self-taught knowledge of GraphQL** from online courses or side projects on spare time. So, despite this is the first serious project where this technology will be used, this strength will help reduce its learning curve.

Regarding weaknesses, the most important is the **lack of experience developing with TypeScript** outside the Angular Framework. This will involve an extra investment of time

on fine-tuning the working environment with TypeScript, although given the importance of this Language in today's world the knowledge acquired at the end of the project will be very useful in the world of work.

The other existing weakness, similar to the above, is the **lack of experience working with GraphQL tools and frameworks in a production environment**. Although it has been seen in online courses and tested, no major project has ever been carried out with it.

Third, the opportunities. This project will provide the **opportunity to improve personal skills on GraphQL, TypeScript, React and other technologies from modern Full Stack Development**. It will also provide other people a guide to help them create a modern application with the same or similar stack.

The other opportunity is to **learn to use a set of tools and frameworks to develop future projects**. It is expected that after learning how to use the technologies from the project it will be easier to start new ones with them.

Finally, we have threats. Those external factors to the project that may involve a danger to this. The first of them is the **constant evolution of technologies which can make the project deprecate in the future.** There will be a moment when this project will be no longer updated with the current technologies of the time because other ones will be used more which happens to almost every software soon or later when other technologies appear with more advantages that make using them produce more maintainable software. The thing is that if the technologies stack is not chosen right or the software is not built the most maintainable way possible **It will be deprecated and replaced by new technologies sooner than expected.**

The last threat is related to external factors that can influence **not delivering the project on time**. This thread is related to all those threads that can reduce the time invested in the project and result in not achieving the Minimum Viable Product to deliver.

Risks Analysis

The main risk to be faced, as previously discussed in the weaknesses, will be the **lack of experience** when developing with the stack of technologies mentioned before (React, GraphQL, and TypeScript). This means that there is a risk of not acquiring the necessary knowledge adequately to develop the project.

Another real risk, as mentioned before too, is the one of **not reaching the objective within the proposed time limit**. This risk may depend directly on the previous one since the lack of knowledge and experience will mean dedicating time prior to personal research and learning in those technologies.

Finally, the risks that do not depend on the project must be taken into accounts, such as unforeseen personal situations, illnesses, or increased work in my workplace. Those are **risks that would hinder the planned deadlines and generate delays**.

After analyzing the risks, it has been concluded that certain measures must be taken to avoid delays or at least try to minimize and contain them. The first of the measures and the most important one is the **planning and organization** of the project. The planning must be objective and accord with the capacities and time available. Objectives will be classified and ranked by priority to tackle the most important ones first.

STATE OF THE ART

The State of The Art (SoTA) is part of the description and comparison of the different technologies used nowadays to build a web application. It will be focused on the JavaScript ecosystem. This part will not only demonstrate the novelty of my project but also will accomplish other important objectives:

- It will teach a lot about my project problems by reading and researching them which will end in learning from other developers and will make it easier to understand and analyze the project.
- It will prove that the project has relevance. If many people are trying to develop a similar project, and if it can be demonstrated in the SoTA, that will mean that the pretended project to be done is important.
- It shows different approaches to a solution. By seeing many different approaches taken in other projects, the approach will be evaluated and realize its novelty (or lack of it) easily. I will also show which approaches are the most popular and which are dead ends.
- Will show what can be reused from what others have done. Especially when doing researching new software, it is amazing how many people could be making similar software by simply searching on places like GitHub.

The main problem to solve or objective to achieve of the project is to develop a modern web application with GraphQL. That is why the research in this section will be focused on technologies about this topic. To structure this section is the easiest to comprehend the way it will start with a SoTA of modern web development in general. Then the SoTA of GraphQL, including technologies outside the JavaScript Ecosystem. Next, a start will be given of the JavaScript Ecosystem to develop modern web applications like the project one. Finally, The SoTA of GraphQL on the JavaScript Ecosystem. After this, the SoTA in Similar Projects and in the end the Technologies Chosen.

Modern Web Development

Nowadays the range of possibilities to build web applications is so wide in terms of tools, methodologies, frameworks that there is no standard for it. Therefore, this section will be divided into the three most common sections any project or web application has nowadays: A Frontend and Backend.

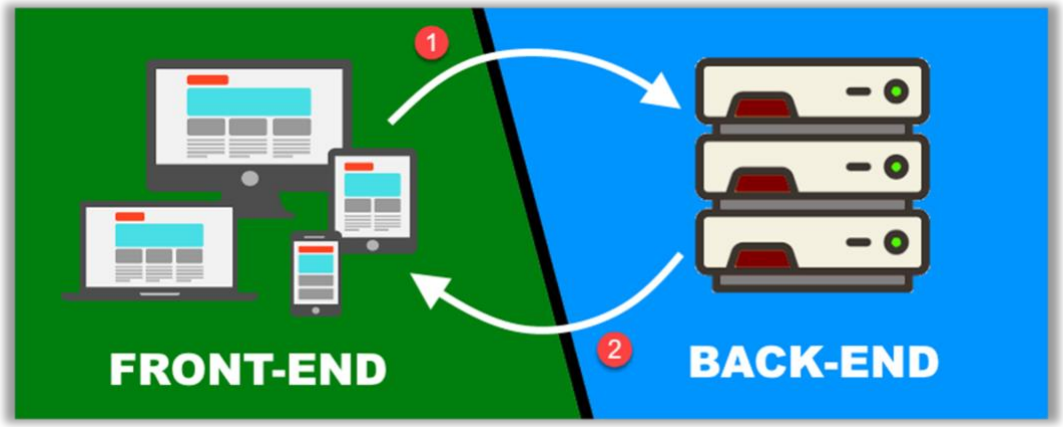


Figure 15 Frontend vs Backend.

Frontend

It refers to **the client part**, the one that our users will see and interact with. This interaction can be done with a mobile, desktop, browser, even a TV depending on the specifications of the application.

Here, depending on the language and framework adopted the technologies range is huge. For mobile frontends, the most popular ones are Android for Java and Kotlin, iOS for swift, and React Native for building native apps in both Android and iOS.

In the browser's world, it depends on the kind of website we are building, but it is commonly used a **Javascript framework** like React, Vue, or Angular. Although a new wave of leaner frameworks and application compilers may change the status quo over the next few years.

Backend

The backend part of a project is the one that the client does not see, the servers, databases, API services, etc. In this field, there are a lot of web frameworks because of the diversity of languages. The tool Wappalyzer will be used in this section to represent the total of the percentages of the Backend web frameworks and languages used on the web. This tool has access to the websites' technology stacks and collects the information from all of them to perform statistics. Here are the top 10 of the web frameworks used in the backend:

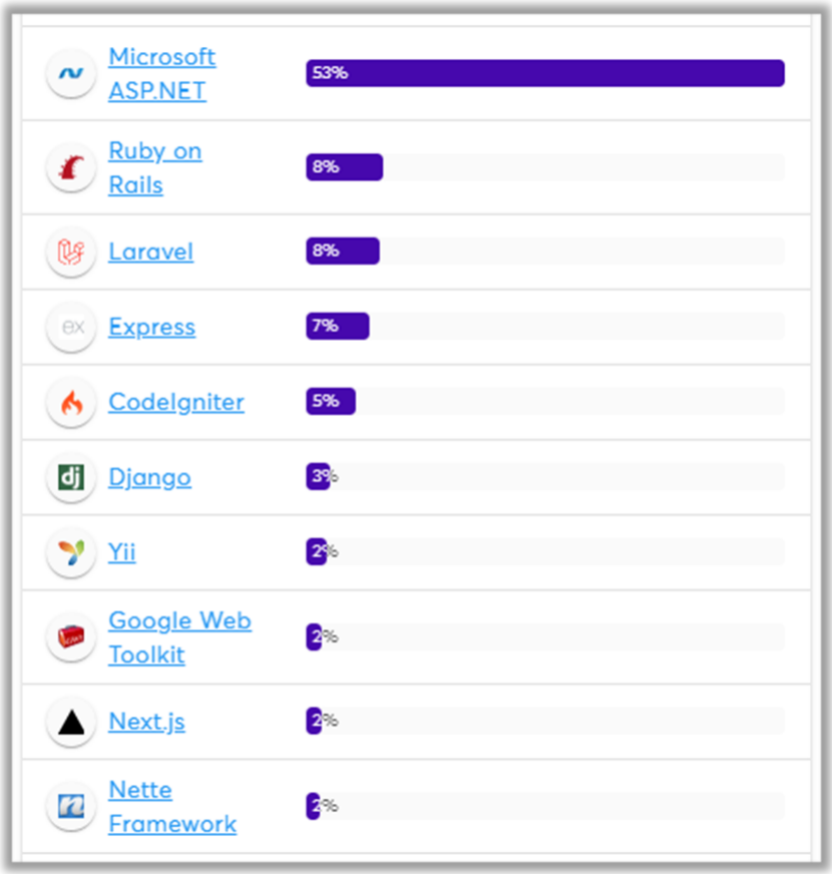


Figure 16 Top 10 Backend web frameworks.

But no matter what is built, if it is a blog, a CMS, an app, we can confirm that **the majority of the websites nowadays have PHP** behind them due to the popularity of this language in the 2000s, which caused that a lot of PHP frameworks emerged and were

used even to this day like **WordPress**, **Wikimedia**, **Moodle**, etc. As it can be seen non the top 10 languages chart:

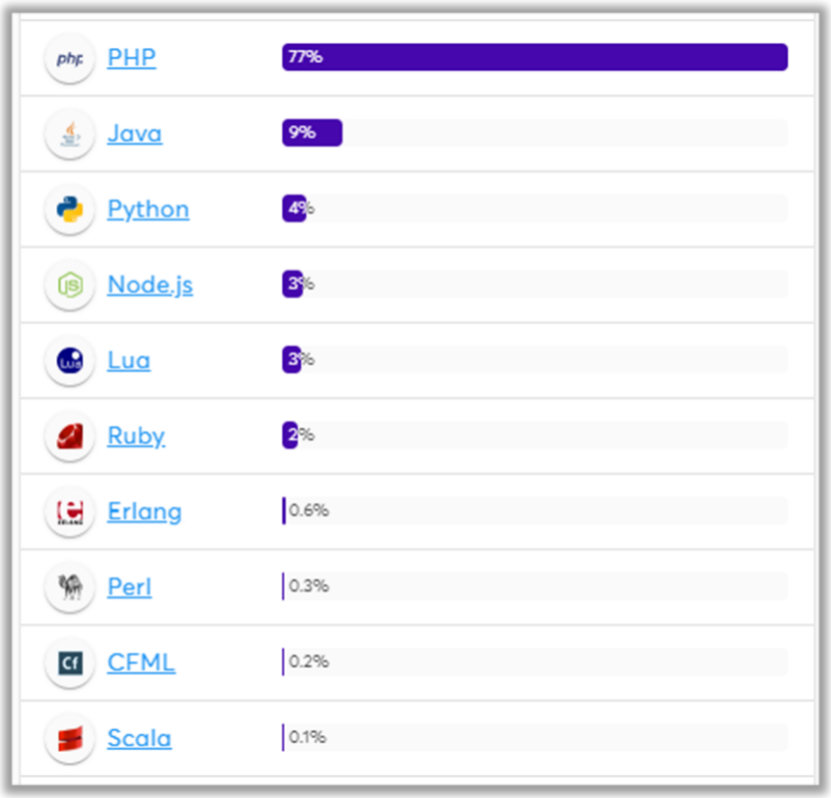


Figure 17 Top 10 Web languages.

GraphQL in Web Development

A lot of tools and libraries have been developed to grant the use of GraphQL in the different programming languages and frameworks described above for web development. The following list names some of the more popular server-side frameworks and client libraries:

- graphql-java. A library for people who want to create a GraphQL server in **Java**. It requires some Spring Boot and Java knowledge.
- graphql-dotnet. Which is an implementation of Facebook's GraphQL in **.NET** being developed by the GraphQL Foundation. The project uses a lexer/parser originally written by Marek Magdziak and released with an MIT license.

- Graphene. Graphene-Python is a library for building GraphQL APIs in **Python** easily, its main goal is to provide a simple but extendable API for making developers' lives easier.
- For **PHP** there are mainly 3: graphql-php which is based on the reference implementation in JavaScript, Lighthouse, a GraphQL framework that integrates with Laravel applications and wp-graphql a plugin that brings the power of GraphQL to WordPress.
- **Javascript and Typescript** also have 3 main libraries: GraphQL.js a reference implementation of GraphQL for JavaScript, express-graphql to create a GraphQL HTTP server with Express and apollo-server which is a server for Express, Connect, Hapi, Koa and others.
- graphql-ruby A **Ruby** implementation of GraphQL.

Javascript Ecosystem in modern web development

Given how much the JavaScript and web ecosystem have grown, it is no longer practical to refer to it as a client scripting language like it was explained in the introduction. It is now used in both Frontend and Backend **and it is a required language if you want to develop modern web applications** because soon or later you will face it and will have to work with it.

TypeScript has graduated to late majority status and is by far the most widely adopted JavaScript variant that has made substantial progress over the past few years, and most JavaScript frameworks now leverage its tooling and infrastructure.

Frontend

In this section, an analysis will be performed on the most used JS frameworks and libraries for frontend development. The tool npm trends will be used which allows us to compare Node.js packages popularity and stats over time. Starting with the three most used frameworks: **React**, **Angular**, and **Vue**.

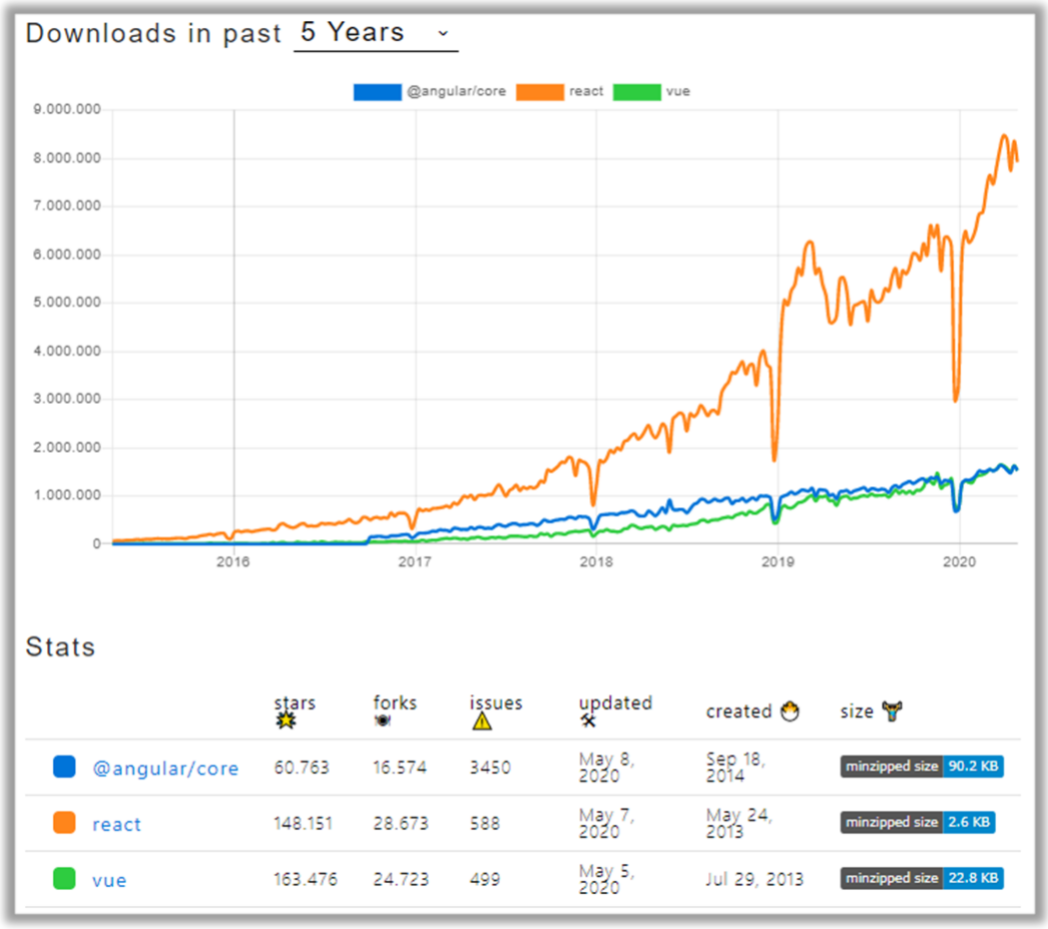


Figure 18 React, Angular, and Vue on npm trends.

This figure shows the package download counts and other metrics over the last 5 years of the three along with the libraries GitHub Stars.

Vue had the top star count with 163k+, showing how it's increasing in popularity. For the first time, it surpassed React in the ratings. However, GitHub stars aren't the only way to understand JS frameworks. While they give a sense of the trend, other statistics suggest a fuller picture of these top three frameworks.

Unlike GitHub stars, the number of downloads shows React in the lead in terms of the sheer bulk of use. These downloads are a good indicator of what developers are actually using, instead of hot trends. Also, the 2020 Stack Overflow Developer Survey also measured what frameworks developers love verses which are most wanted. Like GitHub stars, this shows how developers feel about these frameworks. Here is what

percentage of developers who are developing with the language or technology and have expressed interest in continuing to develop with it.

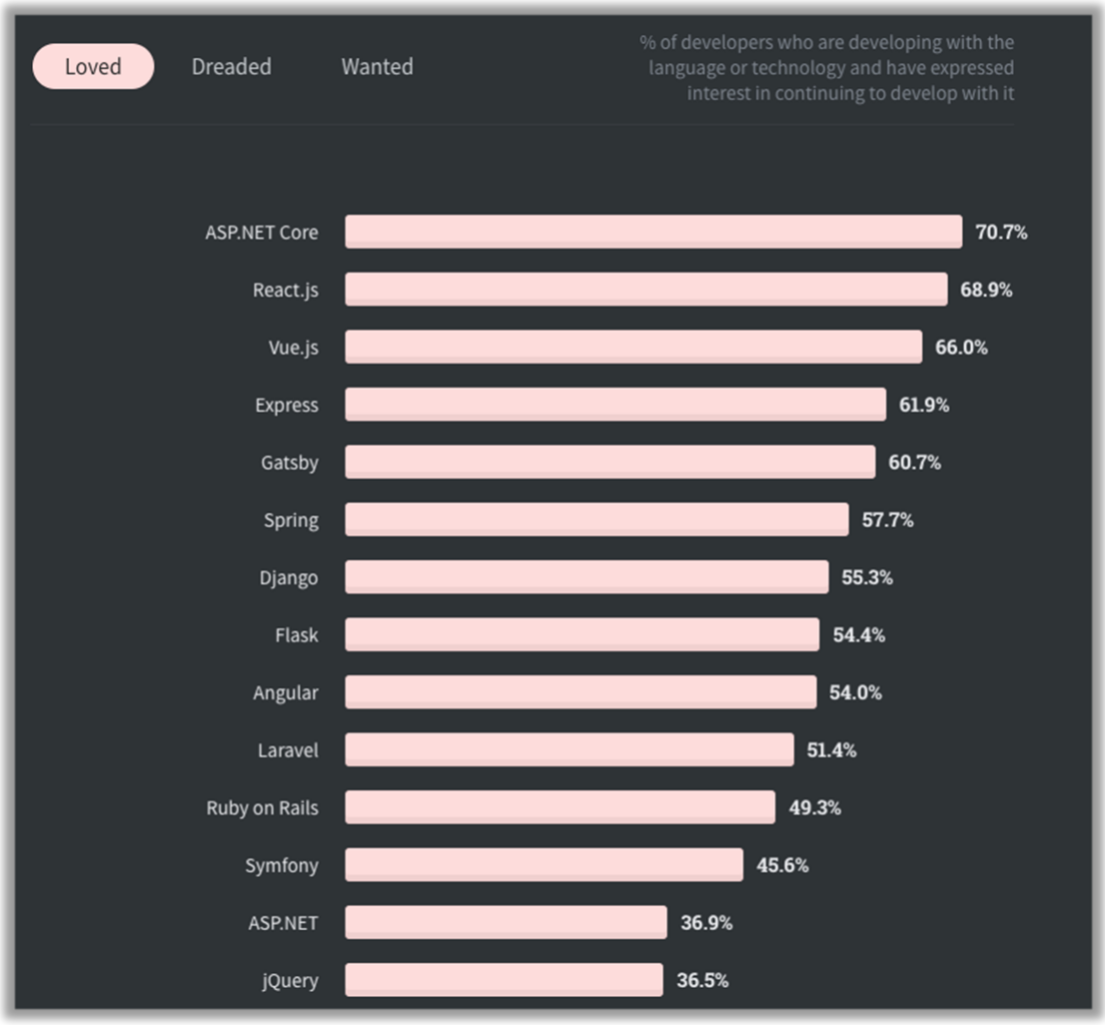


Figure 19 Stackoverflow 2020 survey most loved web frameworks.

Between the 3 discussed frameworks React and Vue is highly above Angular in this figure. Now the % of developers who are developing with the language or technology but have not expressed interest in continuing to do so:

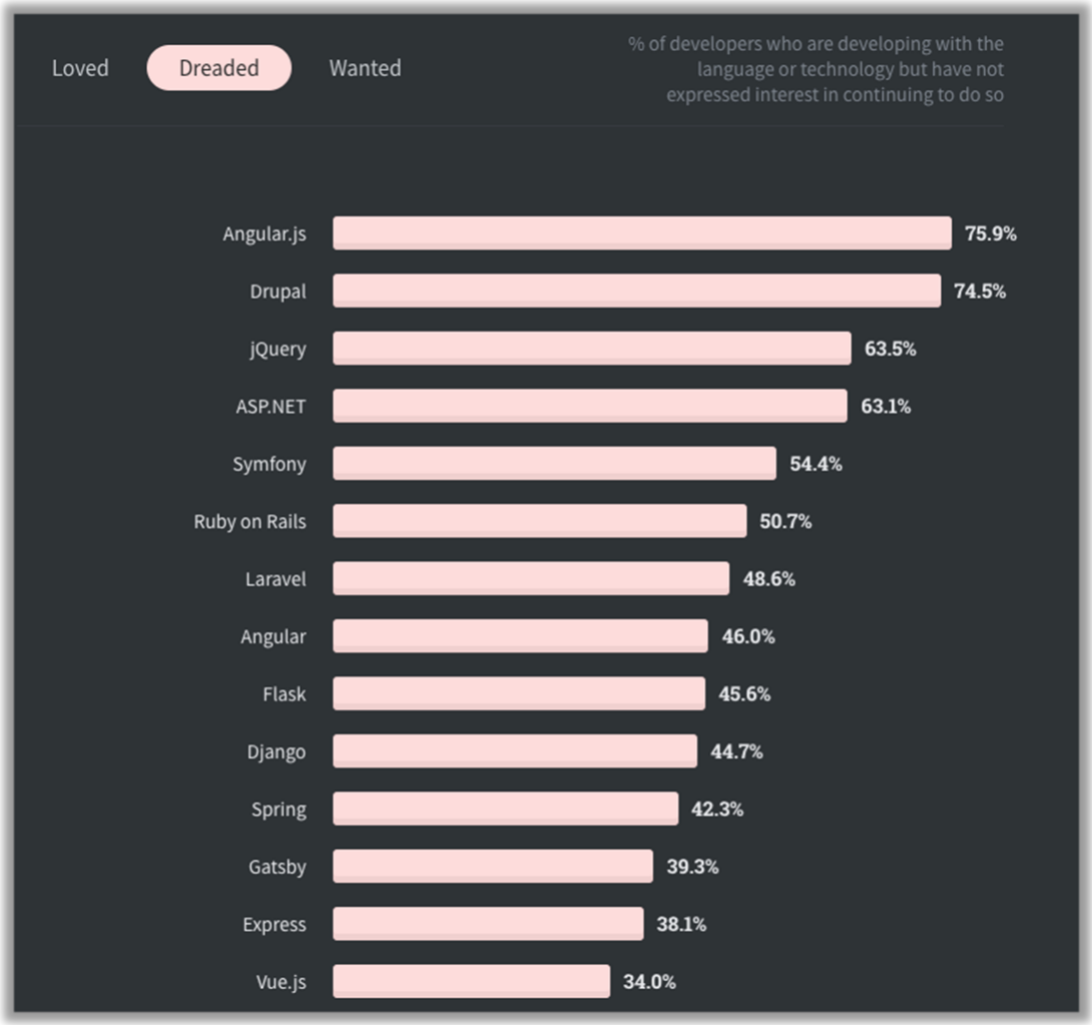


Figure 20 Stack overflow 2020 survey most dreaded web frameworks.

