UNIVERSIDAD DE GRANADA

MÁSTER EN DESARROLLO SOFTWARE

TRABAJO FIN DE MÁSTER

Micro-Frontend architecture for web application development: research, analysis, and implementation

Presentado por:

D. Illia Nechesa Dernovyi

Tutor:

Prof. Dr. Juan Antonio Holgado

Curso académico 2020 / 2021

ABSTRACT

[Context] Nowadays, most web applications are developed with a monolithic architecture, or based in a microservice architecture, such as the one of my current company, a single page application whose purpose is to manage and monitor different solar plants around the world. However, micro frontends are increasing in popularity, as it enables the splitting of monolithic frontends into independent end-to-end applications. Huge amounts of companies, such as Spotify, Ikea, or Starbucks among many others, have already adopted this new architecture. On the other hand, many others reject it due to the lack of knowledge or information.

[Objective] The goal is to study the micro frontend architecture, its advantages, and disadvantages, and to check its feasibility for our company, whether it suits us or not, and why many companies are adopting it.

[Method] We will perform deep research about the micro frontend architecture and different ways to achieve it, making a proof of concept in those ways, to try out their feasibility. This might be achieved by developing small applications and trying to establish communication between them in different ways.

[Result] We created a fully working application, which is perceived as a monolithic one, but in the background, it is a composition of many micro-applications, which communicate between them with window events.

[Conclusions] Microfrontend is an emerging architecture, which has been adopted by a lot of companies in the last few years. It has plenty of benefits and many ways to achieve it. However, as it is something relatively new, it is difficult to find concrete information about the best way to implement it. A huge independent and personal research must be done for each company to see its feasibility.

CHAPTER 1: INTRODUCTION

1.1. MOTIVATION

Active internet users are increasing faster than in the last few years. There are more and more people who have access to it, and use it every day. Therefore, the content on the internet is also increasing symmetrically, as there is a need to fulfill users' necessities or inquiries. Just take a look at Facebook. It started as a simple web page to meet and talk with people from Harvard, where Zukerberg was studying. This simple and small social network has evolved so much that now it owns plenty of other applications, and has increased its number of features by a huge number, being the biggest social network worldwide with more than two and a half billion active users.

Some web applications which were launched taking into account no more than thousands of people would use them, now have millions of requests per day, like the one I have mentioned before. They are evolving, becoming bigger, more efficient, and optimized, thanks to the new frameworks, technologies, design patterns, internet providers…

However, even with the current technologies, some web applications are so big that even for a big company it starts to get difficult to maintain. Not only because of the huge amount of code, which at that scale might be unreadable but also for the different business models in the same application. As I said before, current applications, mostly talking about the ones that have a huge amount of requests per day, or too much content, must be modularized or separated, trying to avoid mixing between independent business models.

In my company Grupo EOSOL, the same issue occurs, and we decided to find a solution to it. This is why I decided to find out a new way, or a new architecture, that could solve the issue of having too much code in the same application, and can manage the same application independently, not depending constantly on other features or components of it which are completely independent.

Current companies, especially big ones which I am going to mention later, are trying to solve those issues by applying their solutions. Some of them are implementing their libraries, studying current architectures, and trying to modify them to solve this issue. However, most of them are using ready-to-use frameworks which try to solve this problem in some way.

This is an outstanding moment to find out a way to solve this problem since it is now the moment when the internet is growing too fast, there are more and more web applications and users, therefore more programmers and engineers trying to maintain all those applications.

1.2. GOALS

1.2.1. GENERAL

The general goal is to find a solution that could solve the problem of the maintenance of a big application, oriented mostly to developers. The main challenge is to find a way to restructure, organize, modularize and divide the current application into small ones, where each micro-application has a specific business model and is maintained and run independently from each other. This led to improvements in many company aspects, such as the evolution of the company's business that requires the incorporation of new services and the substitution of current services without affecting the rest of the existing applications of the company.

1.2.2. SPECIFIC

To achieve the general goal, we need to perform some specific tasks or goals. Next, a full list of all the specific goals we want to achieve during the development of the project is presented

- Analysis of the current web application architectures such N-Tier architecture or model view controller paradigm, among others for understanding the main components that support those applications.
- Study the architecture designs used for the development of web applications, such as the Single Page Applications one, the Microservices architecture, and the Serverless one, among others.
- Research about the methodologies generally used for web application development. This allows us to extract patterns for a novel development methodology.
- Investigation of the frameworks that could be used for the project development of web applications, specially oriented to front-end based on Javascript language
- Identification of the main issues to take into account when designing the application.
- Research frontend technologies based on Javascript, and analyze them by comparing them with each other.
- Explore the methodologies that developers used for web development to identify the main issues or aspects to be considered in our project.
- Investigate the approaches proposed in the web community related to support the execution of web micro-applications. A comparison between these approaches allows us to understand the main concerns to be considered for this project.
- Analysis of the Micro-Frontend architecture, including reasons to adopt it, its benefits and issues, and different kinds of approaches to it.

- Test a pilot of the novel framework to a web development team to extract an opinion about the improvements that a new approach could have and obtain feedback from developer users.
- Development of a POC implementing Micro-Frontend architecture, including the requirement specification, the analysis, and the design.

CHAPTER 2: WEB APPLICATIONS

2.1. INTRODUCTION

Nowadays, most web applications are developed based on traditional three-tier architectures (front-end, back-end, database), and follow the typical schema of client-server, which denotes a relationship between cooperating programs in an application, composed of clients initiating requests for services and servers providing that function or service. The client-server relationship communicates in a request-response messaging pattern and must adhere to a common communication protocol, which formally defines the rules, language, and dialog patterns to be used.

Front-end frameworks are evolving providing dynamic single web pages on the client-side that behave as native responsive applications, using Javascript as a base programming language with frameworks such as Angular, React, and Vue among many others. However, before getting focused on the Front-end, it is a good idea to perform a little study of the most common web architectures.

2.2. WEB APPLICATION ARCHITECTURES

The client-server model, or client-server architecture, is a distributed application framework dividing tasks between servers and clients, which either reside in the same system or communicate through a computer network or the Internet. The client relies on sending a request to another program to access a service made available by a server. The server runs one or more programs that share resources with and distribute work among clients.

Currently, there are some architectures based on the client-server one, which tries to improve the initial model, dividing the logic of the web application and implementing one architecture or another depending on the behavior we want to achieve or the way which suits best for us. Those are the N-Tier architectures, and I will also include the Model-View-Controller one, which I am going to explain next.

2.2.1. N-TIER ARCHITECTURES

Before starting to talk about web application architectures, it is a good idea to explain the low-level architecture behind them. In software applications, many layers are being used, and it is important to understand which those are, explaining the N-Tier Architecture.

In N-tier, “N”, refers to a number of tiers or layers that are being used, such as 1-tier, 2-tier, 3-tier… It is also called “Multi-Tier Architecture”. There are different types of N-Tier Architectures, as I mentioned before.

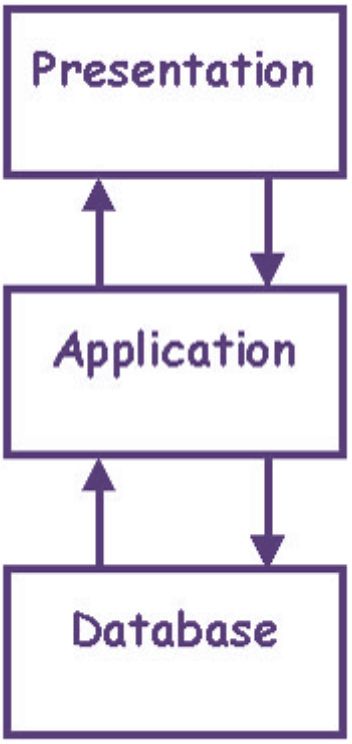

Figure 2.2.1.1. Web application layers

1-TIER ARCHITECTURE

This is the simplest one as it is equivalent to running the application on a personal computer. All of the required components for an application to run are on a single application or server. The presentation layer, Business logic layer, and data layer are all located on a single machine.

2-TIER ARCHITECTURE

It is like Client-Server architecture, where communication takes place between client and server. In this type of software architecture, the presentation layer or user interface layer runs on the client-side while the dataset layer gets executed and stored on the server-side. There is no business logic layer or immediate layer in between client and server.

3-TIER ARCHITECTURE

Three-tier architecture is a client-server software architecture pattern in which the user interface (presentation), functional process logic ("business rules"), computer data storage, and data access are developed and maintained as independent modules, most often on separate platforms.

The presentation tier is the topmost level of the application. It displays information related to such services as browsing merchandise, purchasing, and shopping cart contents. It communicates with other tiers by which it puts out the results to the browser/client tier and all other tiers in the network. In simple terms, it is a layer that users can access directly.

The application tiers are the logical ones. It is pulled out from the presentation tier and, as its layer, it controls an application's functionality by performing detailed processing. The data tier includes the data persistence mechanisms and the data access layer that encapsulates the persistence mechanisms and exposes the data.

However, in the web development field, three-tier is often used to refer to websites, commonly electronic commerce websites, which are built using three tiers, which are the front-end web server, a middle dynamic content processing, and generation level application server, and a back-end database or data store.

N-TIER ARCHITECTURE

This architecture divides an application into logical layers, which separate responsibilities and manage dependencies, and physical tiers, which run on separate machines, improve scalability and add latency from the additional network communication. N-Tier architecture can be closed-layer, in which a layer can only communicate with the next layer down, or open-layer, in which a layer can communicate with any layers below it.

2.2.2. MODEL-VIEW-CONTROLLER

The MVC is a software architectural pattern that with three components breaks up the application logic from the view logic. It is an important architecture since it is used not only in basic graphic components but also in big companies. Most of the current frameworks use MVC for the architecture, for example, Ruby on Rails, Django, and AngularJS among many others.

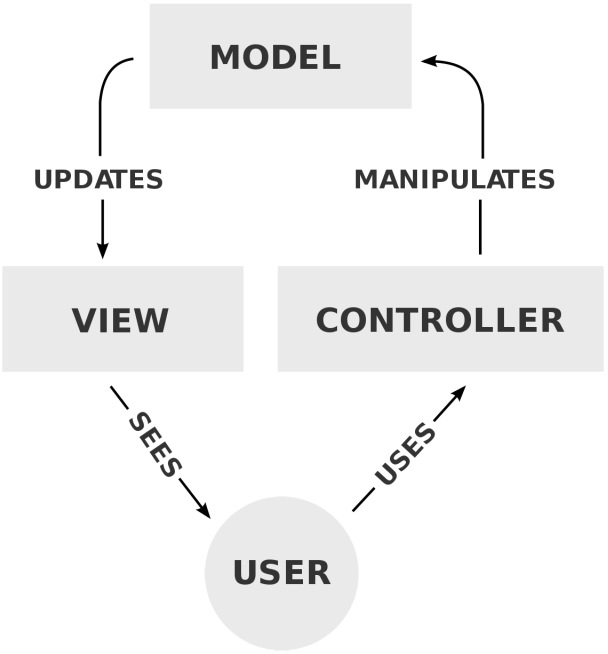

Figure 2.2.2.1. Model-view-controller diagram

The controller is in charge of orchestrating the application, as it receives the user commands and its responsibility is to request the data from the model and communicate it to the view. The model is in charge of the data and often it sends queries to the database, such as updates, reads, deletes, additions…

Finally, the view is the visual representation of the data. All related to the graphic interface goes here. Neither the controller nor the model is aware of how the view is being rendered, it is only the view that is aware of this.

2.3. ARCHITECTURE DESIGN OF A WEB APPLICATION

Currently, there are three primary types of web application architectures that we should take into account, and we can categorize them in frontend and backend layers, which are the following.

2.3.1. FRONTEND

SINGLE-PAGE APPLICATION ARCHITECTURE

This architecture is based on loading all the content on one single page, and updating it dynamically with the necessary resources as the user interacts with it. This single page is an HTML file, and all the application runs inside it, giving the user a better and faster experience. The HTML, CSS, and JavaScript code is loaded one single time, and we can navigate between different sections and the content will be ready since it has been loaded before.

Many available techniques can help us to develop and achieve this single-page application. Frameworks such as AngularJS, React, and Vue.js among many others. As an example, Gmail, Google Maps, Facebook, or GitHub are single-page applications.

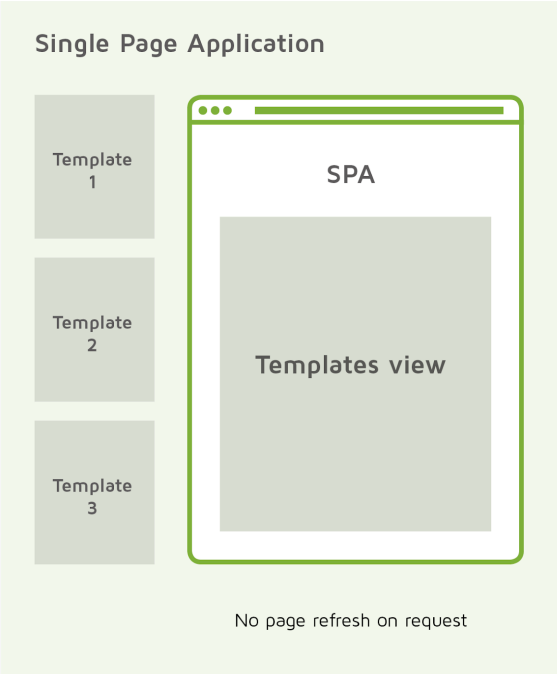

Figure 2.3.1.1 Single page application

2.3.2. BACKEND

MICROSERVICE ARCHITECTURE

This architecture aims to split the system into individual parts so that they can be treated and developed independently. Meanwhile, monolithic architecture is developed as a single unit, a microservice-based architecture works as a set of small services which are executed independently. Each microservice can be written in its language and plays an independent role. They communicate between them using APIs and have their storage systems.

It is an architectural style that structures an application as a collection of services that are highly maintainable and testable, loosely coupled, independently deployable, organized around business capabilities, and often owned by a small team.

The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

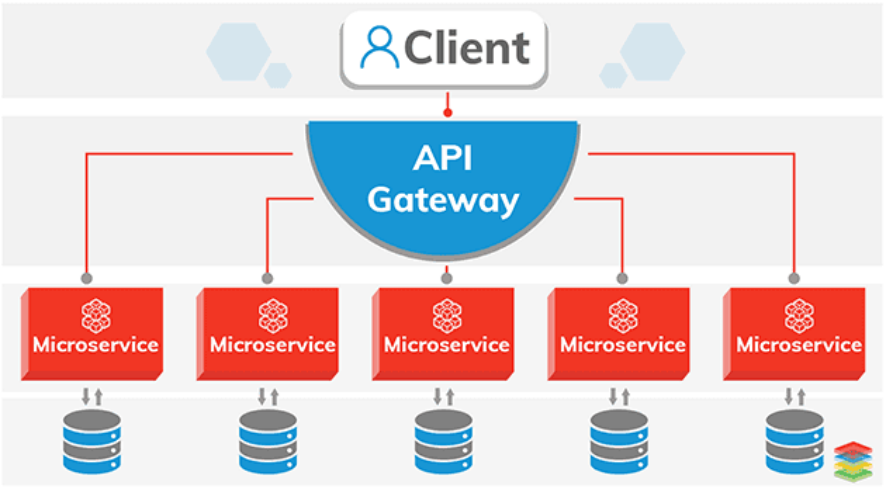

Figure 2.3.2.1 Microservices

SERVERLESS ARCHITECTURE

Architecture without a server is a way of creating and executing applications and services without the infrastructure, where the cloud provider, such as AWS, Azure, or Google Cloud, takes the responsibility for the code execution through dynamic resource allocation. This code which is sent to the cloud provider usually is in a function form, this is why serverless sometimes is called "Functions as a Service" or "FaaS". While serverless abstracts the underlying infrastructure to the developer, the servers still participate in the execution of the functions.

2.4. METHODOLOGIES FOR WEB APPLICATION DEVELOPMENT

The current trend in web development is to build applications based on the monolithic architectures mentioned before, with split disciplines, usually powered by microservices. There are different approaches to overall application structuring, but the most common are the following.

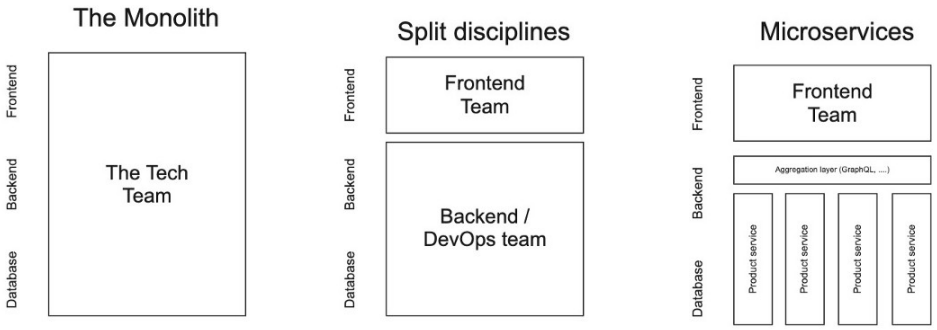

Figure 2.4.1 Types of application structuring

The monolithic architecture consists of a simple principle, all the application code is inside one project, and works as an entire module. The monolithic discipline is easy to understand. It is usually used in small companies where it is difficult to have many different roles or teams, so the leaders decide to have only one Tech Team who will do all the required tasks related to the development.

In most cases, the companies have two distinguished teams related to web development, which are the Frontend team and the Backend team. One only focuses on the frontend layer and the other one on the backend and database.

Moreover, the backend team can manage the backend layer with microservices, which makes the entire application decentralized and decoupled into services that act as separate entities. Unlike the monolithic architecture wherein a failure in the code affects more than one service or function, there is the minimal impact of a failure using microservices. Even when several systems are brought down for maintenance, your users won't notice it.

However, even with those disciplines and architectures, the code becomes bigger and focuses explicitly on one technology or framework. The frontend layer becomes massive, and different features are maintained by different teams, which requires a lot of coordination between them. The frontend layer could become difficult to maintain, especially when a lot of different code modules developed during different projects should be reused and integrated into the same application.

However, micro frontends are increasing in popularity, as it enables the splitting of monolithic frontends into independent end-to-end applications. Huge amounts of companies, such as Spotify, Ikea, or Starbucks among many others, have already adopted this new architecture.

CHAPTER 3: FRONTEND WEB DEVELOPMENT TECHNOLOGIES

3.1. INTRODUCTION

Most of the web development frameworks offer the developer tools and support the unit testing and can ensure backward compatibility. Stability and community support are also the factors not to be forgotten therefore the availability of experienced specialists on the market won't be the issue. To achieve the desired results, we should find the best suitable web front-end framework that will meet the general and your requirements.

Trends are fully dependable on developers and their framework decisions. According to the specific survey, most market belongs to such JavaScript frameworks as React, Angular, Vue.js.

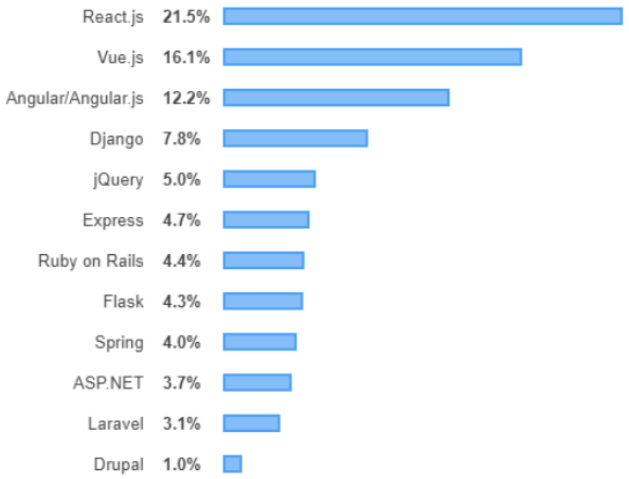

Figure 3.1.1. Most wanted to use web development frameworks

3.2. TECHNOLOGIES BASED ON JAVASCRIPT

Before studying different ways to achieve Micro-Frontend architecture, it is necessary to be familiar at least with one front-end framework, such as Angular, Vue, or React among many others. It should also be achieved only by making static HTML applications with their one javascript and CSS, however, it is not practical as nowadays applications are developed mostly with the mentioned frameworks before. Each framework has its strengths which we are going to look for below. But let's see an image of the frameworks that big companies are using.

ANGULAR.JS		REACT.JS		VUE.JS	
The Guardian	theguardian	AirBnB	airbnb	Alibaba	Alibaba.com
Upwork	upwork	Instagram		Grammarly	grammarly
Pay Pal	PayPal	UberEats	eats	IPL dashboard	IPL
Sony	SONY	Dropbox	Dropbox	Gitlab	GitLab

Figure 3.2.1. Frameworks and companies who use them

I have worked only with Angular. However, I think it is a good idea to learn some other popular frameworks to develop and implement this project. I am not going to perform deep research about each one, and the idea is to do some crash courses to get slightly familiar with them.

I am only going to understand the basics of each framework, as the project we are going to implement is going to be as simple as possible. This is why I am going to analyze those frameworks briefly, explaining their basics and how they are structured. Before starting, it is important to install NodeJS on our computer, which is an execution environment based on JavaScript.

3.2.1. VUE

VueJS is a JavaScript framework for building front-end UI's. In view, we can start simple and then progressively add the tools and features that we need to build a complex web application. Its core provides a way to build components that encapsulate data or states in our JavaScript and then connect that state reactively to a template in HTML.

First, it's interesting to understand that VueJS was designed to be incrementally integrated. That means if we have an existing frontend application, we do not have to redo everything. We just make a new part in VueJS and quickly integrate it with the rest. This framework is also the easiest to pick up of all the frontend frameworks.

Like with many other frameworks that I will explain below, VueJS has also taken good ideas from its competitors. It allows data binding. The data and the DOM are coupled and reactive to changes. We also find the concept of virtual DOM with VueJS. It is a tree-like data structure or collection of JavaScript objects representing DOM nodes that are managed by VueJS and that should be rendered on the page. These objects are called "virtual nodes". The main purpose of virtual DOM is faster and more efficient DOM manipulation. When there are lots of nodes in the DOM, updating them can be expensive in terms of processing power and resources required.

The framework is partly inspired by the MVVM architecture pattern (Model-View-ViewModel), which facilitates the separation of the development of the graphical user interface. VueJS links the DOM, the view part, with the view instance, the Model-View part. These two parts are linked by the data-binding system. Finally, the data in your view instance is the Model part. It provides the application with data.

A VueJS application is composed of one or more components. When the instance of the global application is launched, there is first a root instance. This root instance is composed of a tree of components that have their root instance.

VueJS uses what's called a one-way data flow. Data is passed to child components from a given parent component using a prop or a custom attribute that becomes a property on the child component instance. When the parent component updates a prop value, it's automatically updated in the child component. Mutating a property inside a child component should not be done. Also, it does not affect the parent component (unless it is an object or array). The child component can communicate back to the parent via an event. The parent can assign a handler to any event emitted by the child component instance and data can be passed back to the parent. The child component can emit a special event for updating the props passed to it.

VueJS is a server-side rendering, which allows developers to pre-populate a web page with custom user data directly on the server. It is generally faster to make all the requests within a server than making extra browser-to-server round-trips for them. This is what developers used to do before client-side rendering.

For a better understanding of the framework, I have developed a small project which I am going to explain briefly now. The simplest way to create the project with the Vue CLI, a full power system for rapid Vue.js development, which provides interactive project scaffolding, zero-config rapid prototyping, a runtime dependency, a rich collection of official plugins integrating the best tools in the frontend ecosystem, and of course a full graphical user interface to create and manage Vue.js projects. It is the standard tooling baseline for the Vue ecosystem.

I am not going to explain how to create each component and explain deeply how the code is structured. Nevertheless, I am going to show how the project is structured and the main fundamentals of it.

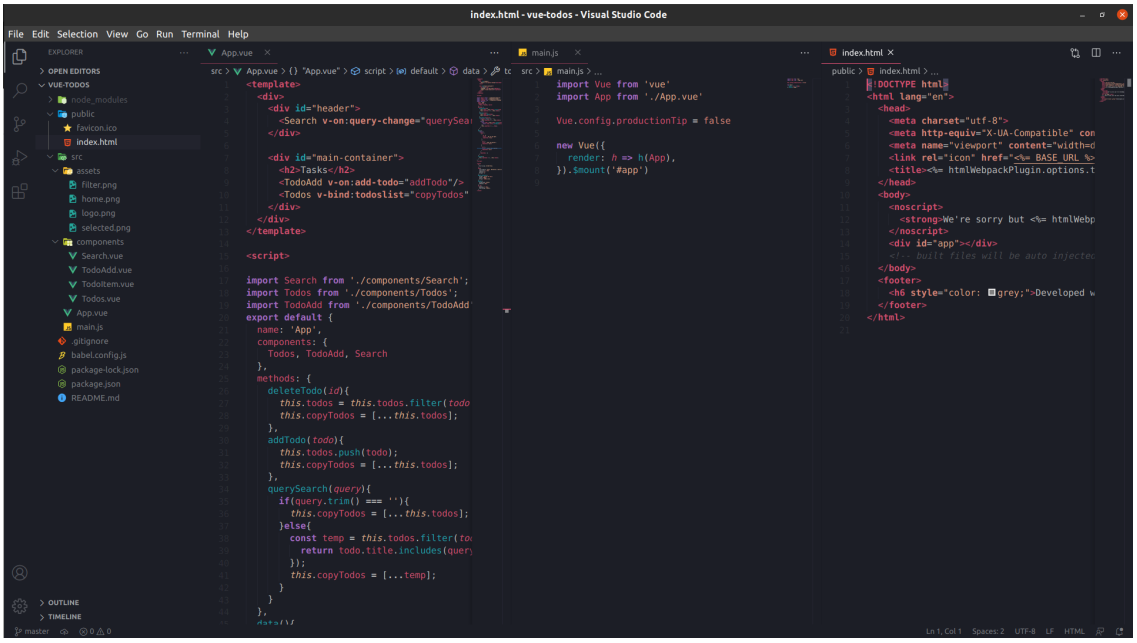

Figure 3.2.1.1. Structure of Vue project

So this is how a common project looks like. We have a node_modules folder which we should not touch. The public folder contains the index.html file, which will be the point of entry. As we can see, there is a div with id “app”, which we are going to initialize in our main.js file. The App.vue file will be the main file where we are going to import and use all the other components we have developed.