Now it is Angular the one that appears on the top from the 3. Not only Angular.js, the v1 of this framework but its current version is also the one from the 3 that developers are not interested in working with. Finally, the most wanted ones:

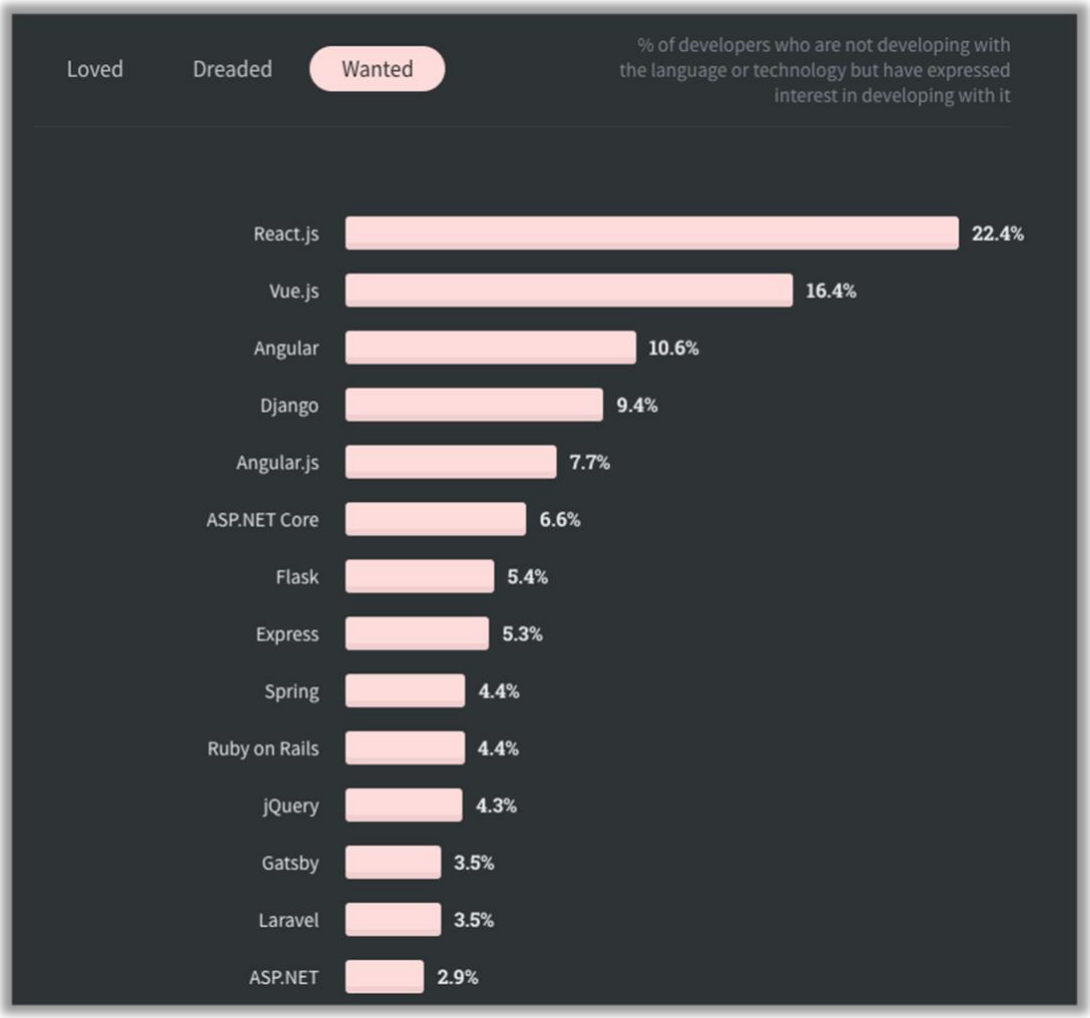


Figure 21 Stack overflow 2020 survey most wanted web frameworks.

Here, the % of developers who are not developing with the language or technology but have expressed interest in developing with it shows how React is the favorite one.

Backend

Since the Node.js appearance JavaScript has been used in the backend ecosystem and dozens upon dozens of frameworks have cropped up, giving developers a wide arsenal of tools to create dynamic, powerful web applications using JavaScript.

Node is a popular choice for building backends in large part due to the fact that it shares the same base language, JavaScript, as many front-ends do. This allows

developers and teams to contribute, share, and reuse code on the stack, and manage entire projects with a single package manager (such as **NPM**).

In addition to the obvious congruences in the code stack, Node backends are known for their event-driven, non-blocking model that allows simultaneous data processing at very high speeds. It is for this reason that Node frameworks have become the back-end of choice for microservers that require flexible and highly scalable data processing.

Some popular Node.js backend frameworks include **Express**, **Sails**, and **Koa**. These are, of course, just a few of the dozens of frames used in current production, and as is the case throughout the JavaScript community, the landscape for frames has never been so diverse.

- **Express**, one of the oldest and most widely used frameworks, has become a child of Node.js. It is a component of the MEAN software stack that includes the MongoDB database platform and the Angular front-end framework.

- **Sails** was built on Express and provides a more traditional **MVC** framework in the style of other familiar backends like Rails.

- **Koa**, created by the same team behind Express, is rapidly gaining ground as a lighter framework for building minimal interfaces using asynchronous functions on callbacks.

GraphQL in the JavaScript Ecosystem

GraphQL is an open specification, and there are indeed other implementations in the realm of JavaScript (like the Apollo Server) and other languages, each with their own characteristics and strengths. However, the **GraphQL.js** client that explores this piece undoubtedly acts as the best entry point to the query language and establishes a strong fundamental understanding that can then be extended with other clients.

All these benefits that GraphQL offers over REST APIs have made it an attractive option for JavaScript frameworks like React, which bases much of its upcoming Concurrent Mode functionality on GraphQL, along with the GraphQL client for the framework. Since React as seen before is the most popular JavaScript framework, this will have important implications with the adoption of GraphQL for the foreseeable future.

Those benefits have made that a GraphQL service can be introduced to an application without interruption, due to two of its features:

- **GraphQL services are decoupled on the client-side and reside on the server one**. A GraphQL service can be developed in isolation and run alongside its traditional API services with a single endpoint: commonly "/graphql". It will not interfere with the rest of your application ecosystem.
- **GraphQL queries are done with fetch () requests**: no additional middleware or specialized packages are required to use GraphQL. In fact, GraphQL queries can simply be done with cURL requests in Terminal if you choose to do so, with support for both GET and POST requests.

Some of the frameworks used to implement GraphQL on JavaScript are:

- **GraphQL.js.** The reference implementation of the GraphQL specification, designed to run GraphQL in a Node.js environment.
- **express-graphql.** The reference implementation of a GraphQL API server on an Express server. You can use this to run GraphQL in conjunction with a regular Express web server, or as a standalone GraphQL server.
- **Apollo.** a platform for building a data graph, a communication layer that seamlessly connects your application clients (such as React and iOS apps) to your back-end services. Is an implementation of GraphQL designed for the needs of product engineering teams building modern, data-driven applications. It's a community that builds on top of GraphQL and provides

different tools to help you build your projects. The tools provided by Apollo are mainly 2: Client and Server.

- **Apollo Client.** helps your Frontend communicate with a GraphQL API. It has support for the most popular frameworks such as React, Vue, or Angular and native development on iOS and Android.
- **Apollo Server.** is the GraphQL server layer in your backend that delivers the responses back to the client requests.

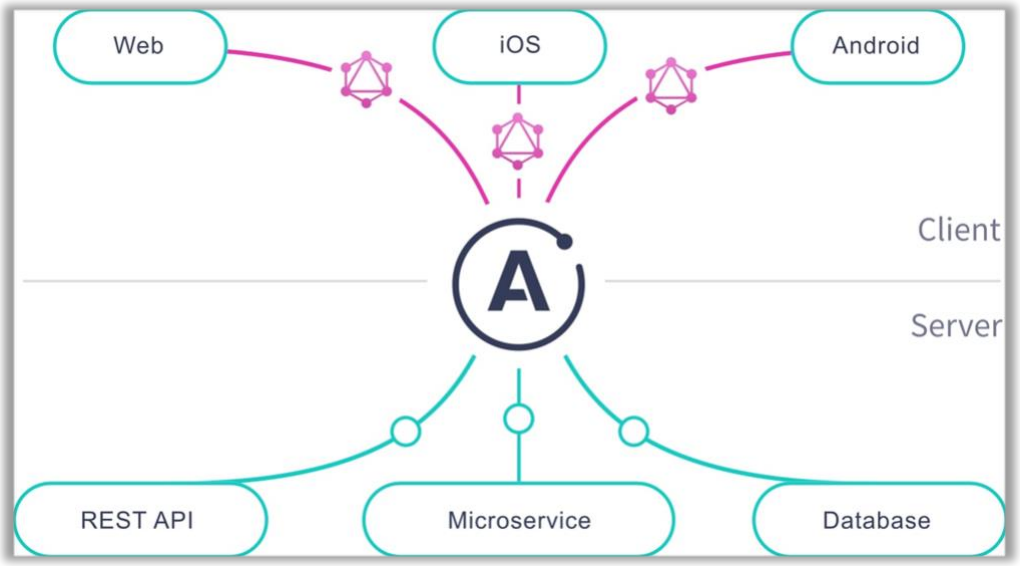


Figure 22 Apollo GraphQL diagram.

Technologies chosen

After reviewing the state of the art of the current technologies in the GraphQL ecosystem and more specifically in the JavaScript ecosystem the technologies chosen to build a social network with GraphQL have been the next ones:

TypeScript

It has already been explained in the introduction and it does not need extra information. It will be used along with the full project as the main language because its

advantages such as being more intuitive and type-safe than JavaScript. Both, the Node.js Server and the React Client will be done with this language.

React.

Will be used to develop the client application. Although it was compared with Angular and Vue it has not been explained properly. ReactJS is an open source JavaScript library developed by Facebook that is used to build the user interface. Usually used to create single page applications. Additionally, you'll use it to create cross-platform applications, suggesting that it can also be rendered on the server side in addition to the client side. React creates a VIRTUAL DOM in memory so Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM. This way React finds out what changes have been made, and changes only what needs to be changed.

React made using a pattern called **Flux** which is based on a one-way data flow. Recall that the Model in MVC represents the persisted data that will be rendered by the View. Flux splits the responsibilities of the MVC Model; it uses Actions/APIs for business logic and the “Store” for handling state which is a passive model for the entire app.

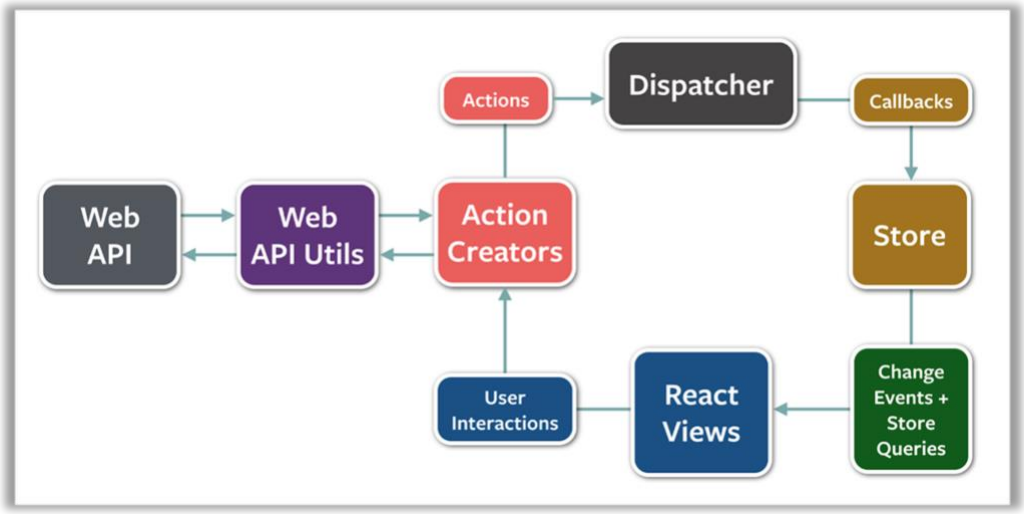


Figure 23 Flux diagram.

Styled-Components.

To style the React application using tagged template literals (a recent addition to JavaScript) and the power of CSS, stylish components will allow us to write real CSS code to design the components. It will also remove the assignment between components and styles.

Apollo.

It will be used in both Client and Server with the libraries explained before to handle all the GraphQL implementation.

GraphQL Code Generator.

GraphQL Code Generator is a simple CLI tool that operates based on a configuration file and can generate TypeScript types for both Client and Server. With GraphQL Code Generator TypeScript definitions will be generated given a GraphQL schema, and a set of GraphQL documents if they are presented.

PostgreSQL.

Although there has not been a State-of-the-Art part about persistence, the reason for choosing PostgreSQL as the SQL Database system has been that it is a Relational Database implementation that has tables, constraints, triggers, roles, stored procedures, and views together with foreign tables from external data sources and many features from NoSQL.

OBJECTIVES

This section has defined the application objectives that the Social Site must meet. Additionally, a series of personal objectives are proposed that must be carried out for the resolution of this Final Master's Project.

Application objectives

The objectives that have to be achieved at the end of the implementation of the social site system are the following:

- Create an easy to use and intuitive application with a simple and modern design. If the user does not adapt to the design and the application becomes tedious, it will end up not accessing it. In addition, the design of this will be inspired by current social sites like Instagram, Twitter, and Facebook.
- Design a clear way to add and read posts from other users. The main objective of the app is that the user can express itself and share its content with other users interacting with them.
- Use a modern stack of web development technologies to implement both the frontend and the backend. Use a modern stack of web technologies along with GraphQL instead of REST to create the application.
- Create a secure application both for the development and for the users to use. Implement tests and data encryption to make the application secure to use.

Personal objectives

These are a series of personal objectives proposed and that want to be achieved with the making of the project to gain knowledge and skills.

- Learn to design and create a social site. Nowadays almost every app needs to have a social part related to it with the management of accounts and user content. Developing one will be a nice personal training to learn how to implement this kind of features.
- Design the architecture and develop a software project from start to finish. Gain skills at designing the architecture of a software project and implementing it from the beginning to the end.
- Improve the way of personal planning to take advantage of available time and resources. In order to carry out this project and future projects planning is considered to be one of the most important objectives and, therefore, with this project, it is pretended to improve personal planning.
- Learn to develop with TypeScript and GraphQL. The popularity of these technologies is growing each day because of its multiple advantages and many companies have already implemented them in their development process. For this reason, It is believed that expanding the knowledge in these technologies may be of help in the future beyond this project.
- Apply the knowledge acquired in the Master of Web Engineering. In special the management of a project with technologies like GitHub which were used in some subjects from the Master.

METHODOLOGY

The methodology of a project is nothing more than how it will be developed. Exist multiple types of methodologies for the development of software and applications, for this, the project will use a phased development methodology together with an Agile methodology.

The overall project phases and in each one of the iterations will be:

- **Analysis**: A study of the current market regarding the application sector. Once the results have been analyzed, they proceed to specify the project requirements.

- **Design**: once you have an idea of the project requirements, they will be structured and placed graphically in a conceptual model in the complete design of the application.

- **Implementation**: In this phase the development of all the graphic content and the functionalities of the application based on the designs created in the previous phase.

- **Deployment**: This is the final phase in which it is verified that everything has been implemented. In the correct way and the possible faults that can be found are corrected. Once it performs these subtasks it is put into production and is evaluated how it performs before beginning a new iteration.

As for the Agile methodology, which is based on developing the main functions in a fast and flexible way and on the principles of continuous error detection and possible improvements, it is adapted to unforeseen events and is self-managed in an easy way and balanced. Some of the advantages of this methodology for the development of the application compared to other specific ones:

- **High flexibility in face of changes**. Given the personal inexperience when facing a project of this magnitude, a poor initial contemplation of its requirements can be given. An Agile methodology has a great reaction to possible changes in the requirements of an application.

- **Reduction of market time**. through this methodology, the application will reach a state with the basic functionalities as soon as possible, which would allow a longer period of testing and improvement.

- **Higher quality of the software produced**. The requirement to implement and improve the functionalities through each iteration results in the software being debugged and improved from iteration to iteration.

- **Better time control**. By dividing the functionalities implementation process into iterations also called sprints, you can know the average time to carry out each iteration and its estimated time when certain functionality is implemented.

- **Lower number of risks.** thanks to the previous advantage of better control of development time and the priority of implementation of the main functions, possible delays and future risks in development can be prevented and dealt with in an appropriate way.

I will also follow in each sprint or milestone a software development method called tracer bullets. Essentially, the **tracer bullet method involves implementing a new end-to-end application to test the interaction of each layer or component.** The key benefit of this method is getting quick feedback on a variety of factors. It serves as proof that the architecture is compatible and feasible, as well as providing a functional and demonstrable skeleton to work from the beginning in the development process. The goal is to identify problems early, make something work early, and progress more consistently and safely. This approach is invaluable not only when launching into a new application but can also be applied by adding functionality to an existing application.

Project Management

Additional tools will also be used to carry out the development of the project of a more efficiently, managing time, planning goals and accounting tasks In the backlog, in progress and done. Also tools for developing the project in terms of producing the necessary and appropriate code to carry out the application's functionalities.

Clockify.

Clockify is a web and mobile application whose application is to count the time of the tasks of a project in a precise and simple way. In addition to the main timer tool, Clockify has a history of completed tasks and a system of labels and projects in which to group the records. This makes the differentiation of tasks very useful when working on several sections of the same project. As additional functionality includes the generation of detailed reports of the working times performed by the user. Here is the app reports dashboard where the time is being tracked and filtered by all the tasks tagged as TFM (Final Thesis Master in Spanish *Trabajo de Fin de Master*).

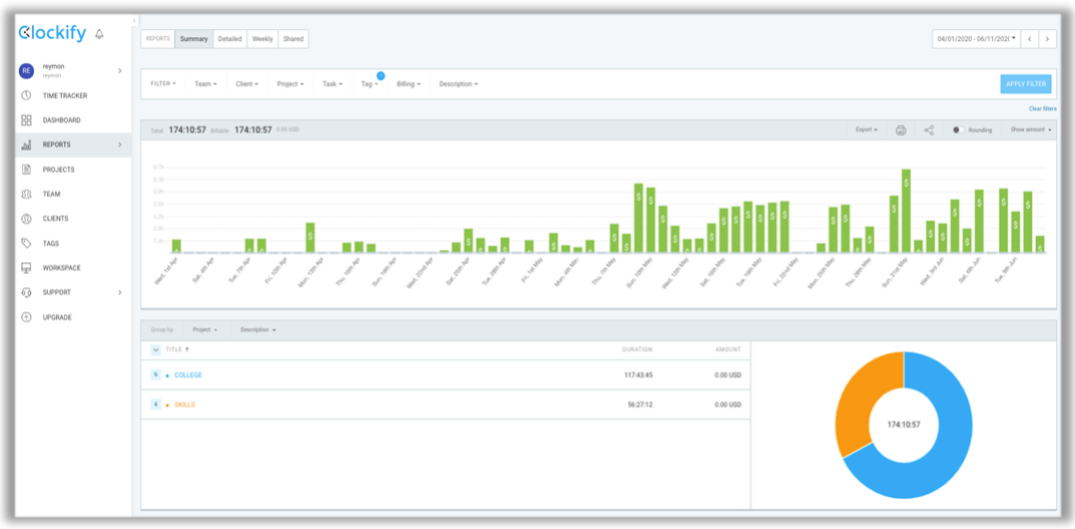


Figure 24 Clockify Reports Dashboard filtered by TFM tag.

As mentioned, before it also generates automatic reports. Here is an example of a report with all the tasks again filtered by the TFM tag.

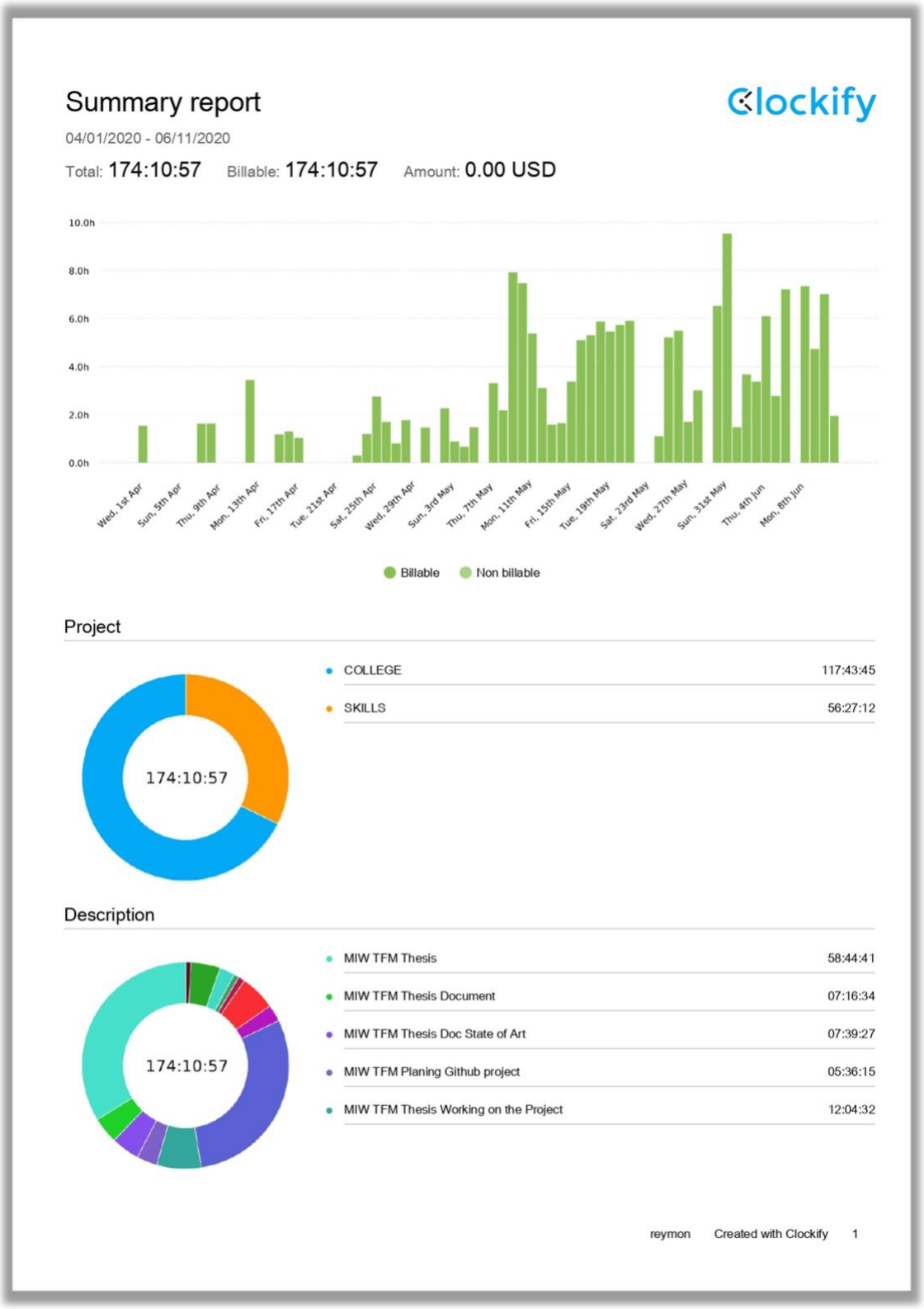
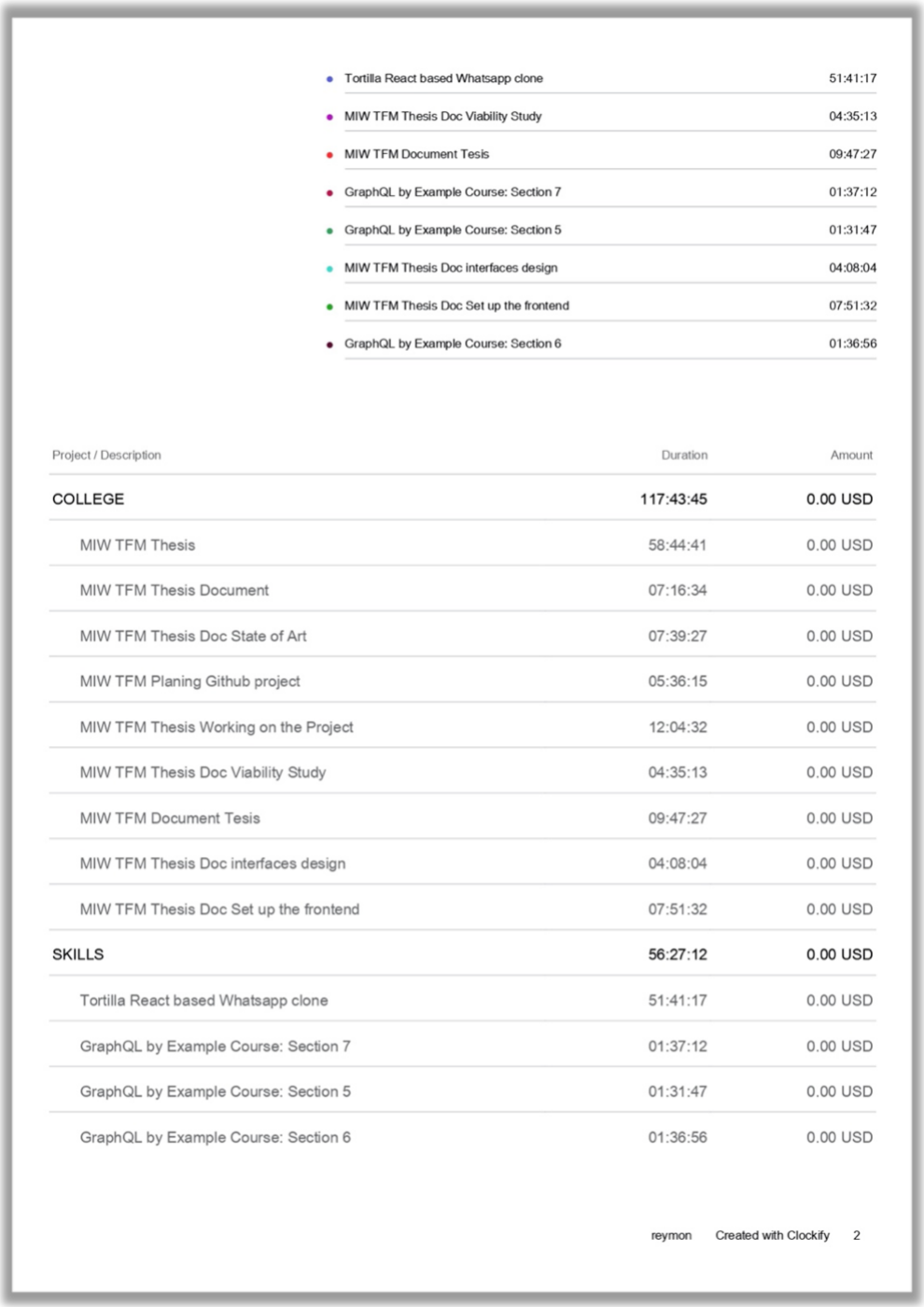


Figure 25 Clockify Summary Report from 04_01_2020 to 06_11_2020 page 1



- Tortilla React based Whatsapp clone 51:41:17
- MIW TFM Thesis Doc Viability Study 04:35:13
- MIW TFM Document Tesis 09:47:27
- GraphQL by Example Course: Section 7 01:37:12
- GraphQL by Example Course: Section 5 01:31:47
- MIW TFM Thesis Doc interfaces design 04:08:04
- MIW TFM Thesis Doc Set up the frontend 07:51:32
- GraphQL by Example Course: Section 6 01:36:56

Project / Description	Duration	Amount
COLLEGE	**117:43:45**	**0.00 USD**
MIW TFM Thesis	58:44:41	0.00 USD
MIW TFM Thesis Document	07:16:34	0.00 USD
MIW TFM Thesis Doc State of Art	07:39:27	0.00 USD
MIW TFM Planing Github project	05:36:15	0.00 USD
MIW TFM Thesis Working on the Project	12:04:32	0.00 USD
MIW TFM Thesis Doc Viability Study	04:35:13	0.00 USD
MIW TFM Document Tesis	09:47:27	0.00 USD
MIW TFM Thesis Doc interfaces design	04:08:04	0.00 USD
MIW TFM Thesis Doc Set up the frontend	07:51:32	0.00 USD
SKILLS	**56:27:12**	**0.00 USD**
Tortilla React based Whatsapp clone	51:41:17	0.00 USD
GraphQL by Example Course: Section 7	01:37:12	0.00 USD
GraphQL by Example Course: Section 5	01:31:47	0.00 USD
GraphQL by Example Course: Section 6	01:36:56	0.00 USD

reymon Created with Clockify 2

Figure 26 Clockify Summary Report from 04_01_2020 to 06_11_2020 page 2

GitHub

GitHub is a **Git** repository hosting service, but it adds many of its own features. While Git is a command-line tool, GitHub does provide a graphical web-based interface. It also provides access control and various collaboration features such as wikis and basic

task management tools for each project. It will be used as the **Version Control System** of the project.

In addition, **GitHub Project Boards** will be used in the project too. Project boards on GitHub will help to organize and prioritize the work creating a customized workflow that will suit the project needs.

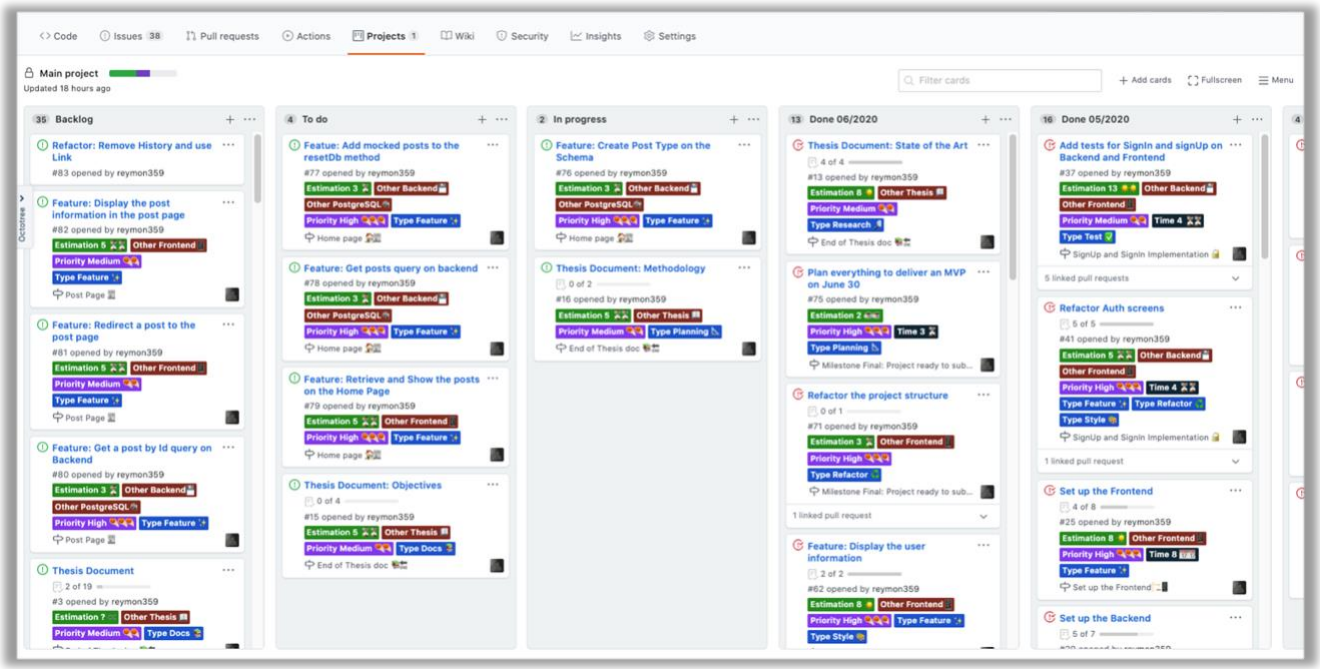


Figure 27 GitHub Project Board

To properly manage the issues the GitHub tagging system will be used in the project with tags that refer to the task/issue that it is being implemented. The color palette was configured using **coolors.co** and the type of tags will be based on the Angular convention:

- **feat**: a new feature
- **fix:** a bug fix
- **docs**: changes to documentation
- **style**: formatting, missing semicolons, etc; no code change
- **refactor** refactoring production code

- **test**: adding tests, refactoring test; no production code change
- **chore**: updating build tasks, package manager configs, etc; no production code change

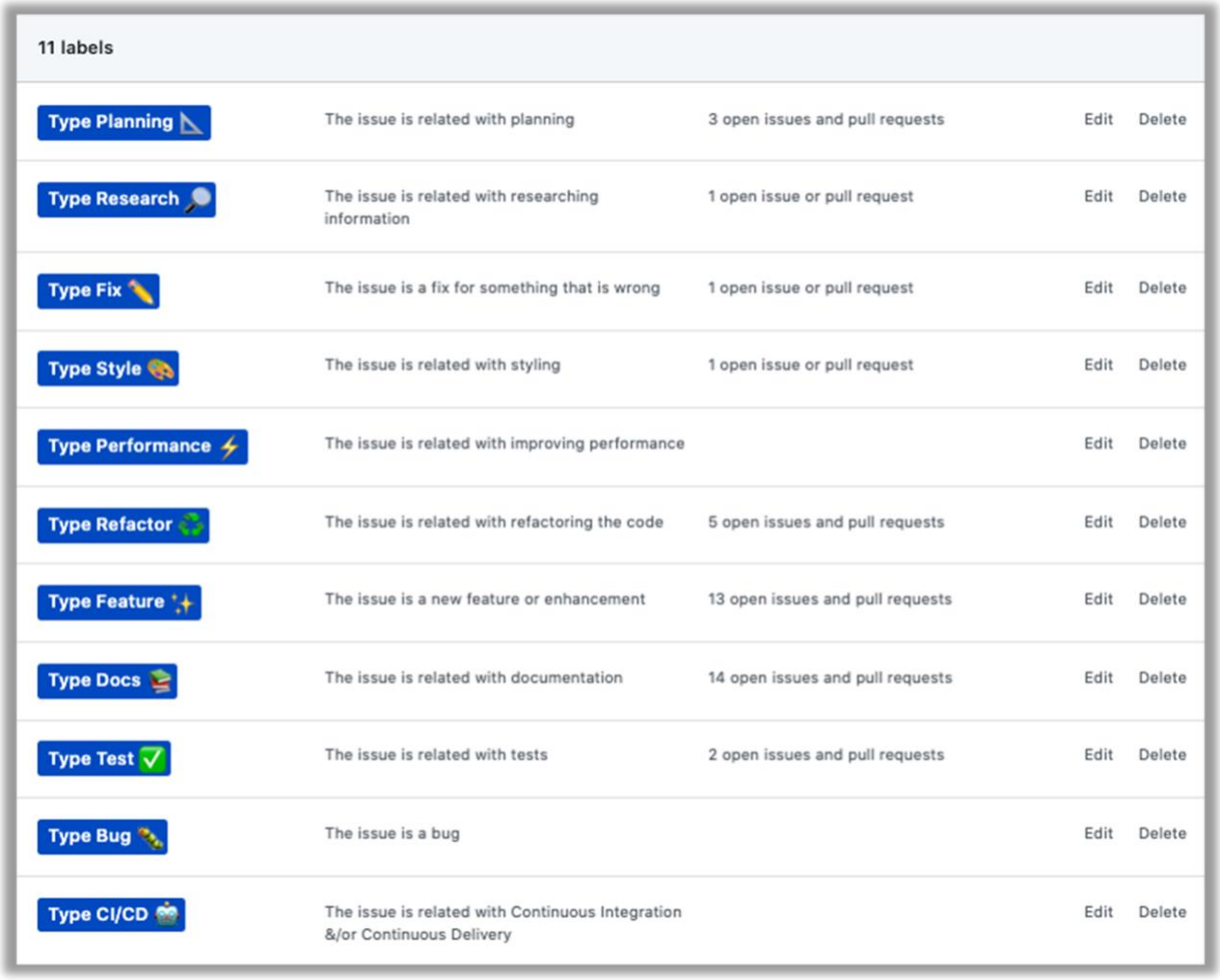
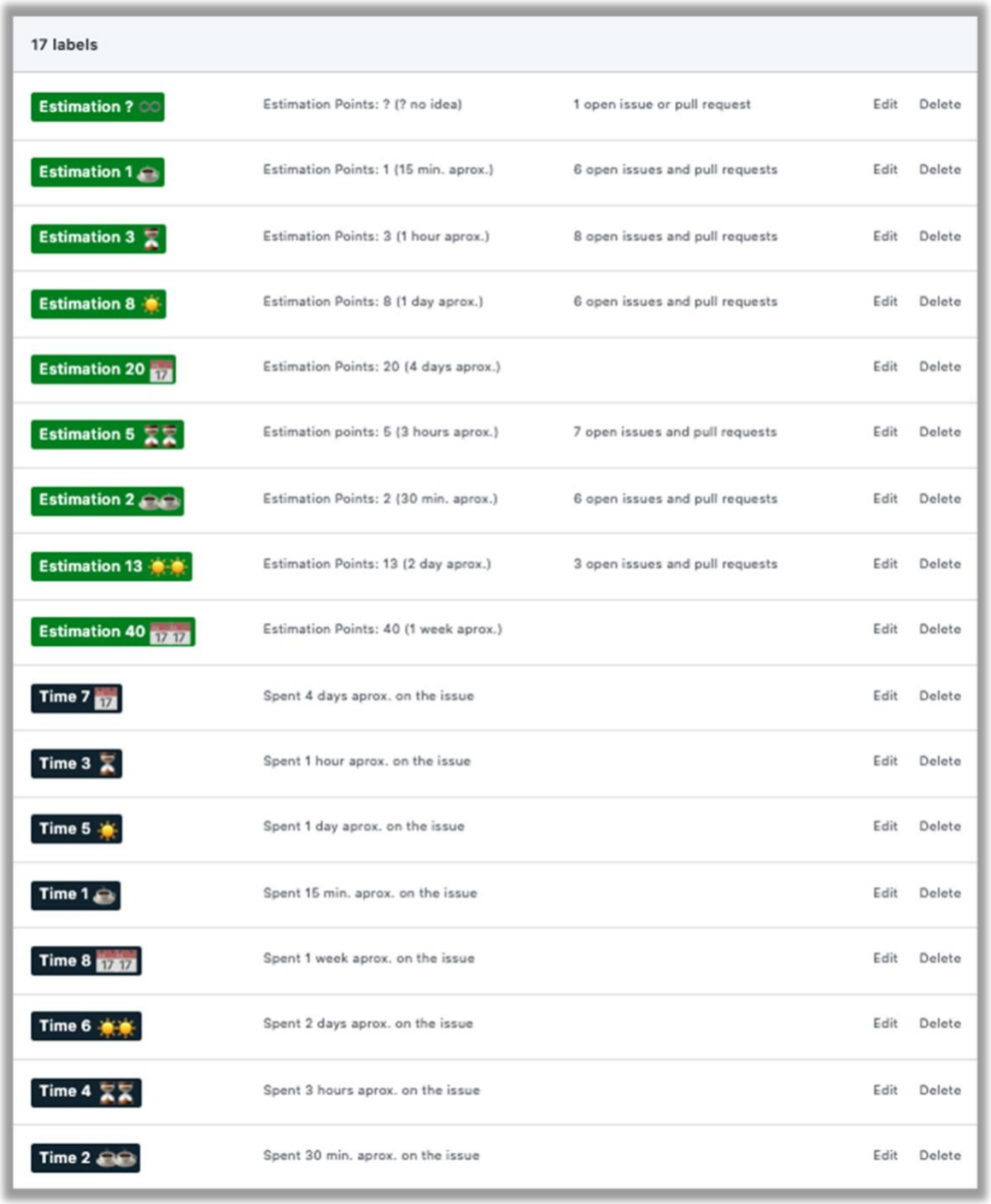


Figure 28 Project Type Tags

To estimate and track the time spent in each issue **story point** tags have been also added. A story point is an abstract measure of effort required to implement a user story. It is a number that will tell about the difficulty level of the story. The difficulty could be related to complexities, risks, and efforts involved.



17 labels			
Estimation ?	Estimation Points: ? (? no idea)	1 open issue or pull request	Edit Delete
Estimation 1	Estimation Points: 1 (15 min. aprox.)	6 open issues and pull requests	Edit Delete
Estimation 3	Estimation Points: 3 (1 hour aprox.)	8 open issues and pull requests	Edit Delete
Estimation 8	Estimation Points: 8 (1 day aprox.)	6 open issues and pull requests	Edit Delete
Estimation 20	Estimation Points: 20 (4 days aprox.)		Edit Delete
Estimation 5	Estimation points: 5 (3 hours aprox.)	7 open issues and pull requests	Edit Delete
Estimation 2	Estimation Points: 2 (30 min. aprox.)	6 open issues and pull requests	Edit Delete
Estimation 13	Estimation Points: 13 (2 day aprox.)	3 open issues and pull requests	Edit Delete
Estimation 40	Estimation Points: 40 (1 week aprox.)		Edit Delete
Time 7	Spent 4 days aprox. on the issue		Edit Delete
Time 3	Spent 1 hour aprox. on the issue		Edit Delete
Time 5	Spent 1 day aprox. on the issue		Edit Delete
Time 1	Spent 15 min. aprox. on the issue		Edit Delete
Time 8	Spent 1 week aprox. on the issue		Edit Delete
Time 6	Spent 2 days aprox. on the issue		Edit Delete
Time 4	Spent 3 hours aprox. on the issue		Edit Delete
Time 2	Spent 30 min. aprox. on the issue		Edit Delete

Figure 29 Estimation and Time Project tags

Visual Studio Code and WebStorm

Both **IDEs** have been used to develop the project code. **Visual Studio Code** is a free source code editor created by Microsoft for Windows, Linux, and macOS. Features

include support for debugging, syntax highlighting, smart code completion, snippets, code refactoring, and embedded Git. It also allows installing extensions that add additional functionality. And **WebStorm** is for complex client-side development and server-side development with Node.js.

It provides smart code insight, autocompletion, refactoring features, on-the-fly error prevention, and much more.

GraphQL Playground (GraphiQL) and Postman

GraphQL Playground is a graphical, interactive, in-browser GraphQL IDE, created by Prisma and based on GraphiQL. It is an environment to perform Queries, Mutations, or Subscriptions to the project GraphQL schema and interact with its data. And **Postman** is a is a collaboration platform for API development that allows to easily perform request and queries to our GraphQL. It also includes other utilities and tools for management of the overall queries that make it an essential tool.

ANALYSIS AND SPECIFICATION

This section focuses on the main functionalities and problems that we want to solve with our application. This requires an analysis and specification of all project requirements. These requirements will be as concrete as possible given that they will be used as a guide when developing the application and if the development phase is advanced, the cost of solving a requirement to solve a problem will be higher.

To carry out the analysis and specification, the IEEE 830 standard for the SRS (Software Requirements Specifications) will be taken as our recommendations and way of proceeding have as a final product a clear systematic definition of the necessary requirements when establishing a solution. software for each problem.

In order to carry out the most specific analysis possible, these requirements have been divided into functional and non-functional requirements. Each requirement will be identified with a unique ID and will also have a name and priority.

• **High**, when it is essential to fulfilling the requirement for the correct development of the functionality.

• **Media**, when you only want to meet this requirement, but not essential for the development of the application.

• **Low**, when the implementation of the requirement is merely optional.

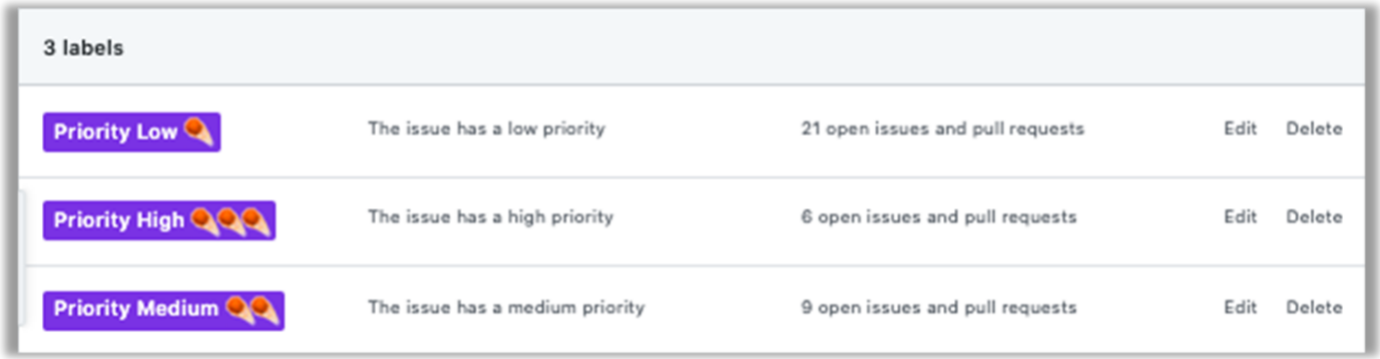


Figure 30 Project Priority Tags

Finally, before starting to analyze the requirements, the user for whom these requirements must be met will be analyzed. The application, having no user roles or a distinction of users by payment categories, since it is free, will only have one type of user. Its characteristics such as age or sex can vary since the application contemplates its use in users of any age and regardless of whether they are men or women.

The type of user will be one that wants to share its posts in a social network and also see other users' posts and interact with them. The user to whom the application is focused should not have extreme knowledge of application management or any particular subject.

Functional Requirements

The functional requirements include the main functionalities of the application and the behaviors in interactions that the system will carry out to face them.

ID Identifier	FR-01
Name	User can Sign up on the Social Site
Priority	High
Description	The user can Sign up on the Social Site and create an account submitting its personal data like the name, email, password, etc.

Figure 31 Functional Requirement 01 - User can Sign Up on the Social Site

ID Identifier	FR-02
Name	User can Sign in on the Social Site
Priority	High
Description	The user can Sign in on the Social Site submitting its email and password in a Sign In form

Figure 32 Functional Requirement 02 - User can Sign In on the Social Site

ID Identifier	FR-03
Name	User can end the current session
Priority	High
Description	The user can end the current session using a properly accessible Logout button.

Figure 33 Functional Requirement 03 - User can end the current session

ID Identifier	FR-04
Name	User can navigate freely through the app
Priority	High
Description	The user is able to go to the main pages of the application using a navbar with links to them.

Figure 34 Functional Requirement 04 - User can navigate freely through the app

ID Identifier	FR-05
Name	User can always navigate to the Home Page
Priority	Medium
Description	The user is able to go to the main pages of the application using a navbar with links to them.

Figure 35 Functional Requirement 05 - User can always navigate to the Home Page

ID Identifier	FR-06

Name	User can See its personal information
Priority	High
Description	The user is able to see all the data submitted on the Sign up in the Profile page.

Figure 36 Functional Requirement 06 - User can See its personal information

ID Identifier	FR-07
Name	User can See other users' personal information
Priority	High
Description	The user can access to other users' profiles and see their personal data

Figure 37 Functional Requirement 07 - User can See other users' personal information

ID Identifier	FR-08
Name	User can edit its personal data
Priority	High
Description	The user is able to edit its personal data, both the one displayed on the profile and also the personal one.

Figure 38 Functional Requirement 08 - User can edit its personal data

ID Identifier	FR-09
Name	User can see its own posts
Priority	Medium
Description	The user is able to see on its profile the posts created.

Figure 39 Functional Requirement 09 - User can see its own posts

ID Identifier	FR-10
Name	User can see its liked posts
Priority	Low
Description	The user is able to see on its profile the posts liked from other users.

Figure 40 Functional Requirement 10 - User can see its liked posts

ID Identifier	FR-11
Name	User can see the posts of another user
Priority	Medium
Description	The user is able to see the posts of another user in that user profile

Figure 41 Functional Requirement 11 - User can see the posts of another user

ID Identifier	FR-12
Name	User can see the posts liked by another user
Priority	Low
Description	The user is able to see the posts that another user has liked in that user profile

Figure 42 Functional Requirement 12 - User can see the posts liked by another user

ID Identifier	FR-13
Name	User can follow another user
Priority	Medium
Description	The user is able to follow other users

Figure 43 Functional Requirement 13 - User can follow another user

ID Identifier	FR-14
Name	User can see the last posts in the Home Page
Priority	High
Description	The user is able to see the last posts submitted on the home page.

Figure 44 Functional Requirement 14 - User can see the last posts in the Home Page

ID Identifier	FR-15
Name	User can access a post from Home Page
Priority	Medium
Description	The user is able to navigate to the Post Page of any of the posts that appear on the Home Page.

Figure 45 Functional Requirement 15 - User can access a post from Home Page

ID Identifier	FR-16
Name	User can like a post

Priority	Medium
Description	The user is able to like a post from another user

Figure 46 Functional Requirement 16 - User can like a post

ID Identifier	FR-17
Name	User can create a post
Priority	High
Description	The user is able to create a post with a title, description, picture, and content.

Figure 47 Functional Requirement 17 - User can create a post

ID Identifier	FR-18
Name	User can edit its own post
Priority	Medium
Description	The user is able to edit a post that has been previously created by itself.

Figure 48 Functional Requirement 18 - User can edit its own post

ID Identifier	FR-19
Name	User can delete its own post
Priority	Medium
Description	The user is able to delete a post that has been previously created by itself.

Figure 49 Functional Requirement 19 - User can delete its own post

ID Identifier	FR-20
Name	User can unlike a post
Priority	Low
Description	The user is able to unlike a post previously liked.

Figure 50 Functional Requirement 20 - User can unlike a post

Non-Functional Requirements

Non-functional requirements are those that describe aspects of the system but that do not describe the way of acting or its functionalities. Some of these requirements should be present from the beginning of any system since they are related to aspects such as security or error control.

ID Identifier	NFR- 01
Name	Availability
Priority	High
Description	The social site must be online whenever the user wants to access it to see the media content or interact with other users

Figure 51 Non-Functional Requirement 01 - Availability

ID Identifier	NFR- 02
Name	Security
Priority	High
Description	The social site must be secure, and all the user data must comply with data protection and security regulations.

Figure 52 Non-Functional Requirement 02 - Security

ID Identifier	NFR- 03
Name	Error handling
Priority	Medium
Description	The social site must provide mechanisms that allow the user to be informed of errors that have occurred at any level.

Figure 53 Non-Functional Requirement 03 - Error handling

ID Identifier	NFR- 04
Name	Responsive
Priority	Medium
Description	The social site must render well on a variety of devices and windows or screen sizes.

Figure 54 Non-Functional Requirement 04 – Responsive.

DESIGN

The Design section is where the functional and non-functional requirements previously raised will be answered, the identifiers of the requirements will be used to relate the design of the solution to each one.

The design of these solutions is a complex task that involves aspects at different levels or views of the project. Therefore, the design will be grouped into different sections according to the theme or aspect that is being designed, so each section will contain everything related to that context. And the sum of these designs will make up the application itself.

Conceptual Architecture

The conceptual architecture of the project will show the modules and functional blocks that the solution will contain. In the case of this project, there are 3 main modules: The React Client, the GraphQL API, and the PostgreSQL Database.

React Client

This one is the React application that will communicate with the user through the interfaces and will also communicate with the GraphQL API.

GraphQL API

This is the module in charge of the application's services and drivers. In this part the user cannot access, it is in charge of managing the information and data between the Client part, where users enter or request the information, and the database where said information is stored.

PostgreSQL Database

It handles the persistence of data, its storage, and distribution in a correct way in a database.