The structure of a vue extension file is simple. We always have three main parts. The template, where we write the HTML code, the script, where we write the JavaScript code, such as functions for example, and the style, where we write in CSS the style of our component.

Figure 3.2.1.2. Structure of Vue file

In the template of the App.vue file, we have defined some divs such as the header and the main container where our tasks are going to be located. It is here where we use our components, and thanks to Vue, we can bind data and make a lot of other things such as respond to events, executing certain functions, which of course should be located in the script label. Here I leave a screenshot of a little part of the App.vue file to make it easier to understand.

Todos.vue App.vue

src > App.vue > {} "App.vue" > script > default > data > todos > title

```vue
<template>
  <div>
    <div id="header">
      <Search v-on:query-change="querySearch" />
    </div>

    <div id="main-container">
      <h2>Tasks</h2>
      <TodoAdd v-on:add-todo="addTodo"/>
      <Todos v-bind:todoslist="copyTodos" v-on:delete-todo="deleteTodo" />
    </div>
  </div>
</template>

<script>

import Search from './components/Search';
import Todos from './components/Todos';
import TodoAdd from './components/TodoAdd';
export default {
  name: 'App',
  components: {
    Todos, TodoAdd, Search
  },
  methods: {
    deleteTodo(id){
      this.todos = this.todos.filter(todo => todo.id != id);
      this.copyTodos = [...this.todos];
    },
    addTodo(todo){
      this.todos.push(todo);
      this.copyTodos = [...this.todos];
    },
    querySearch(query){
      if(query.trim() === ''){
        this.copyTodos = [...this.todos];
      }else{
        const temp = this.todos.filter(todo =>{
          return todo.title.includes(query)
        });
        this.copyTodos = [...temp];
      }
    }
  },
```

Figure 3.2.1.3. Todo app vue file

I have developed a simple To-do application where there is a list of tasks, and you can check them, delete them, add new ones, or even search for a specific task in the list.

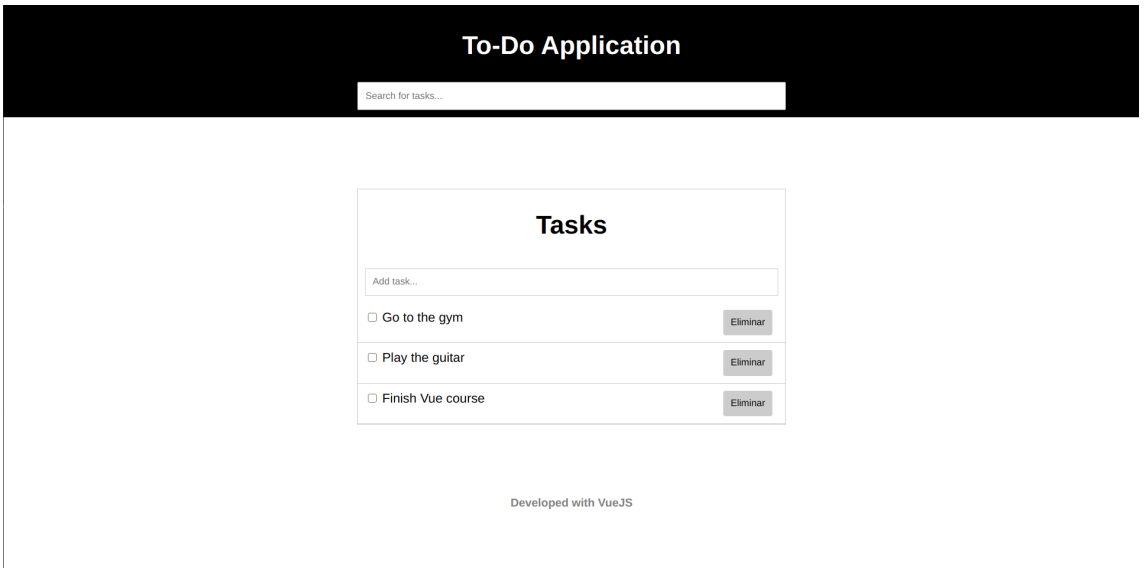

Figure 3.2.1.4. Todo application web page in Vue

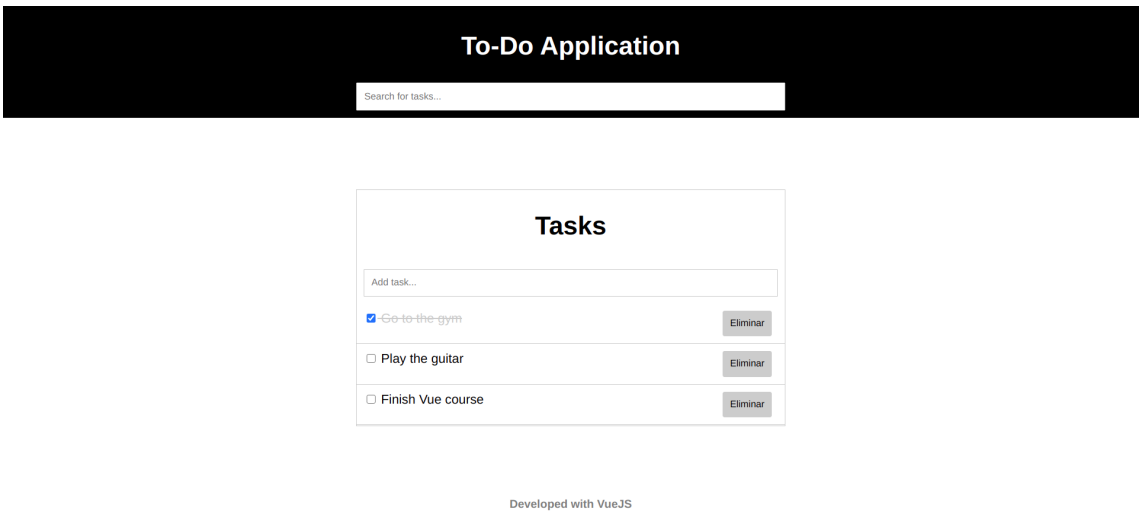

Figure 3.2.1.5. Todo application web page marked task in Vue

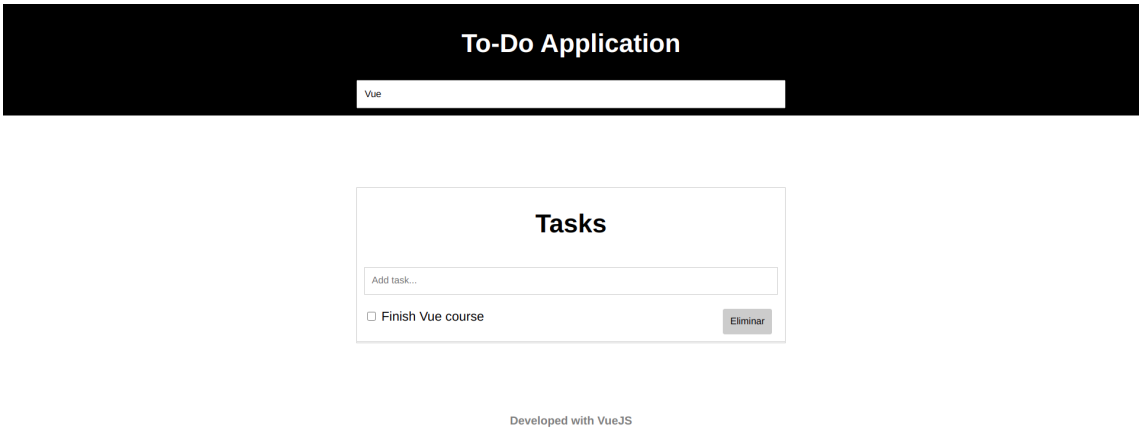

Figure 3.2.1.6. Todo application deleted tasks in Vue

I leave here the GitHub repository if it is in your interest to look at it:
https://github.com/illianechesa/vue-todos

3.2.2. REACT

React is a Javascript library whose purpose is to simplify the development of visual interfaces. It was developed by Facebook in 2013, and currently, it drives some of the most widely used code in the world, powering Facebook and Instagram among many, many other software companies. It is used to build single-page web applications, among many other libraries and frameworks that were available before React came into life.

It is important to mention that React itself is a library and not a framework. Nevertheless, different add-ons that we could add to it, make React similar to a framework in many properties. It is used in many ways so we can create web applications, single-page applications, or even mobile applications. In React there exists a complex ecosystem of modules, tools, and components that can help us to achieve complex targets without making a big effort. Therefore, React represents a solid base above which we can build nearly anything we want with Javascript. Remember that React, the same as Vue, has a server-side rendering.

React creates a virtual DOM. When state changes in a component, it firstly runs a “diffing” algorithm, which identifies what has changed in the virtual DOM. The second step is reconciliation, where it updates the DOM with the result of diff. It is important to understand that the HTML DOM is always tree-structured, and we need to modify it incessantly and a lot, which means a lot of performance and development pain. The Virtual DOM is an abstraction of the HTML DOM. It is lightweight and detached from the browser-specific implementation details.

One of the most important features of React is JSX, a syntax extension to JavaScript, which produces React “elements”. We can embed any JavaScript expression in JSX by wrapping it in curly braces. After compilation, JSX expressions become regular JavaScript objects. This means we can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions. Even though React does not require JSX, it is the recommended way of describing our UI in React app.

React apps are organized as a series of nested components. These components are functional in nature: that is, they receive information through arguments (represented in the “props” attribute) and pass information via their return values (the return value of the “render” function). This is called unidirectional data flow. Data is passed down from components to their children.

Another important feature about React is Redux. The basic idea of Redux is that the entire application state is kept in a single store. A store is simply a Javascript object. The only way to change the state is by firing actions from our application and then writing reducers for these actions that modify the state. The entire state transition is kept inside reducers and should not have any side -effects.

Redux is based on the idea that there should be only a single source of truth for your application state, be it UI state like which tab is active or Data state like the user profile details. All of this data is retained by redux in a closure that redux calls a store. It also provides us with a recipe for creating the said store.

To understand it better, taking into account I have not used React before, I have implemented the same project as I did in Vue, but this time using React. It is not the same, but it is very similar. So thanks to NodeJS, there is a simple way to create and initialize a React application, just by typing “npx create-react-app <app-name>”. And the structure we obtain is very similar to the one we have obtained previously in the vue project.

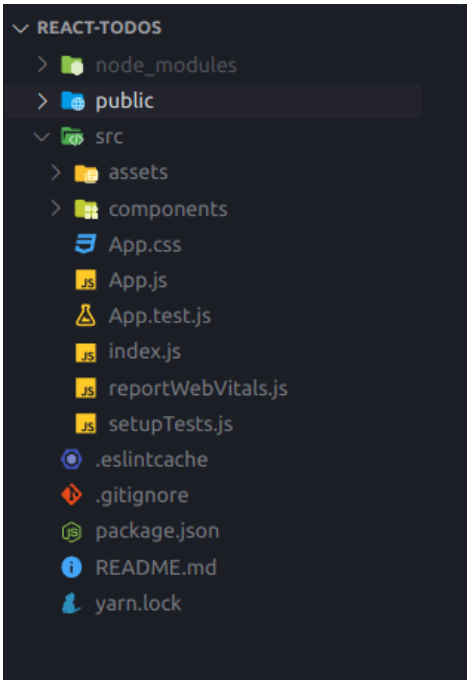

Figure 3.2.2.1. Todo application structure ReactJS

It is necessary to understand the structure of the project. In the node_modules folder, as I have already said before, the ppm dependencies are stored. The public folder is the root of the project, and the main file is stored here, index.html. Inside that file, there will always be a div tag with an id “root” which indicates where the main component of our application will be introduced.

All the components and their styles are stored in the src folder, as the js files, we will also use. By default, the main component is called App.js. There will also be an index.js file whose mission is to introduce the App component in the div tag I have mentioned before.

Now that we have familiarized a little bit with React, I will explain briefly the project I have done, and how it works. The main idea and purpose of the applications are the same as the one I have done before with Vue. Even I have tried to maintain the same style.

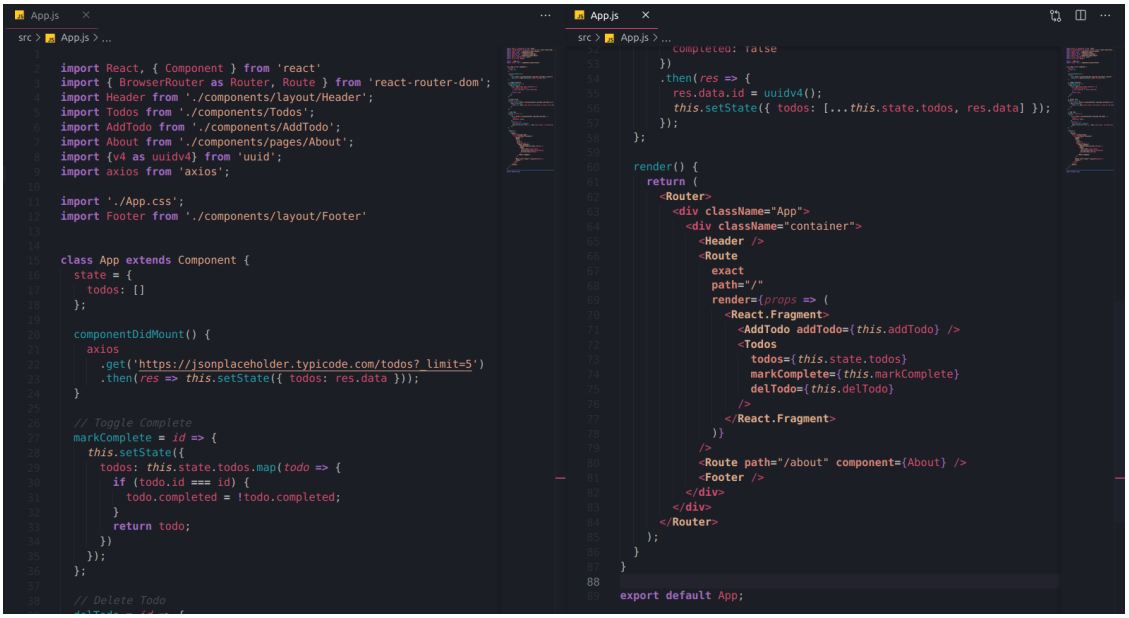

Figure 3.2.2.2. Todo application files structure ReactJS

The most important thing is to understand how it renders the components. We can see that our class App extends from a component, which means, it is a component. Each component should have its render function, with which you can render the view, it is similar to the “template” in VueJS.

Moreover, inside the render, we can easily instance other components. Just by typing <”nameOfComponent” />, we can show it. Additionally, we can add attributes to those components to share or bind data.

I think it is not necessary to explain deeply all its features, so I leave here a small example of a component, and how it is used in the main application.

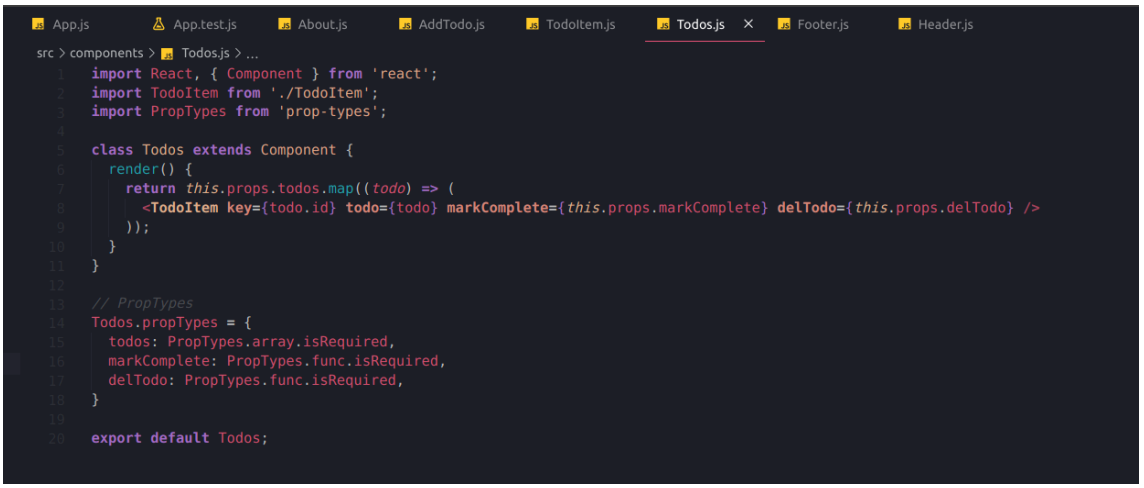

Figure 3.2.2.3. Todo application file structure ReactJS

```
render() {
  return (
    <Router>
      <div className="App">
        <div className="container">
          <Header />
          <Route
            exact
            path="/"
            render={props => (
              <React.Fragment>
                <AddTodo addTodo={this.addTodo} />
                <Todos
                  todos={this.state.todos}
                  markComplete={this.markComplete}
                  delTodo={this.delTodo}
                />
              </React.Fragment>
            )}
          />
          <Route path="/about" component={About} />
          <Footer />
        </div>
      </div>
    </Router>
  );
}
```

Figure 3.2.2.4. Todo application file render function ReactJS

We have only created a component that we export in the end so we can use it in other files. In the App render, we just type the component name and bind the variables we want to.

Finally, let's see a full implementation example. I have made a Todo application where we can add tasks, mark them as completed, and delete them. Moreover, those tasks are being acquired from and URL with an HTTP request, using a famous Javascript library called “Axios”. There are also two sections, the home and the about one. Do not forget that this is a Single Page Application, so we can navigate between different sections without reloading the web page. This can be achieved with paths and routing methods.

The final result of the application is the following.

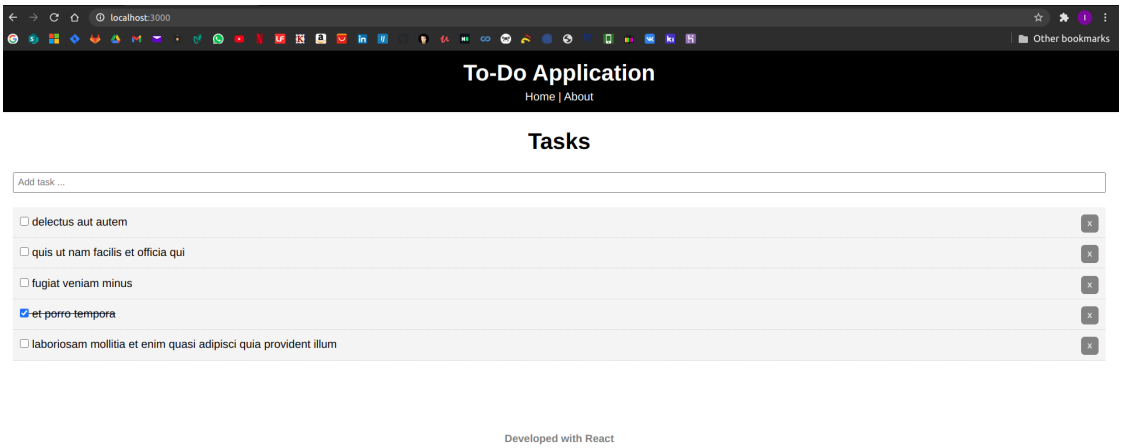

Figure 3.2.2.5. Todo application web page in ReactJS

We can see that when the application launches, we start in the root path, the tasks are shown, and we can add other ones just by typing and submitting.

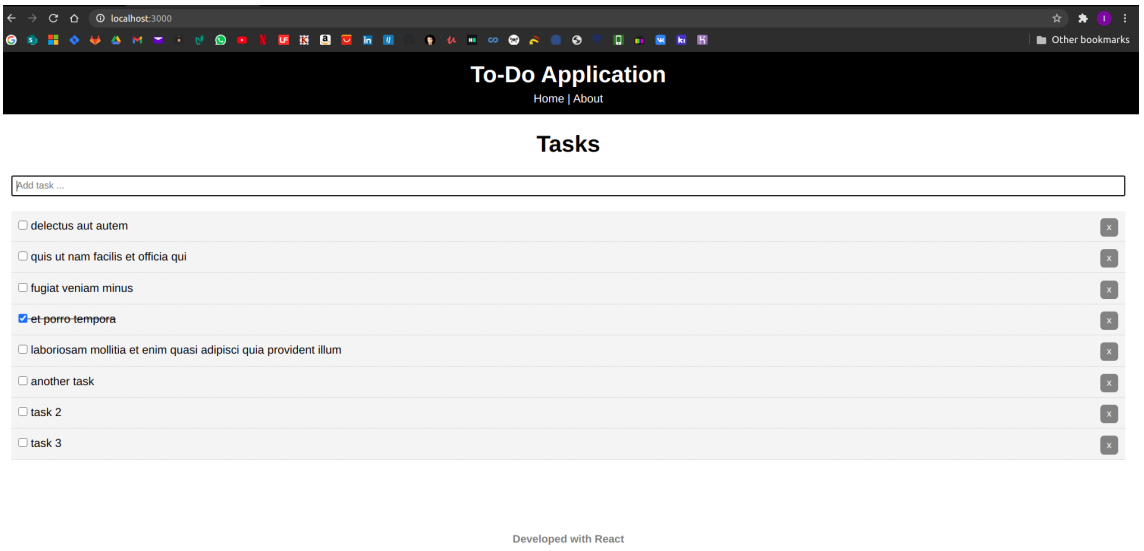

Figure 3.2.2.6. Todo application web page add a task in ReactJS

Now if we select the about link, it will show us the about section, but it does not reload the web page, it only adds the new path in the upper link.

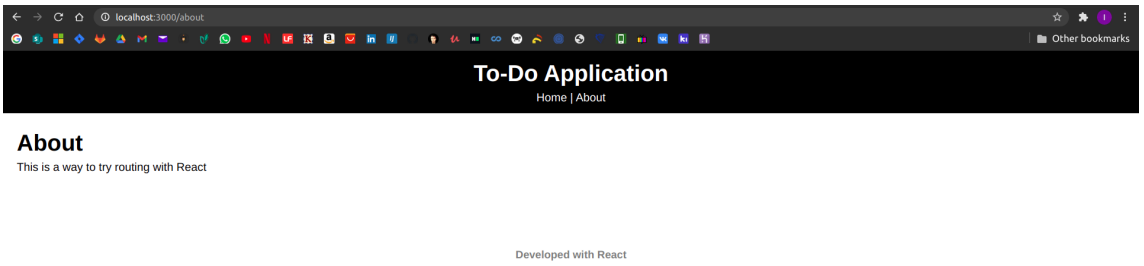

Figure 3.2.2.7. Todo application web page about section in ReactJS

If you have any interest in the code of the project, you can find it in my personal GitHub repository: https://github.com/illianechesa/react-todos.

3.2.3. ANGULAR

Angular is an open-source framework that is used to create single-page applications. It was created by Google, launching its first version named AngularJS in 2012. However, today's Angular has nothing to do with the first version of AngularJS. In 2016, Google rewrote the AngularJS code completely, therefore the new framework called Angular has nothing to do with AngularJS. This framework is the one we use at the company to maintain and develop our web page, which is a single-page application.

Before explaining the project I have done, I found it important to remark some benefits of using Angular. This framework is based on components, and unlike React or Vue which can be used as simple libraries, Angular is used as a complete framework. You can integrate testing tools into the project also. But the most important thing is that we can inject dependencies on it. It means, we use the dependencies directly in the object instead of creating them locally. Finally, one of its best benefits is its support and maintenance by Google.

To follow the previous frameworks' explanations, I am going to show a little bit of the project structure, and comment on it adding some screenshots of the application.

Different from VueJS and React, Angular is a client-side rendering. This allows developers to make their websites entirely rendered in the browser with JavaScript. Instead of having a different HTML page per route, a client-side rendered website creates each route dynamically directly in the browser. This approach spread once JS frameworks made it easy to take.

Client-side rendering manages the routing dynamically without refreshing the page every time a user requests a different route. But server-side rendering can display a

fully populated page on the first load for any route of a website, whereas client-side rendering displays a blank page first.

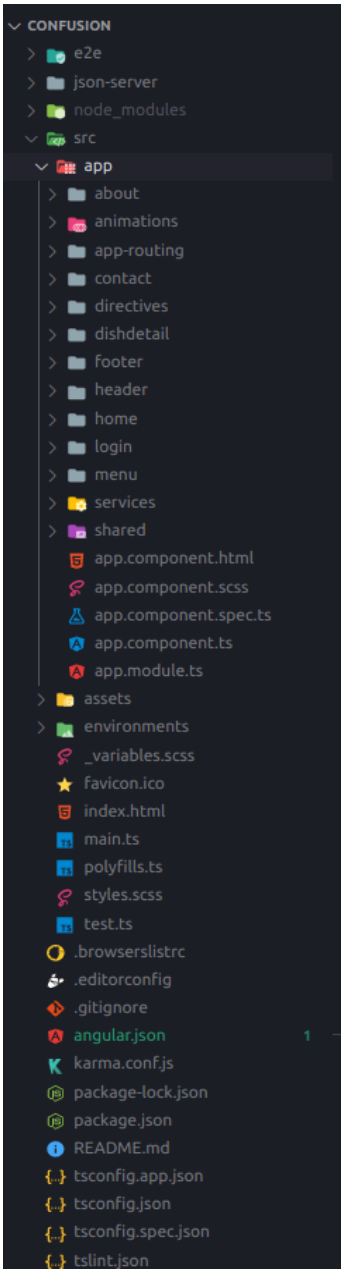

Figure 3.2.3.1. Restaurant application project structure Angular

It is important to understand that Angular is structured with components. We can see there all the folders such as login, home, menu, header… which are components. Those have always the same structure. An HTML file which is going to be the template,

the SCSS file to give a style to our page, the ts file where all the logic is gonna be written in typescript, and finally, the spec.ts file which is our main file for testing our component.

For data providers, we can create and use services, which is the thing I have done in my project. It consists of a pizza restaurant web page, where we have many components, and many sections such as the Home, About, Menu, and Contact. To provide the data of the menu, and other information such as images, dishes, promotions, and feedback among others, I have used a local JSON server, which is running on a specific port, where my project services are accessing to perform the requests for specific information using path routes.