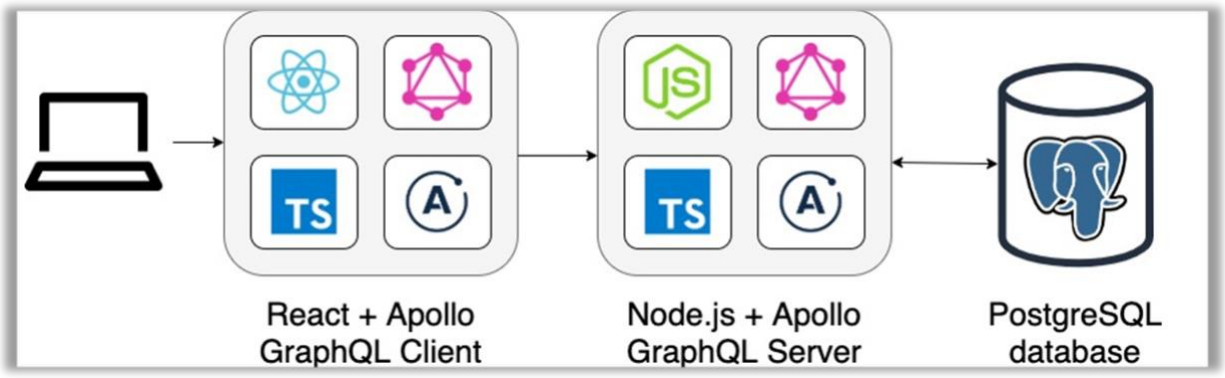


Figure 55 Project Conceptual Architecture Design

Client Architecture

The Client Architecture references the way the React application will be structured and how the integrations with Apollo and GraphQL will be implemented. The main modules and components will be shown and explained.

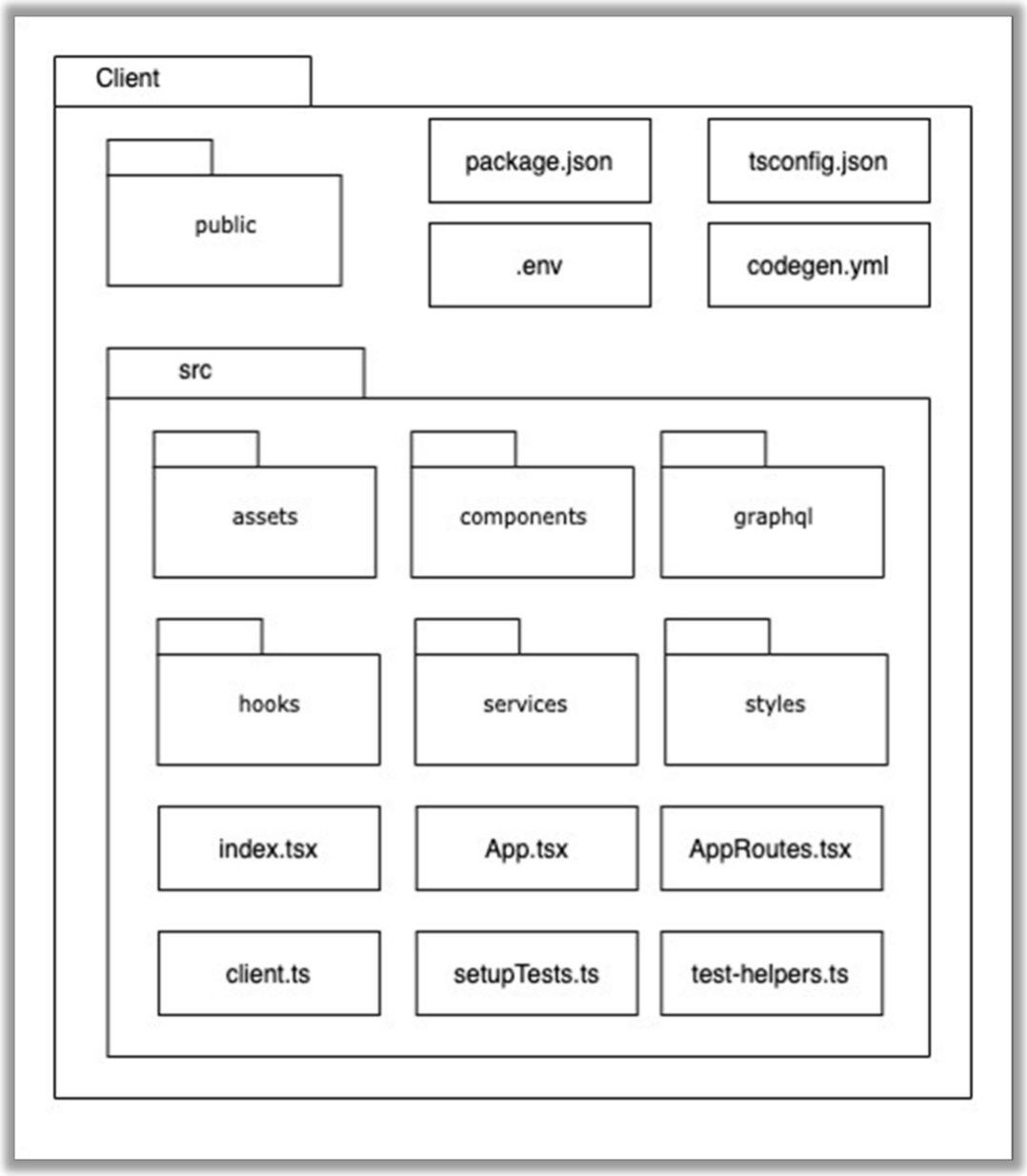


Figure 56 Project Client Architecture Overview

- **public.** This is where the static files reside. If the JavaScript application doesn't import the file and there is no need to keep its name, it will be here. Files in the public directory will keep the same file name in production, which generally means that the client will cache them and never download them again. If the file doesn't have a major file name, like index.html, manifest.json, or robots.txt, it will be on the src instead
- **package.json.** This file contains various metadata relevant to the project and is used to provide information to **npm** that allows you to identify the project and manage project dependencies. It can also contain other

metadata, such as a project description, the project version in a particular distribution, license information, even configuration data, all of which can be vital for both npm and package end users.

- **tsconfig.json** this one is a must if the project is going to be done in TypeScript because the file specifies options for the typing compiler when it takes the code and transforms it from typescript to Javascript. Some notable configuration options for the file are:
    - **target**. What kind of Javascript should the compilers generate which in this project will be the es5 version of JavaScript that is compatible with many browsers.
    - **lib**. If the project is going to be developed using a new Javascript syntax, the compiler can add to the output libraries that would support the new syntax, even if browsers don't know it.
- **.env** It's a simple configuration text file that is used to define some of the external variables that will be passed into the application's environment.
- **codegen.yml** The essence of this file is to provide the GraphQL code generator with the GraphQL schema, documents, and the output path of the type definition files and a set of plugins.
- **src** Here is where the dynamic files reside. If the file is imported by your JavaScript application or changes content will be here. To ensure that the client downloads the most current version of their file instead of relying on a cached copy, Webpack will give the modified files a unique file name in the production build. This will allow to use simple and intuitive file names during development, such as logo.png instead of banner-updated.png. This will also avoid the user of using the deprecated cached copy because Webpack will automatically rename logo.png to logo.unique-hash.png, where the unique hash changes only when logo.png changes. Inside it, the structure has these components
    - **index.tsx.** This is the file that stores the main Render call from ReactDOM. It also imports the App.tsx component which tells React where to render it (in a div in the public/index.html file). The Global

styles and main context providers like the ApolloProvider and the ThemeProvider will be here.

- **App.tsx** Conventionally, App.js acts as the highest level component in the React application structure. It will be the main parent directory and will also handle the routing of the app.
- **AppRoutes** This file will have the project routes predefined so we can reuse them everywhere on the project without having to declare them multiple times and having to change multiple files if there is a need to rename any of them.
- **client.ts.** This file will handle the Apollo Client implementation which will be wrapped around the GraphQL endpoint which essentially uses HTTP requests. It will not only fetch the data, but it also caches the result of the query so it can be seamlessly re-used when the user requests the same data.
- **assets.** This directory will store asset files like the logo, images, or other static elements.
- **graphql.** This directory will be the directory with all the GraphQL elements: Schema, fragments, mutations, queries, and subscriptions.
- **hooks.** Will store useful React hooks that will be used in more than one component of the application.
- **services.** The folder that will handle the logic of services like the authentication or the cache
- **styles.** All the global styles and the theme of the app will be featured here. In the components, the components will be styled individually using the styled-components CSS-in-JS library.
- **components.** Here is where the React components will be stored.

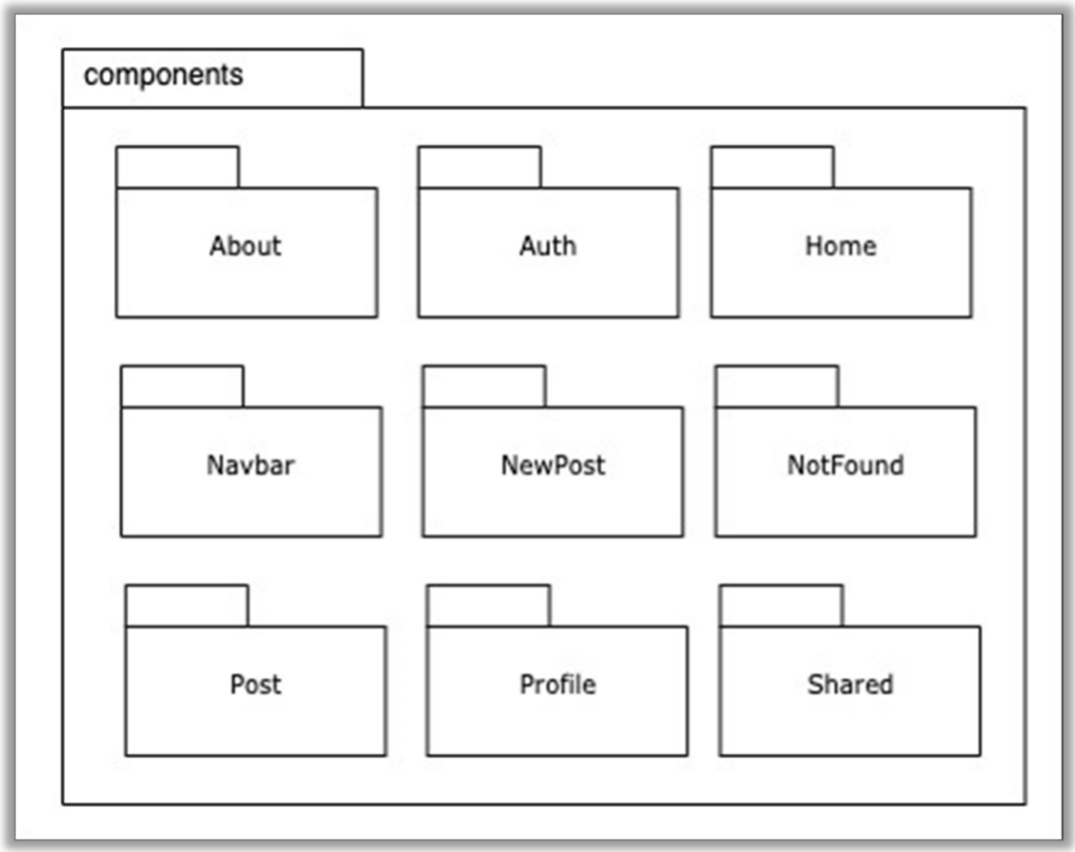


Figure 57 Project Client Architecture Components

The Components are mainly structured making reference to the main user interfaces and other ones used across them like the navbar one. The majority of them will be structured the same way with a container component, the main component, tests file, and an index.js file to export the rest of the other files to be used on the application. The shared directory will contain components that, like the name says, will be shared across other ones. This way we will increase modularity and reusability.

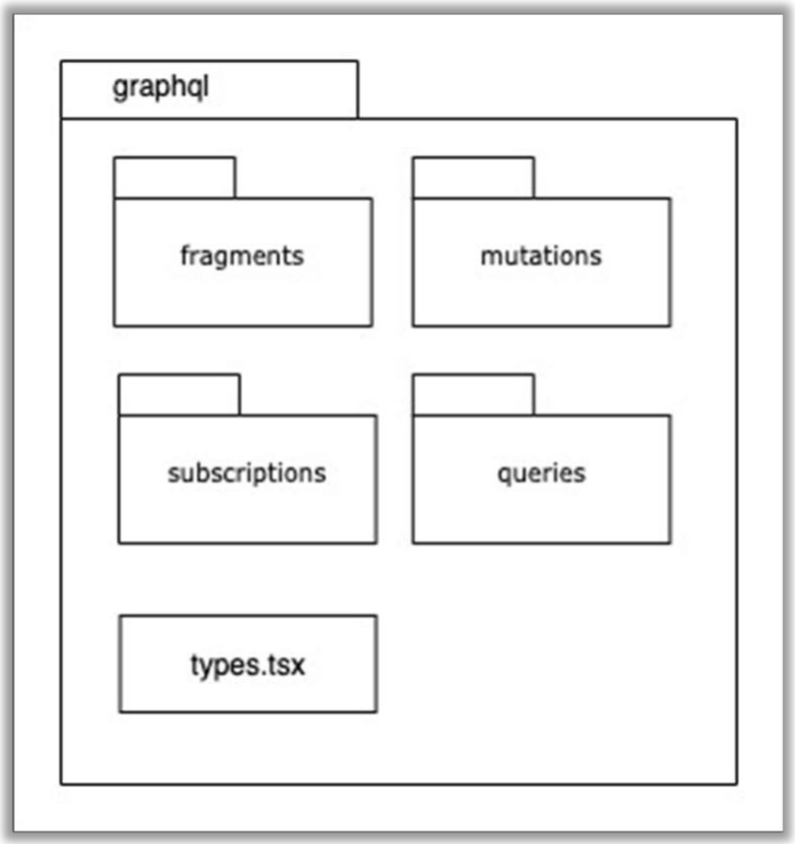


Figure 58 Project Client Architecture GraphQL

As mentioned before, this is the Client's part where the GraphQL components will reside and will be exported to use across the application.

- **types.tsx.** The file autogenerated by the GraphQL Generator will all the types for the schema, queries, mutations, and subscriptions of the app
- **fragments.** Here will be all the GraphQL fragments. A GraphQL fragment is a shared piece of query logic. The most common use of fragments is to reuse parts of queries (or mutations or subscriptions) in various parts of your application.
- **queries.** The GraphQL queries used across the components like getting the user data or the latest posts.
- **mutations.** The GraphQL mutations used across the components like signing in a user or creating a post
- **subscriptions.** The GraphQL subscriptions used across the components like subscriptions to a new post created.

Server Architecture

The GraphQL API will be implemented in a Node.js server and the Apollo Server which will make it compatible with any GraphQL client and is one of the best ways of building a production-ready, self-documenting GraphQL API that can use data from any source.

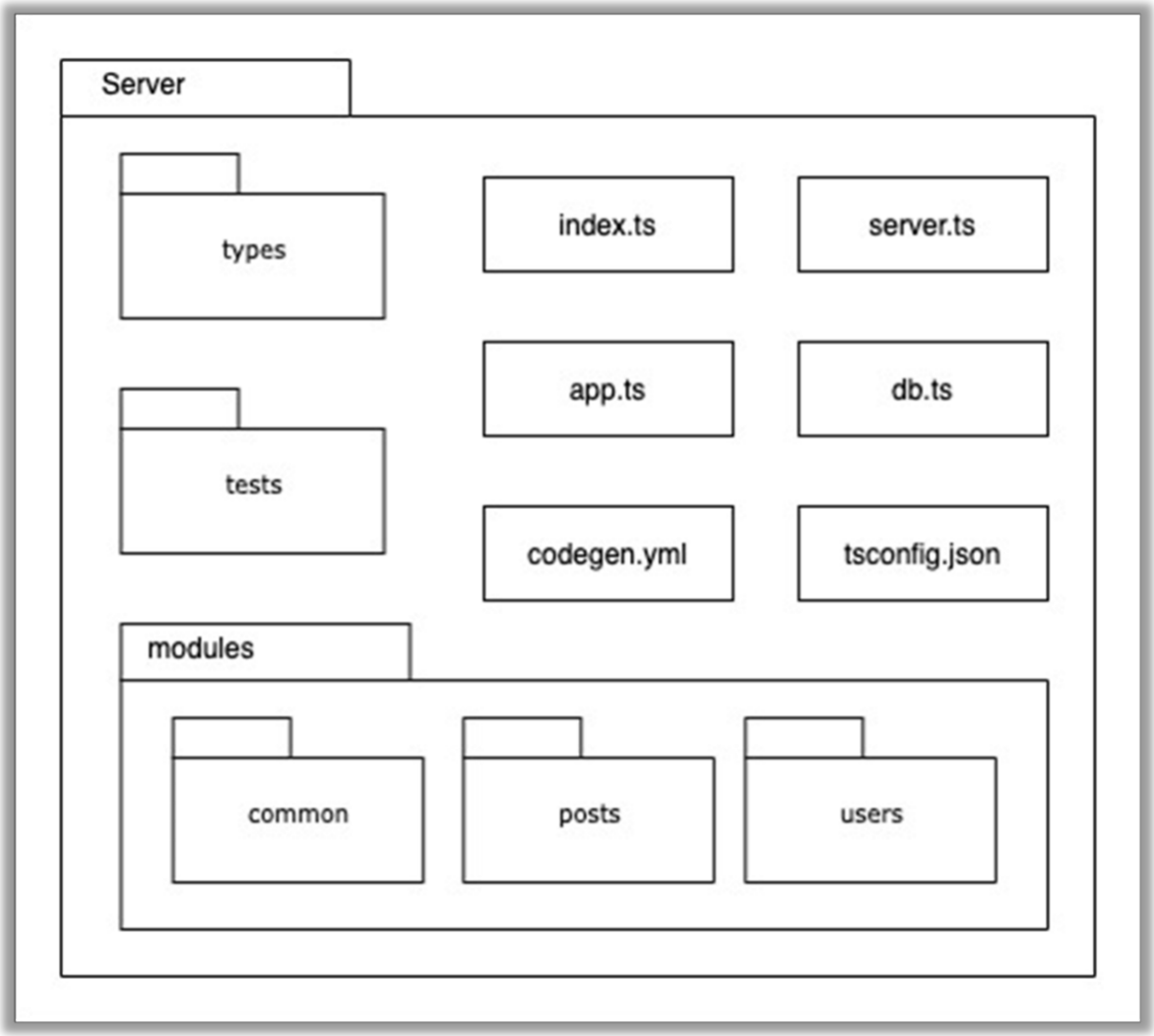


Figure 59 Project Server Architecture Overview

Some of the most important directories and files:

- **index.ts.** The Server main file, it will import the express app configuration from the app.ts and the Apollo server one from the server.ts and setup the HTTP server.

- **server.ts.** Here is where the Apollo server configuration resides. Using the GraphQL modules library it will import all the graphql modules as the main rootModule and use them as the GraphQL schema for the Apollo server. It will also handle the cookies-based session authentication in the context of the Apollo Server.
- **app.ts** Where the express server will be set up. In the end, with GraphQL the project will make POST requests where the query will be passed in the payload (body).
- **db.ts** This file will have some of the Database configurations and also a method to mockup a full database for development.
- **codegen.yml** Similar to the client, it will also configure the GraphQL code generator with the GraphQL schema, documents, and the output path of the type definition files.
- **tsconfig.json** Will work the same way as in the client, it will configure the use of TypeScript on the server-side.
- **types.** This directory will include the autogenerated files from the GraphQL Code Generator. And other types needed to work with TypeScript.
- **tests.** Where the server tests of all the components will reside. The GraphQL queries and mutations will be tested here. Given the importance of testing, its design will be explained ahead in its own section.
- **Modules** Commonly, every app starts small and the difficulty of maintenance grows while features are being implemented. And while the initial GraphQL Schema is a small one, the lack of separation makes the schema harder to maintain, especially if it starts to grow rapidly. To improve reusability and maintenance modules will be created the Schema will be split in to separate parts thanks to the GraphQL Modules library.
    - **Common** Here will be the logic we want to share with all the other modules like the database instance.
    - **users** Everything related to the users, the Schema types, authentication logic, the resolvers, and database methods.
    - **posts** All that has to go with the posts, the Schema types, resolvers, and database methods.

Schema

A GraphQL Schema is at the core of any GraphQL server implementation. It describes the shape of the data, defining it with a hierarchy of types with fields that are populated from the data source and also specifies which queries and mutations are available so the client knows about the information that can be requested or sent. Here are the different parts of the schema modularized.

```
const typeDefs = gql`
 type User {
  id: ID!
  name: String!
  username: String!
  email: String!
  bio: String
  followers: Int!
  following: Int!
  picture: URL
 }

 extend type Query {
  me: User
  user(username: String!): User
  users: [User!]!
  followers(userId: String!): [User!]!
  following(userId: String!): [User!]!
 }

 extend type Mutation {
  signIn(email: String!, password: String!): User
  signUp(
    name: String!
    username: String!
    email: String!
    password: String!
    passwordConfirm: String!
  ): User
  editUser(
    name: String!
    username: String!
    email: String!
    bio: String!
    password: String!
    passwordNew: String!
    passwordNewConfirm: String!
  ): User
  follow(userId: String!): User
  unfollow(userId: String!): User
 }

 extend type Subscription {
  userFollowed: User!
 }
`;
```

Figure 60 User Schema Module

```
const typeDefs = gql`
 type Post {
  id: ID!
  title: String!
  picture: URL
  description: String!
  content: String!
  createdAt: DateTime!
  likes: Int!
  user: User
 }

 extend type Query {
  post(postId: ID!): Post
  lastPosts: [Post!]!
  userPosts(userId: ID!): [Post!]!
  userLikedPosts(userId: ID!): [Post!]!
 }

 extend type Mutation {
  addPost(
   title: String!
   picture: String!
   description: String!
   content: String!
  ): Post
  editPost(
   title: String!
   picture: String!
   description: String!
   content: String!
  ): Post
  removePost(postId: ID!): ID
  likePost(postId: ID!): ID
  unlikePost(postId: ID!): ID
 }

 extend type Subscription {
  postLiked: Post!
  postAdded: Post!
  postRemoved: ID!
 }
`;
```

Figure 61 Post Schema Module

Testing

In the context of software, there are constantly changes. It's also impossible to make all functions completely independent of each other, so things in the project will break as it is updated or maintained. This is why the project needs a set of tests that can be run on-demand, so when a new feature is implemented, the tests can simply be run and see which feature was broken due to the latest changes.

Figure 62 Main test frameworks in the JavaScript Ecosystem

There are currently 3 main test frameworks in the **JavaScript** ecosystem: **Jasmine, Mocha**, and **Jest**. Each test frame has its pros and cons but given its popularity, current use and other features Jest will be used in the project, a test framework developed by Facebook. The good thing about Jest is that it can be used to test client and server logic as it runs as a Node.JS application, but it also emulates the browser environment every time we run it, thanks to JSDOM.

It will be used to test the React Components on the client and Apollo-GraphQL on the server. There are 3 types of tests:

- Unit tests, which are used to test a single component, independently of other components in the system.
- Integration tests: which are used to test a component in relation to other components in the systems.
- E2e tests (end-to-end) - used to test an entire process, from the moment the user clicks a button in any interface until the data comes back from the server and is displayed on the screen.

Test efficiency goes from bottom to top (unit -> e2e), but maintenance and complexity go from bottom to top (e2e -> unit). Consequently, the testing design will have to find a good balance in which there is not too much time spent writing tests and have a good indicator of how well the system works. This is why a lot of unit tests, a fair amount of integration tests, and a handful of e2e tests will be written.

Persistence

Persistence is referred to as the worrying thing about the application's data storage. Analyzing the requirements of the project, it has been concluded that user data will be needed to access the application, to be displayed around the application and to store the posts created by all the users. The type of database will be a Relational Database PostgreSQL one as specified when the technologies where chosen. The database tables will be:

- **Users**. The main table of the database that contains the information of the users registered in our application. The data that will be stored in this table will be the name
    - Name: users
    - Fields: id, name, username, password, email, bio, picture.