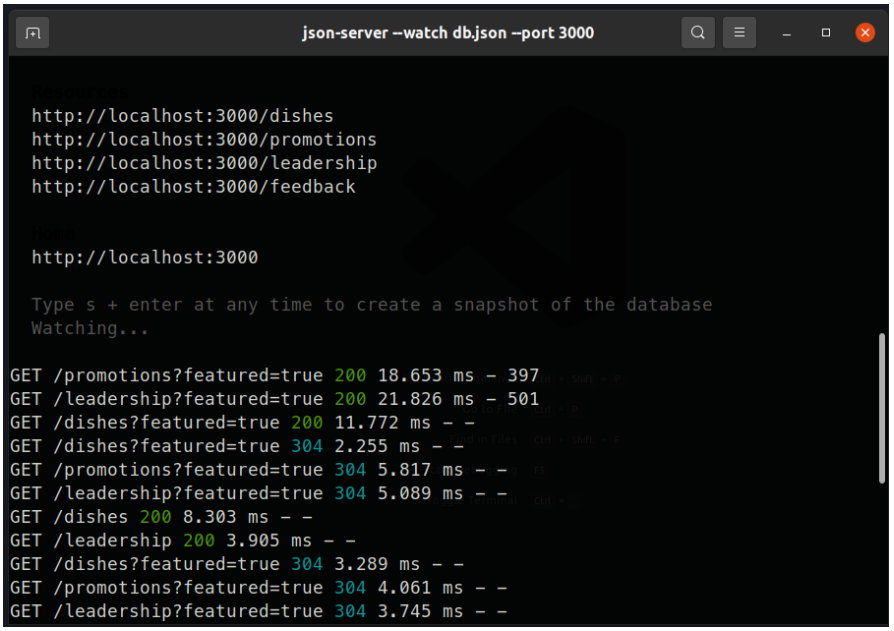

Figure 3.2.3.2. JSON-server running on localhost

This is how my services folder looks like.

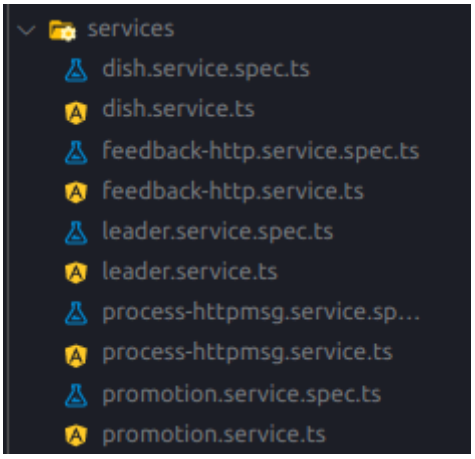

Figure 3.2.3.3.Restaurant application project services Angular

And this is how the service code I made looks like.

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { catchError, map } from 'rxjs/operators';
import { baseURL } from '../shared/baseurl';
import { Leader } from '../shared/leader';
import { ProcessHTTPMsgService } from './process-httpmsg.service';


@Injectable({
  providedIn: 'root'
})
export class LeaderService {

  constructor(private http: HttpClient,
              private processHTTPMsgService: ProcessHTTPMsgService) { }

  getLeaders(): Observable<Leader[]> {
    return this.http.get<Leader[]>(baseURL + 'leadership')
      .pipe(catchError(this.processHTTPMsgService.handleError));
  }

  getLeader(id: number): Observable<Leader> {
    return this.http.get<Leader>(baseURL + 'leadership/' + id)
      .pipe(catchError(this.processHTTPMsgService.handleError));
  }

  getFeaturedLeader(): Observable<Leader> {
    return this.http.get<Leader[]>(baseURL + 'leadership?featured=true').pipe(map(leaders => leaders[0]))
      .pipe(catchError(this.processHTTPMsgService.handleError));
  }
}
```

Figure 3.2.3.4. Restaurant application project service Angular

Angular has many features but I will not explain them, since the aim of this thesis is not to talk about specific frameworks, but to talk about micro frontends. However, I will leave some screenshots of the project I have developed.

This is the homepage, and we can see all the sections, and the most relevant information of our restaurant here, such as the principal dish, the owner, and in the footer, we can access the restaurant information if we want to.

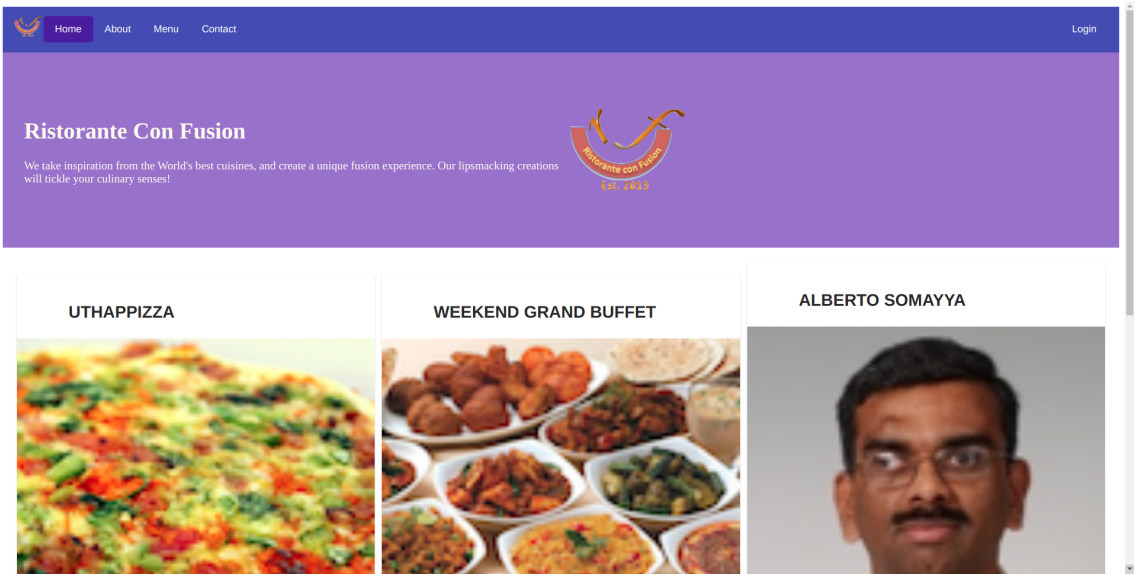

Figure 3.2.3.5. Restaurant application web home Angular

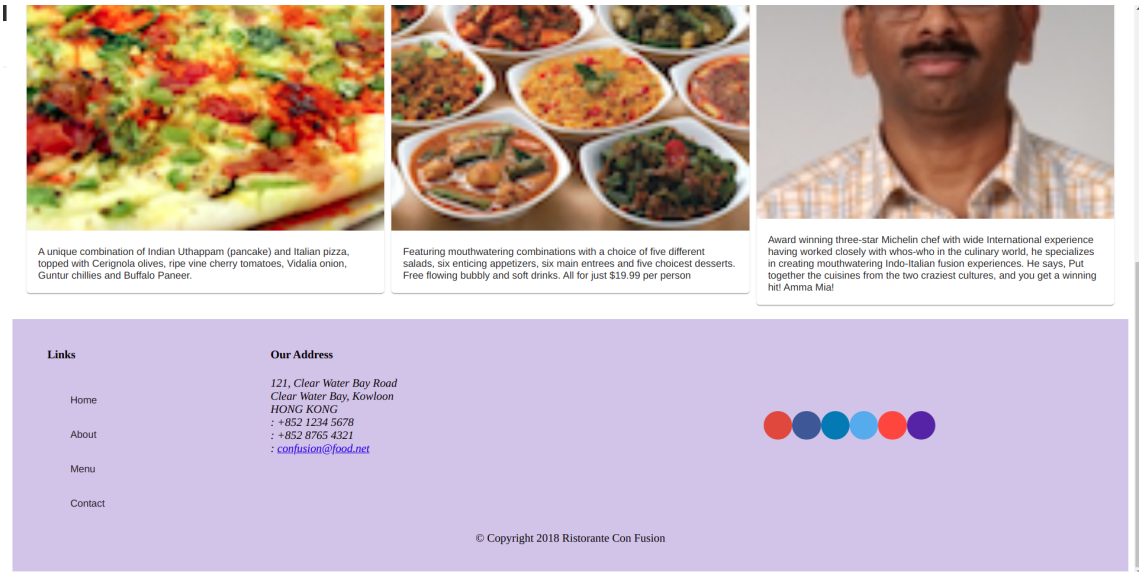

Figure 3.2.3.6. Restaurant application web home Angular

This is the about section where the history of the company is written, and the corporate leadership section.

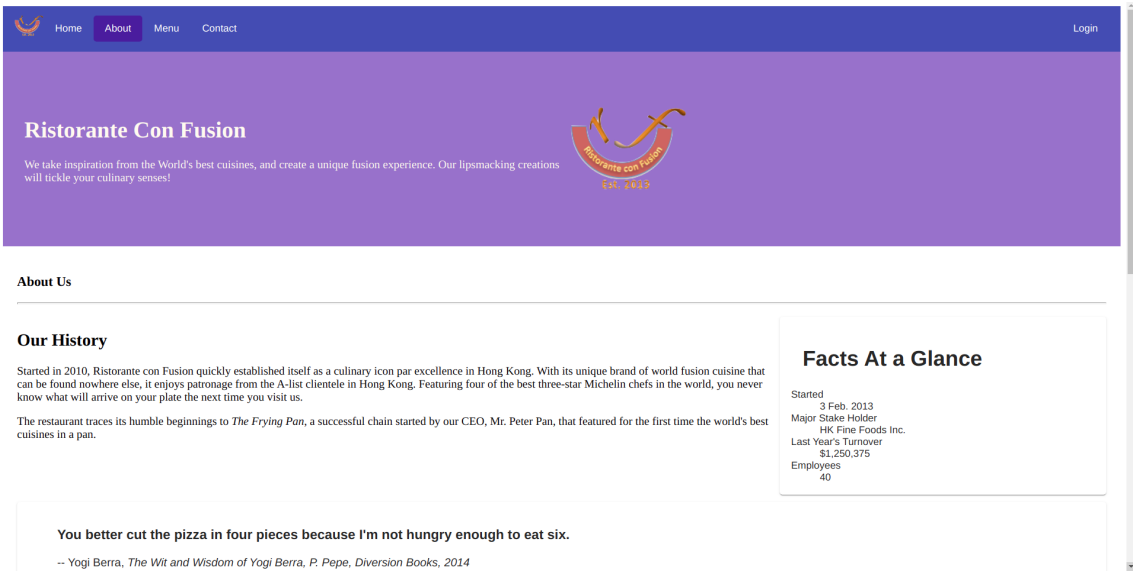

Figure 3.2.3.7. Restaurant application web about Angular

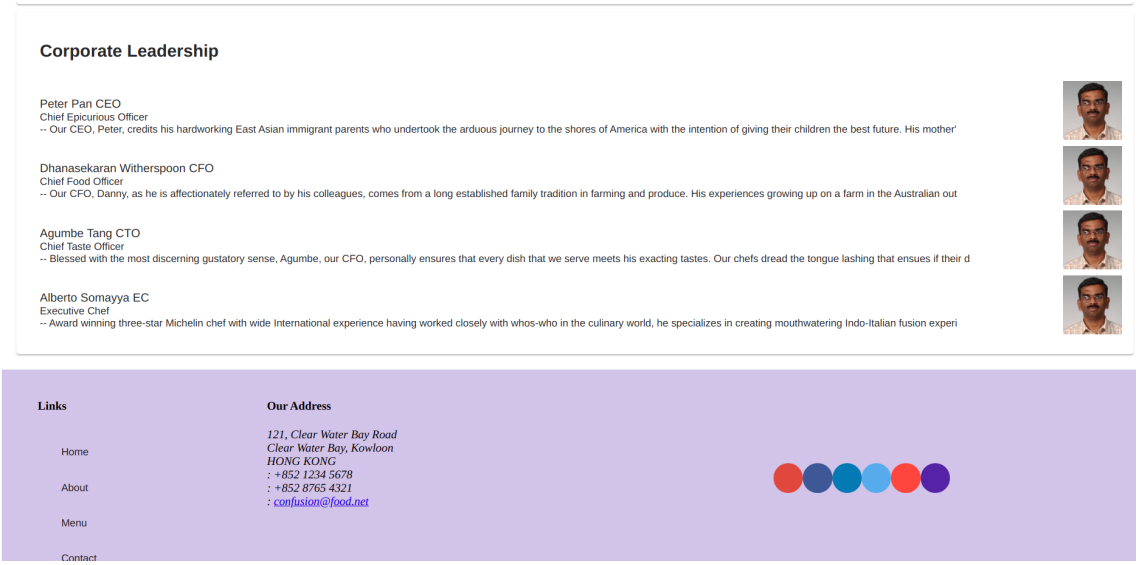

Figure 3.2.3.8. Restaurant application web home Angular

This is the menu section where we can access all the dishes, such as the pizza, the cake, and some other ones. If we click on one of them, we will access its specific information, which you can see below the following image.

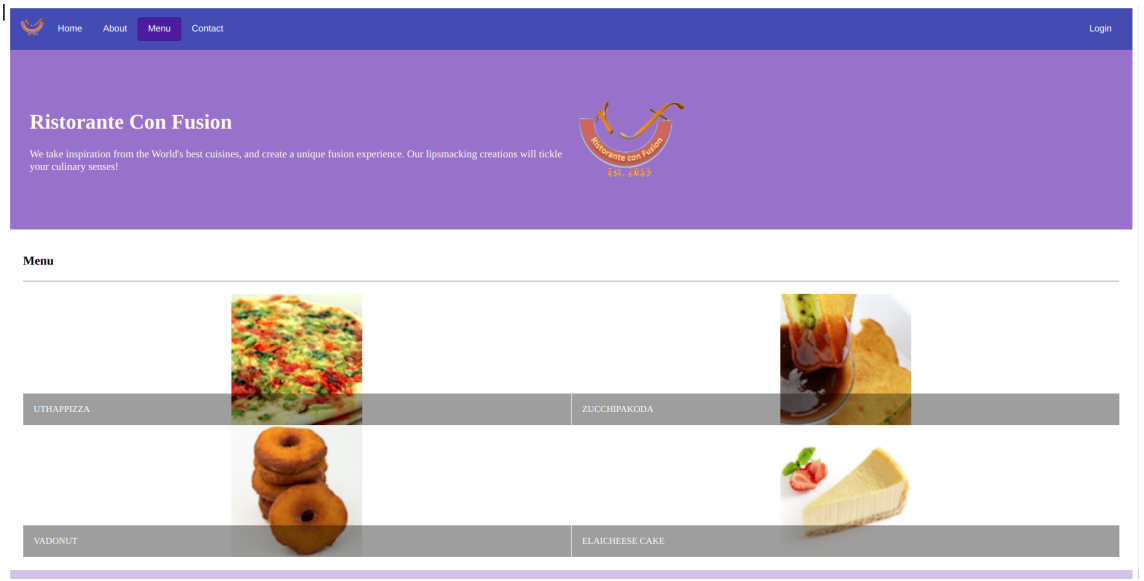

Figure 3.2.3.9. Restaurant application web menu Angular

Here we can see the big image of the dish, the comments the people left on them, and finally, we can submit our comments, and like, share, or navigate between the dishes.

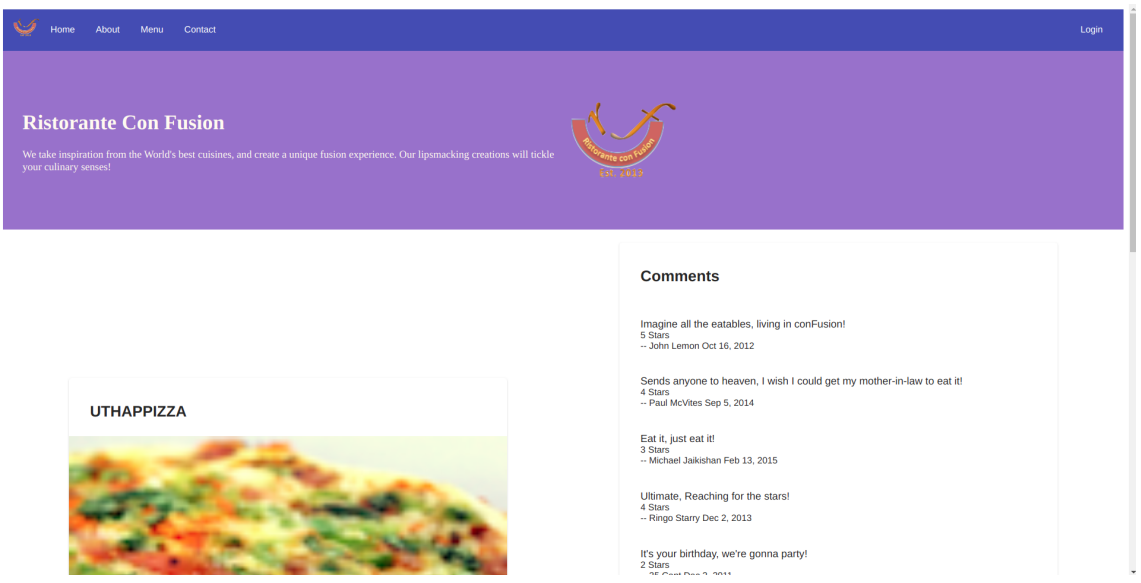

Figure 3.2.3.10. Restaurant application web dish comments Angular

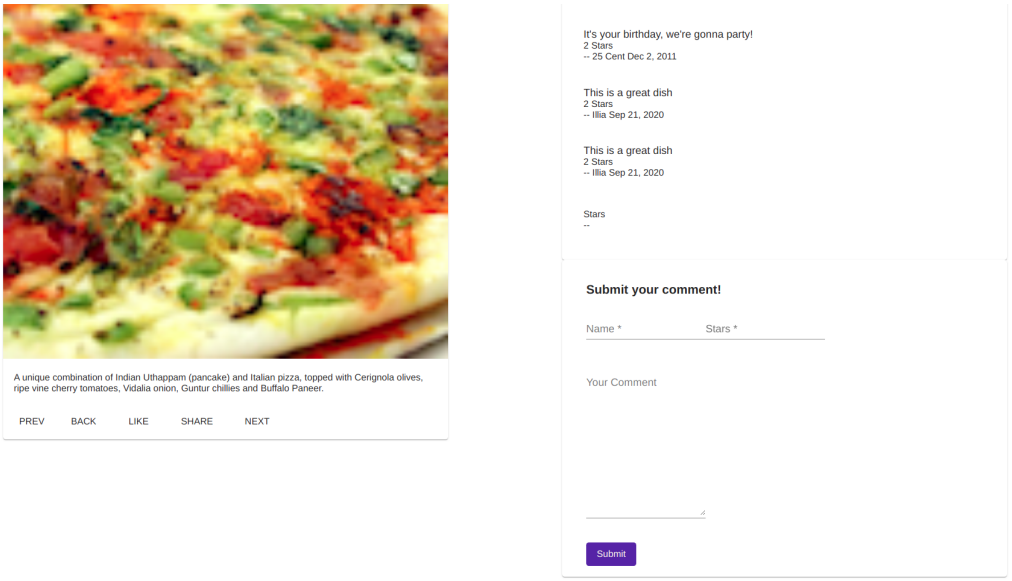

Figure 3.2.3.11. Restaurant application web dish add comment Angular

This is the final section where the customer can contact the restaurant, writing its personal information and leaving feedback.

Figure 3.2.3.12. Restaurant application web contact Angular

If you are interested to try out the project by yourself or to take a look at the code, you can just access the repository at my personal Github account: https://github.com/illianechesa/confusion

3.3. COMPARISON BETWEEN FRAMEWORKS

Having explained the three main frameworks and libraries in Javascript for frontend development, I found it important to share a comparison between those three, to make it easier to find out the one that best suits us.

ATTRIBUTES	ANGULAR	REACT	VUE.JS
Type	JavaScript framework	Open Source JS Library	Progressive JavaScript Framework
Npm weekly downloads (2018)	444,794	5,036,078	996,293
Size	167 KB production 1.2 MB development	109.7 KB production 774.7 KB development	30.67 KB production 279 KB development
Easy to learn	Steep (Learn TypeScript)	Moderate	Easy
Coding speed	Slow	Normal	Fast
Documentation	✓	✓	✓
Performance	✓	✓	✓
Startup time	Longer due to its large codebase	Quick	Quick
Complete web apps	Can be used on standalone basis	Needs to be integrated with many other tools	Requires third party tools
Data binding	Bi-directional	Uni-directional	Bi-directional
Rendering	Client side	Server side	Server side
Model	MVC	Virtual	Virtual
Code reusability	Yes	No, only CSS	Yes, CSS & HTML
When to use	Production, esp. enterprise apps with Material UI	Production, custom UI apps	Startups, production

Figure 3.3.1. Comparison between most famous javascript-based frontend frameworks

CHAPTER 4: MICRO-FRONTENDS

4.1. MICROSERVICE ARCHITECTURE

The aim of this architecture is to split the system in individual parts so that they can be treated and developed independently. Meanwhile, monolithic architecture is developed as a single unit, a microservice-based architecture works as a set of small services which are executed independently. Each microservice can be written in its language and plays an independent role. They communicate between them using APIs and have their storage systems.

It is an architectural style that structures an application as a collection of services that are highly maintainable and testable, loosely coupled, independently deployable, organized around business capabilities, and often owned by a small team.

The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

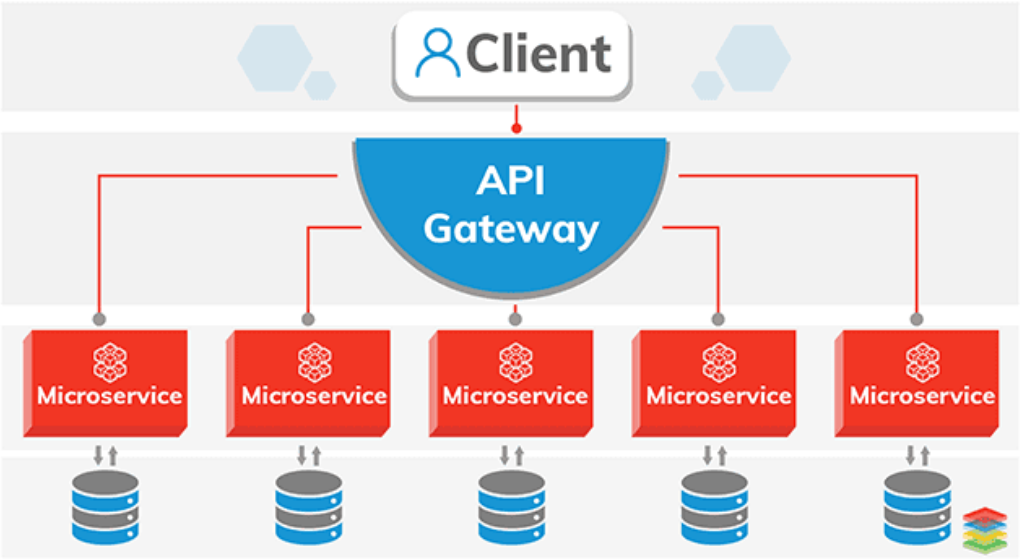

Figure 4.1.1. Microservices architecture

4.2. SERVERLESS ARCHITECTURE

Architecture without a server is a way of creating and executing applications and services without the infrastructure, where the cloud provider, such as AWS, Azure, or Google Cloud, takes the responsibility for the code execution through dynamic resource allocation. This code which is sent to the cloud provider usually is in a function form, this is why serverless sometimes is called “Functions as a Service” or “FaaS”. While serverless abstracts the underlying infrastructure to the developer, the servers still participate in the execution of the functions.

4.3. APPROACHES TO OVERALL APPLICATION STRUCTURING

The current trend in web development is to build applications based on the monolithic architectures mentioned before, with split disciplines, usually powered by microservices. There are different approaches to overall application structuring, but the most common are the following.

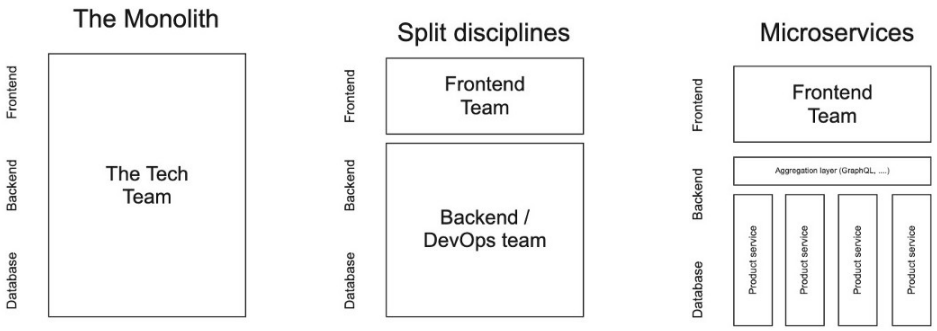

Figure 4.3.1 Approaches to application structuring

Because of that, the code becomes bigger and focuses explicitly on one technology or framework. The frontend layer becomes massive, and different features are maintained by different teams, which requires a lot of coordination between them. The frontend layer could become difficult to maintain especially when a lot of different code modules developed during different projects should be reused and integrated into the same application.

However, micro frontends are increasing in popularity, as it enables the splitting of monolithic frontends into independent end-to-end applications. Huge amounts of companies, such as Spotify, Ikea, or Starbucks among many others, have already adopted this new architecture.

4.4. MICRO-FRONTENDS

4.4.1. MICROSERVICES AND MICRO-FRONTENDS

I found it important to understand not only the difference but the relation between the microservices and the micro-frontends. Oftentimes, when companies adopt a microservice architecture on the backend, they leave their front-end apps as monoliths. This common architecture looks like this.

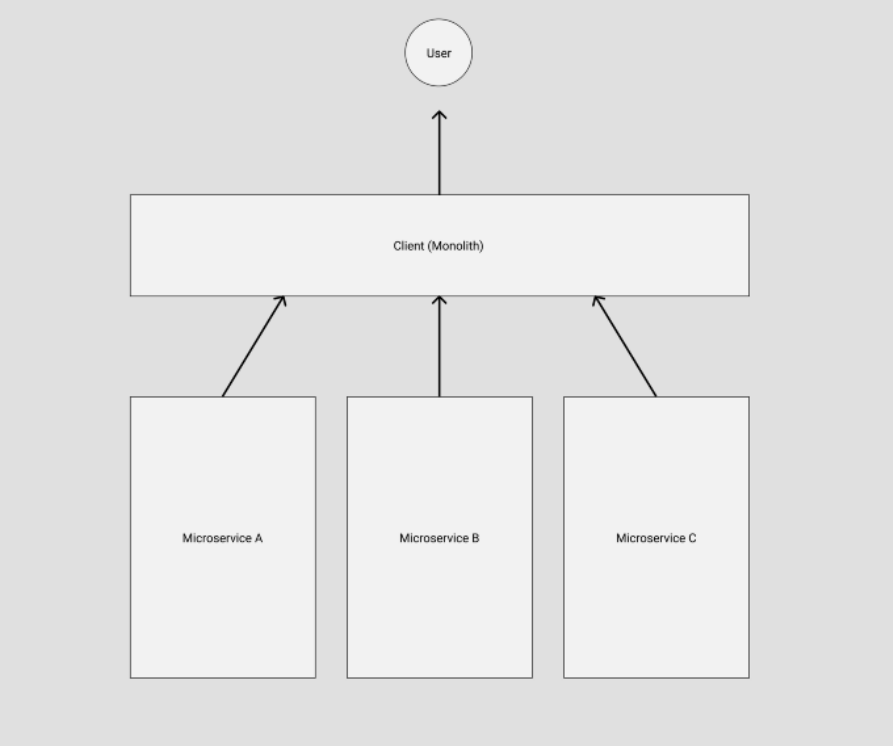

Figure 4.4.1.1. Microservices architecture monolithic

However, this is not truly an end-to-end microservices architecture. While the backend is split up according to business requirements, the frontend is still all in one app. Over time, problems can begin to present themselves, especially if our microservices and UI demand go hand in hand.

With a micro-frontends architecture, we can split the entire application by business domain across the entire stack, which enables the front-end teams the same level of flexibility, testability, and velocity that our backend teams are getting from their microservices. On a high level, using microservices looks more like this.

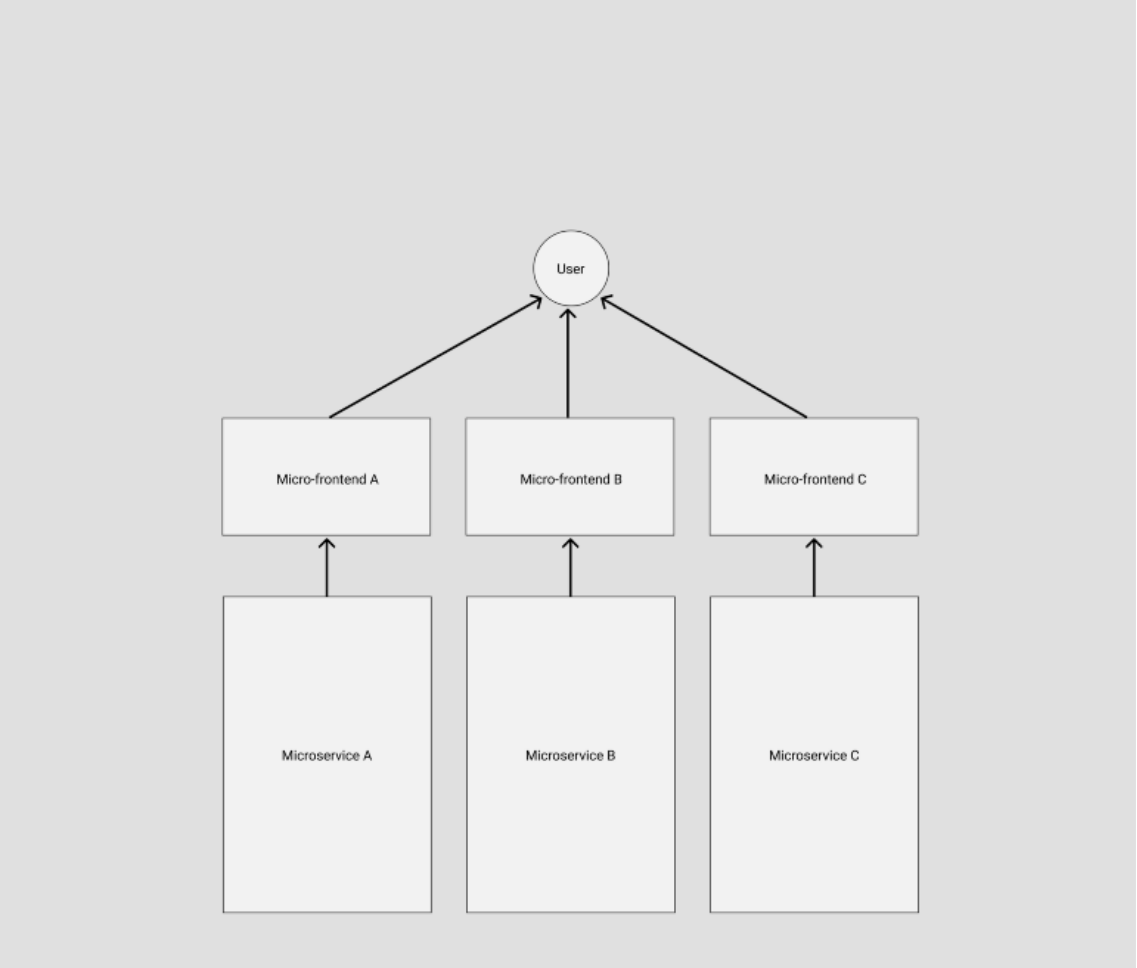

Figure 4.4.1.2. Microservices architecture micro frontend

4.4.2. PURPOSE

The aim of the micro-ui or micro frontend is to give complete autonomy to the teams, in a way that would let them have complete control over their features without having to align to other team ideas, frameworks, or technologies. We need to think about a web app as a composition of features, which are managed by independent teams. Each team has a distinct area of business or mission it works with. Each one is cross-functional and develops its features end-to-end, from database to user interface.

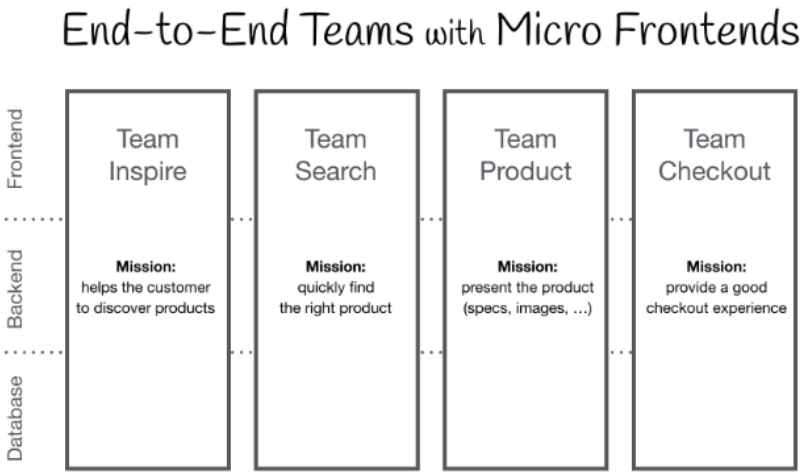

Figure 4.4.2.1. Micro Frontend functionality

4.4.3. STRUCTURE

From a development perspective, we see separate applications that run and build independently, they may have different tech stacks but the result is a web application or website that works on your browser and behaves as it was defined.

However, not all browsers work in the same way. Each one behaves differently with JavaScript, and some features are not supported by some browsers. The best way is to use APIs that work correctly in all browsers.

From a user's perspective, they see one web application that links or navigates different micro-ui's. However, they do not see or notice this, or at least, should not.

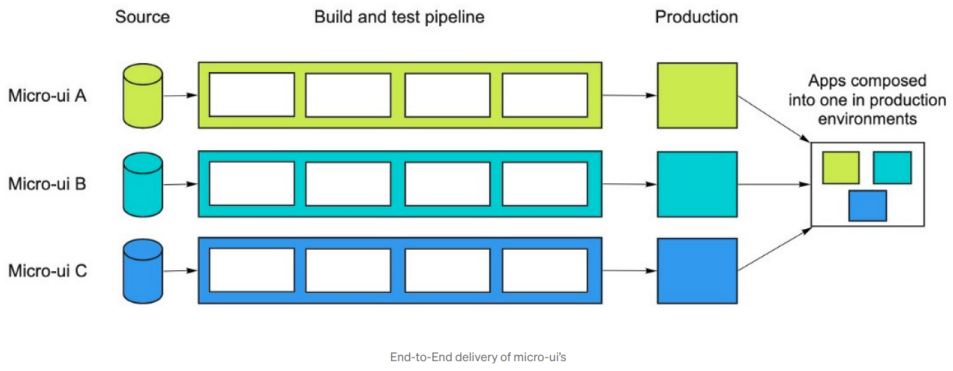

Figure 4.4.3.1. Micro Frontend deployment diagram

Each micro-ui should have its source code and is built and tested independently. When all the work is done, those micro-ui's are deployed on the web, and we start tying all of them into one application. This can be achieved by using different routing methods which we are going to study during the development of this project.

The main challenge in this architecture is to share data or establish communication across these different applications. We would like to share styling and some functions which are commonly used in those applications. The most common technique to achieve that is to create separate NPM libraries and pull those into each of our micro-ui's. This should be designed in a way so that the micro-ui could extend on these shared components. Changing theming, padding, and so on, should be easy to do.

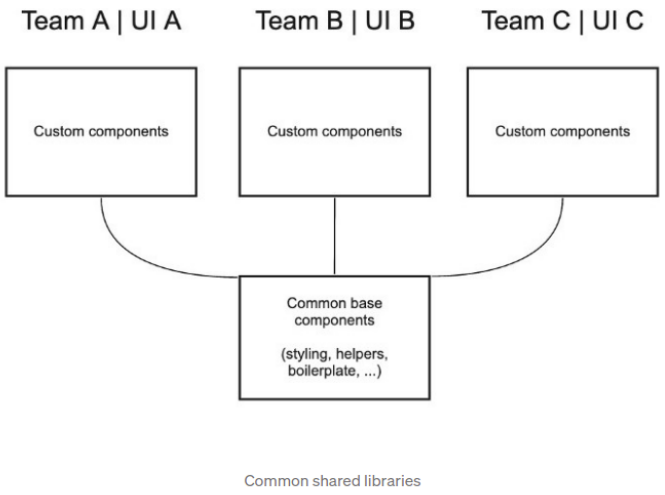

Figure 4.4.3.2. Micro Frontend shared resources graph

Finally, the idea of the micro-ui's architecture will be the following.

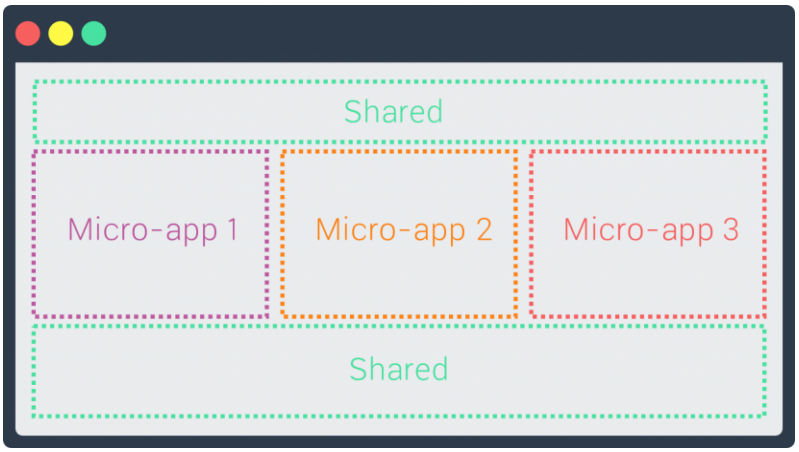

Figure 4.4.3.3. Micro Frontend shared resources diagram

4.4.4. REASONS TO ADOPT

Famous and huge companies such as Zalando, Ikea, and Spotify among many others, are adopting Micro-Frontends. However, the reasons why those big companies are adopting them are not yet clear for the developers. As we said before, the main reason is the increased complexity of the monolithic frontend and the need to scale the development teams. Also, it is curious to mention that several companies are adopting Micro-Frontends because a lot of other companies are adopting them. Let's see now which are the main reasons why those companies decided to perform this change.

I. Frontend Growth

As frontend growth, it becomes harder and harder to maintain. There are three motivations for the adoption of Micro-Frontends related to frontend growth.

A. Large codebase

In the past years, the backend side has moved from monolithic backend to microservices architecture. However, the frontend side remains monolithic. Over time, the front-end grows so much that it becomes very difficult to understand how the application works. Therefore, the monolithic frontend becomes hard to scale from the development point of view, and it is mainly impossible to keep evolving it. If we adopt Micro-Frontends, the code of each one is going to be much simpler and easy to understand and maintain long term.

B. Increased complexity

The front-end will eventually become more and more bloated and front-end projects will become more and more difficult to maintain and the application becomes unwieldy. All the functionalities in the application depend on each other. This means if one function stops working, the whole application goes down.

C. Organizational problems

Most of the development teams are working in an agile managed project delivery process that advocates cross-functional over a cross-technical team, for example, Angular team, Java team, DB team, etc. Micro-Frontend provides the flexibility to have a cross-functional team over a cross-technical team that focuses on end-to-end delivery.

II. Scalability

A. Scale development teams

As a monolithic application is being developed, its life cycle expands the more the application grows. Over time, the amount of features teams need to support grows also. While multiple teams are contributing to a monolithic application, the more tedious the development and release coordination becomes.

B. Independent deployments

Every time we update a monolithic web application, it is necessary to update it completely. We cannot update only one functionality while keeping the rest of the functionalities old, because it can produce problems in the website. This increases the chance of breaking the application in production, introducing new bugs and mistakes especially when the code base is not tested as a whole. By adopting Micro-Frontends, development teams achieve flexibility in development and operations.

C. Fast delivery

Nowadays, it is important to be fast in terms of deploying. The software industry is moving fast forward and companies are dependent on applications and new features on them. With multiple teams working on the same code base this target is hard to achieve.

III. Code-base rules evolution

In most companies, there are code-based rules we should follow and write our code according to those. Oftentimes, those rules are defined once and remain the same for months or even years. The main reason for this is that changing any rule would require a lot of time and effort across the entire code-base and be a large investment for the organization.

IV. Slow onboarding

The initiation of a new developer is time-consuming since the large codebase is confusing. This is because the application has grown too large and has too many edges to explore.

V. Killing innovation

Using a monolithic codebase forces developers to introduce new techniques and apply them to the entire project for maintaining codebase consistency. Due to the nature of the Micro-Frontends development team can evolve part of the application without affecting the entire system. In this way, testing a new version of a library or even a completely new UI framework will not provide any harm to the application’s stability.

4.4.5. BENEFITS

After studying the reasons why to adopt Micro-Frontends in our application, we have to think also about the benefits we achieve by doing that, which are a huge amount.

I. Different technologies support

From my point of view, this is one of the best benefits, since each development team can choose in which stack or framework they want to work, such as Angular, React, Vue, etc. It is a great benefit that a new framework can be chosen without having to rewrite the existing system, and the technologies can be selected by the development team, based on their needs and their skills.

II. Autonomous cross-functional teams

As I explained before, Micro-Frontends bring the concept and benefits of Microservices to front-end applications. Multiple teams can work on different parts of the application without affecting each other. They can create and innovate architectural decisions inside their applications because it would not affect the other teams, it means, the other applications. It is important to understand that each team has a distinct area of business that it specializes in.

III. Independent development, deployment and managing, and running

Remember that each Micro-Frontend is an independent application, and changing it will not affect the other applications and it is also more maintainable. On a large application, many teams can work parallel and produce features fast without the need to coordinate with other teams. Therefore, Micro-Frontends enable teams to develop independently, quickly deploy, and test individually, helping with continuous integration, continuous deployment, and continuous delivery.

IV. Better testability

Testing as well becomes simple because we do not have to touch the entire application if we have done a small change in our code, and there is no need to run the whole test suite every time.

V. Improved fault isolation

In case something goes wrong with a certain application, there is no need to shut down the entire system. We can inform the user about the issue, but keep running the other applications while this one is being fixed or reloaded.

VI. Highly scalable

Micro-Frontends make it easier to add new features or spin up teams when needed. We can distribute the development of end-to-end features between many teams, and each one will focus on developing, deploying, and maintaining their solutions. Since Micro-Frontend has a modular structure, we can easily upgrade it according to our business needs or market trends. There is no need to upgrade the entire front-end.

VII. Faster onboarding

Structuring our frontend Layer with Micro-Frontends makes it easier for new developers when they join the development team. It is easier to understand the system and the architecture, and it can be done almost immediately.

VIII. Fast initial load

Application shell loads micro applications based on the route when the user comes to the web application.

IX. Improved performance

Unlike monolithic applications, if a specific application does not load or is slow loading, this would not affect the rest of the application. We do not have to wait until all the applications are loaded to start using the system, since they are independent.

X. Future proof

New frameworks and technologies can be tested and integrated easily, same with their rejection. Teams are free to choose their technology of choice, which makes the application future-proof.

4.4.6. ISSUES

Even though Micro-Frontends have many benefits, they also have some drawbacks we should take into account, which are the following.

I. Increased payload size

Using multiple technology stacks may have a negative impact on end-users. This is because the browser needs to fetch a lot of data if our application uses more than one JS framework, such as Angular and Vue for example. This may slow the loading time of the application.

II. Code duplication

Micro-Frontends can cause duplication of common dependencies, increasing the number of bytes applications have to send over the network to end-users. For example, if every Micro-Frontend includes its copy of Angular, then we are forcing our end users to download Angular n times.

III. Shared dependencies

The dependency redundancy between sub-projects after integration increases the complexity of management. At the end of the day, Micro-Frontends will have shared dependencies and shared code. This is hard to nail and requires more testing.

IV. UX consistency

We need to understand that the user experience may become a challenge if the autonomous individual teams go with their direction without consulting the other teams and make some basic common decisions. The purpose of Micro-Frontends is to abstract teams from each other so that they can work separately. However this could lead to a system with many applications with different styles each one, and that is kind of chaotic.

A possible way to increase UX consistency is to use a shared CSS stylesheet, but it means that all applications would depend on one common source. Having a common style guide helps to keep the look and feel consistent (but not identical) across the whole system. Alternatively, we could use a common component library included in each application.

V. Monitoring

Tracking and debugging problems across the entire system is complex and difficult.

VI. Increased level of complexity

Micro-Frontends add complexity at the technical and organizational levels. This can become a problem if we need to support a large number of significantly different clients implemented with different technologies, such as web, mobile, and desktop among others. The integration of multiple sub-projects applications becomes complicated. As a distributed architecture, Micro-Frontends will inevitably lead to having more subjects to manage.

VII. Governance

As the applications are developed independently, there is a need for collaboration between the multiple teams working on one product. They should be aligned and have a common understanding, though when there is a change in multiple directions in terms of organizational and technology strategy.

VIII. Islands of knowledge

It is understandable that if each team works on its own application end-to-end, its code, they do not interact with each other too much, and do not expose to other teams what they are doing. To improve this, it is important to constantly make internal meetups or scrum sessions so at least the developers would know what their coworkers are doing.

IX. Environment differences

There exists a risk associated with the development of the application in a different environment from production since if the applications development-time container behaves differently than the production one, the team might find that their Micro-Frontend application is broken, or behaves differently when they deploy it to production.

X. Higher risk when releasing updates

Just as teams can distribute new changes instantly across many services, they are also able to distribute bugs and errors. These errors also surface at application run-time rather than at build time or in continuous integration pipelines.

XI. Accessibility challenges

Some of the implementations of Micro-Frontends, particularly looking at embedding iFrames, can cause huge accessibility challenges. It is recommended that if the application has accessibility requirements it is simply to avoid using iFrames.

4.4.7. APPROACHES

It is essential to understand that Micro-Frontends are not frameworks or libraries, it is an approach to organize and divide big applications into smaller ones, conducive to better maintainability. It is about some kind of orchestration of independent apps, in a way that the final user sees only a single application on its screen. Different approaches could be implemented. I am going to explain the high-level architecture of those.

IFRAMES

Iframes are just HTML documents that can be embedded inside another HTML document. You can place them wherever you want. You only have to write the source of the iframe and the height or width you want. If not, it is gonna take the width and height of the frame in the parent document.