  - Primary key: id

- **Posts.** Database table with all the posts from all the users. Each post will contain the title, description and the main content of the post.
  - Name: posts
  - Fields: id, title, picture, description, content, created_at, user_id.
  - Primary key: id
  - Foreign key: user_id to table users
- **Post Liked by Users** A table that defines the many to many relationships between multiple posts liked and the users that liked them.
  - Name: posts_liked_users
  - Fields: post_id, user_id
  - Foreign key: post_id to table posts
  - Foreign key: user_id to table users
- **Follows.** Table to be able to create a "following" relationship between users.
  - Name: follows
  - Fields: following_user_id, followed_user_id
  - Foreign key: following_user_id to table users
  - Foreign key: followed_user_id to table users

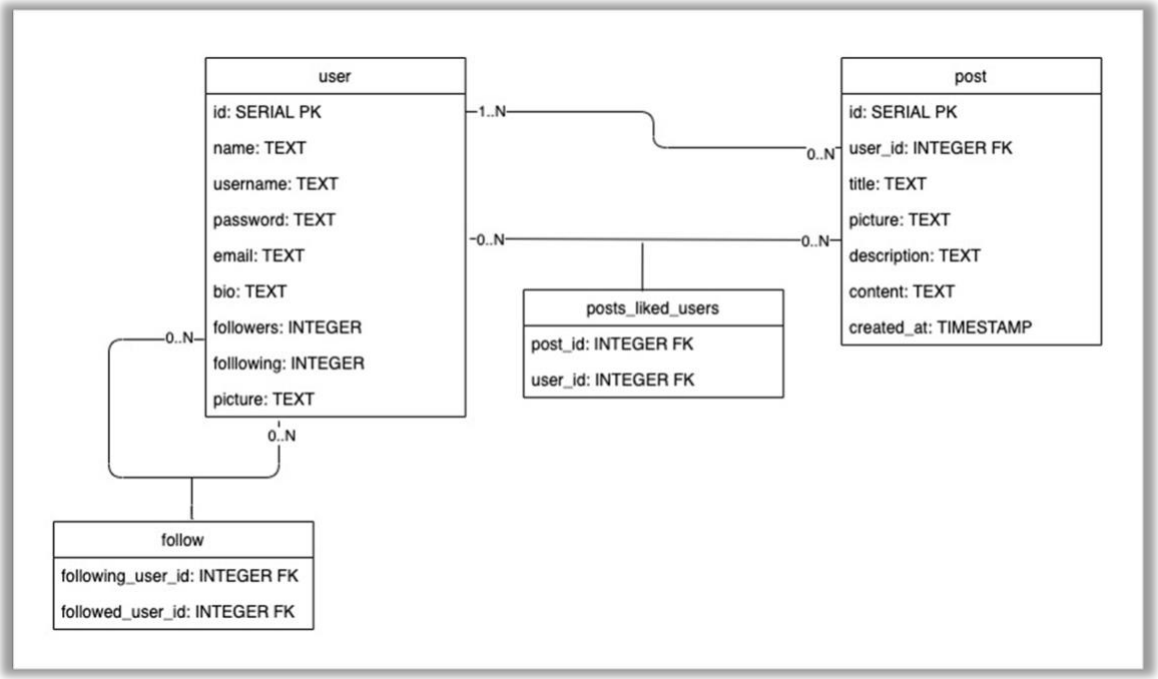


Figure 63 Project Database Design

Interfaces Design

Before starting to carry out the application, the design of the interfaces will be done for decision-making in design stages and not in development, which will speed up this last phase. Those designs will reflect the functional requirements of the app previously described. A thing to keep in mind is that the more accurate the interface design is now, the fewer decisions and developments will have to be made later. Now the previous designs made will be exposed, explaining the requirements applied in each one.

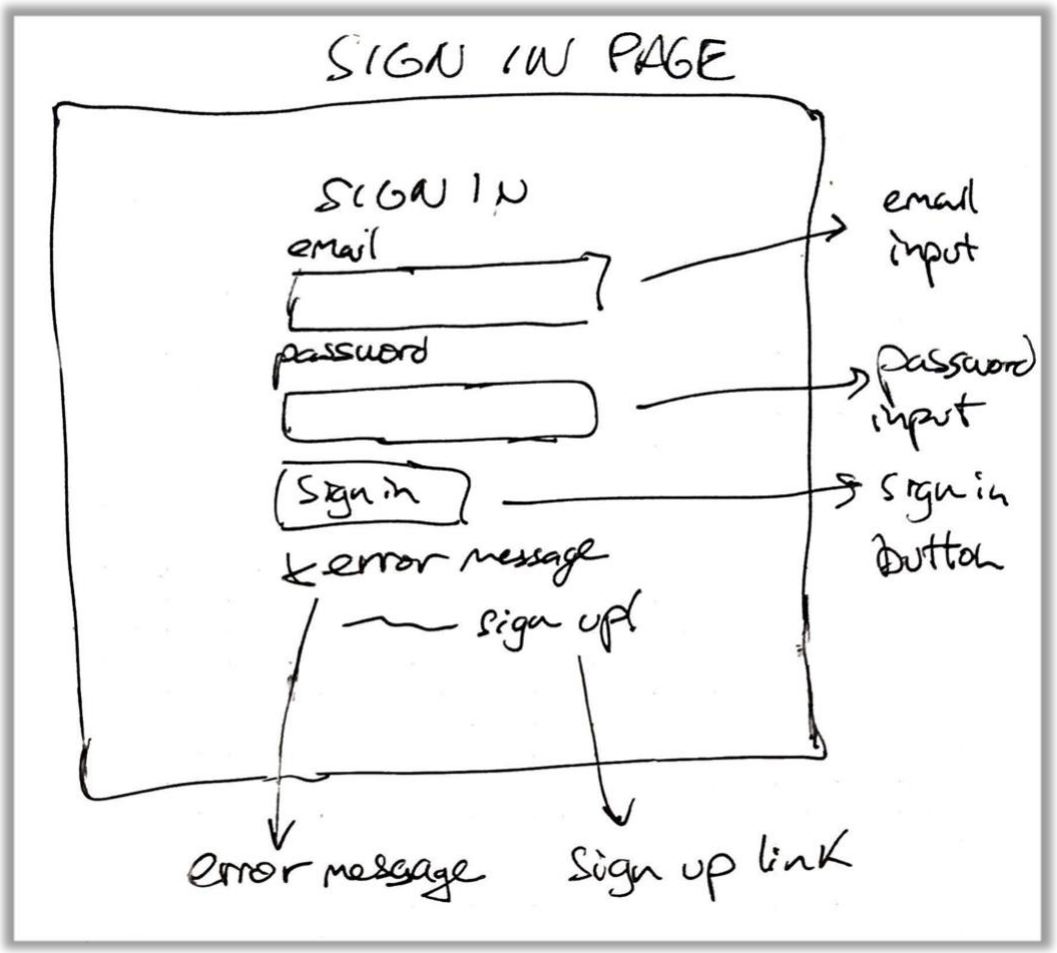


Figure 64 Sign In page mockup

The first interface the user will see will be the Sign In page where the user will be able to sign in into the app using the email and password. If the user does not have an account, there is a link to the Sign Up page to sign up and create a new account. Here would be the functional requirement RF02 in which the user can sign in into the application.

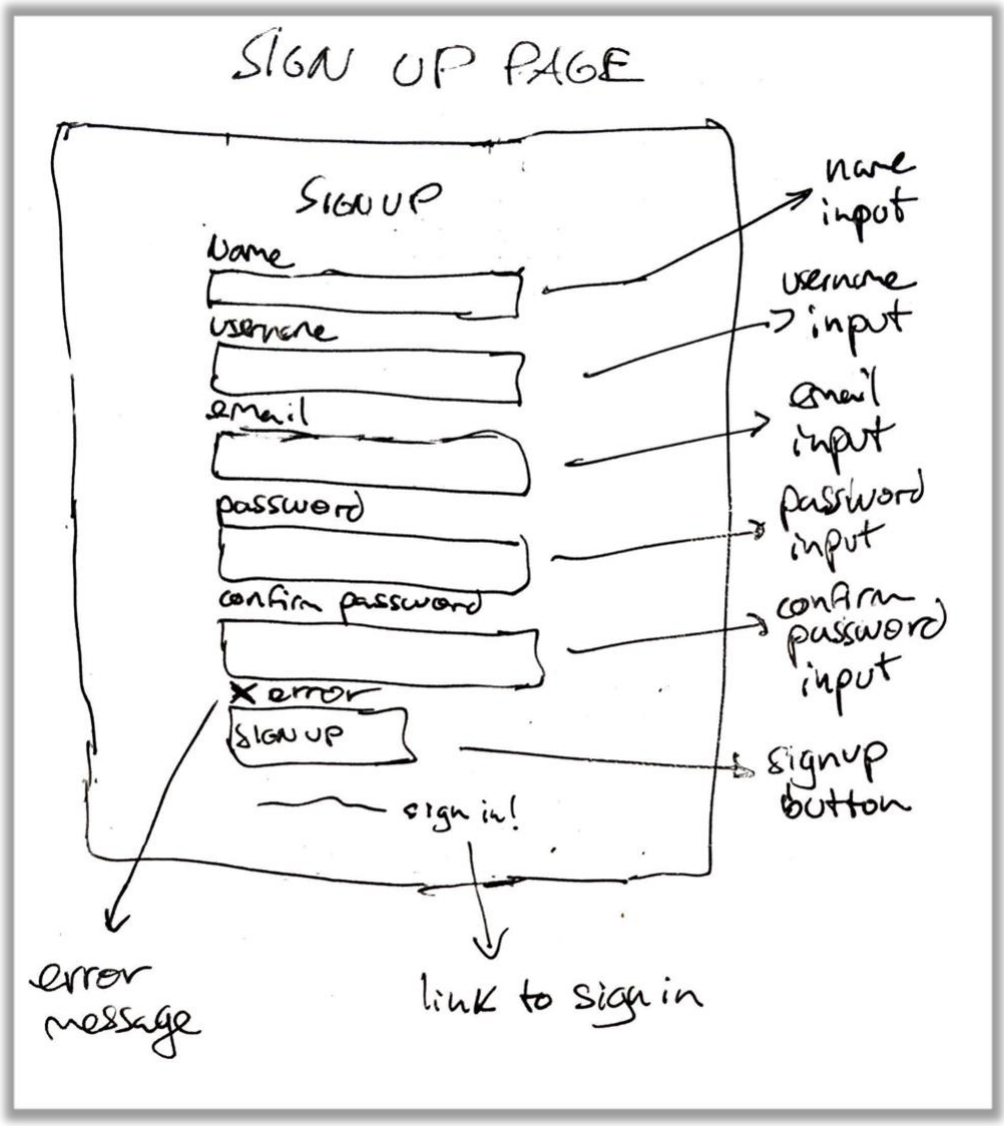


Figure 65 Sign Up page mockup

In the Sign Up Page, the user will be able to create a new account submitting the data required to do so. There is also a link to the Sign In page in case the user already has an account created. This interface reflects the functional requirement RF01 since the user is able to sign up in the application and create a new account.

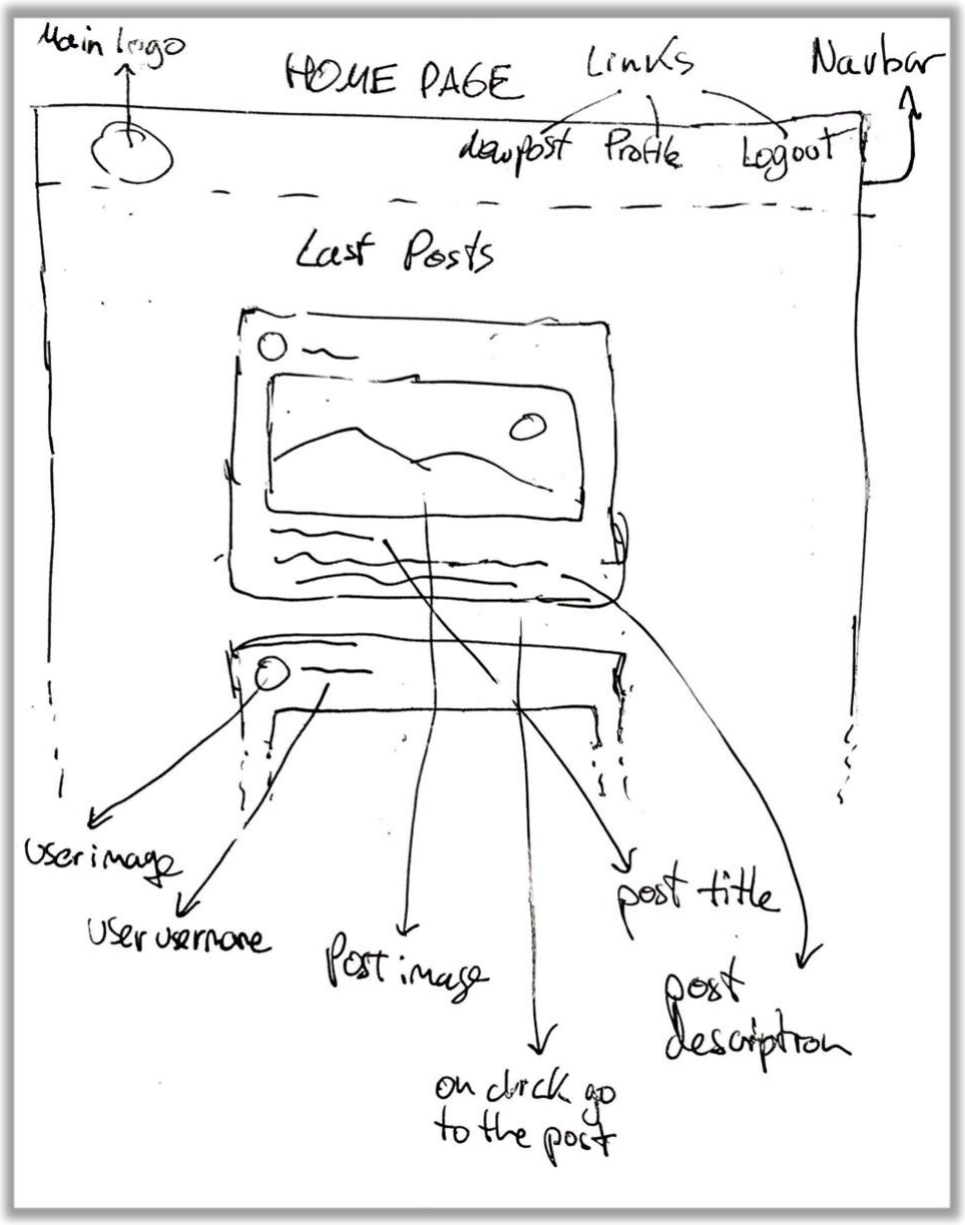


Figure 66 Home Page and Navbar mockup

Once the user has successfully Signed In the application it will be redirected to the Home Page. On top of it and the other pages of the project, there will be the Navbar. The Navbar will have the main logo which will redirect to the Home Page on the left side and three links on the right side to redirect to the New Post Page, Profile Page, and to Logout from the app and end the current session. This last button will redirect the user to the Sign In Page. Thanks to these elements the project will accomplish with the Functional requirements RF03, RF04, and RF05.

The main section of the Home Page will display the lasts posts submitted by the users in a scrollable way. When the user clicks the post, it will be redirected to the Post

Page with the content of the post selected. The Functional requirements RF14 and RF15 are accomplished in this interface too.

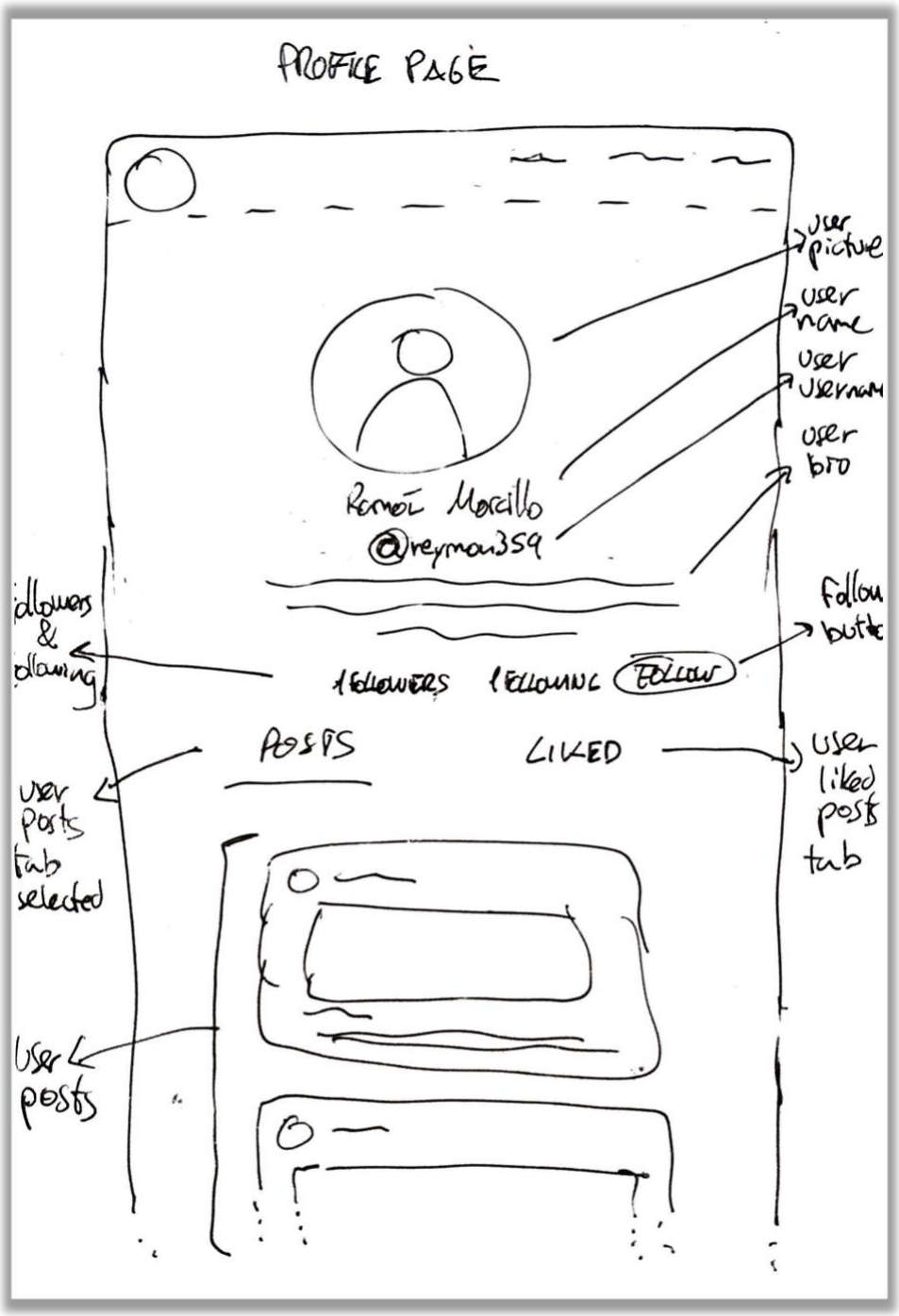


Figure 67 Profile Page mockup

When the user navigates to the Profile page it will be able to see the information of a user and its posts and the ones liked from other users. The current user will be able to follow the user clicking the follow button. If the profile is the one from the current

user, then the button will be replaced with an edit button that will redirect the user to the Edit User information. Here would be the functional requirements RF06, RF07, RF09, RF10, RF11, RF12, RF13.

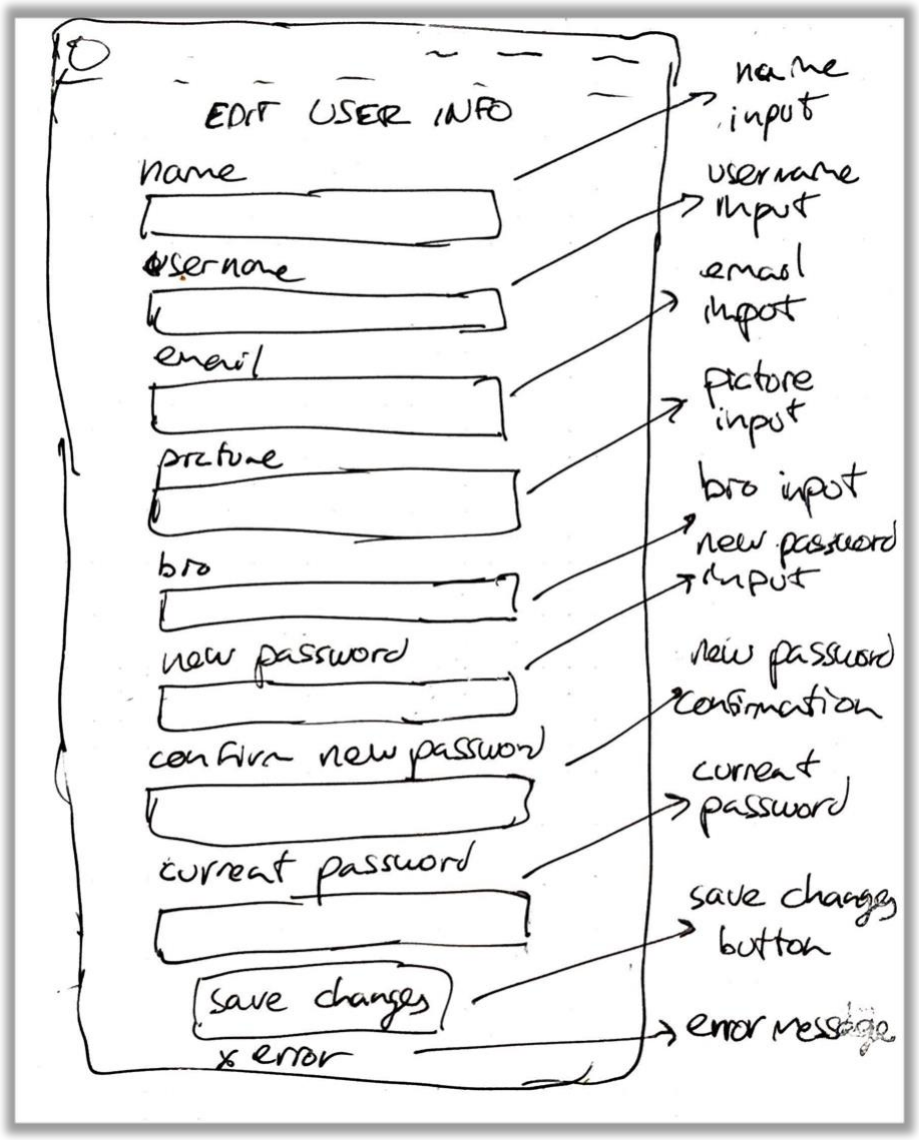


Figure 68 Edit User Information mockup

In this interface, the user will be able to edit any field of its own information if it has the need to. The Functional Requirement RF08 would be accomplished here.

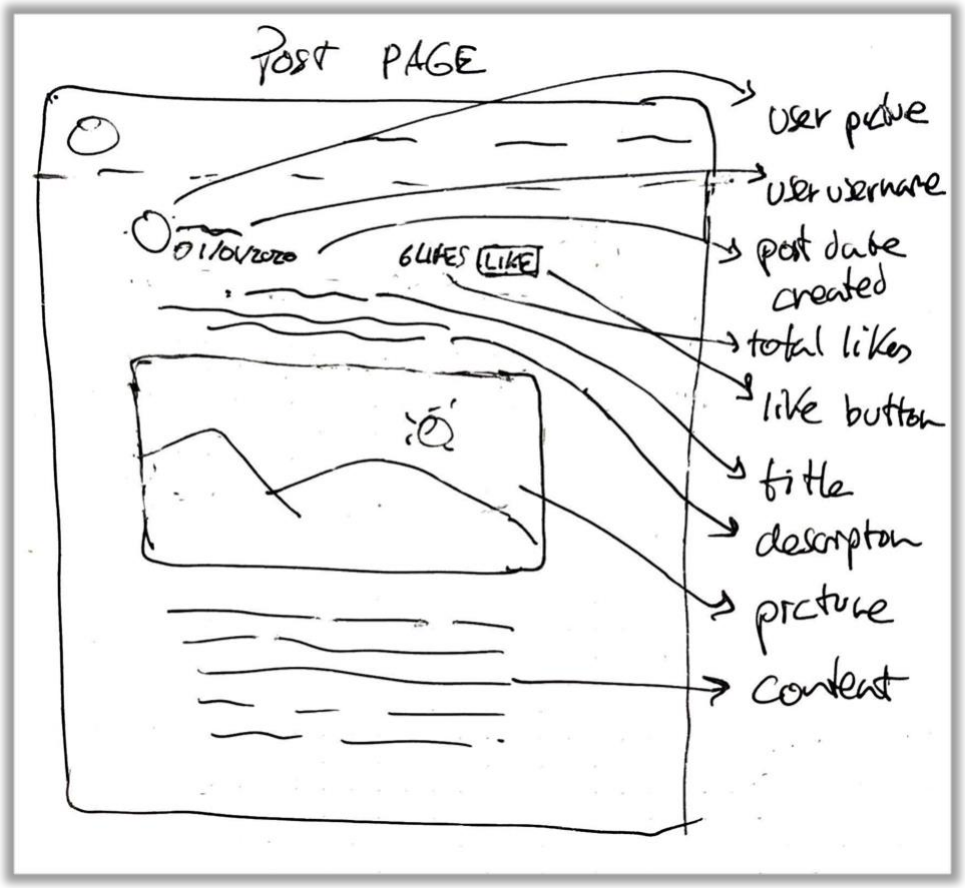


Figure 69 Post Page mockup

The Post Page will display all the information of a post and the picture and username from the user that created it. It will also have a Like button so the user can like the post or unlike it if it is already liked. This button will be replaced by an Edit button and a Delete button if the post is from the current user signed in. The Edit button will redirect the user to the Edit Post Page and the Delete one will delete the post. The Functional Requirements RF11, RF16, RF19, and RF20 will be implemented here.