```html
<!DOCTYPE html>
<html>
<body>

<iframe src="http://localhost:4300" width="500" height="500">
</iframe>

<iframe src="http://localhost:4301" width="500" height="500">
</iframe>

</body>
</html>
```

Figure 4.4.7.1. Iframes approach to achieve Micro Frontend architecture

In the previous example, we are running two applications on port 4300 and 4301, which are the sources for our iframes. This kind of approach better suits the project where all the functionality resides on the same page without any navigation. It does not suit when there is navigation and routing involved in the project.

NGINX

NGINX can be used as a web server or reverse proxy to serve the static content. We can use NGINX to route the appropriate Micro App based on the context path. If we look at the following diagram we have an NGINX web server in between to serve each Micro-Frontend based on the context path or routing, for example, /app1 loads the Micro App 1, /app2 loads the Micro App 2, and so on.

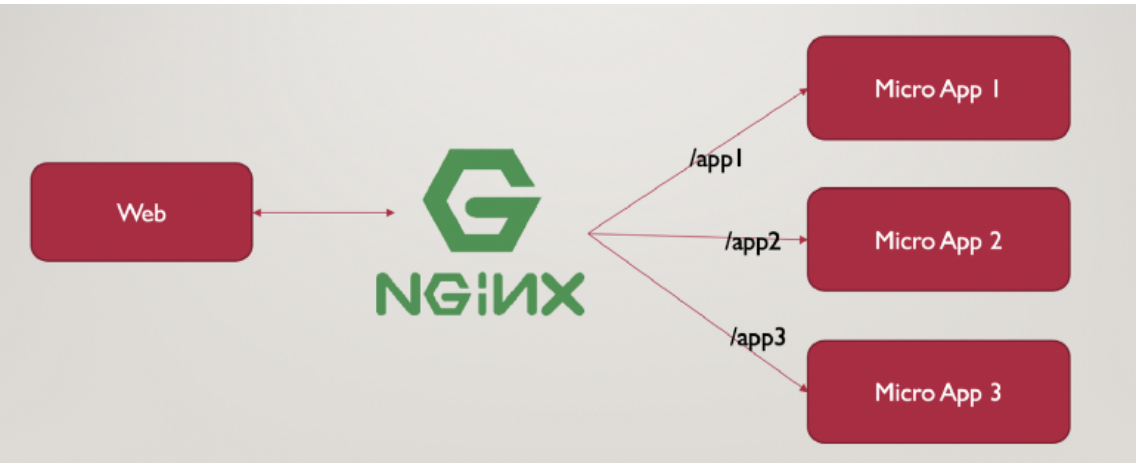

Figure 4.4.7.2. Nginx approach to achieve Micro Frontend architecture diagram

This is quietly easy to achieve with the NGINX configuration file. We only need to define block directive location for each Micro Frontend and load the appropriate app based on the location path or context root.