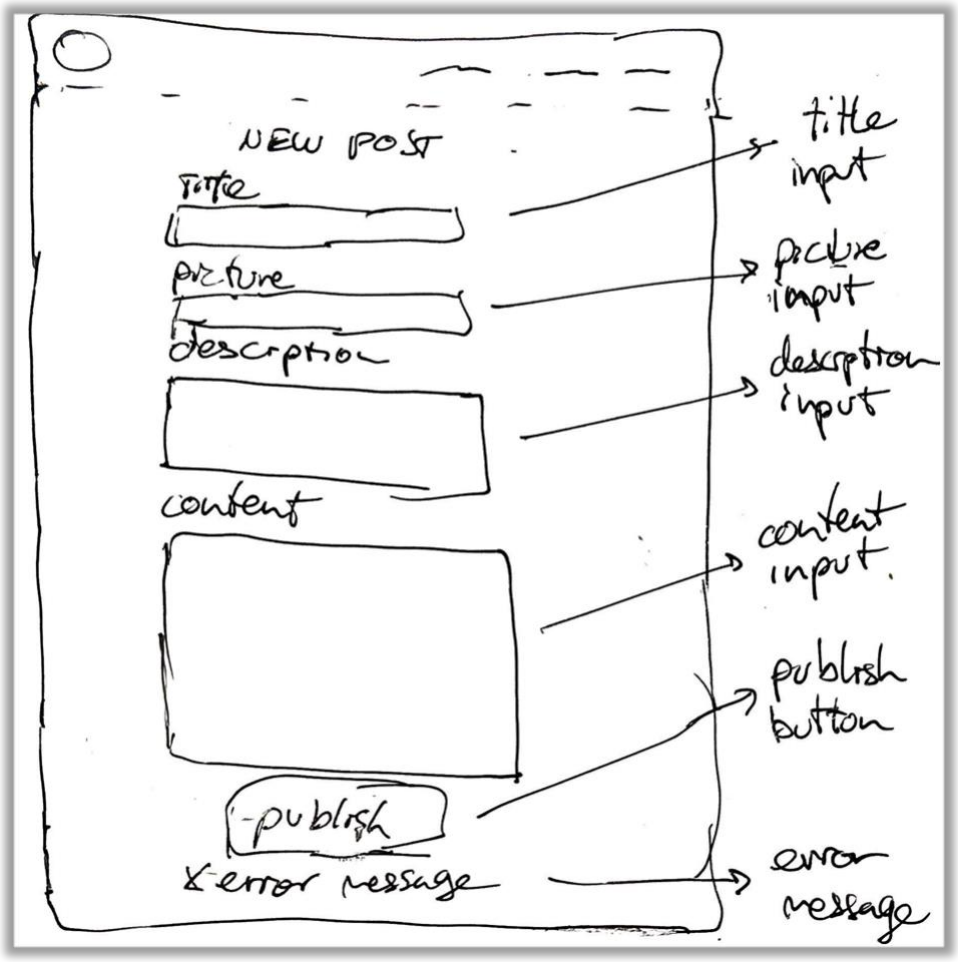


Figure 70 New Post Page mockup

The New Post Page will be the one where the user will be able to submit a new post after submitting the required data representing the Functional Requirement RF17.

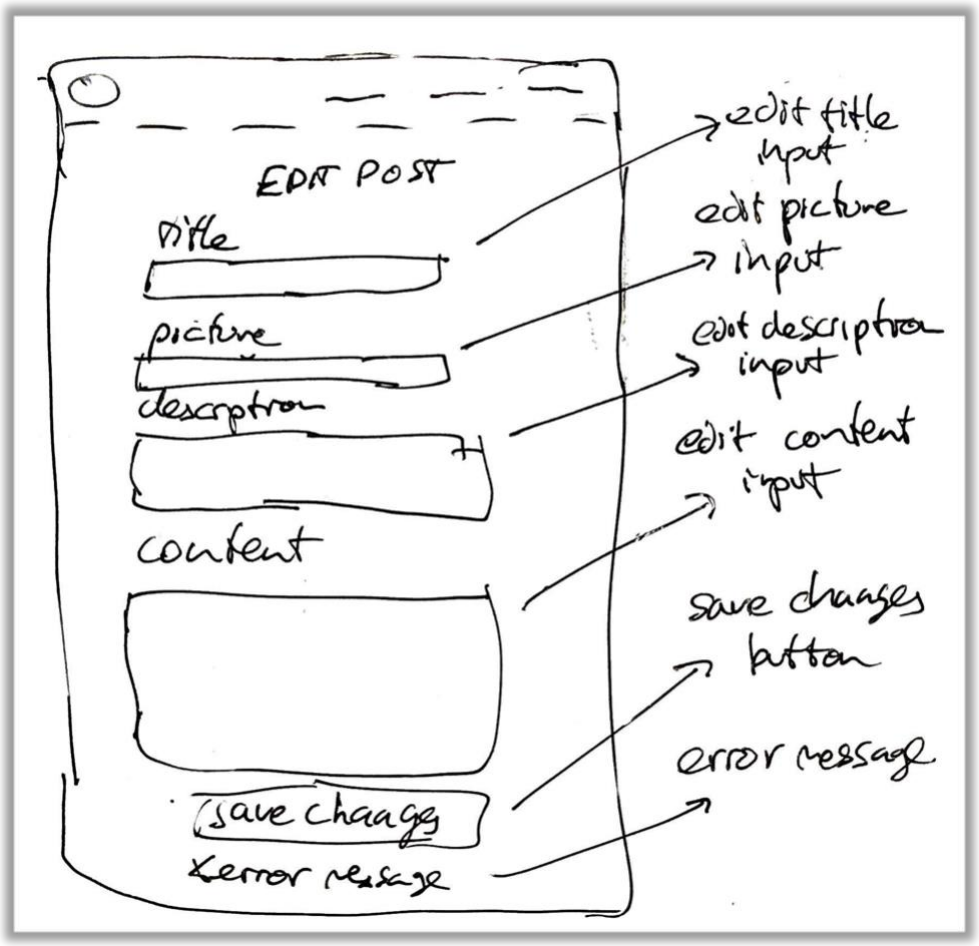


Figure 71 Edit Post Page mockup

In the Edit Post page, the user will be able to edit its post by updating the fields desired and save the changes. The Functional Requirement RF18 is accomplished here.

IMPLEMENTATION

This section describes the development process that has been carried out to develop the project once the previous phases have been completed and having a clear design on which to base it. As previously mentioned, I have followed a Tracer Bullet Development (TBBD) which main idea is to have the first version of the software from start to finish with all the pieces to work together, from start to finish.

With this skeleton in place, then proceed, iteratively, in time boxes to add functionality to the growing system. This style of development dramatically reduces risk and helps keep feedback loops short. Other features of TBD are:

- It makes the developer look for the important requirements, the ones that define the system. Look for the areas where there are doubts, and where there are the biggest risks. Then prioritize the development so that these are the first areas that are coded.
- Is consistent with the idea that a project is never finished: there will always be changes required and functions to add. It is an incremental approach.
- While Prototyping generates disposable code, Tracer code is lean but complete and forms part of the skeleton of the final system.

Authentication

The first point to implement end to end has been the Authentication which consisted of the user being able to access the application, create a new account, and sign in with it into the app. This one has probably been the hardest because of the previous steps needed to develop it, setting up the backend, database, and React Client between others.

The authentication method implemented has been a cookie-based authentication system and the authentication process will be this one:

- A user logs in with a username and a password. The server compares the received username and password to the ones stored in the database.
- If the comparison was successful, the server will generate a token and will set it as a cookie.
- Each time a request is sent, the server will retrieve the username from the stored token on the cookie header and will send the data back accordingly.

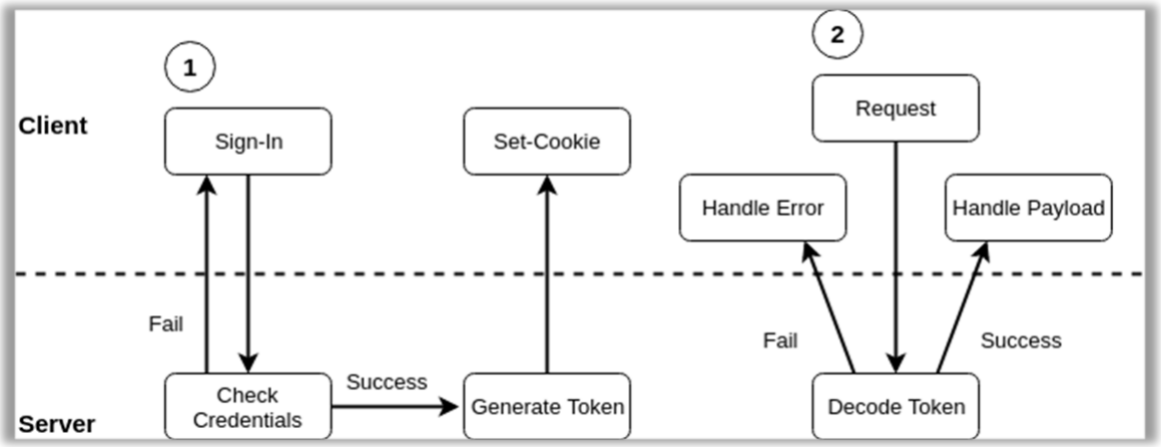


Figure 72 Cookie Based Authentication System of the Project

Everything in the authentication process will be encrypted, confidential information will never be stored or sent in its raw form to prevent the fact that the data could be stolen in case of a database violation or request hijacking. To follow this approach in the application:

- Passwords will always be stored in an encrypted form in the database using an algorithm called Bcrypt which has the ability to compare the password in its raw form with the encrypted one, which will help to authorize the user.

- The tabs are independent. That means that once the encrypted string is decoded the username string can be obtained. This kind of encrypted tokens is called JSON Web Token (JWT).

The issues created for this milestone have been

ID Identifier	01_AUTH01
Name	Set up the GraphQL API on Node.js Server
Description	Set up the backend part related to the GraphQL API. The Apollo server, TypeScript, and other features
Types	Feature, Backend
Estimation Points	13
Time Spent	8 (1 week approx.)
Tasks related	- Add Git and GitHub for Version Control - Create the Node.js Express Server - Implement Apollo Server on it with GraphQL - Configure the server project to use TypeScript
Comments	Although not having worked with most of those technologies some procedures where highly helped by the previous time spent researching and doing courses and tutorials.

ID Identifier	01_AUTH02
Name	Set up the PostgreSQL Database
Description	Set up the PostgreSQL Database, the authentication on it, tables, etc. Configure Docker to have the option of having it running in a container.
Types	Feature, Backend
Estimation Points	3
Time Spent	5 (1 day approx..)
Tasks related	- Install PostgreSQL and configure the user and password. - Create database and tables. - Configure Docker to run the database on it.

Comments	First time working with PostgreSQL and some extra research was needed regarding its installation. Using **pgAdmin** helped a lot in the task and in later development.

ID Identifier	01_AUTH03
Name	Implement Users on the backend
Description	Add the User types and methods into the backend.
Types	Feature, Backend
Estimation Points	5
Time Spent	4 (3 hours approx.)
Tasks related	- Create the User type on the GraphQL schema with its Queries, Mutations, and Subscriptions. - Configure GraphQL Code Generator. - Implement the resolvers to retrieve the user data from the database - Add mocked data for users.
Comments	There were some issues using the GraphQL modules and GraphQL code generator with the User types given the lack of knowledge using these libraries which were solved in the end.

ID Identifier	01_AUTH04
Name	Implement the client app
Description	Set up the React application in the frontend with TypeScript and the Apollo Client.
Types	Feature, Frontend.

Estimation Points	8
Time Spent	8 (1 week approx.)
Tasks related	- Use Create React App to create the client. - Configure the client to use TypeScript on it. - Configure Apollo client on it.
Comments	Some other libraries were added like Styled components although they were not used yet in the project.

ID Identifier	01_AUTH05
Name	Implement Sign Up on the backend
Description	Implement the Sign Up method on the Backend so the user can create a new account.
Types	Feature, Backend
Estimation Points	3
Time Spent	4 (3 hours approx.)
Tasks related	- Implement the Sign Up mutation resolver on the user module - Implement the Sign Up provider on the user module - Add validations to the data submitted - Add error messages
Comments	GraphQL Playground and Postman were used to test the mutation properly.

ID Identifier	01_AUTH06
Name	Implement Sign In on Backend

Description	Implement the Sign In method on the Backend so the user can sign in with its account
Types	Feature, Backend
Estimation Points	3
Time Spent	3 (1 hour approx.)
Tasks related	- Implement the Sign In mutation resolver on the user module - Implement the Sign In provider on the user module - Add validations to the data submitted - Add error messages
Comments	It helped to have created the Sign Up before implementing this one. Although they are not the same similarities between both appeared while coding.

ID Identifier	01_AUTH07
Name	Implement sign up on frontend
Description	Implement the Sign Up page on the frontend so the user can submit its info and create an account.
Types	Feature, Frontend
Estimation Points	8
Time Spent	3 (1 hour approx.)
Tasks related	- Create Auth component - Create the Sign Up form - Sign Up with Apollo Client - Style the Sign Up - Add tests to Sign Up

Comments	The estimation on this one was hideously good, it was thought that it would take more time than it actually did. Although more experience was had with CSS-modules than Styled Components, it was a joy to work with this CSS-in-JS library.

ID Identifier	01_AUTH08
Name	Implement Sign In on frontend
Description	Implement the Sign In page on the frontend so the user can submit its email and password to access the app.
Types	Feature, Frontend
Estimation Points	5
Time Spent	4 (3 hours approx.)
Tasks related	- Implement an auth service to handle the sessions. - Sign In with Apollo Client - Create the Sign In form - Style the Sign In - Add tests to Sign In
Comments	After submitting it, the user was redirected to the home page although it was not implemented yet.

ID Identifier	01_AUTH09
Name	Refactor the authentication
Description	Refactor the overall authentication process to reduce tech debt.
Types	Refactor, Feature, Frontend, Backend
Estimation Points	5

Time Spent	4 (3 hours approx.)
Tasks related	- Improve the error messages and add success ones. - Fix the Auth pages background screen - Refactor the folder structure - Remove unused files and styles. - Refactor the use of the styles
Comments	This issue improved the project quality and reduced the technical debt. Also settled up the main theme to use it globally across the app and as an extra, a dynamic background was added to the Authentication Pages.

Navigation

Now that the user is able to access the application properly it will need to access the different pages and navigate between them before loading content to them. So, In order to navigate between different pages, a router will be configured using the react-router-dom package to manage the routes of the application. The router will be implemented in the APP main component of the client. The app routes will be:

- Root: '/'. Default route that will redirect to the Home route which will redirect to the Home Page in case the user is authenticated. If it is not authenticated it will be redirected to the Sign In page.
- Home: '/home'. If the user is authenticated redirects to the Home Page If not it will redirect to the Sign In page.
- Sign Up: '/sign-up'. Redirects to the Sign Up Page.
- Sign In: '/sign-in'. Redirects to the Sign In Page.
- Post: '/post/:postId'. If the user is authenticated redirects to the Post Page with passing the post id as a param. If it is not authenticated it redirects to Sign In.
- New Post: '/new-post'. It redirects to the New Post Page where the user can create a new post if it is authenticated. If not, redirects to Sign In.

- Profile: '/:username'. It redirects to the Profile Page passing the username as the param to see the profile of that specific user. If it is not authenticated it redirects to Sign In.
- All: '*'. This route means that when the URL submitted does not match any of the previous ones the user will be redirected to the Not Found route which redirects to the Not Found Page (404).
- Not Found: '/not-found'. This route redirects the user to the Not Found Page (404).

Main issues created for this Implementation:

ID Identifier	02_NAV01
Name	Set up the router and routes
Description	Add the routing to the application so when the user enters an URL in the browser it is redirected to that page.
Types	Feature, Planning, Frontend
Estimation Points	5
Time Spent	5 (1 day approx.)
Tasks related	- Set up the Component Pages structure - Configure react-router-dom - Set up the routing on the App component - Implement Authentication condition
Comments	The definition of the routes was abstracted in a separate file, so the application imports them from it instead of just writing the names of the routes in every Link. This means that in case renaming a route there is no need to change every component using it.

ID Identifier	02_NAV02
Name	Create the Navbar Component
Description	Create the Application top navbar so the user can move freely between the different pages.
Types	Feature, Frontend
Estimation Points	8
Time Spent	6 (2 days approx.)
Tasks related	- Create the navbar component - Add the Logo - Add the Links to pages - Pass the Current User id in the params to the Profile Page - Add tests to the navbar
Comments	The logo was made to be dynamic as an extra component, so it changes with the main theme defined for the application. Extra time was needed in tests due to problems with authentication on them.

ID Identifier	02_NAV03
Name	Add Logout functionality
Description	Give the user the ability to close the current session in a safe way
Types	Feature, Frontend, Backend
Estimation Points	5
Time Spent	4 (3 hours approx.)
Tasks related	- Remove the current session cookies - Redirect to Sign In after successful Logout

Comments	When talking about removing the cookie, the field "expires" represents the lifespan of a cookie. Beyond the added date the cookie will be deleted by the browser. “document.cookie = `authToken=;expires=${new Date(0)}`;”

Profile

With the user being able to navigate through the app and having its data mocked up on the backend I proceeded to display the user’s information in the Profile Page. This would implement important features for future iterations like dynamic routes using the params or displaying data from the Apollo Server in the client.

To develop this part, it was needed to work making queries being already authenticated because otherwise could not obtain the data. To proceed with this There was previous research on a proper way to do so and in the end Postman pre-request scripts where used. Pre-request scripts in Postman execute JavaScript before a request runs by including code in the Pre-request Script tab for a request pre-processing such as setting variable values, parameters, headers, and body data can be carried out.

Here is the Postman Pre-request Script implemented to log in before the requests.

```
// Sign In Pre-request Script

const apiUrl = pm.variables.get("API_URL");
var graphql = JSON.stringify({
  query: `mutation signIn($email: String!, $password: String!) {
  signIn(email: $email, password: $password) {
    id
    name
    username
  }}`,
  variables: {"email":"uri@uri.com","password":"111"}
})
pm.sendRequest({
    url: `${apiUrl}/graphql`,
    method: 'POST',
    header: {
        'content-type': 'application/json',
        'accept': 'application/json',
        //'x-site-code': pm.environment.get("x-site-code")
        'X-CSRF-TOKEN': 'fetch'
    },
    body: graphql,
    redirect: 'follow'
}, function (err, res) {
    !err ?
    console.log('Sign in Pre-request done! Auth Cookie saved') :
    console.log(`Sign in Pre-request error: ${err}`);
});
```

Figure 73 Postman Sign In Pre-request Script

Main issues created for this Implementation:

ID Identifier	03_PROF01

Name	Implement GraphQL query to retrieve the user
Description	Implement in the Apollo server the GraphQL query to get the user by its username
Types	Feature, Performance, Backend
Estimation Points	8
Time Spent	6 (1 day approx..)
Tasks related	- Investigate and create a Postman Pre-request Script - Add tests to the query. - Create the resolver. - Add the provider to retrieve the data properly. - Implement cache in the query to improve performance.
Comments	Researching and implementing the Pre-request script it was a great discovery.

ID Identifier	03_PROF02
Name	Retrieve user information in Profile Page
Description	Get the user information in the profile page based on the username provided in the params.
Types	Feature, Frontend
Estimation Points	5
Time Spent	4 (3 hours approx.)
Tasks related	- Create the Profile component structure - Get username by route params - Create a query in Apollo Client
Comments	After the creation of the Profile component the structure developed acted as an example to structure the next components.

ID Identifier	03_PROF03
Name	Display user information in Profile Page
Description	Display the user information in the Profile Page based on the mockups.
Types	Feature, Style, Frontend
Estimation Points	8
Time Spent	5 (1 day approx.)
Tasks related	- Display the User information - Add tests to the Profile Page - Add Styles to the Profile Page - Implement the Follow button just in other users' profiles
Comments	For the default user image the API robohash.org was used which gives you a unique image based on the param that you send in the route. For example: "https://robohash.org/<unique_param>" The param used in the application was the username which is unique to each user.

Home

The main app interface, the Home Page, was the next one to be implemented. Adding this page meant more features and work than the later ones because a new Type had to be added to the code: The Post Type. This addition meant that full implementation was required from end to end, similar to the Authentication one.

Main issues created for this Implementation:

ID Identifier	04_HOME01
Name	Implement Schema Post Type
Description	Implement the Post Type on the Schema in the Apollo GraphQL server.
Types	Feature, Backend
Estimation Points	5
Time Spent	5 (1 day approx.)
Tasks related	- Add the Post type on the GraphQL schema with its Queries, Mutations, and Subscriptions. - Create database tables. - Add mocked data to the database. - Configure GraphQL Code Generator.
Comments	While it was estimated to cover this in less than a day in the end it took that time more or less. This and the next one helped to increase the PostgreSQL knowledge which ended in refactoring previous code.

ID Identifier	04_HOME02
Name	Implement Post GraphQL resolvers
Description	Create the resolvers and database providers for the post queries.
Types	Feature, Backend
Estimation Points	3
Time Spent	4 (3 hours approx.)

Tasks related	- Implement the resolvers in the Schema to retrieve the data from the providers. - Implement the database queries in the providers to retrieve the posts. - Add snapshot tests to the resolvers
Comments	Although the only query needed to be implemented for the Home Page was the get last posts query, the get post by id query was implemented too for the next implementation because at that time it was not very difficult to implement and maybe later it was harder to remember the way to do so.

ID Identifier	04_HOME03
Name	Request Posts on Client and display them.
Description	Request the last posts on the home page and display them according to the mockups
Types	Feature, Frontend
Estimation Points	5
Time Spent	4 (3 hours approx.)
Tasks related	- Create the Home component structure - Implement query to get the last posts in Apollo Client - Display the information of the last posts - Add tests to the Home Page - Add Styles to the Home Page
Comments	In order to increase the performance of the application by not including extra libraries and packages for simple implementation a custom function was used to implement the transformation of the date created of each post into a more user-friendly sentence like “2 days ago”. Also, Unsplash API was used to display the post images.

Post

The next most natural interaction the user will have after seeing the posts on the home page will be to click on one of them to read its content. This is why the next implementation will be the Post Page where the user will be able to read the full post. In some way, it will be similar to the Profile Page where according to the params received it will render a specific post.

Thanks to having implemented the get post by id query in the last implementation (although it was not required there) the backend part was covered to start working on the frontend one. Anyway, it was a step from this implementation and so its issue will be represented here.

ID Identifier	05_POST01
Name	Implement Get post by id in the Apollo Server
Description	Implement the GraphQL query Get post by id from the Schema.
Types	Feature, Backend
Estimation Points	3
Time Spent	4 (3 hours approx.)
Tasks related	- Implement the resolver in the Schema to retrieve the data from the providers. - Implement the database queries in the providers to retrieve the post by its id. - Add snapshot test to the resolver
Comments	Nothing to comment on this one.

ID Identifier	05_POST02
Name	Redirect a post card to its Post Page
Description	Implement the feature to click on a post card to be redirected to its Post Page
Types	Feature, Frontend
Estimation Points	5
Time Spent	2 (30 min. approx.)
Tasks related	- Wrap post card into a router Link element - Update styles to it - Send the post id in the params to the Post Page
Comments	When wrapping the card in a Link some of the styles where lost because the Link default styles where being applied so had to update those too.

ID Identifier	05_POST03
Name	Display the post information
Description	Display the post information in the Post Page according to the post passed by params
Types	Feature, Frontend.
Estimation Points	5
Time Spent	4 (3 hours approx.)

Tasks related	- Set up the component structure - Get post id by params - Make the Apollo query - Display the post information - Add styles to the Post Page - Add Like button - Add Edit and Delete buttons - Add tests to the Post Page
Comments	There was implemented a check the current user logic in order to display the like button or the edit and delete ones. I was inspired by Medium posts to style it.

New Post

The next implementation consisted of giving the user the ability to create posts which are the main content of the social site. To do so the user would navigate to the New Post Page where there would be a form to fill with all the information related to it. Once the user submits the data and passes the validations the post will be created, and the user will be redirected to the Post Page to see it. The mutation returns just the id which is passed by params to the Post Page.

ID Identifier	06_NEW_POST01
Name	Implement the add post mutation
Description	Implement the GraphQL mutation add Post from the Schema.
Types	Feature, Backend

Estimation Points	5
Time Spent	4 (3 hours approx.)
Tasks related	- Implement the mutation resolvers - Implement the database providers - Check and sanitize the data - Add error and success messages - Add tests to the mutation
Comments	The delete post mutation was also implemented here although it was not technically needed for the implementation for developing purposes.

ID Identifier	06_NEW_POST02
Name	Create the New Post Page component
Description	Create the New Post Page component, add the new post form and styles to it
Types	Feature, Style, Frontend
Estimation Points	8
Time Spent	4 (3 hours approx.)
Tasks related	- Create the New Post page Structure - Add form with necessary fields - Add styles to the New Post page
Comments	The content input was updated to be a text area with resizing just vertical so it adapts to the user's necessities.

ID Identifier	06_NEW_POST03
Name	Implement the add post mutation on New Post Page

Description	Implement the Apollo Client add post mutation to submit a new post and redirect the user to the Post Page.
Types	Feature, Test, Frontend
Estimation Points	4
Time Spent	4 (3 hours approx.)
Tasks related	- Implement the add post mutation - Add tests to the submit button - Add tests to the Apollo mutation - Redirect the user to the post page once created
Comments	The React Testing Library has a lot of methods that were very useful for the tests here.

Likes System

In this implementation, the goal was the user being able to like and unlike other posts and also visualize the posts itself and other users like in the Profile Page. The behavior to like a post starts when the user enters a post from another user in the Post Page. Here, if the user has not liked the post yet the “Like” button will appear for it to like it. If the user has already liked the post this button will change to the “Unlike” one.

Once the user has liked a post if it navigates to its profile and selects the “Liked Posts” tab the liked posts will appear in the order where the most recently liked to appear first. In this implementation the user posts have been displayed on the profile too.

ID Identifier	07_LIKES01
Name	Implement the Like mutation on Backend
Description	Implement the GraphQL mutation like Post from the Schema.

Types	Feature, Backend
Estimation Points	3
Time Spent	3 (1 hour approx.)
Tasks related	- Implement the mutation resolvers - Implement the database providers - Check and sanitize the data - Add tests to the mutation
Comments	Nothing to comment.

ID Identifier	07_LIKES02
Name	Implement the Unlike mutation on Backend
Description	Implement the GraphQL mutation unlike Post from the Schema.
Types	Feature, Backend
Estimation Points	3
Time Spent	2 (30 min. approx.)
Tasks related	- Implement the mutation resolvers - Implement the database providers - Check and sanitize the data - Add tests to the mutation
Comments	Pretty similar to the like post one which made it easier to implement.

ID Identifier	07_LIKES03

Name	Implement the get user liked posts query
Description	Implement the GraphQL query get user liked posts from the Schema.
Types	Feature, Backend
Estimation Points	3
Time Spent	3 (1 hour approx.)
Tasks related	- Implement the query resolvers - Implement the database providers - Check and sanitize the data - Add tests to the query
Comments	Here not only the get user liked posts query was implemented but the get user posts too. So later in the profile both tabs would be implemented, the user posts and the user liked posts.

ID Identifier	07_LIKES04
Name	Display the user liked posts on the profile
Description	Display the user liked posts in the liked posts tab on the Profile Page.
Types	Feature, Frontend
Estimation Points	5
Time Spent	4 (3 hours approx.)
Tasks related	- Create a tabs component - Retrieve data from the backend - Display the user posts on the posts tab - Display the user liked posts on the liked posts tab - Style the tabs component - Add tests to them

Comments	The displaying of the user posts was not implemented yet and although it was not required to implement this task it was done to improve the quality of the content from the Profile Page

ID Identifier	07_LIKES05
Name	Implement Like System on Post Page
Description	Implement the Like System on the buttons from the post so the user can like and unlike a post.
Types	Feature, Frontend
Estimation Points	8
Time Spent	4 (3 hours approx.)
Tasks related	- Add the Logic to the button - Implement the mutations on the frontend - Add tests to the button to check it works properly - Refactor Post Page
Comments	A refactor was needed and all the like logic and buttons were moved to a new component to separate it from the Post Page.

RESULTS

The final result of the project has been a Full Modern Web Stack Social Site Application with a Client and Server implementation with interfaces that meet the main requirements. Some of the resulted interfaces variate from the originally designed mockups the reason behind this was to experiment with the Styled Components library and also because of the inspiration of the moment. The following sections will describe the main application interfaces.

Sign In Page

This is the first interface the user will see when entering the application. There is the sign-in form with the fields required to access the application and a link to the Sign Up Page to create an account in case it is its first time. All the styles are based on the main theme configuration, so it adapts to whatever theme configuration is provided.

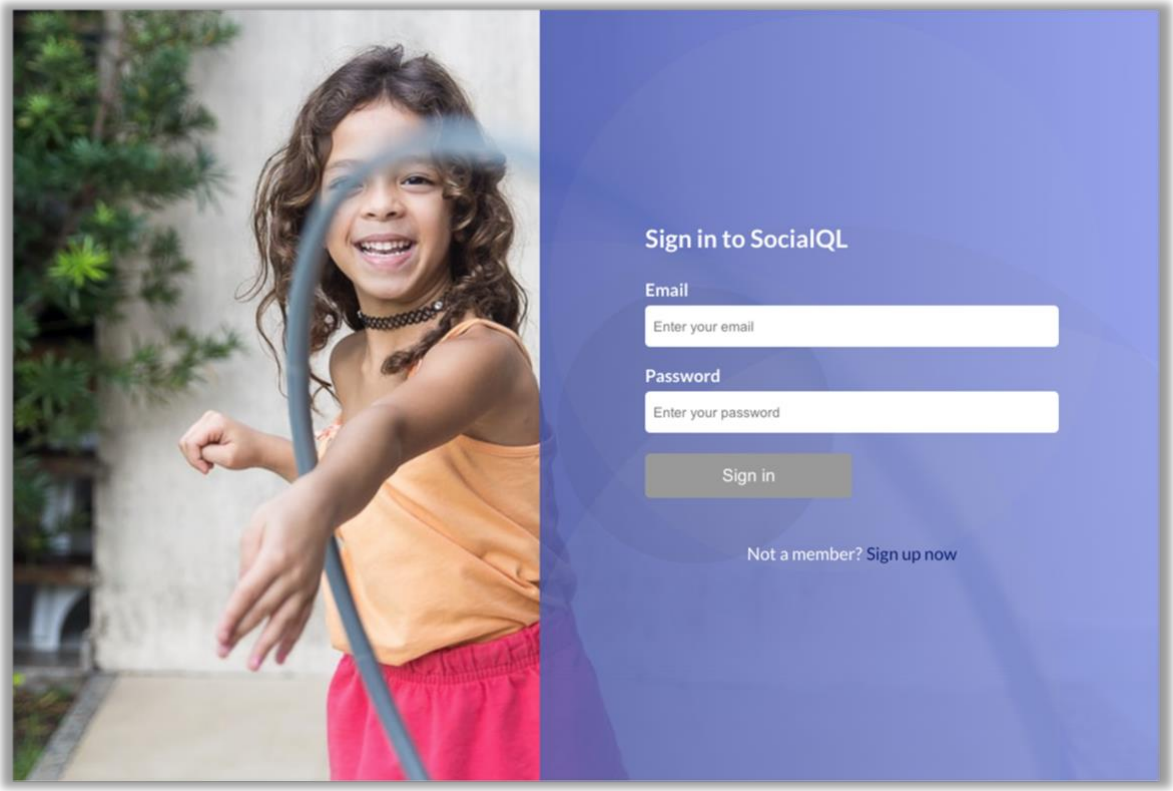


Figure 74 Sign In Page result

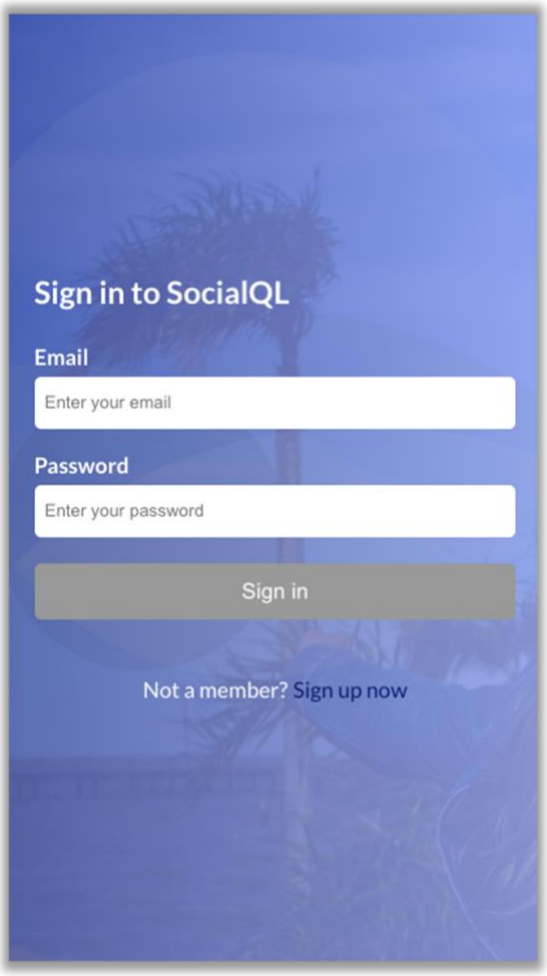


Figure 75 Sign In Page mobile version result

Sign Up Page

If it is the first time the user enters the application and does not have an account, it will end in this interface after clicking the link from the Sign In Page. The Page has a form with all the fields required to create an account to access the application. It is also adapted to the main theme configuration of colors and fonts like the Sign in Page.

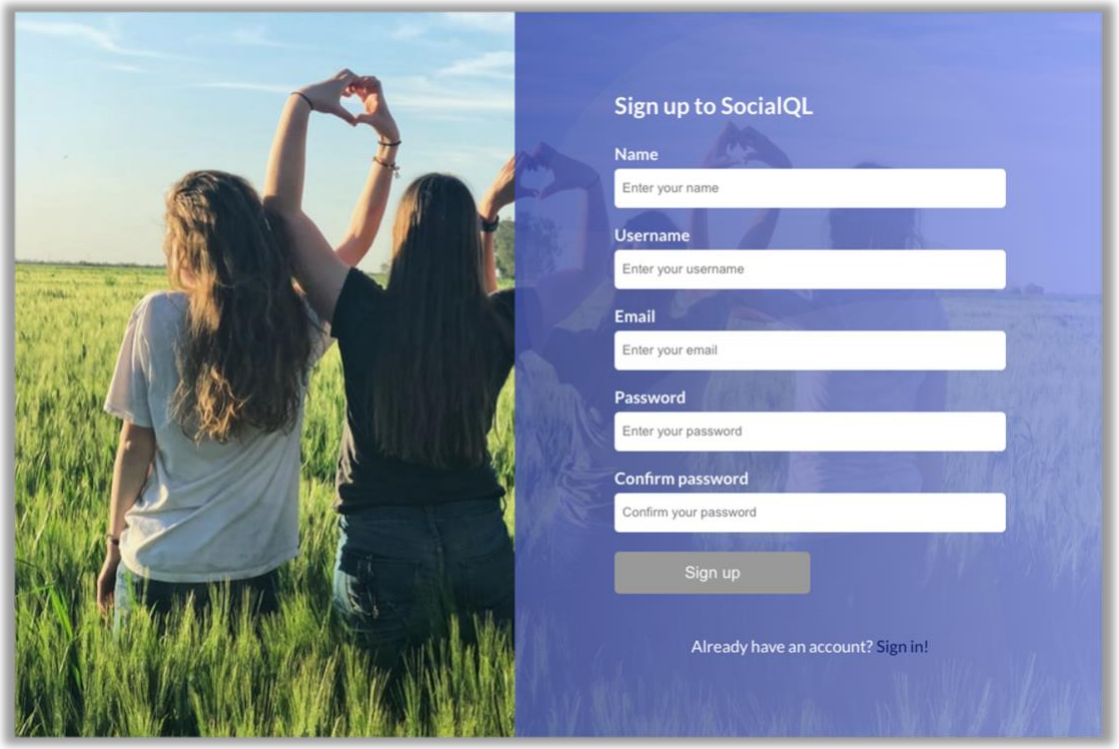


Figure 76 Sign Up Page result

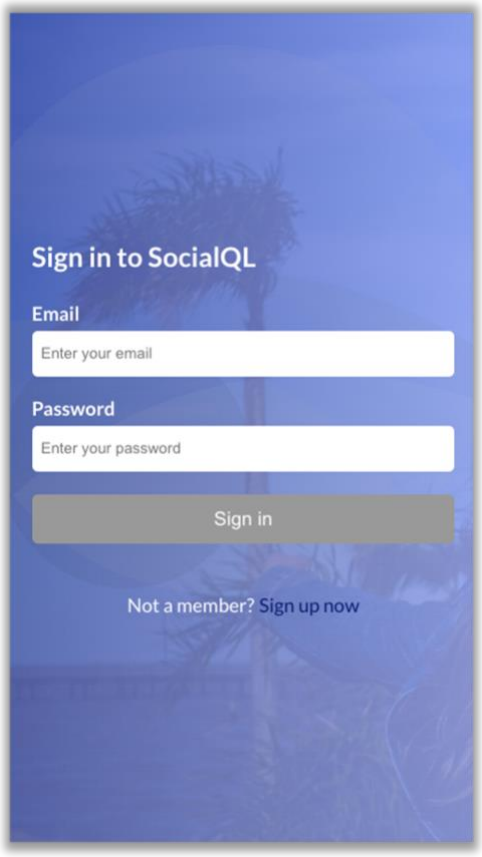


Figure 77 Sign Up Page mobile version result

Home Page

Once the user has successfully signed in it will find the Home Page interface. This interface is more similar to the mockups made in the design phase and its main function remains to be able to see the last posts submitted by the users and to redirect to any of them if selected.

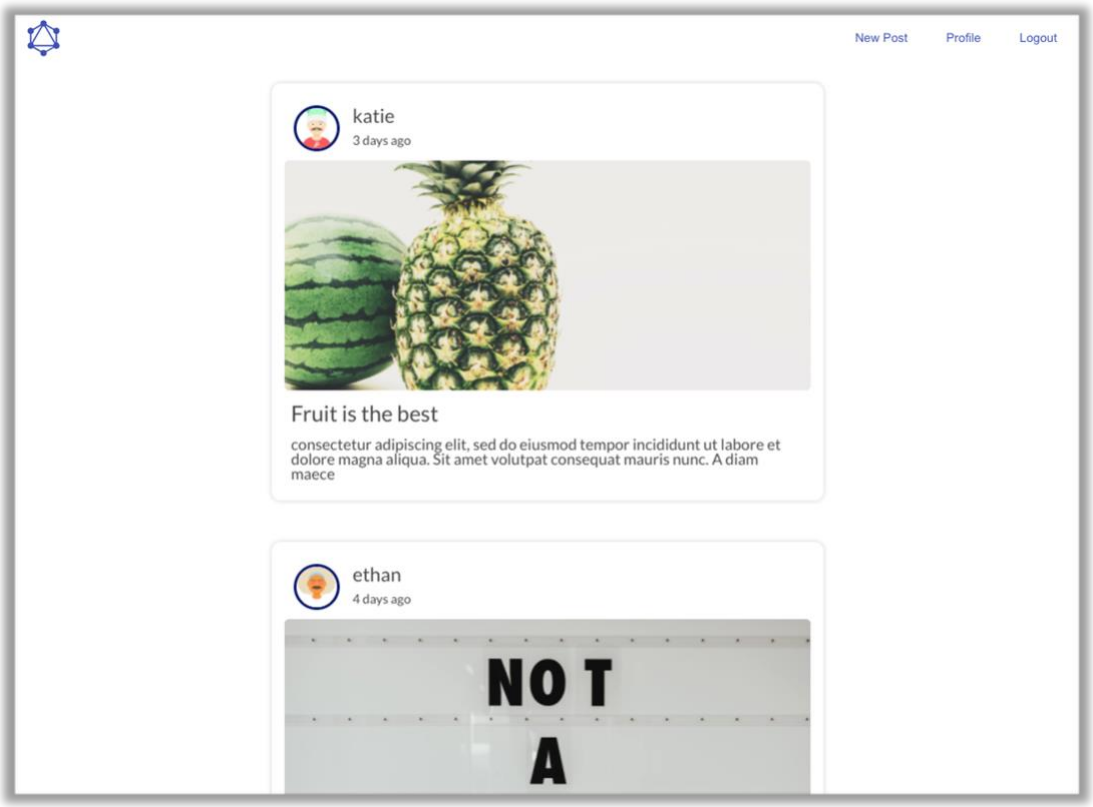


Figure 78 Home Page result

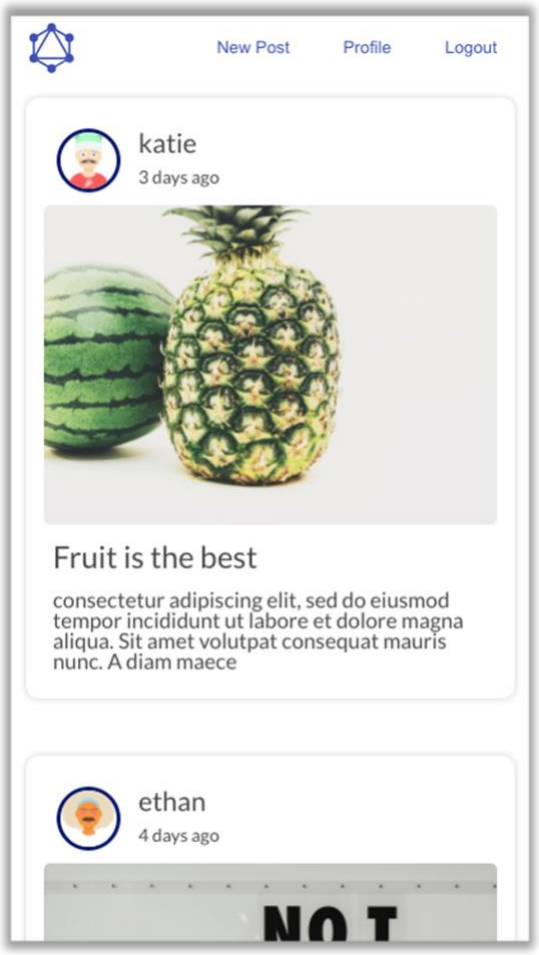


Figure 79 Home Page mobile version result

Figure 80 Navbar with link selected result

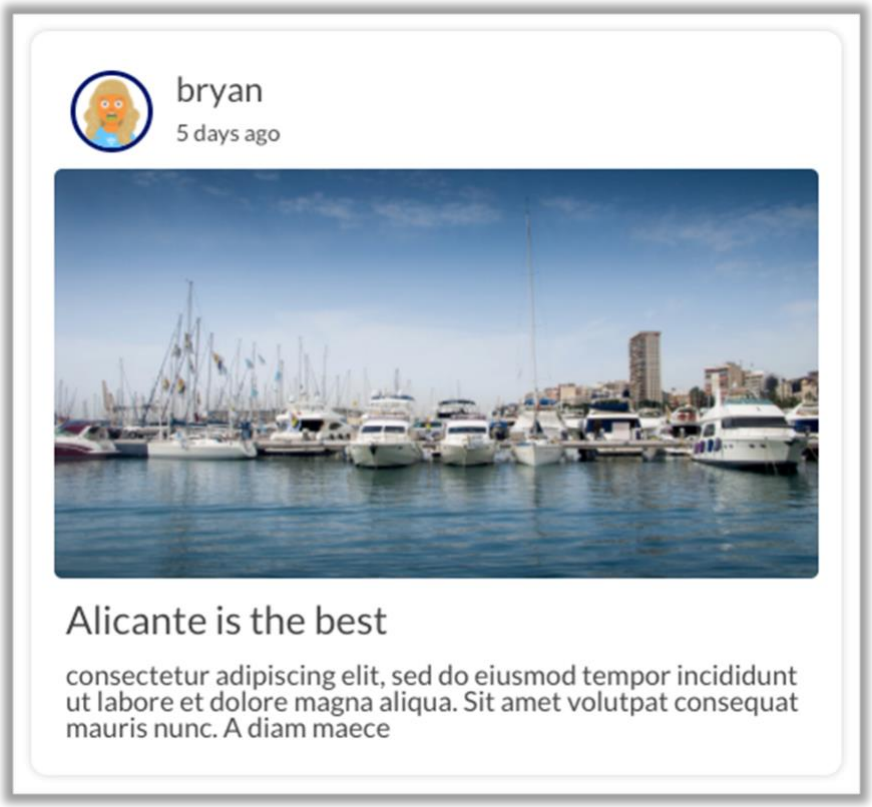


Figure 81 Post Card result

Profile Page

If the user navigates to the Profile Page clicking on the navbar its profile will load in the Profile Page and if he just submits in the params the username of another user, it will be redirected to that user's Profile Page. If the username does not match any user means that the user does not exist and will be redirected to the 404 page. The final result is missing some parts from the mockups because other implementations were prioritized before them and given the lack of time and other external factors and they will be implemented after the Minimum Viable Product.

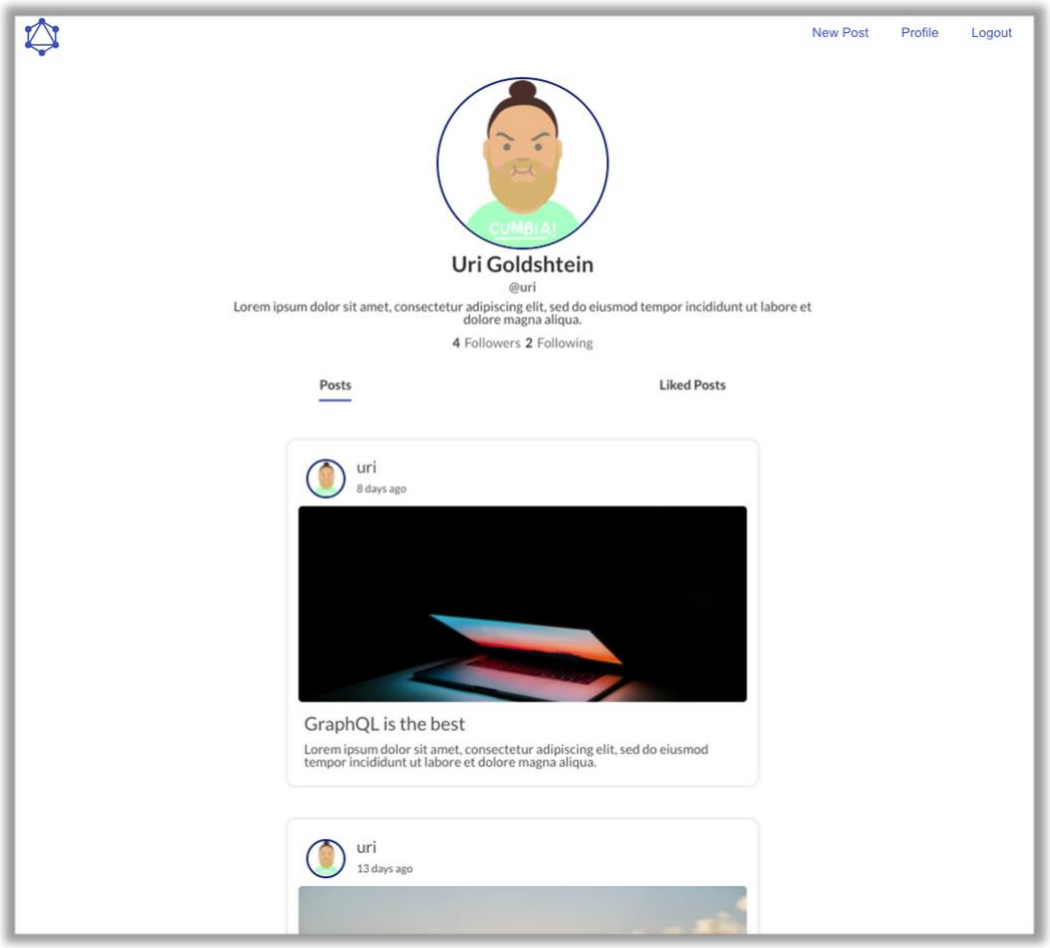


Figure 82 Profile Page result

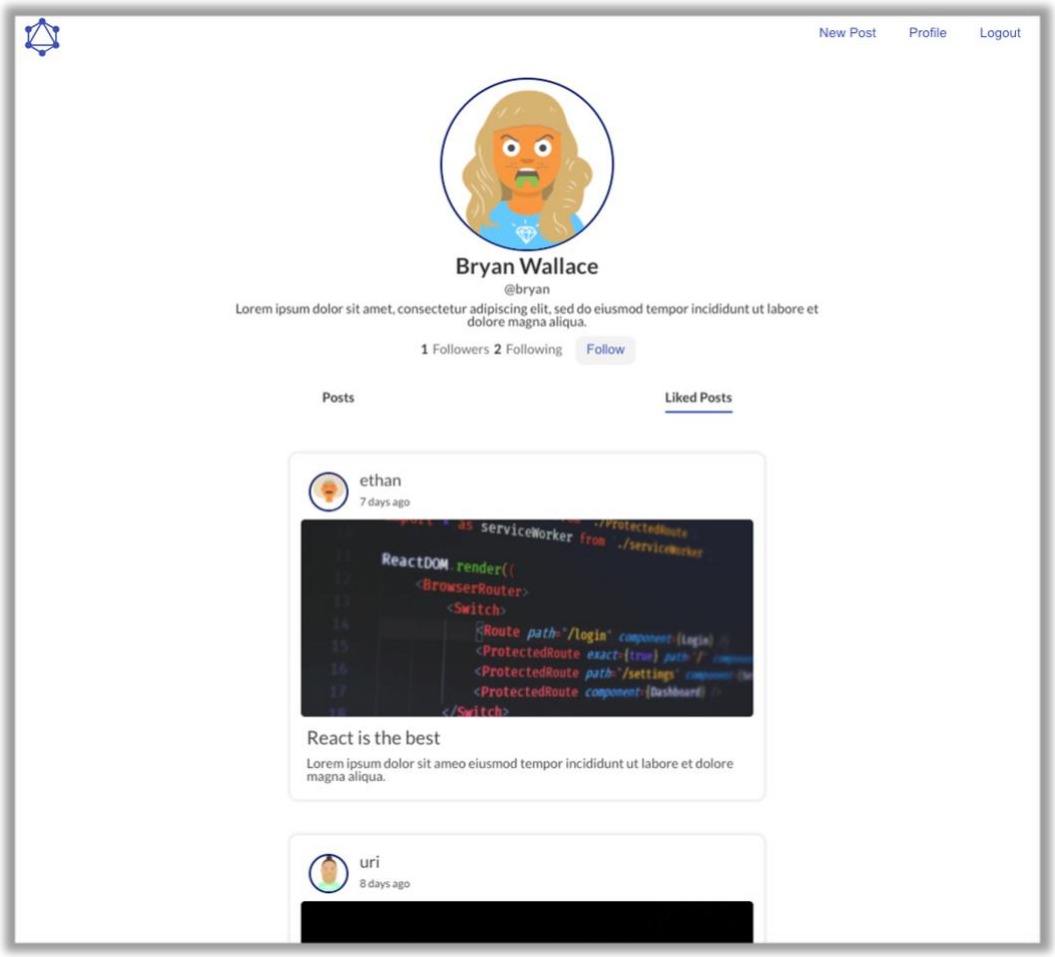


Figure 83 Profile Page Liked Posts Tab Selected Results

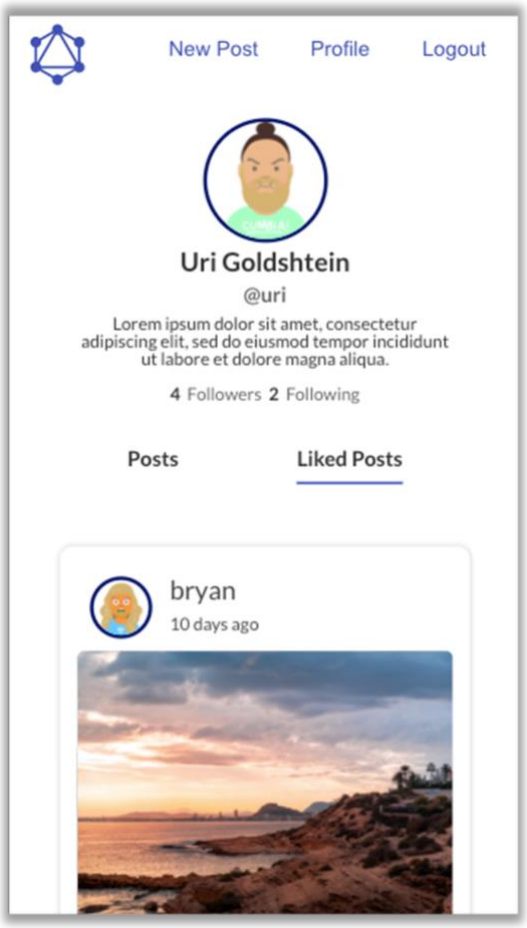


Figure 84 Profile Page mobile version result

Post Page

This is the interface where the user will be able to read its own post or the posts from other users. The design of this interface is pretty similar to the mockups one. The user here is able to read the content of the post, like it or edit and or delete it if it is the author. Also, he will navigate to the author's profile if he clicks on the user info on the top.