```
worker_processes 4;

events { worker_connections 1024; }

http {
    server {
        listen 80;
        root  /usr/share/nginx/html;
        include /etc/nginx/mime.types;

        location /app1 {
            try_files $uri app1/index.html;
        }

        location /app2 {
            try_files $uri app2/index.html;
        }

        location /app3 {
            try_files $uri app3/index.html;
        }
    }
}
```

Figure 4.4.7.3. Iframes technical approach to achieve Micro Frontend architecture

As I said before, this kind of approach suits better in a project where there is navigation or routing involved, and the application itself is divided into multiple applications based on features.

The communication happens through the NGRX store and the local storage as shown in the diagram below. During the navigation between the different micro applications, the app state is always stored and transferred through some kind of state management tool and local storage.

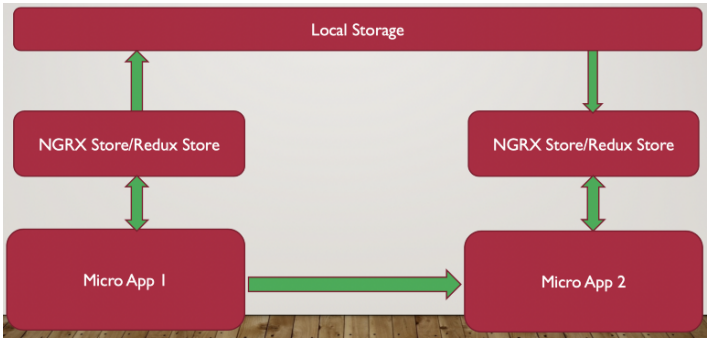

Figure 4.4.7.4. Nginx approach to achieve Micro Frontend architecture diagram

The main disadvantage of using NGINX for micro frontends is that the page refreshes each time we switch between the micro-apps. Therefore, it is advisable to maintain some shared components on each app such as header and footer, which is beneficial to maintain the same layout.

WEB COMPONENTS

Web components are the combination of different technologies that allow us to create reusable components on the web. It consists of three specific technologies. Custom elements allow us to create elements that can be used wherever we want on the page. Shadow DOM allows us to run our code in a separate DOM other than the main DOM which provides encapsulation. Finally, the HTML templates allow us to write markup templates that can be used multiple times.

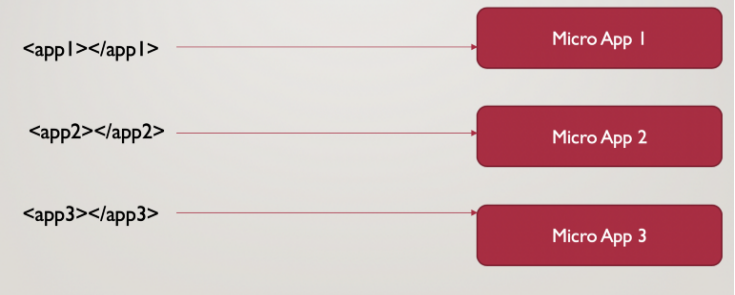

Figure 4.4.7.5. Web components approach to achieve Micro Frontend architecture diagram

LIBRARIES

Each Micro-Frontend can be a library that can be simply pushed as a node module into some private repository. There is a need for a shell app that pulls that repo wherever we need it with lazy loading (if it's Angular) or dynamic imports. In the diagram below, three Micro applications canto be converted to three libraries and pushed into the repository.

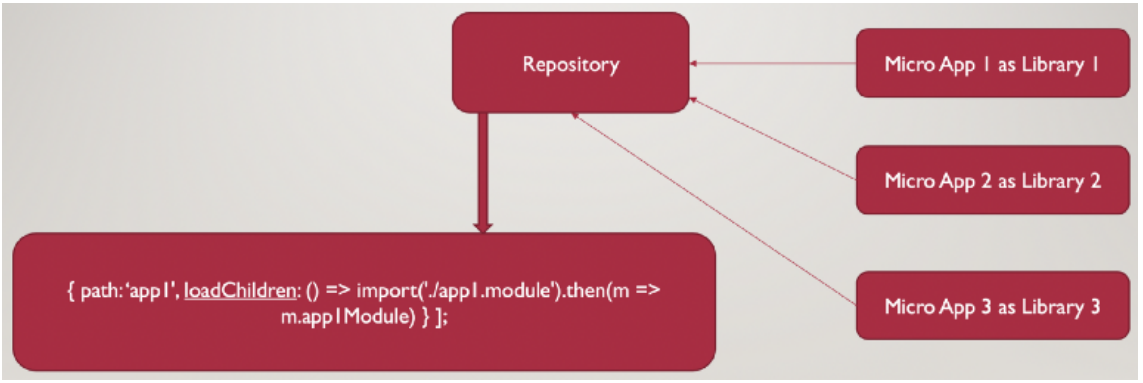

Figure 4.4.7.6. Library approach to achieve Micro Frontend architecture diagram

Communication is not a big deal here since all the individual libraries end up in the same project or application.

MONOREPOS

This is a strategy used by many companies, where we can put multiple related projects under one repo. By using this, we do not have to push shared code to a separate repo as a library or module and pull it for use. In the diagram below, there is only one mono repo that contains all the projects and shred code as well. In this way, we do not need to create any separate libraries for the sharable code. All the shared code and actual projects live in the same repo.

Figure 4.4.7.7. Monorepo approach to achieve Micro Frontend architecture

FRAMEWORKS

Micro frontends have been implemented for at least two years and it is still a green field. However, there are a couple of libraries and frameworks which can help us to achieve the Micro Frontend Architecture.

SINGLE-SPA

This is a Javascript framework oriented for frontend microservices and can be implemented with all three popular frameworks such as Angular, React, and Vue.js. It can lazy load the applications based on the need.

FRINT.JS

This is a modular Javascript framework for building scalable and reactive applications. It does not support Angular at the moment but it supports React. If you are building a reactive application from scratch and you are just starting, this is the framework for you.

CUSTOMIZED ORCHESTRATOR

The main idea of this approach is to define plain Javascript files to orchestrate the entire workflow of the application and its micro-projects. All of them are deployed independently, and the orchestrator can load each project based on the URL.

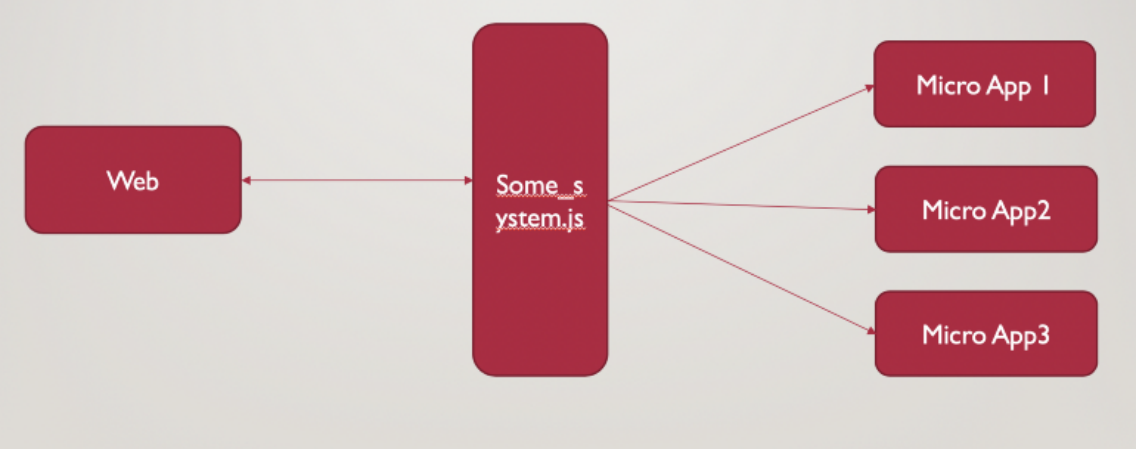

Figure 4.4.7.8. Customized Orchestrator approach to achieve Micro Frontend architecture diagram

For communication, we can define some global namespace and objects in the orchestrator. These global objects are available for all the projects so that you can send data among applications.

I would say this is the most difficult implementation since it is completely bound to the developers. You start from scratch and have to define your orchestrator according to your needs, and I think this is what we are going to try to develop our project.

DAZN

I am going to take the example of DAZN, a sports streaming platform that implemented the Micro-Frontends architecture, with the customized orchestrator approach.

It is important to understand what a Micro Frontend is for DAZN. The first thing they have done is define a manifesto, which talks about independent implementation avoiding sharing logic, modeled around a business domain, and owned by a single team.

Their idea is they have one domain, which is the application. But what they have done, is to think about how to divide it into many subdomains, based on different business domains.

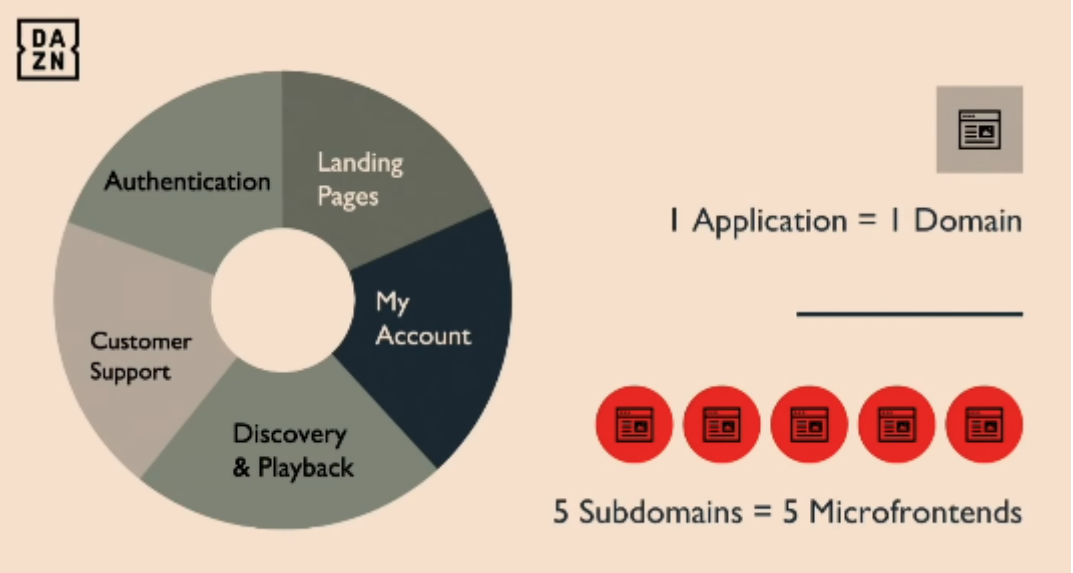

Figure 4.4.7.9. DAZN approach to achieve Micro Frontend architecture diagram

Their micro frontends are one hundred percent autonomous single-page applications, and only one micro frontend is loaded per time. One team, one subdomain, one micro frontend. We need to think about a micro frontend as a normal single-page application, composed of HTML, CSS, and javascript files.

This is when we reach the most important question. We ask ourselves how these micro frontends fit all together. This is the moment we think about an orchestrator. DAZN here decided to select bootstrap as its orchestrator.

Bootstrap is the first thing the web page loads, and it downloads the Micro Frontend. It means the micro frontend gets loaded by bootstrap.

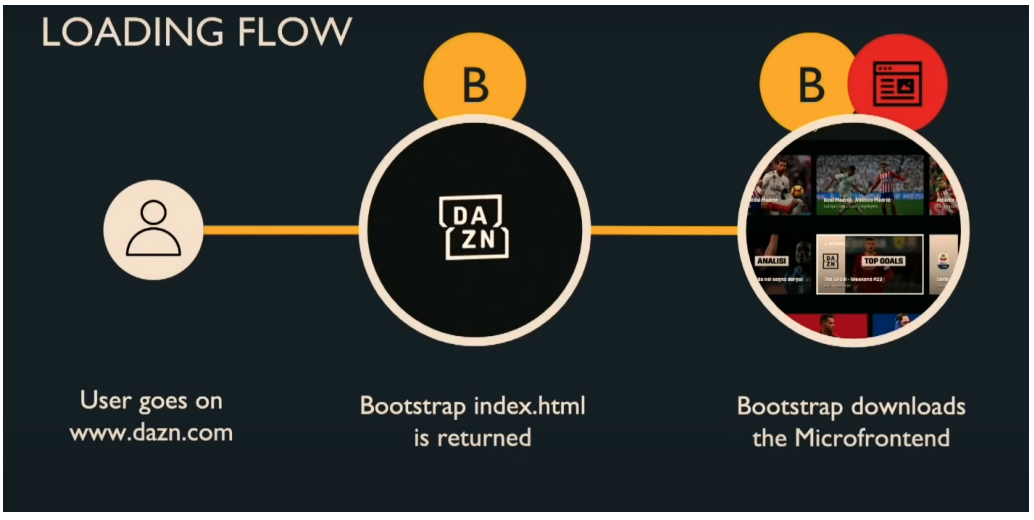

Figure 4.4.7.10. DAZN application loading flow

However, bootstrap is not only that, it is much more, and it has more responsibilities and benefits. It is in charge of the application startup as I have already said. The way their application starts is bootstrap responsibility. Another benefit is the client-side routing, which means we can go from one micro frontend to another, just using bootstrap.

Those transitions are handled by bootstrap and lifecycle APIs, since it is important to understand at every time if there is a need to load a micro frontend, or is loaded, or is loading, or needs are unloaded, and so on. This is how a standard communication between the different micro frontends, thanks to bootstrap.

And last but not least, I / O Operations. DAZN not only centers on web pages but on many other devices such as Smart TVs, Play Stations… and all of these require a layer of abstraction.

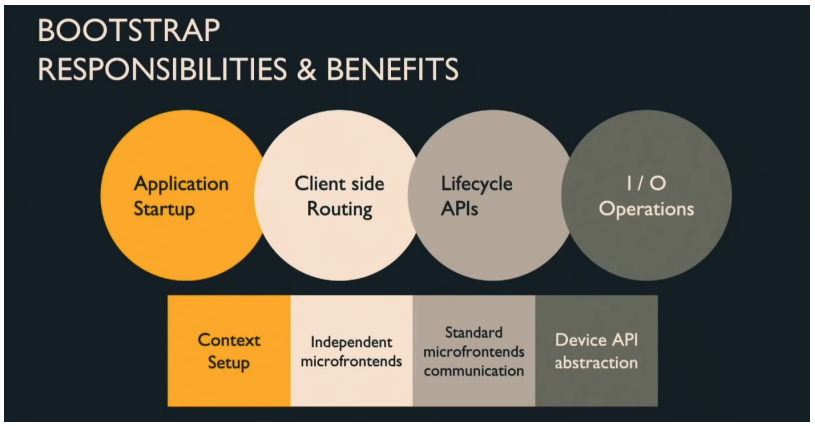

Figure 4.4.7.11. DAZN bootstrap responsibilities and benefits

So by this, what DAZN archives are scaling the Teams, reducing communication overhead, and innovating.

CHAPTER 5: PLANNING AND PROJECT MANAGEMENT

We are going to use GitLab for project management as well as our version control system. It is a very powerful tool, which is used in a huge amount of companies all around the world, including mine.

Gitlab itself has many features, and due to its popularity, we selected it as our main software. One of the main benefits of GitLab is the fact we can create groups, with which we can assemble related projects and grant members access to several projects at once. I have created a group named “Thesis Illia”, and I store all the little projects and the main one here.

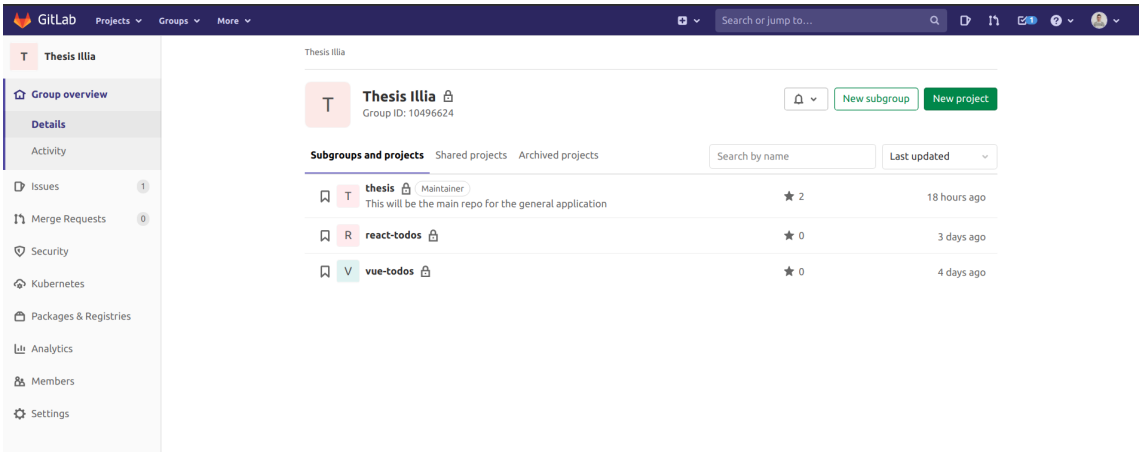

Figure 5.1. Gitlab repository with all the small and big applications

I have also added some members. My teaching coordinator, the responsible for software development and the responsible for DevOps from my company, so they can follow up on the project.

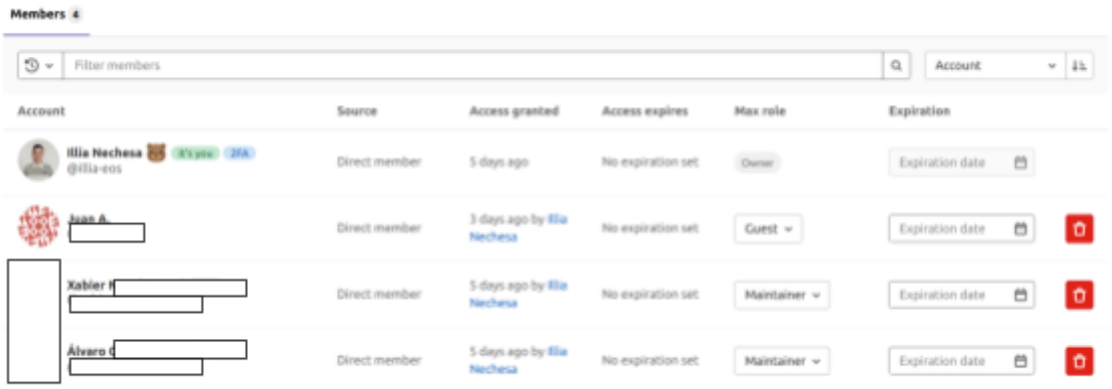

Figure 5.2. Gitlab users to the thesis repository

All the documentation is reachable from the Wiki section of the project, where the index page is located. This page contains the links to all the files we create or use during the development. All of those are located in a shared folder in google drive with my teaching coordinator, and each file is accessible from the wiki page.

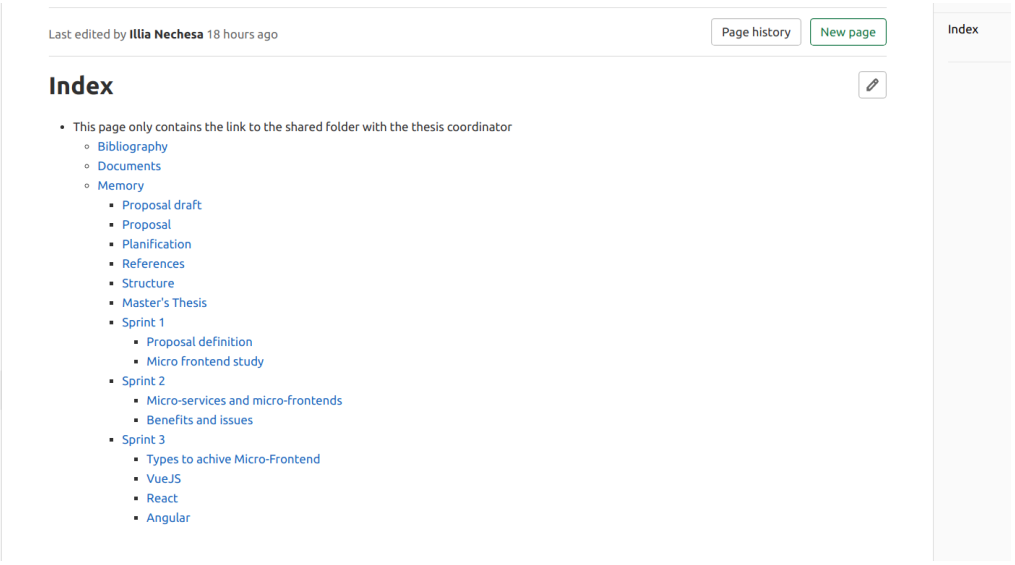

Figure 5.3. Gitlab index redirection to google documents

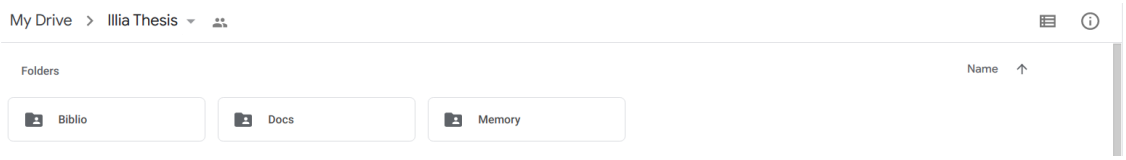

Figure 5.4. Google documents folder

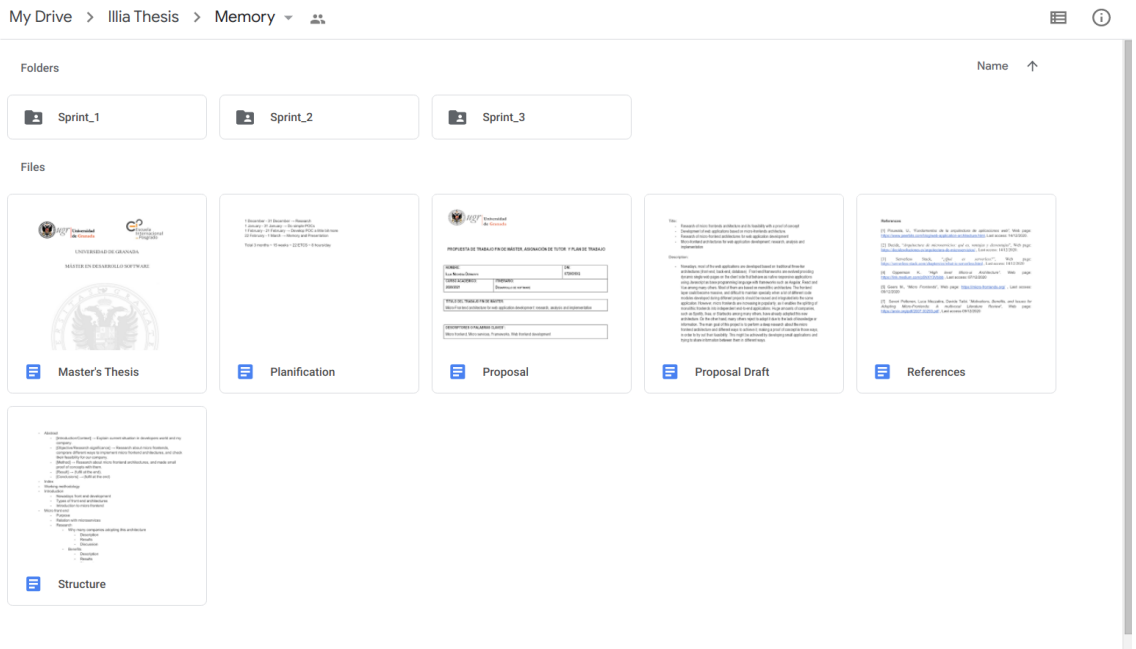

Figure 5.5. Google documents memory folder

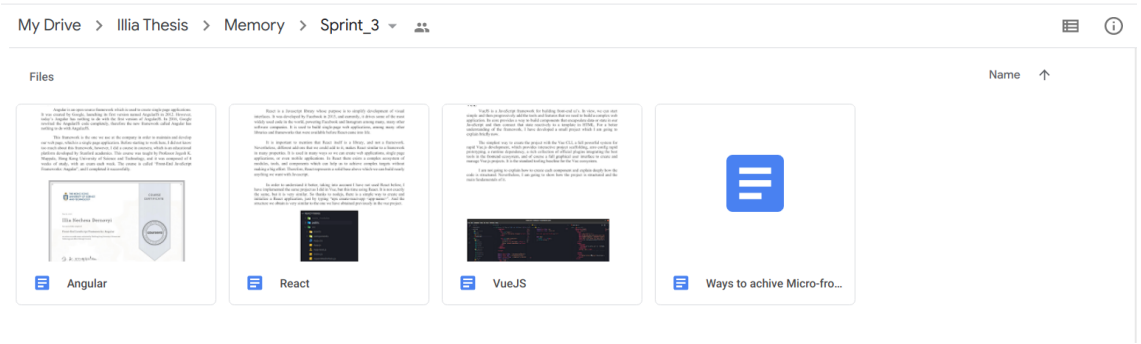

Figure 5.6. Google documents sprint folder

The working methodology used is a combination of SCRUM and Kanban. This is due that we do not follow SCRUM accurately. We made sprints every one or two weeks, defining the tasks to do each sprint. To organize those tasks better, we use a Kanban board with four basic columns. Open, which is used as the backlog, To Do, Doing, and Closed. Remark that each project of the group has its issues and milestones, which makes it easier to organize the projects and the development of each one independently.

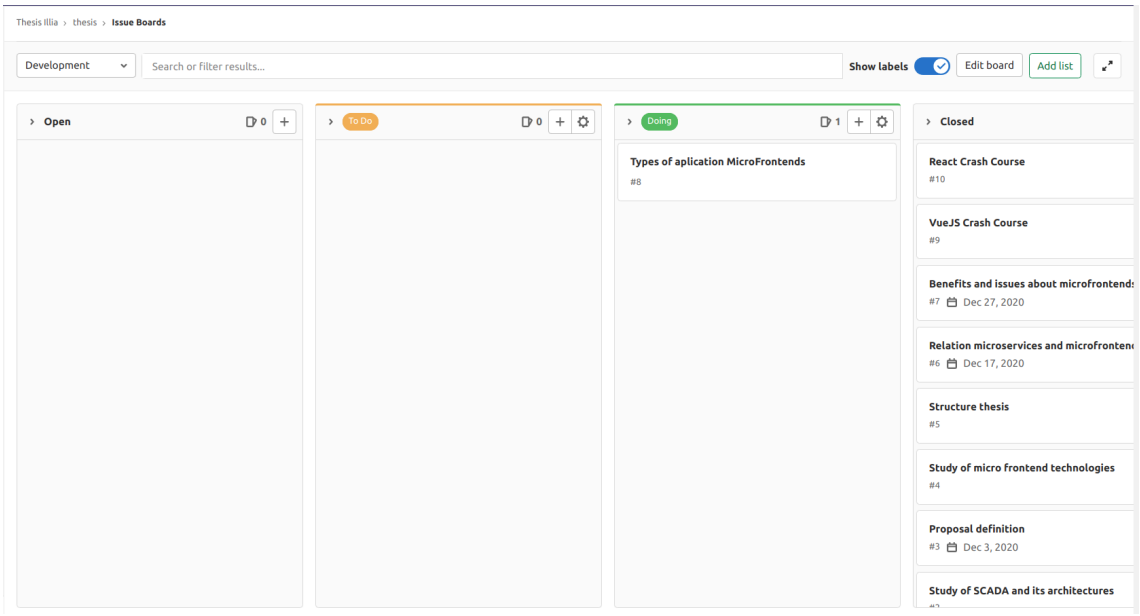

Figure 5.7. Gitlab tasks and sprint board

Finally, tasks are treated as issues in GitLab and can be assigned to a specific milestone. This milestone is what we know as Sprint.

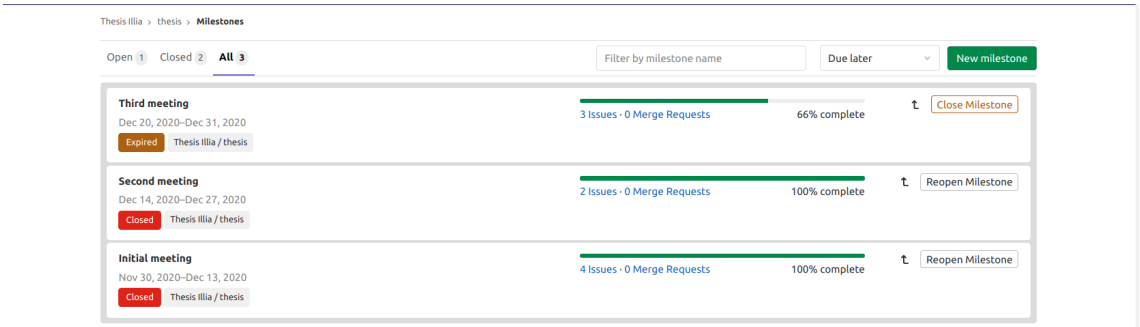

Figure 5.8. Gitlab sprints

The issues can be open or closed. The open ones can be on the backlog state, and on doing state. When the issue is finished, we just close it. If in the future we need, we can reopen it and continue or change something on it.

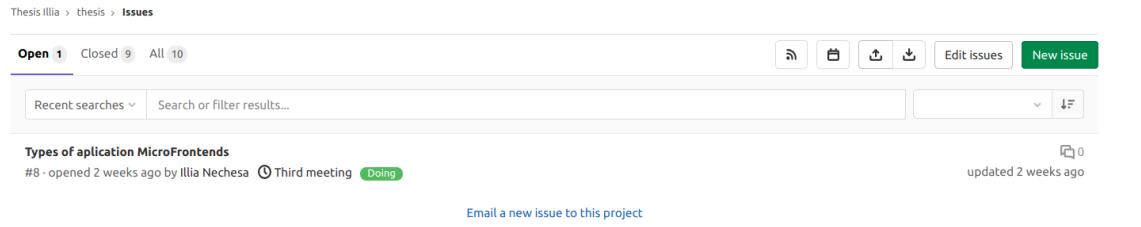

Figure 5.9. Gitlab issues

The most interesting about the issues is that we can assign them to any member of the group, and also assign them to a specific milestone, also named Sprint in SCRUM methodology. Apart from that, we can assign labels and establish a due date for the issue.

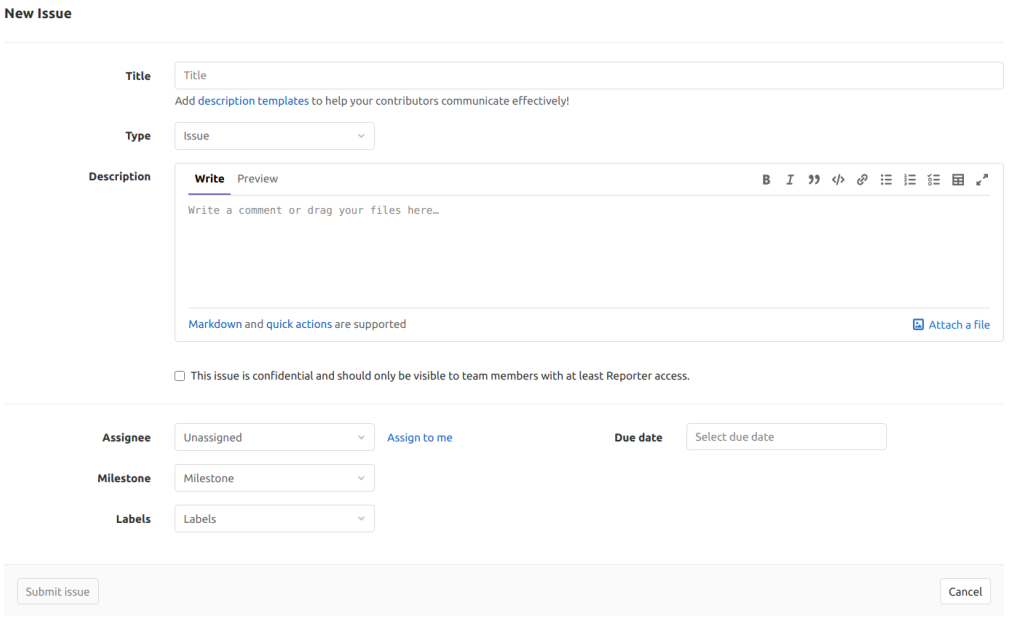

Figure 5.10. Gitlab issue creation

All those projects are being stored in the company Gitlab repository. However, as I do not reveal any confidential information, I have also stored them in my GitHub personal repository. This was achieved by defining a git remote that points to multiple git remotes.

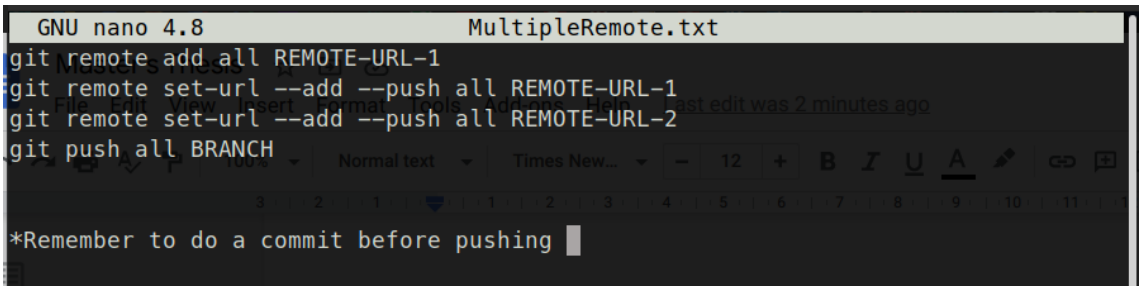

Figure 5.11. Gitlab setup for multiple remote

In the upper figure, we can see that we add two remote URLs to the “all” git remote. With this, all the commits are stored in all our remote repositories, and by pushing to all and to the branch we want, we will push at all at the same time. Shame that we cannot make groups on GitHub. However, all the projects are being stored in the same way.

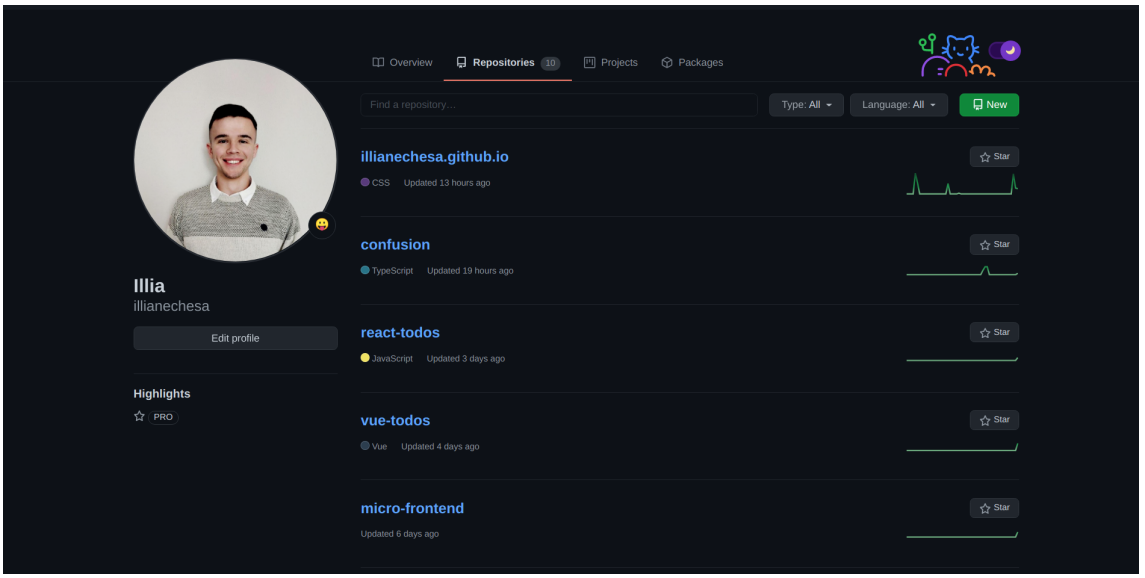

Figure 5.12. Github where the main repository is stored

CHAPTER 6: FRAMEWORK DEVELOPMENT

6.1. INTRODUCTION

For the application development and in order to achieve the micro-frontend architecture, we need to use some kind of framework that can permit us to do so. This is why I chose single-spa since it is quite common in the micro-frontend community. Its purpose is to bring together multiple JavaScript micro-frontends in a frontend application

6.2. REQUIREMENT SPECIFICATION

The Single-spa npm package is not opinionated about the build tools, CI process, or local development workflow. However, its team has put together a "recommended setup" that gives an opinionated approach to solving the practical problems of micro-frontends. It is highly recommended to use a setup that uses in-browser ES modules plus import maps (or SystemJS). This setup has several advantages such as the following ones:

- Common libraries are easy to manage and are only downloaded once. If we are using SystemJS, we can also preload them for a speed boost as well.
- Sharing code, functions, or variables is as easy as import/export, just like in a monolithic setup.
- A lazy loading application is easy, which enables us to speed up initial load times.
- Each application can be independently developed and deployed. Teams have the freedom to work at their own speed, experiment, QA, and deploy on their own schedules.
- Great developer experience.

6.2.1. USE CASES

Architecting our frontend using this framework enables many benefits, such as:

- Use multiple frameworks on the same page without page refreshing (React, AngularJS, VueJS, Ember...)
- Deploy the micro-frontends independently.
- Write code using a new framework, without rewriting an already existing app.
- Lazy load code for improved initial load time.

6.3. SOFTWARE ARCHITECTURE

Single-spa takes inspiration from modern framework component life cycles by abstracting lifecycles for entire applications. Born out of Canopy's desire to use React + react-router instead of being forever stuck with our AngularJS + ui-router application, single-spa is now a mature library that enables frontend microservices architecture or "micro frontends". Micro frontends enable many benefits such as independent deployments, migration and experimentation, and resilient applications.

Single-spa apps consist of the following:

1. A single-spa root config, which renders the HTML page *and* the JavaScript that registers applications. Each application is registered with three things:
    - A name
    - A function to load the application's code
    - A function that determines when the application is active/inactive
2. Applications can be thought of as single-page applications packaged up into modules. Each application must know how to bootstrap, mount, and unmount itself from the DOM. The main difference between traditional SPA and single-spa applications is that they must be able to coexist with other applications as they do not each have their HTML page.
   For example, your React or Angular SPAs are applications. When active, they can listen to URL routing events and put content on the DOM. When inactive, they do not listen to URL routing events and are removed from the DOM.

6.4. COMMUNICATION

The main challenge of the Micro-Frontends Architecture, as I have already said, is to establish some kind of communication between the different applications, since they might share functions, components, logic, states… Functions, components, common logic, could be placed on a third package and imported on each app, and that should not require much effort. But sharing state is completely different, and there are some possible approaches for this. I will explain some of them, which are the most popular options.

6.4.1. CUSTOM EVENTS

Using synthetic events is one of the most common ways to communicate using event listeners and CustomEvent. The setup is simple, it is customizable, framework agnostic, and Micro-Frontends do not need to know their parents. Here is an example illustrating a simple communication between two Micro-Frontends.