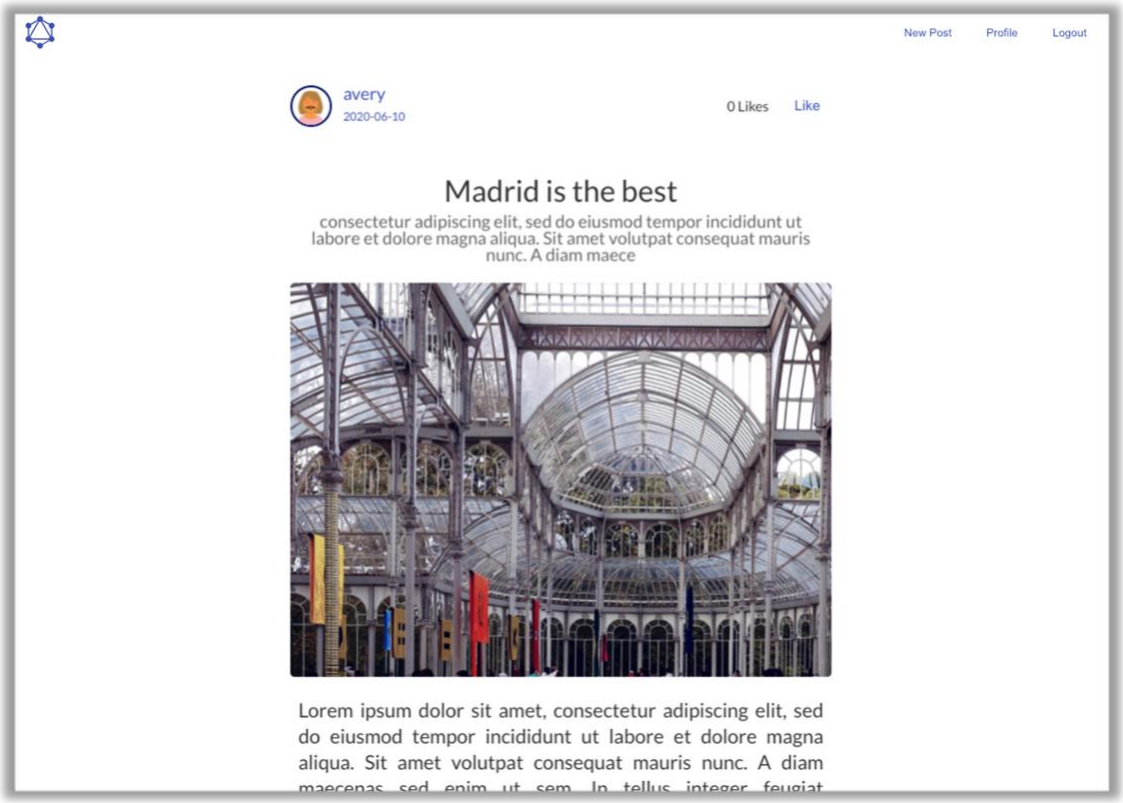


Figure 85 Post Page result

Figure 86 Post Page mobile version result

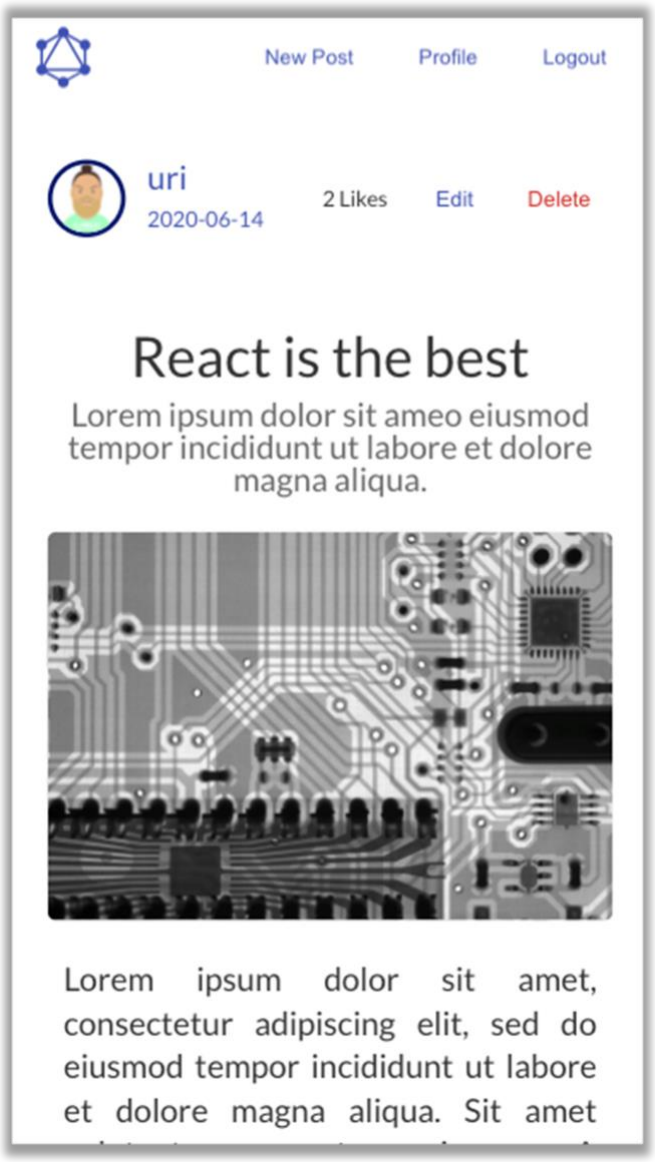


Figure 87 Post Page mobile version own post result

New Post Page

The result of the New Post Page is a form like in the mockup of the same interface with all the inputs to submit the post information: Title, description, image, and content. Once the post has been successfully submitted the user is redirected to its Post Page to see it.

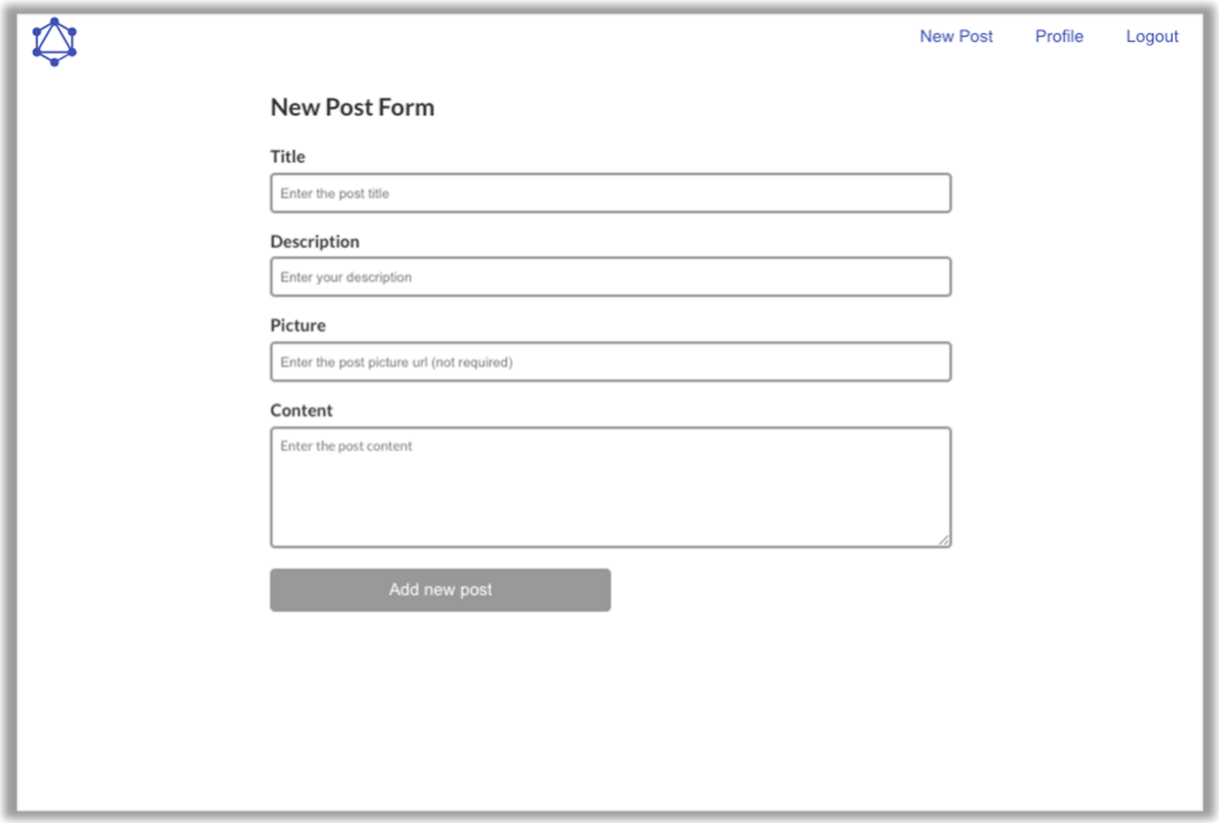


Figure 88 New Post Page result

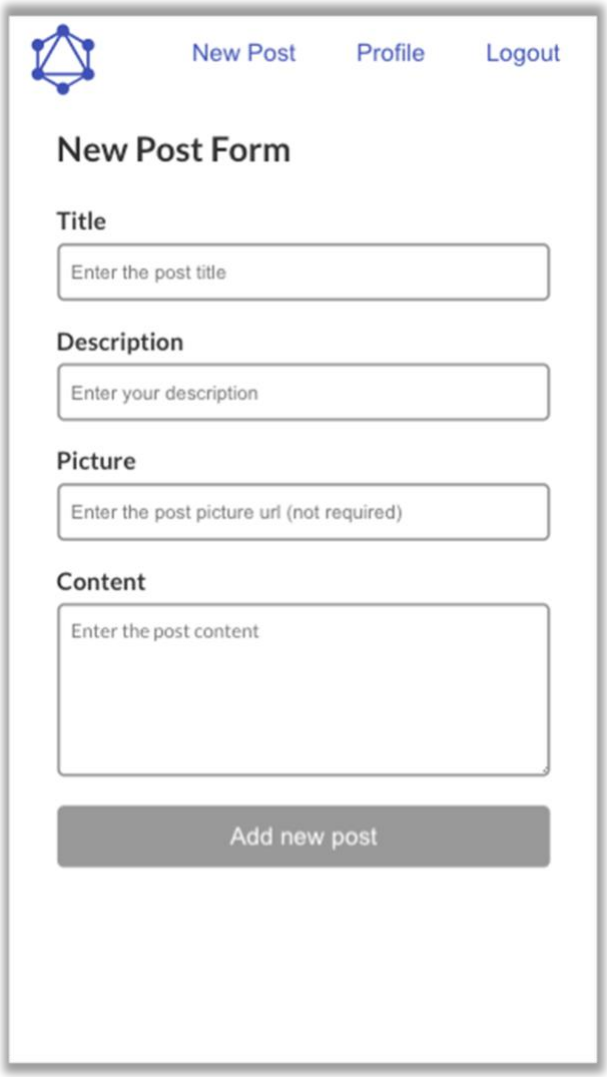


Figure 89 New Post Page mobile version result

CONCLUSIONS

As a conclusion of this project, it can be affirmed that the main objectives of both the application and the personal ones that were proposed at its beginning have been accomplished.

The application fulfills the main objective of a Social Site where users can share its content and interact with other users' content. The application is secure, the authentication method is up to date and the data is encrypted properly. Also, the design is simple which helps the user to focus on the important things when using the application.

I would have liked to add more features in some of the project parts but because of the workload in the work environment and most important the global situation (COVID-19) and personal problems derived from it, there was a lack of temporary resources. All of this caused to restructure the planning to achieve a proper Minimum Viable Product and some features planned where postponed. In fact, this was good in some way because it made me learn a lot about prioritizing and adapting an actual planned project to new requirements and resources.

The GraphQL API is fully functional and serves the queries it receives from the application successfully. Furthermore, the PostgreSQL database has been successfully created and maintained. And as for the Apollo framework it has been totally implemented in both client and server.

With this application, the value of online teaching and being self-taught has been improved, since the knowledge about the technologies used to carry out the project was acquired through online courses: The use of TypeScript, the React client application, the GraphQL API in NodeJS, the PostgreSQL database and the Apollo framework.

I am very satisfied with myself because I had almost no knowledge about most of these technologies and other technologies used in the project like the testing libraries, which was a great handicap at the beginning of the project and made the research phase longer than expected. But in the end, I have been able to learn about all of them and increase my skills at them. In addition, and to exit further of the comfort zone, this is the first project of this magnitude that I have written totally in English which has helped me to improve my vocabulary in software development in an international way.

I have also experienced the importance of refactoring, testing, and have a decent test percentage coverage of the application which has helped a lot in the development process when implementing new features and to prevent increasing technical debt making the project more maintainable.

But not all the knowledge has been learned from online resources. The design of Architectures like the one used in this project, the importance of refactoring and testing, and the software development methodologies used were acquired throughout the Master in subjects such as Architecture and Pattern designs for Web Applications or Web Development Methodologies.

FUTURE WORK

A software project is never completed at its total and this one is no exception. This is why a series of future ideas will be exposed that can be followed once the Minimum Viable Product has been presented:

- Implement comments in the application. The next natural step for a social site would be to give the users the ability to comment on other users' posts to express their opinions about them. This would add extra value to the posts and would also work as a feedback way to the post author.

- Implementation of real-time chats or messaging system for communication between users. Another feature provided on most social networks nowadays is a way to communicate directly with other users using real-time chats or a direct message. This would increment the use of the social site and the interaction between users.

- Add a tagging system. With this feature, users would be able to tag their posts and also filter posts by a specific tag. Adding this would help users to access the content they are more interested in.

- Improve the content of the post giving the users the ability to add images, video, markdown, and other features to the post.

- Implement a search system. Add a search page or search bar so the users can access content by specific text on it or tags. This would also help users to access the content they are more interested in.

- Push notifications. Add notifications to the application so the user gets notified when another user starts following them or a post has been liked by another user.

REFERENCES

This section includes the works and materials consulted and used in the elaboration of the thesis.

1. *HTML Introduction.* Introduction to HTML, the standard markup language for creating Web pages. https://www.w3schools.com/html/html_intro.asp
2. *Javascript.* Javascript information from Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/JavaScript
3. *Adobe Flash.* Adobe Flash information from Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Adobe_Flash_Player
4. *Java Applet.* Java Applet information from Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Java_applet
5. *MVC to Modern Frameworks.* From MVC to Modern Web Frameworks. https://hackernoon.com/from-mvc-to-modern-web-frameworks-8067ec9dee65
6. *Ajax.* Ajax information from Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Ajax_(programming)
7. *GraphQL.* GraphQL information from Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/GraphQL
8. *Basics of GraphQL.* GraphQL - A Query Language for APIs. https://www.howtographql.com/basics/0-introduction/
9. *Risk analysis.* Risk analysis and management. https://www.pmi.org/learning/library/risk-analysis-project-management-7070
10. *SWOT analysis.* SWOT analysis information from Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/SWOT_analysis
11. *TypeScript - Wikipedia.* TypeScript information from Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/TypeScript
12. *TypeScript - Javascript that Scales.* Typescript Website. https://www.typescriptlang.org/
13. *Why and how to write the state-of-the-art.* Tips on writing a good State of The Art. https://blog.babak.no/2007/05/22/why-and-how-to-write-the-state-of-the-art/

14. *How to write a "state of the art" chapter.* Advice on Structuring a State of The Art section. https://writing.stackexchange.com/questions/16724/how-to-write-a-state-of-the-art-chapter/16725
15. *Web Programming languages.* Web Programming languages tracked by Wappalyzer. https://www.wappalyzer.com/technologies/programming-languages
16. *Web Frameworks.* Web Frameworks tracked by Wappalyzer. https://www.wappalyzer.com/technologies/web-frameworks
17. *Wappalyzer.* Wappalyzer, a technographics data provider. https://www.wappalyzer.com/technologies/programming-languages
18. *GraphQL Code.* List of programming languages that support GraphQL. https://graphql.org/code/
19. *Graphql-dotnet.* GraphQL for .NET. https://github.com/graphql-dotnet/graphql-dotnet
20. *Graphql-java.* GraphQL Java implementation. https://github.com/graphql-java/graphql-java
21. *Graphene.* GraphQL framework for Python. http://graphene-python.org/
22. *Graphql-php.* A PHP port of GraphQL reference implementation. https://github.com/webonyx/graphql-php
23. *Lighthouse.* A framework for serving GraphQL from Laravel. https://lighthouse-php.com/
24. *A brief history of the web*. Slides and text on the history of the web. https://www.lopezferrando.com/a-brief-history-of-the-web/
25. *Wp-graphql.* GraphQL API for WordPress. https://www.wpgraphql.com/
26. *Graphql-js.* A reference implementation of GraphQL for JavaScript. http://graphql.org/graphql-js/
27. *Express-graphql.* Create a GraphQL HTTP server with Express.. https://github.com/graphql/express-graphql
28. *Apollo-server.* GraphQL server for Express, Connect, Hapi, Koa and more. https://github.com/apollographql/apollo-server
29. *Graphql-ruby.* Ruby implementation of GraphQL. https://github.com/rmosolgo/graphql-ruby

30. *JavaScript Trends.* JavaScript and Web Development InfoQ Trends Report 2020. https://www.infoq.com/articles/javascript-web-development-trends-2020/
31. *Npm trends.* Compare package download counts over time. https://www.npmtrends.com/
32. *Web Programming languages.* Web Programming languages tracked by Wappalyzer. https://www.wappalyzer.com/technologies/programming-languages
33. *State of JS.* State of the modern JavaScript development. https://stateofjs.com/
34. *Stackoverflow 2020 survey.* Most loved web frameworks. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-web-frameworks-loved2
35. *Stackoverflow 2020 survey.* Most dreaded web frameworks. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-web-frameworks-dreaded2
36. *Stackoverflow 2020 survey.* Most wanted web frameworks. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-web-frameworks-wanted2
37. *React.* React, a JavaScript library for building user interfaces. https://reactjs.org/
38. *Vue.* Vue, The Progressive JavaScript Framework. https://vuejs.org/
39. *Angular.* Angular, One framework. Mobile & desktop. https://angular.io/
40. *Express.* Fast, unopinionated, minimalist web framework for Node.js. https://expressjs.com/
41. *Sails.js.* Sails makes it easy to build custom, enterprise-grade Node.js apps. https://sailsjs.com/
42. *Koa*. Koa, next generation web framework for node.js. https://koajs.com/
43. *express-graphql*. GraphQL HTTP Server Middleware. https://github.com/graphql/express-graphql
44. *Apollo.* The Apollo Data Graph Platform. https://www.apollographql.com/
45. *Apollo Client.* Complete state management library for JavaScript apps. https://www.apollographql.com/docs/react/
46. *Apollo Server*. Open-source, spec-compliant GraphQL server. https://www.apollographql.com/docs/apollo-server/

47. *GraphQL Code Generator.* Generate code from your GraphQL schema and operations with a simple CLI. https://graphql-code-generator.com/
48. *Flux*. An application architecture for React utilizing a unidirectional data flow. https://github.com/facebook/flux
49. *Flux architectural pattern.* An introduction to the Flux architectural pattern. https://medium.com/swlh/an-introduction-to-the-flux-architectural-pattern-674ea74775c9
50. *PostgreSQL.* The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/
51. *Styled Components*. Use the best bits of ES6 and CSS to style your apps without stress. https://styled-components.com/
52. *Clockify*. Simple time tracker and timesheet app for tracking work hours across projects. https://clockify.me/
53. *Git*. Free and open source distributed version control system designed to handle everything from small to very large projects. https://git-scm.com/
54. *GitHub*. GitHub is a development platform to host and review code, manage projects, and build software alongside 50 million developers. https://github.com/
55. *Story Points estimation.* Estimate Story Points in Agile. https://www.tothenew.com/blog/how-to-estimate-story-points-in-agile/
56. *Coolors*. Create the perfect palette or get inspired by thousands of beautiful color schemes. https://coolors.co/
57. *Gitmoji*. An emoji guide for your commit messages. https://gitmoji.carloscuesta.me/
58. Stackoverflow 2020 survey. Stackoverflow 2020 survey most loved web frameworks. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-web-frameworks-loved2
59. *Commitlint*. Lint commit messages. https://github.com/conventional-changelog/commitlint
60. *Visual Studio Code*. Free source-code editor made by Microsoft for Windows, Linux and macOS. https://code.visualstudio.com/
61. *WebStorm*. The smartest JavaScript IDE. https://www.jetbrains.com/webstorm/

62. *GraphQL Playground*. GraphQL IDE for better development workflows (GraphQL Subscriptions, interactive docs & collaboration). https://github.com/prisma-labs/graphql-playground
63. *GraphiQL*. A graphical interactive in-browser GraphQL IDE. https://github.com/graphql/graphiql/tree/master/packages/graphiql#readme
64. *Postman*. The Collaboration Platform for API Development. https://www.postman.com/
65. *GraphQL Schema*. GraphQL Schemas and Types. https://graphql.org/learn/schema/
66. *Resolvers*. How Apollo Server processes GraphQL operations. https://www.apollographql.com/docs/apollo-server/data/resolvers/
67. *Types*. Define the exact same type in multiple services. https://www.apollographql.com/docs/apollo-server/federation/value-types/
68. *React File Structure*. Recommended way to structure React projects. https://reactjs.org/docs/faq-structure.html
69. *Whatsapp Clone Tutorial*. A React based Whatsapp clone. https://www.tortilla.academy/Urigo/WhatsApp-Clone-Tutorial/master/next/step/0
70. *GraphQL Modules*. Toolset of libraries and guidelines dedicated to create reusable, maintainable, testable and extendable modules out of your GraphQL server. https://graphql-modules.com/
71. *Jasmine*. Behavior Driven Development testing framework for JavaScript. https://github.com/jasmine/jasmine
72. *Mocha*. Simple, flexible, fun JavaScript test framework for Node.js & The Browser. https://github.com/mochajs/mocha
73. *Jest*. Delightful JavaScript Testing Framework with a focus on simplicity. https://github.com/facebook/jest
74. *Jest vs Mocha vs Jasmine*. NPM use comparison between Jest, Mocha and Jasmine. https://www.npmtrends.com/jest-vs-mocha-vs-jasmine
75. *Tracer Bullet Development*. Tracer Bullet Development TBD. https://gunnarpeipman.com/tracer-bullet-development/

76. *Bcrypt*. A library to help you hash passwords. https://github.com/kelektiv/node.bcrypt.js
77. *Json Web Token JWT*. JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties. https://jwt.io/
78. *PgAdmin*. pgAdmin is the most popular and feature rich Open Source administration and development platform for PostgreSQL. https://www.pgadmin.org/
79. *Vanilla timeFromNow*. A vanilla JS alternative to the moment.js timeFromNow() method. https://gomakethings.com/a-vanilla-js-alternative-to-the-moment.js-timefromnow-method/
80. *React-router-dom*. Declarative routing for React. https://github.com/ReactTraining/react-router
81. *Pre-request scripts*. Pre-request scripts in Postman to execute JavaScript before a request run. https://learning.postman.com/docs/postman/scripts/pre-request-scripts/
82. *Robohash*. web service that makes it easy to provide unique, robot/alien/monster/whatever images for any text. https://robohash.org/
83. *Unsplash*. The internet's source of freely usable images. https://unsplash.com/