```
microfrontend-1.js
home > illia.nechesa > Escritorio > microfrontend-1.js > App
function App() {
  const [messages, setMessages] = useState([]);

  const handleNewMessage = (event) => {
    setMessages((currentMessages) => currentMessages.concat(event.detail));
  };

  useEffect(() => {
    window.addEventListener('message', handleNewMessage);

    return () => {
      window.removeEventListener('message', handleNewMessage)
    }
  }, [handleNewMessage]);

  ....
}
```

Figure 6.4.1.1. Custom events communication

```
microfrontend-2.js
home > illia.nechesa > Escritorio > microfrontend-2.js > App
function App({ onNewMessage }) {
  const handleSubmit = (e) => {
    e.preventDefault();

    const { target: form } = e;
    const input = form?.elements?.something;

    const customEvent = new CustomEvent('message', { detail: input.value });
    window.dispatchEvent(customEvent)
    form.reset();
  }

  ....
}
```

Figure 6.4.1.2. Custom events communication

6.4.2. PUBSUB LIBRARY

Nowadays, all services, apps, and frontends have one thing in common, which is distributed systems. And reading about the microservices environment, there is a pretty much popular communication mode which is pub/sub-queues. Taking into account that every micro frontend including the container is at the window, it is a good idea to hold a global communication using pub/sub implementation. There are already some open-source libraries that we could use to establish this communication, as "windowed-observable". Here is an example exposing an observable attached to a topic to publish, retrieve, and listen to new events on its topic.

```
import { Observable } from 'windowed-observable';

// Define a specific context namespace
const observable = new Observable('cart-items');

const observer = (item) => console.log(item);

// Add an observer subscribing to new events on this observable
observable.subscribe(observer)

// Unsubscribing
observable.unsubscribe(observer);

...

// On the publisher part of the app
const observable = new Observable('cart-items');
observable.publish({ id: 1234, name: 'Mouse Gamer XyZ', quantity: 1 });
```

Figure 6.4.2.1. Pubsub library communication

In this library, there are also some features such as retrieving the latest event published, getting a list with every event, clearing every event, and more. Now let's see an example.

microfrontend-1.js

home > illia.nechesa > Escritorio > microfrontend-1.js > App

```
import { Observable } from 'windowed-observable';

const observable = new Observable('messages');

function App() {
  const [messages, setMessages] = useState([]);

  const handleNewMessage = (newMessage) => {
    setMessages((currentMessages) => currentMessages.concat(newMessage));
  };

  useEffect(() => {
    observable.subscribe(handleNewMessage);

    return () => {
      observable.unsubscribe(handleNewMessage)
    }
  }, [handleNewMessage]);

  ...
}
```

Figure 6.4.2.2. Pubsub library communication

Figure 6.4.2.3. Pubsub library communication

The setup is also simple, it is pretty much customizable, it has extra features to retrieve dispatched events… and it is open source.

6.5. IMPLEMENTATION

In this section, we are going to explain how to implement this framework. If we want to achieve the micro-frontend architecture our application must contain a container app, which serves as the main page container and coordinates the mounting and unmounting of the micro-frontend applications.

Let's imagine we have this container app, and then we want to have a navigation bar application that will always be displayed, and two other apps that only will be shown when active. Those four applications are completely independent apps and located in different repositories for example. To use the framework, it is easier to use a command-line interface tool called create-singe-spa.

To create the container app, which is mostly named as root config, we should execute the following commands:

```
mkdir single-spa-demo

cd single-spa-demo

mkdir single-spa-demo-root-config

cd single-spa-demo-root-config

npx create-single-spa
```

Figure 6.5.1. SingleSPA Application creation

During these steps, we should select our dependencies manager, which yarn or npm, and the “single spa root config” option, followed by the organization name. And this is for the container app.

According to the micro-frontend applications, it is quite similar:

```
cd ..

mkdir single-spa-demo-nav

cd single-spa-demo-nav

npx create-single-spa
```

Figure 6.5.2. SingleSPA Application creation

However at this point, differently from the previous one, we need to select the “single-spa application” option, followed by our dependencies manager and the framework in which we are going to develop this micro application, whether Angular, React, VueJS, or any other available. Finally, we shall also add the organization name, which should be the same when creating the root config app, and enter a project name for this micro-application.

Let’s now think that we have done this for all of our 4 applications. At this point, we have generated our 4 apps, container one, and three micro-frontend apps. So, how can we put them all together? As I already said before, one of the container app’s primary responsibilities is to coordinate when each app is “active” or not. In other words, it handles when each app should be shown or hidden.

To help the container app understand when each application should be shown, we provide it with what are called “activity functions” Each app has an activity function that simply returns a boolean, true or false, for whether or not the app is currently active. To achieve that, we need to open up the “activity-functions-js” file inside the “single-spa-demo-root-config” directory, and write something like this:

```
export function prefix(location, ...prefixes) {
  return prefixes.some(
    prefix => location.href.indexOf(`${location.origin}/${prefix}`) !== -1
  );
}
```

```
export function nav() {
  // The nav is always active
  return true;
}
export function page1(location) {
  return prefix(location, 'page1');
}
export function page2(location) {
  return prefix(location, 'page2');
}
```

Next, it is needed to register all the micro-frontend apps with single-spa. To do that, we need to use the “registerApplication” function. This function accepts a minimum of three arguments: the app name, a method to load the app, and an activity function to determine when the app is active.

Inside the “single-spa-demo-root-config” directory, in the “root-config.js” file, we’ll add the following code to register the applications:

```
import { registerApplication, start } from "single-spa";
import * as isActive from "./activity-functions";
registerApplication(
  "@illia/single-spa-demo-nav",
  () => System.import("@illia/single-spa-demo-nav"),
  isActive.nav
);
registerApplication(
  "@illia/single-spa-demo-page-1",
  () => System.import("@illia/single-spa-demo-page-1"),
  isActive.page1
);
```

```
registerApplication(
  "@illia/single-spa-demo-page-2",
  () => System.import("@illia/single-spa-demo-page-2"),
  isActive.page2
);
start();
```

Now that we have set up the activity functions and registered our apps, the last step before we can get this running locally is to update the local import map inside the index.ejs file in the same directory. It is needed to add the following code inside the head tag to specify where each app can be found when running locally:

```
<% if (isLocal) { %>
  <script type="systemjs-importmap">
    {
      "imports": {
        "@illia/root-config": "http://localhost:9000/root-config.js",
        "@illia/single-spa-demo-nav": "http://localhost:9001/illia-single-spa-demo-nav.js",
                                                    "@illia/single-spa-demo-page-1":
"http://localhost:9002/illia-single-spa-demo-page-1.js",
                                                    "@illia/single-spa-demo-page-2":
"http://localhost:9003/illia-single-spa-demo-page-2.js"
      }
    }
  </script>
<% } %>
```

Each app contains its startup script, which means that each app will be running locally on its development server during local development. As you can see, our navbar app is on port 9001, our page 1 app is on port 9002, and our page 2 app is on port 9003.

6.6. EXECUTION

For the execution, it is necessary to open as many tabs as applications we have. For the root config, in the "single-spa-demo-root-config" directory, we run "yarn start --port 9000". For the other applications we open each directory and execute the following command on each one: "yarn start --port <<here will be a different port for each application>>". Finally, when all the applications are running, only need to access localhost at port 9000, and inside this container application, we will have all the micro-frontend applications.

6.7. DEPLOYMENT

At this point, the application would be running perfectly locally, but we want to go further and see how we can deploy it. There are several ways to do so, but I am going to explain a specific one that is the most used inside the community. We are going to use AWS S3 to store our assets, and we are going to use Travis CI to run a building job and an upload job as part of a continuous integration pipeline.

- AWS S3 Set up
    - Create an AWS account.
    - New IAM User: An access key ID and a secret access key will be given to us.
    - Create the bucket with whatever name.
    - We add CORS configuration rules.
- Now we are going to create a Travis CI job to upload artifacts to AWS
    - As all the applications are independent, we remember that each app lives in its repository on GitHub. What we are going to do is to integrate Travis CI with each of our repositories and set up continuous integration pipelines for each one.
    - To configure Travis CI for any given project, we create a .travis.yml file in the project's root directory, and insert the following code:

```
language: node_js
node_js:
  - node
script:
  - yarn build
  - echo "Commit sha - $ILLIA_COMMIT"
  - mkdir -p dist/@illia/root-config/$ILLIA_COMMIT
  - mv dist/*.* dist/@illia/root-config/$ILLIA_COMMIT/
deploy:
  provider: s3
  access_key_id: "$AWS_ACCESS_KEY_ID"
  secret_access_key: "$AWS_SECRET_ACCESS_KEY"
  bucket: "single-spa-demo"
  region: "us-west-2"
  cache-control: "max-age=31536000"
  acl: "public_read"
  local_dir: dist
  skip_cleanup: true
  on:
    branch: master
```

Figure 6.7.1. Configuration of Travis file

- Now each time we commit and push new code to the master branch, the Travis CI job will run, which will build the JavaScript bundle for the app and then upload those assets to S3.

- Now the only thing left is to implement the same Travis CI configuration for our other micro-frontend apps, but swapping out the directory names in the .travis.yml file as needed.

We have currently all our applications living in separate GitHub repositories. Each repository is set up with Travis CI to run a job when code is merged into the master branch, and that job handles uploading the build artifacts into an S3 bucket.

But how do these new build artifacts get referenced in our container app? In other words, even though we're pushing up new JavaScript bundles for our micro-frontends with each new update, the new code isn't actually used in our container app yet.

If we think back to how we got our app running locally, we used an import map. This import map is simply JSON that tells the container app where each JavaScript bundle can be found. But, our import map from earlier was specifically used for running the app locally. Now we need to create an import map that will be used in the production environment.

So, using the original import map as a template, we can create a new file called importmap.json, place it outside of our repositories and add JSON that looks like this:

Figure 6.7.2. Import map JSON configuration

Afterward, we just upload it manually to our bucket in S3. And finally, we can now reference this new file in our index.ejs file instead of referencing the original import map.

- One of many options is to host it on Heroku. To do that, we will need to create a simple node.js and Express server to serve our file.
    - In the single-spa-demo-root-config directory, we will install express by running yarn add express (or npm install express). Next, we will add a file called server.js that contains a small amount of code for starting up an express server and serving our main index.html file.

```
const express = require("express");
const path = require("path");
const PORT = process.env.PORT || 5000;

express()
  .use(express.static(path.join(__dirname, "dist")))
  .get("*", (req, res) => {
    res.sendFile("index.html", { root: "dist" });
  })
  .listen(PORT, () => console.log(`Listening on ${PORT}`));
```

Figure 6.7.3. Code for starting up an express server

    - Finally, we update the NPM script in our package.json file to differentiate between running the server in development mode and running the server in production mode:

```
"scripts": {
  "build": "webpack --mode=production",
  "lint": "eslint src",
  "prettier": "prettier --write './**'",
  "start:dev": "webpack-dev-server --mode=development --port 9000 --env.isLocal=true",
  "start": "node server.js",
  "test": "jest"
}
```

Figure 6.7.4. Script to differentiate between running in dev or prod

- Deployment to Heroku
    - Production server ready.
    - Create Heroku account
    - In single-spa-demo-root-config we execute “heroku create illia-single-spa-demo”
    - Git push Heroku master
    - Heroku open
- Successful deployment

CHAPTER 7: CASE OF STUDY

7.1. REQUIREMENT SPECIFICATION

After this deep research of different frontend frameworks based on Javascript and the micro frontend architecture, I found it interesting to develop a small application based on it and to obtain my conclusions, whether it is a good idea to use this architecture or not. Since I have already developed some small “todo” applications in different frameworks, I think it is a good idea to try out to develop a full “todo” application in which we are going to create some kind of CRUD, where each component will have its responsibility.

Those components will be completely independent, developed each one on a different project, using one or more frameworks. The aim of this is not to prove that we can develop an application in different frameworks, but to prove that different applications can, not only coexist with each other but also communicate and share information between them.

The idea is quite simple. We will have a root application that will be in charge of mounting and unmounting the different applications. The first application displayed will be the navigation bar, which will be displayed always independently if the other micro applications are being displayed or not, which means it will always be fixed on the top of the page. This application will be in charge of mounting and unmounting the different applications and establishing the navigation between all of them.

In the left section, we are going to have another application with a list of all the tasks as we can see in the prototype image below. We can check them or not depending on the fact if they are completed or not. Apart from that, by clicking on each task, a detail section will appear on the right side, but it is important to remark that this section will be a completely independent application developed in a different project. In this section, we are going to display different information from the task, such as the title, description, if it is completed or not, and the due date of that task. From this one, we will be able to edit the task or delete it. The edit button would invade another micro application whose purpose would be to add or edit (depending on if the edit or the add button were clicked) the task.

This last application will be in charge as I already said of creating or editing a current task. Depending on this, it will pre-load the information of the selected task, or load an empty form, in which we can update the name, the description, and the due date of the task.

7.1.1. ANALYSIS AND DESIGN

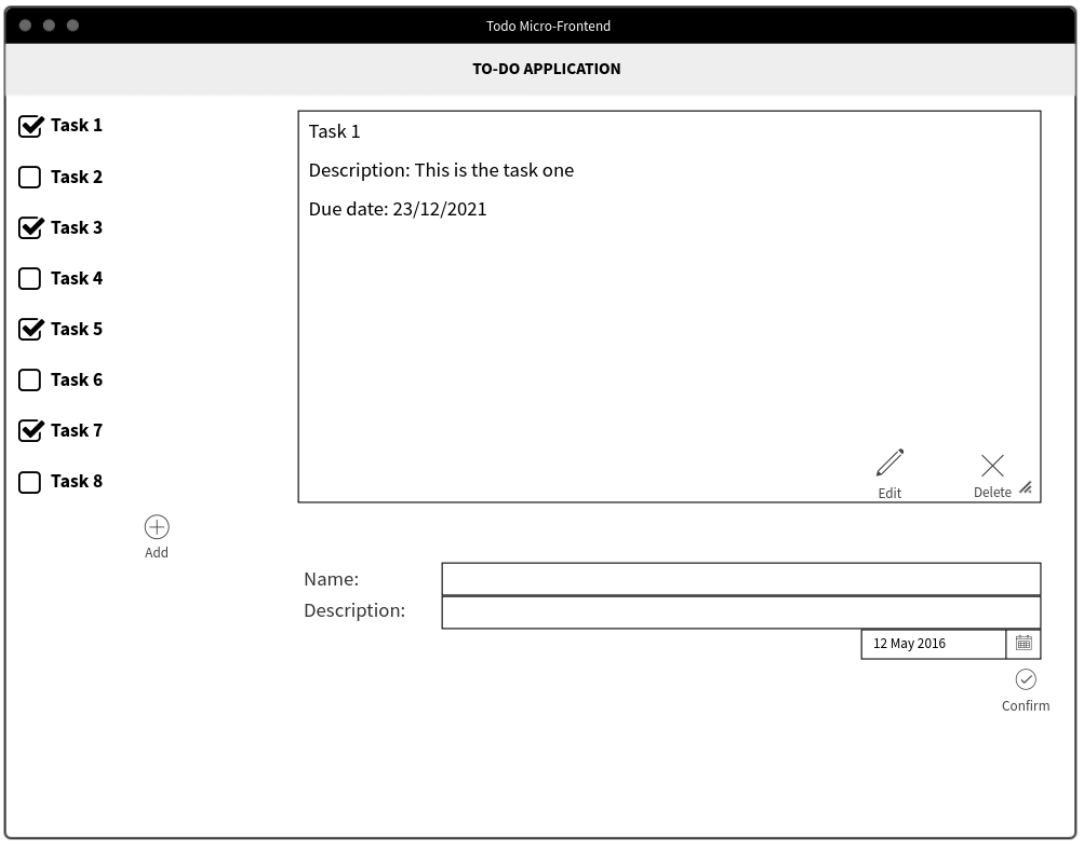

Figure 7.1.1.1. Todo Micro Frontend Application prototype

This is the initial prototype I have done to have a reference of the design of the web application, and also for the different workflows that we could have. Of course, this design may change during the development of the application.

This prototype was done before the implementation of the application, and during the development came across the fact that we will have to update the URL depending on the application we are working in at a specific moment, or which micro application we are executing at that moment. This is why I decided to add another navigation application that will help me achieve this.

7.1.2. USE CASES

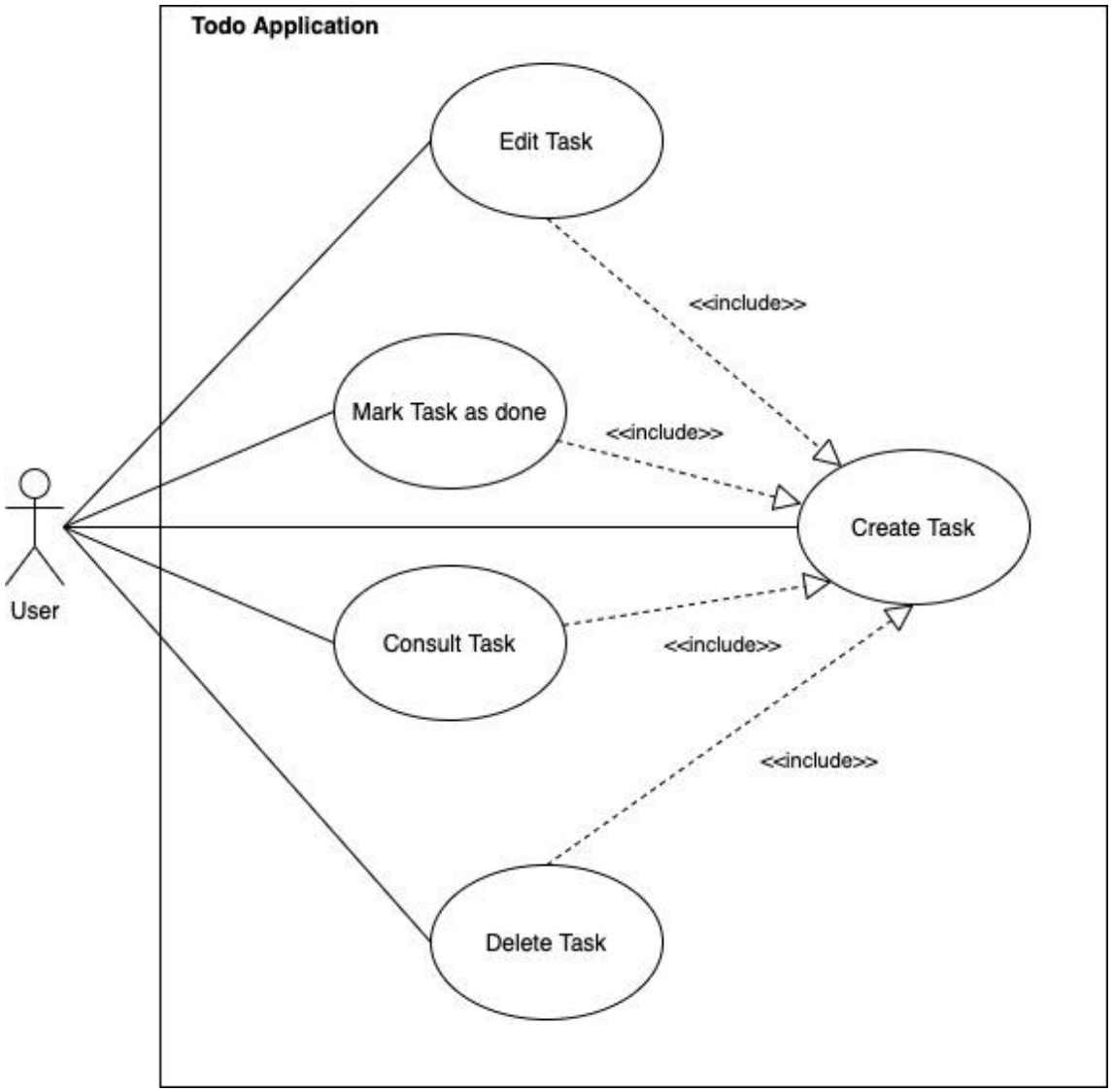

Figure 7.1.2.1. Todo Micro Frontend Application use cases diagram

7.1.3. SOFTWARE ARCHITECTURE

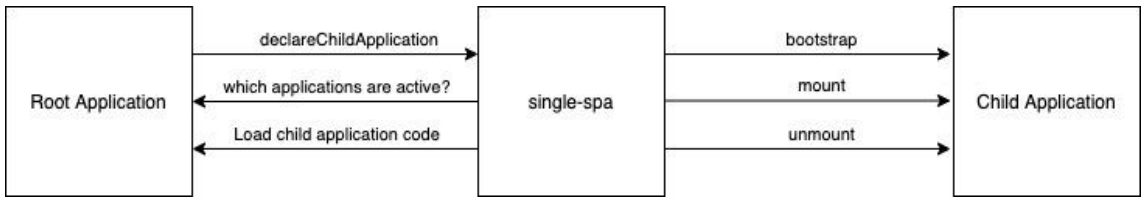

Figure 7.1.3.1. Todo Micro Frontend Application architecture

7.2. IMPLEMENTATION BASED ON SINGLE-SPA FRAMEWORK

In this section, I am going to explain how I implemented the todo application with the single-spa framework and how I managed to establish communication between the micro applications. First of all, let's see all the applications I have created and how I create them.

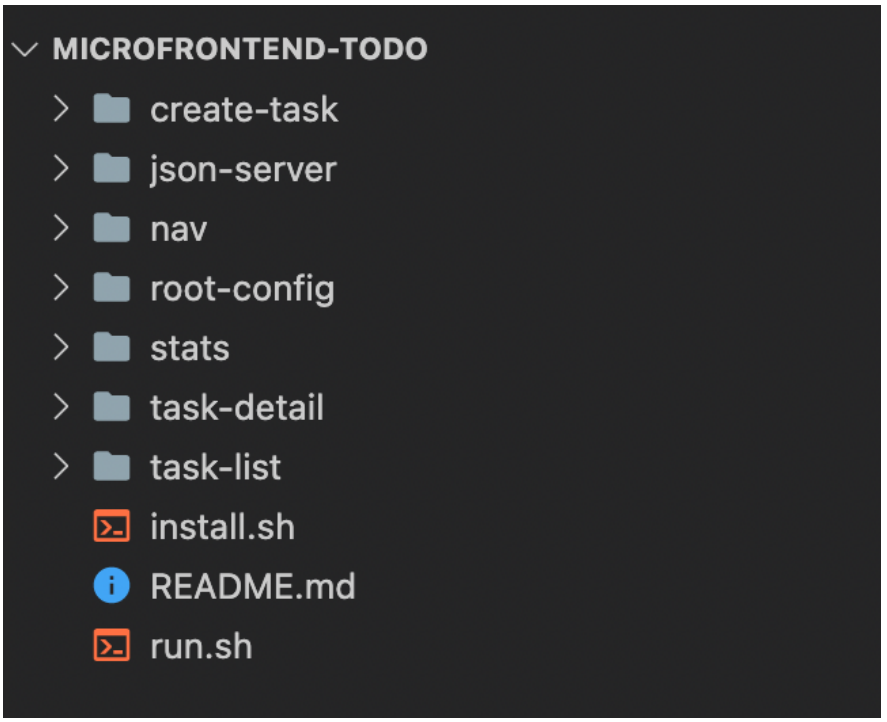

Figure 7.2.1. Todo Micro Frontend Application project structure

7.2.1. APPLICATIONS

- *NAVIGATION*

  This application is located on the top of the web application and gives us the option to navigate to other pages or to mount or unmount different applications. This can be achieved thanks to the ReactJS browser rooter, and it looks like that:

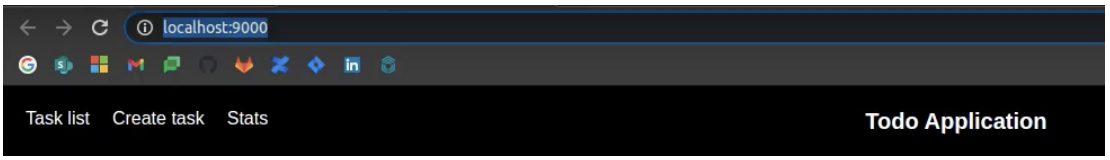

Figure 7.2.1.1. Todo Micro Frontend Application navigation app

I cut it because now we could not see the text properly. The navigation is achieved thanks to the browser router as I already said, which is implemented in the following way:

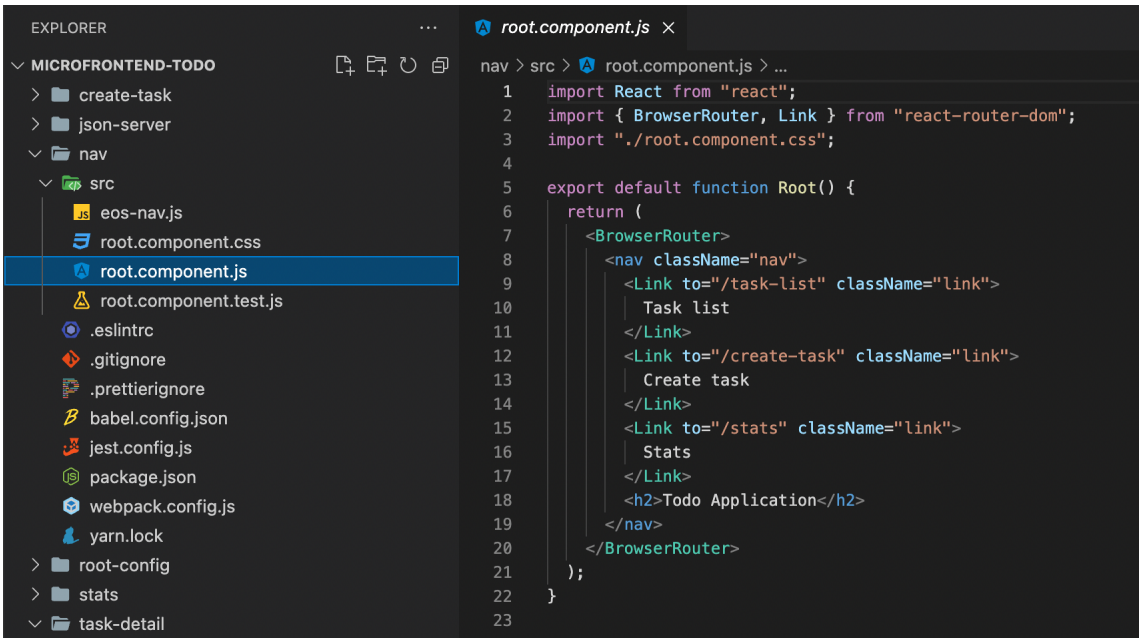

Figure 7.2.1.2. Todo Micro Frontend Application navigation code

- *TASK LIST*

This application displays the list of all the tasks that are currently retrieved from the JSON server which I will explain later, and gives you the ability to check whether they are completed or not. It looks like this:

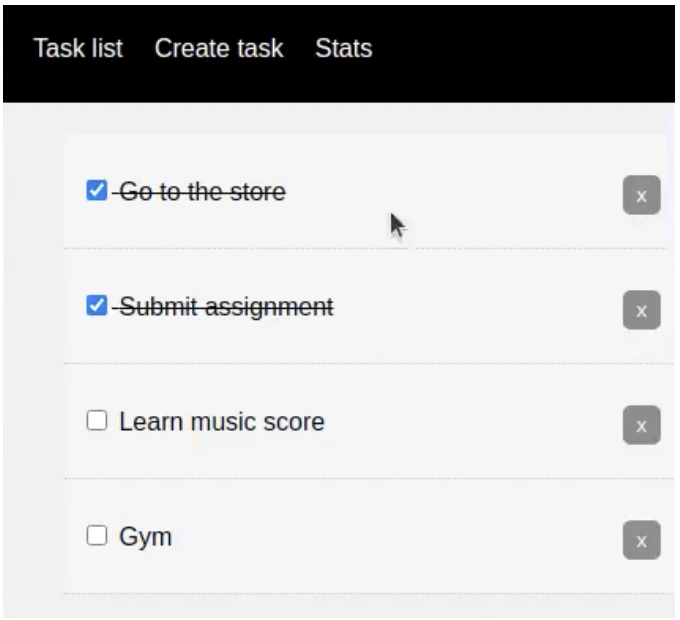

Figure 7.2.1.3. Todo Micro Frontend Application task list app

The implementation of this is quite simple. When we mount this application, it automatically does an API request to get the list of tasks available in our backend.

```
componentDidMount() {
  axios
    .get("http://localhost:3000/tasks")
    .then((res) => this.setState({ todos: res.data }));
}
```

Figure 7.2.1.4. Todo Micro Frontend Application task request

Thereafter, we have a specific function that executes and updates the task when we check when it is done, and another function that deletes it from the database when we click on the delete button. Also, when some of those functions execute, it throws an event so the other applications will be informed, and will react to it if it's the case. This will be better explained in the communication section coming in a few lines below.

- *TASK DETAIL*

Task detail is a similar application, but in this case, it is loaded when any task in the Task List application is selected. When we click on a task, we create an event, we send the id of the task from the task list application, and the task detail application handles that event and renders the detail of that task which is calculated thanks to the id:

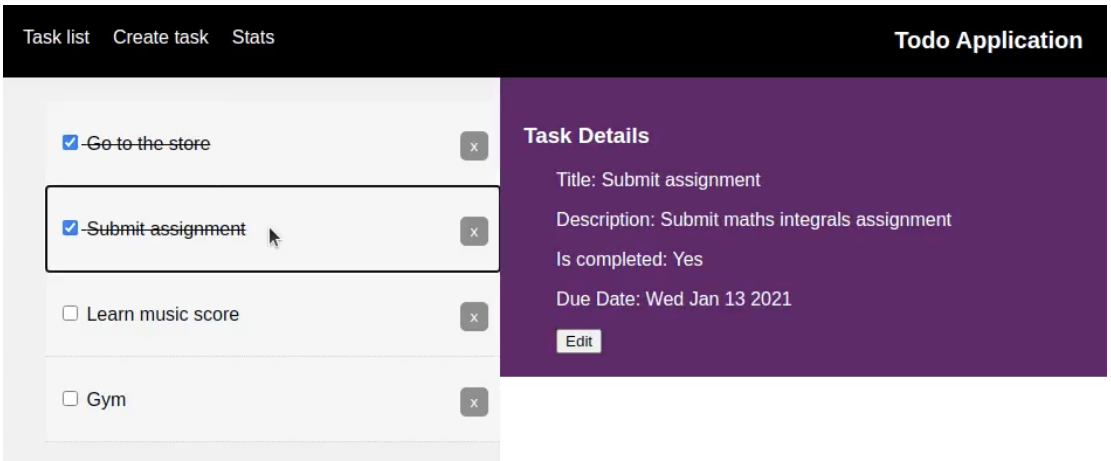

Figure 7.2.1.5. Todo Micro Frontend Application task detail app

Here we can see things such as the task title, description, if it is completed or not, and the due date.

- *CREATE TASK*

This is the last application that we can see in the actual web application. It aims to create or edit a current existing task. It can be called through the navigation when we click on it, or also when we click the edit button, it opens the application with the already pre-filled information of the task:

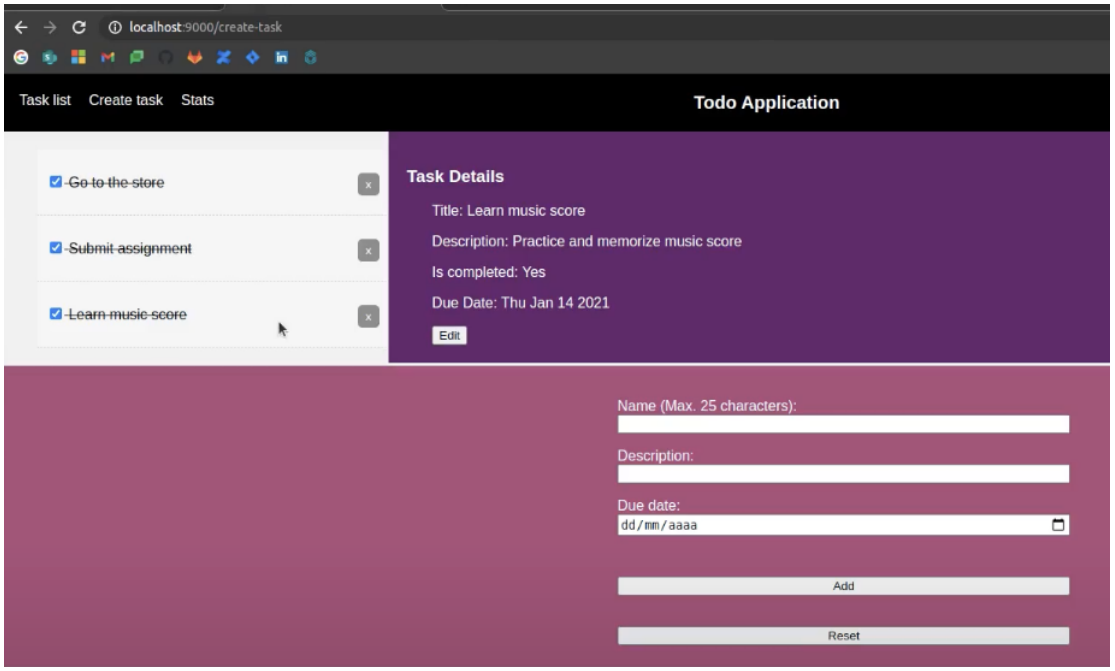

Figure 7.2.1.6. Todo Micro Frontend Application create an app

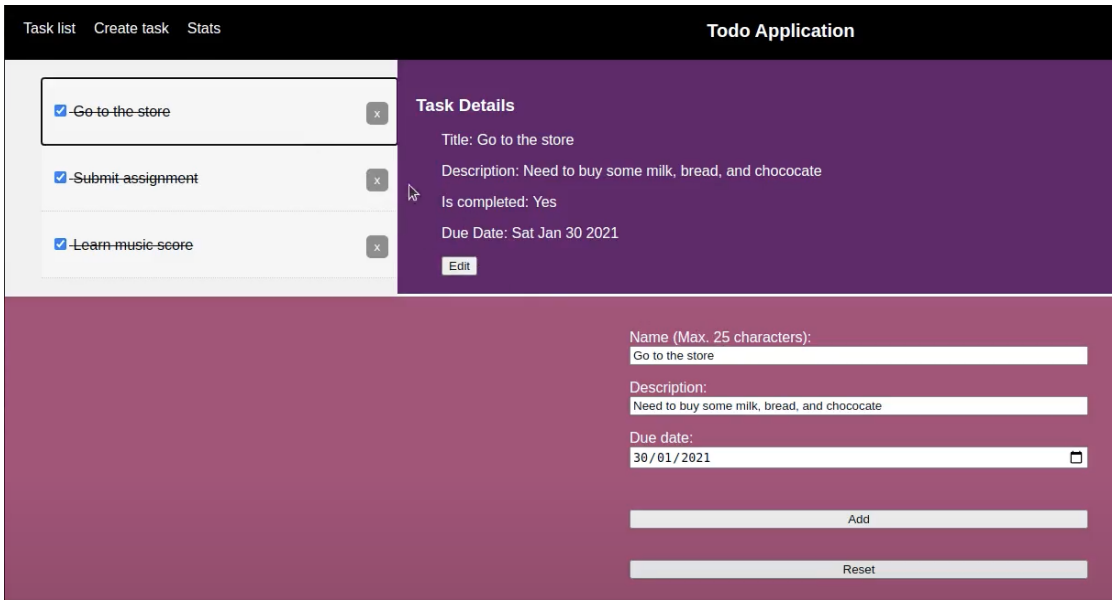

Figure 7.2.1.7. Todo Micro Frontend Application edit app

- *STATS*

This application's purpose was to show some statistics about the tasks, but due to lack of time and not necessary information, it was not developed. This is why I decided to write some errors in the application so that it will not execute correctly. I wanted to do this to prove the fact that even if one of the applications has errors and does not run correctly, it does not mean that the entire application will fail. This is thanks to the micro-frontend architecture. Even if one of the many applications is failing or not working, it will not affect the entire application because all of them are running completely independently.

- *ROOT CONFIG*

This is the core application. The most important one. It is in charge of registering all the micro applications, mounting and unmounting there, and also mapping them.

At this point, we have generated our 5 apps. So, how can we put them all together? As I already said before, one of the container app's primary responsibilities is to coordinate when each app is "active" or not. In other words, it handles when each app should be shown or hidden.

To help the container app understand when each application should be shown, we provide it with what is called "activity functions" Each app has an activation function that simply returns a boolean, true or false, for whether or not the app is currently active. To achieve that, we need the "activity-functions-js" file inside the "root-config" directory, and I wrote this:

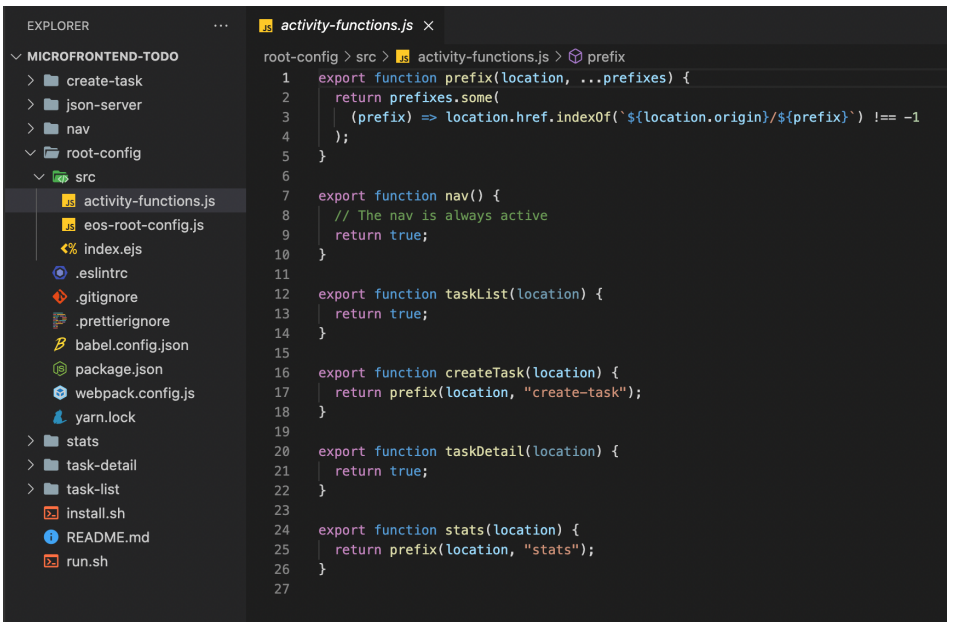

Figure 7.2.1.8. Todo Micro Frontend Application root activity functions

What we are doing here is saying that for example, the task list and the task detail windows are always going to be displayed as the navigation one. However, the stats application and the create task application will only be displayed when we have their prefix on the URL.

Next, it is needed to register all the micro-frontend apps with single-spa. To do that, we need to use the "registerApplication" function. This function accepts a minimum of three arguments: the app name, a method to load the app, and an activity function to determine when the app is active.

Inside the "root-config" directory, in the "root-config.js" file, we'll add the following code to register the applications:

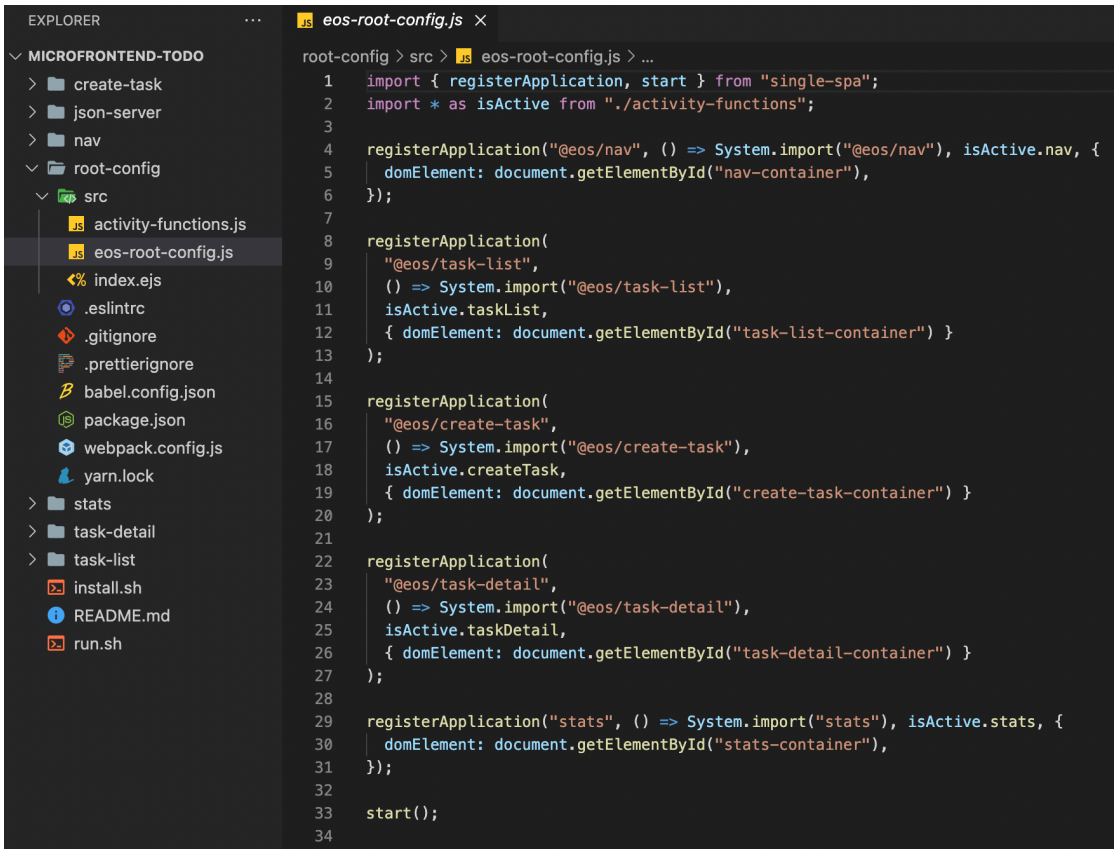

Figure 7.2.1.9. Todo Micro Frontend Application root register application

Now that we have set up the activity functions and registered our apps, the last step before we can get this running locally is to update the local import map inside the index.ejs file in the same directory. It is needed to add the following code inside the head tag to specify where each app can be found when running locally:

```
<% if (isLocal) { %>
<script type="systemjs-importmap">
  {
    "imports": {
      "react": "https://cdn.jsdelivr.net/npm/react@16.13.1/umd/react.development.js",
      "react-dom": "https://cdn.jsdelivr.net/npm/react-dom@16.13.1/umd/react-dom.development.js",
      "@eos/root-config": "http://localhost:9000/eos-root-config.js",
      "@eos/nav": "http://localhost:9001/eos-nav.js",
      "@eos/task-list": "http://localhost:9002/eos-task-list.js",
      "@eos/create-task": "http://localhost:9003/eos-create-task.js",
      "@eos/task-detail": "http://localhost:9004/eos-task-detail.js",
      "stats": "http://localhost:9005"
    }
  }
</script>
<% } %>
```

Figure 7.2.1.10. Todo Micro Frontend Application root index

Each app contains its startup script, which means that each app will be running locally on its development server during local development. As you can see, our navigation app is on port 9001, our task list app is on port 9002, the create task app on port 9003, the task detail on port 9004, and the stats app on port 9005. The root application runs on port 9000 by default.

7.2.2. API

In a regular enterprise application, we work with many teams and third-party APIs. Imagine we have to call a third-party restful web service that will get you JSON data to work on. We are on a tight schedule, so we can't wait for them to finish their work and then start you. JSON-server is a mockup Rest Web service in place to get demo data, and I have used it in my application for this purpose. The db.json file is the one I am working with, and initially it loads the following values:

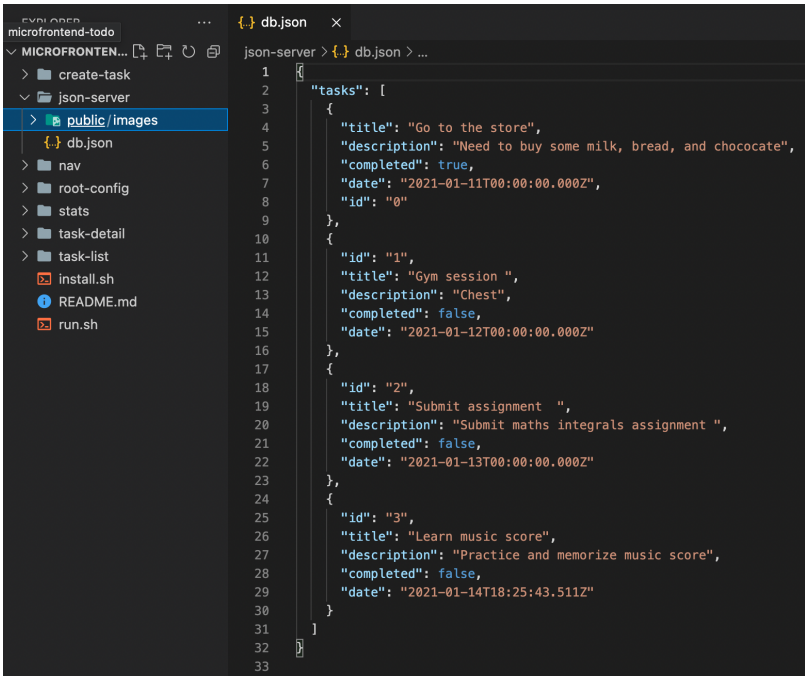

Figure 7.2.2.1. Todo Micro Frontend Application backend

The only command to execute is “json-server --watch db.json”, and we obtain the following window:

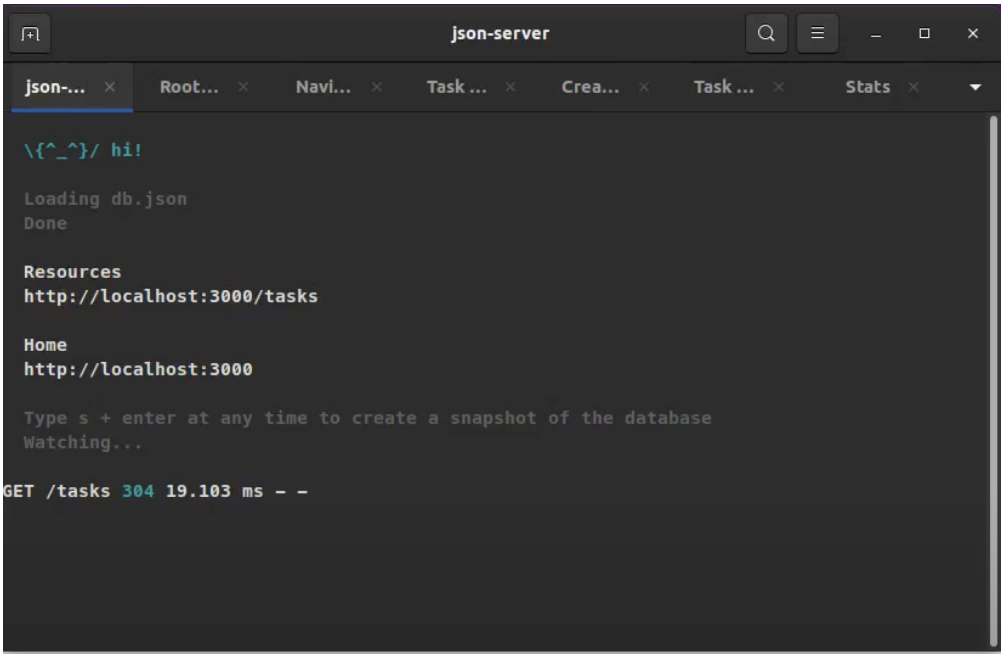

Figure 7.2.2.2. Todo Micro Frontend Application JSON-server execution

7.2.3. COMMUNICATION

Communication as I said before is achieved with window events. From one application we can throw an event with a specific name with parameters if we want to. Then, in whatever application in whatever part of the code, we can listen to this event, and whenever we receive something, we can execute the code we want. This looks in the following way:

```
render() {
  const { id, title, completed } = this.props.todo;
  return (
    <div
      role="button"
      tabIndex={-1}
      style={this.getStyle()}
      onClick={() => window.showDetail(id)}
    >
      <p>
        <input
          type="checkbox"
          defaultChecked={completed}
          onChange={this.props.markComplete.bind(this, id)}
        />{" "}
        {title}
        <button onClick={this.props.delTodo.bind(this, id)} style={btnStyle}>
          x
        </button>
      </p>
    </div>
  );
}
```

Figure 7.2.3.1. Todo Micro Frontend Application events communication

Let's see this code here. It is the code from a specific task in the task list application. When we click on one of them, it dispatches the "showDetail" function

with the id parameter. On the other side, we have this handler which executes whenever the function showDetail is executed in the task list application:

```
render() {
  window.showDetail = (id) => {
    axios
      .get(`http://localhost:3000/tasks/${id}`)
      .then((res) => this.setState({ task: res.data }));
  };
  window.addEventListener("editedTask", () => {
    axios
      .get(`http://localhost:3000/tasks/${this.state.task.id}`)
      .then((res) => this.setState({ task: res.data }));
  });
  return (
    <div className="container3">
      <h3>Task Details</h3>
      <div className="taskDetail">
        <React.Fragment>
          <TaskDetail task={this.state.task} />
        </React.Fragment>
      </div>
    </div>
  );
}
```

Figure 7.2.3.2. Todo Micro Frontend Application events communication

In this case, we are representing the case when a user clicks on a specific task on the task list application, and the task detail application receives this event and loads the information of the specific task.

7.2.4. EXECUTION

As there are many applications to run and to install packages in, I decided to create two scripts, one for installing all the dependencies from all the applications, and the other one to launch all the applications at once.

- install.sh

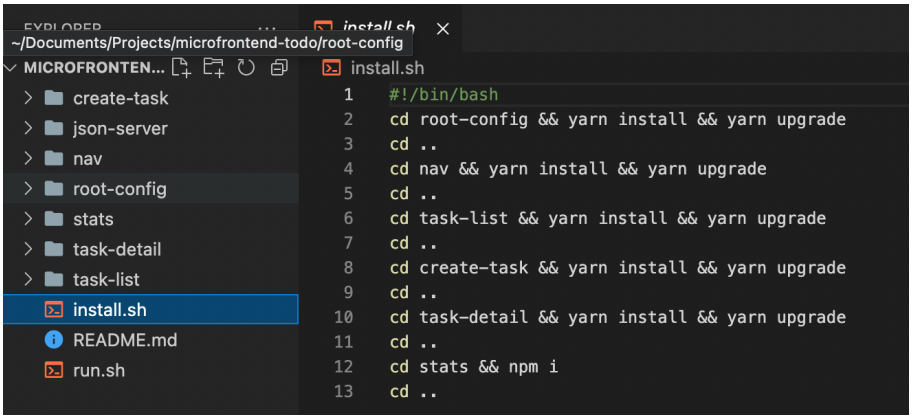

Figure 7.2.4.1. Todo Micro Frontend Application install script

- run.sh

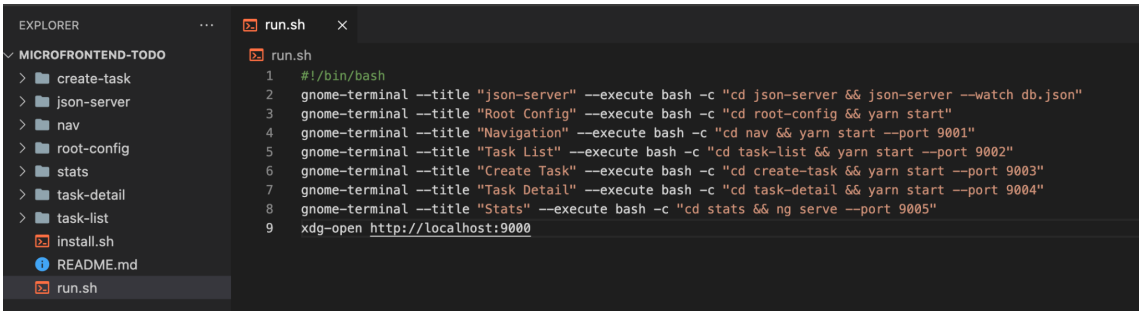

Figure 7.2.4.2. Todo Micro Frontend Application run script

In this case first of all we install all the required dependencies, and thereafter we open several tabs of a terminal, and we launch the json-server, all the applications in their correspondent port, and finally open the web page application. The terminal looks like this:

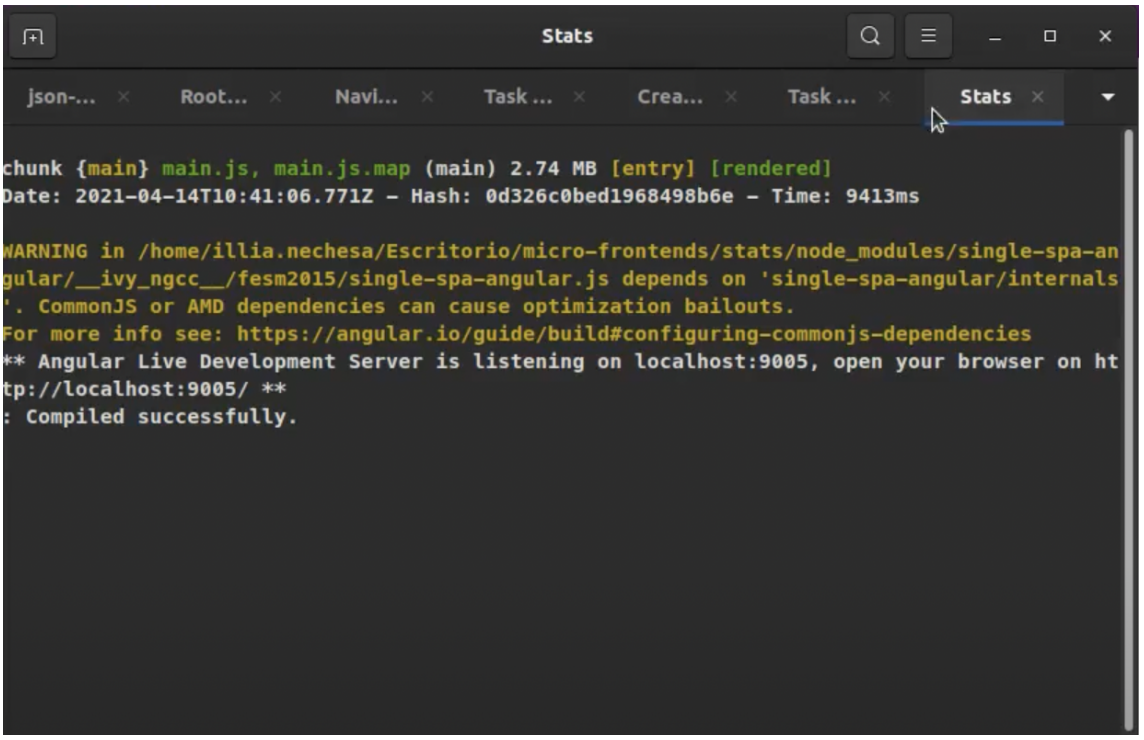

Figure 7.2.4.3. Todo Micro Frontend Application all applications running

Each one looks the same as this one, which means, it is executed independently on a separate tab.

The web page initially loads like this:

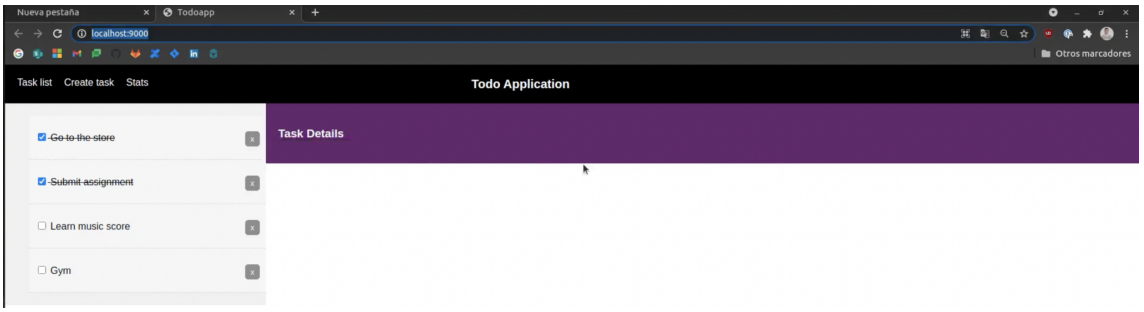

Figure 7.2.4.4. Todo Micro Frontend Application initial load

During the navigation, it may change and one screenshot with all the applications opened except the stats one:

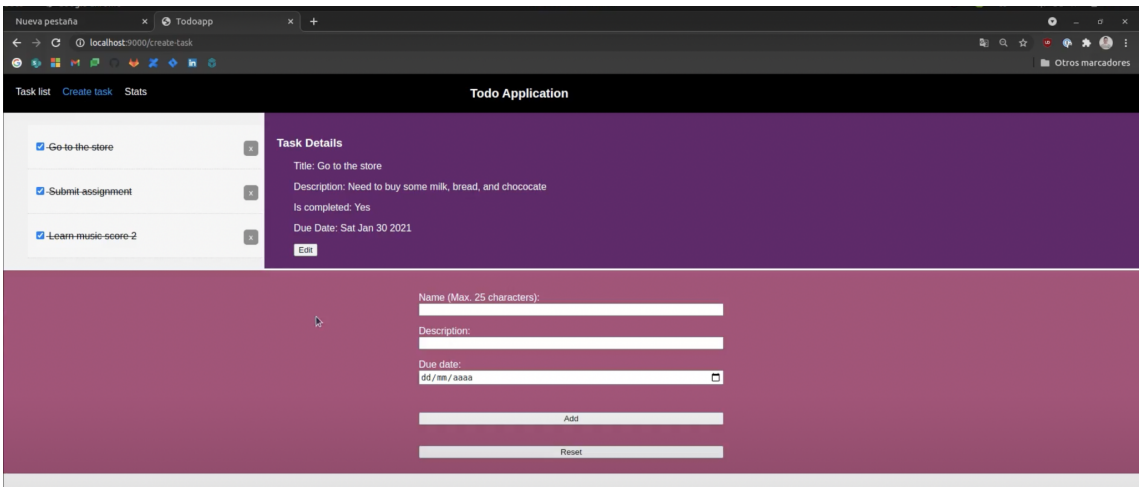

Figure 7.2.4.5. Todo Micro Frontend Application many components loaded

This is the full implementation of the todo application based on a micro frontend framework, with a custom method of communication.

CHAPTER 8. CONCLUSIONS

8.1. GENERAL CONCLUSIONS

The development of this thesis has been through many phases. Initially, the company where I started to develop it had one specific purpose. Do a proof of concept using Micro Frontend architecture and see its feasibility in our company projects.

This development would be composed of many parts. One of them would be studying different frontend frameworks to have the ability to develop the proof of concept with which we could obtain a response to our doubts. Thereafter, I should perform deep research about Micro-Frontends and the best ways to achieve them. There is no one best way to do that, however, there were many options to choose from, and after looking at the benefits and drawbacks of each one, I went for one specific framework which helps us to achieve this, Single-SPA.

But before starting developing the final application, I needed to improve my skills in some frontend frameworks such as Angular, React and VuejS, this is why I developed some small applications based on each of these frameworks. After that, I implemented the final application with the Single-SPA framework.

Some weeks later, I had a presentation at work where I showed the results of my studies and the actual web application which was created as a proof of concept. After a long and interesting discussion, we obtain some conclusions. My previous company was not fully technical, and there was a software development department composed of 3 developers including me. Usually, micro-frontends are used to create end-to-end applications, with a specific business model for each one, so that independent teams can work on a fraction of the application without affecting other fractions. At least, for 3 developers, it does not make a lot of sense to implement Micro Frontend architecture just for that.

Nevertheless, the independent deployment of micro applications, the abstractions of features into small applications, all the lifecycle from testing to deployment are completely different and much more efficient than the monolithic one. So taking these things into account and all the benefits from the micro-frontends, we have concluded.

The three of us agreed that micro-frontends are the future of the frontend, the same as happened with microservices for the backend, and we should work looking forward to it. However, our team did very little to do this huge refactoring, and because of all the projects coming up, we would not have time to work on that. Of course, it is not a thing of two or three weeks. It may take months or even years depending on how many people are working on that or the application size.

Another important thing we agreed with was the fact that micro-frontend is an emerging topic and is still not very welcomed by the frontend community. A lot of companies are achieving this architecture with custom methods, others are using already created dedicated frameworks for that. Many companies are adapting Micro Frontends because other big ones are doing the same, such as Zalando, DAZN, Spotify… However, there is still not any complete guide of how to achieve this architecture entirely and not have any issues because of using any framework that can block you during the development because they do not give you complete freedom over the development.

Due to this, my previous company decided to leave this project apart from the moment, and wait a little bit more until this topic is more developed, and more frameworks or tools appear giving more freedom for the development of a web application based on Micro-Frontends, and the team size grows so they could go ahead with this big refactoring.

Now, from my perspective, I completely agreed with their decision. In addition, I also felt during the development of the application with the Single-SPA framework that you cannot do all that you want. One of the main drawbacks is for example that all the projects must be Javascript-based ones. Apart from that, their command-line interface gave me many problems, and it looked at certain times more like a beta framework than a complete usable framework. However, this is completely normal and the job they are doing is just impressive. Updates are being released each month nearly, and taking into account they are building this themselves and how new the Micro Frontend concept is, they are doing a great job.

Finally, as I already said, most of those frameworks are always in constant development and may have bugs or block conditions for our purposes. This is why famous companies are using custom orchestrators to achieve this architecture and adapting it to their own needs, which I think is the best solution. However you need a specific team for this, that could work on this topic for a huge amount of time, so it is not suitable for small companies of for example five developers.

8.2. ANALYSIS OF THE PROPOSED OBJECTIVES

SPECIFIC OBJECTIVES	ACHIEVEMENT LEVEL	OBSERVATIONS
Analysis of the current web application architectures such N-Tier architecture or model view controller paradigm, among others for understanding the main components that support those applications.	★★★★	Studied different web application architectures which I already was familiar with.
Study the architecture designs used for the development of web applications, such as the Single Page Applications one, the Microservices architecture, and the Serverless one, among others.	★★★★★	Studies different approaches for web architecture designs. Fortunately I was already familiar with these architectures.
Research about the methodologies generally used for web application development. This allows us to extract patterns for a novel development methodology.	★★★★★	Thanks to the fact that I had worked in more than one company, I was familiar with some methodologies.
Investigation of the frameworks that could be used for the project development of web applications, specially oriented to front-end based on Javascript language	★★★★★	Researched many frameworks that could be used for web development, some of them which I have already worked on.

Research frontend technologies based on Javascript, and analyze them by comparing them with each other.	★★★★★	Improved my skills in Angular, and studied new frameworks such as ReactJS and Vue.
Explore the methodologies that developers used for web development to identify the main issues or aspects to be considered in our project.	★★★★	As I have always worked as a frontend developer, I considered many facts during the development of this project.
Investigate the approaches proposed in the web community related to support the execution of web micro-applications. A comparison between these approaches allows us to understand the main concerns to be considered for this project.	★★★	A deep research was done, but there was not a concrete solution or best option to the problem.
Analysis of the Micro-Frontend architecture, including reasons to adopt it, its benefits and issues, and different kinds of approaches to it.	★★★★★	Read many articles and a book about this architecture. Have discussed it in some companies.
Test a pilot of the novel framework to a web development team to extract an opinion about the improvements that a new approach could have and obtain feedback from developer users.	★★★★	I performed a small test in my company, and we decided not to go on since more people is needed and there is still not that much documentation according to migration purposes.

Development of a POC implementing Micro-Frontend architecture, including the requirement specification, the analysis, and the design.	★★★★★	A proof of concept was done and I am really happy with the results.

8.3. FUTURE LINES

As a continuation of this project, it will be interesting to go deeper into the Micro-Frontend approach. What I exactly mean is to try to migrate a full monolithic application into a Micro-Frontend architecture. This may take months or even years for a big team to complete this task for a relatively big application. Only in this way, this architecture can be fully evaluated.

I have done a small proof of concept in which we can see the way this architecture works, how it behaves, how it communicates… But what about big applications with plenty of functionalities? What if we want to share big libraries between these components? How do we share logic or code or components between applications? Let's imagine we have the same feature in two different applications. Why implement it twice? Can we do it only on one separate project and use it on others? Maybe, but this must be fully implemented and tested.

BIBLIOGRAPHY

[1] Pisuwala, U., *"Fundamentos de la arquitectura de aplicaciones web"*, Web page: https://www.peerbits.com/blog/web-application-architecture.html, Last access: 14/12/2020.

[2] Decide, "*Arquitectura de microservicios: qué es, ventajas y desventajas*", Web page: https://decidesoluciones.es/arquitectura-de-microservicios/ , Last access: 14/12/2020.

[3] Serverless Stack, "*¿Qué es serverless?*", Web page: https://serverless-stack.com/chapters/es/what-is-serverless.html , Last access: 14/12/2020.

[4] Opperman K. "*High level Micro-ui Architecture"*. Web page: https://link.medium.com/p5NXY3Vbibb , Last access: 07/12/2020.

[5] Geers M., "*Micro Frontends*", Web page: https://micro-frontends.org/ , Last access: 09/12/2020.

[7] Severi Peltonen, Luca Mezzalira, Davide Taibi. "*Motivations, Benefits, and Issues for Adopting Micro-Frontends: A multivocal Literature Review*", Web page: https://arxiv.org/pdf/2007.00293.pdf , Last access: 09/12/2020.

[8] Johnson B., "*Exploring micro-frontends*", Web page: https://medium.com/@benjamin.d.johnson/exploring-micro-frontends-87a120b3f71c , Last access: 15/12/2020.

[9] Flavio C., "*A developer's introduction to React*", Web page: https://jaxenter.com/introduction-react-147054.html , Last access: 30/12/2020.

[10] Bachina B., "*6 different ways to implement Micro-Frontends with Angular*", Web page: https://medium.com/bb-tutorials-and-thoughts/6-different-ways-to-implement-micro-frontends-with-angular-298bc8d79f6b , Last access: 04/01/2021.

[11] Bachina B., "*How to implement Micro-Frontend Architecture with React* ", Web page: https://medium.com/bb-tutorials-and-thoughts/how-to-implement-micro-frontend-architecture-with-react-5ab172a0fec7 , Last access: 04/01/2021.

[12] Mezzalira L., "*Micro-frontends decisions framework*", Web page: https://medium.com/@lucamezzalira/micro-frontends-decisions-framework-ebcd22256513 , Last access: 04/01/2021.

[13] Neary A., "*Rearchitecting Airbnb's Frontend*", Web page: https://medium.com/airbnb-engineering/rearchitecting-airbnbs-frontend-5e213efc24d2 , Last access: 04/01/2021.

[14] Müller K. R., "*Easy Micro-Frontends*" , Web page: https://itnext.io/prototyping-micro-frontends-d03397c5f770 , Last access: 04/01/2021.

[15] Breux G., "*Client-side vs Server-side vs Pre-rendering for Web Apps*", Web page: https://www.toptal.com/front-end/client-side-vs-server-side-pre-rendering Last access: 05/01/2021.

[16] Mezzalira L., "*Micro-frontends, the future of Frontend architecture*", Web page:

https://medium.com/dazn-tech/micro-frontends-the-future-of-frontend-architectures-5867ceded39a , Last access: 05/01/2021.

[17] Mezzalira L., "*Adopting a Micro-frontends architecture*", Web page: https://medium.com/dazn-tech/adopting-a-micro-frontends-architecture-e283e6a3c4f3 , Last access: 05/01/2021.

[18] Nagy G., "*Introduction to micro frontends architecture*", Web page: https://levelup.gitconnected.com/brief-introduction-to-micro-frontends-architecture-ec928c587727 , Last access: 05/01/2021.

[19] Hawkins, T., "*How to Develop and Deploy Micro-Frontends with Single-SPA*." FreeCodeCamp.Org. https://www.freecodecamp.org/news/developing-and-deploying-micro-frontends-with-single-spa/ , Last access: 25/05/2021