POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

User Interface Development of a Modern Web Application

Supervisors

Prof. Luca ARDITO

Candidate

Marzieh SOMI

December 2021

Summary

This thesis aims to explore the advantages of using a frontend library 'React.js' that used to create a user-friendly and modern web application. React is an open-source JS library designed based on components that are usually small and reusable building blocks. The primary development approach employed for this project was the Agile methodology, chosen for its adaptability to the project's evolving requirements. This approach helps teams deliver value to their customers more quickly and efficiently by focusing on collaboration, adaptability, and continuous improvement.

The thesis examines how react.js can reduce the effort needed to develop and maintain applications. The development focused on react integration with External Libraries. During the development we experienced utilization of Tailwind CSS framework and libraries such as Formik, Redux, Axios and React-chartjs2.
External libraries often provide pre-built components, utilities, and tools that can enhance the functionality of react application and save a significant amount of development time and costs avoid writing repetitive code. Furthermore, libraries tend to receive timely security updates and patches, enhancing application security.
Tailwind allows for rapid development of modern websites by enabling us to write CSS directly in markup. It is very customizable, and empower developers to create unique, efficient, and consistent UI while adapting to specific project. Installing tailwind, it generates two files: 'postcss.config.js' which is a single export JavaScript object that will instruct to run tailwind first. Utilizing 'PostCSS' significantly enhances the build speed of our projects, leading to improved performance. Another file is 'tailwind.config.js'. This configuration file allow customize and configure various aspects of the tailwind. Tailwind' directives (@tailwind base; @tailwind components; @tailwind utilities) should be added at main css file index.css for each of Tailwind's layers. This is like importing tailwind styles to be recognized as real css syntax and then import this css file to 'index.js'.
Formik is a third-party library used to implementing and manage forms in application together with yup. This library helps with taking user input, form submission and validation. We implemented components, creating a skeleton for each type of input: TextArea, SelectField, TextFiel, this makes them easily reusable at different

sections of our application such as developing report pages and landing page. These components make use of 'useField' hook, provided by formik. we get advantages of 'Yup' which is a schema-builder that allows us to define validation rules and schemas for data validation. For using yup we should first define object schema and its validation and then define all criteria for this object.

Redux is used to manage the application's state in a single place and is suitable when dealing with large amounts of application state. It consists of three key components:

- Actions, which are events to communicate with the redux store and trigger state updates.
- Store, there is a central store that holds the application state and wach component can access the it without having to send down props from one component to another.
- Reducers, they are pure function that interpret the action. They accept the initial state and the action type. They take an action and the previous state of the application and returns the new state, sent back the updated state to the view components of the react.

The store is created using 'createStore' function. 'applyMiddleware' function is employed to incorporate the 'thunk middleware' for handling asynchronous actions within Redux. Additionally, integration with the Redux DevTools extension is achieved through the 'compose' function, allowing for debugging within the browser's Redux DevTools extension, added as an extension in the browser. To access data stored in Redux, the 'useSelector' hook from Redux is utilized, which simplifies the process of accessing and using data from the Redux store in functional components, making it a crucial tool when working with Redux in react applications.

The client-side application for receive and update data from the database use API. Communication between the client and the api is done through Http request which can be achieved using 'Fetch' or 'Axios'. We use Axios for its ease to use and extensive browser support due to its use of the Bluebird Promise library, providing better compatibility. It is a JS library that handle asynchronous operations and manage the resulting values or errors. It can intercept and transform request and response data, as well as cancel requests or automatic transforms for Json data, so unlike FetchAPI we don't need converting our request body to a Json. It is implemented by creating two components:

- 'axios.js' which contains an axios instance configured for making HTTP requests to an API. Firstly, it checks if the application is running on the localhost, saved result in a constant. Then retrieves the 'token' from local

storage if it exists and parses it as Json. Then create an instance of Axios with 'axios.create()' function. This instance can be used to make Http requests to the specified 'baseURL' which is including the Authorization header with the token value. The code examines if token is exists, the authorization header will be set as Bearer token, otherwise it set an empty string.

- 'requests.js' where an object defied that holds various api endpoint URLs.

The development process was involved implementing several components, each assigned to different team members using a collaborative approach. I participated in the development of 'Theme Selector', 'Report pages', 'Landing page' and 'Impostazioni'.

Theme Selector, allowing users to choose their desired color palette from a range of pre-created themes. This feature prioritizes user preferences, enhancing the design of a modern web application. To implementation of this part, two files is created; 'theme.js' where we defined all color palettes constants and the 'utils.js' which is contains an exportable function 'applyTheme'.

The constants defined at them.js imported into the 'ThemeSelector' component, where dropdown list is render. To developing this component, we get advantages of headlessUI library which is designed specially to work with Tailwind CSS and provided pre-build components. This component imported necessary dependencies from react, along with color palettes defined in theme.js component and the 'applyTheme' function from'utils.js'. By importing these constants into 'App.js' we make the theme selector drop-down available on other pages of application as well. At App component, the useEffect hook is used to set the theme of the application based on a value stored in the browser's local storage. This hook runs once, when the component is mounted and sets the initial theme based on the stored preference in local storage.

Reports play a crucial role in modern web applications. They provide data transparency, which is particularly important for businesses and organizations that require regulatory compliance. We used 'react-chartjs-2' because of its easy-to-use. 'chart.js' also installed which is like a core library used for any kind of application. Data assigned to reports are fake because of lack of real data. For the initial phase we implemented 3 report pages implemented different charts such as Bar, Pie, Line. All Report pages contain three common components: a 'Date Picker' for selecting a date range for visualizing reports, and two drop down selector 'Gender' and 'Sector' for filtering data based on these two options. These components are written individually and imported to main components of each report page. HeadlessUI is integrated to development of reports. hide/show specific charts was implemented managing by a button. This functionality is particularly useful for data privacy, allowing users to hide specific charts when in the presence of some group of people. To implementing this feature, charts are categorized into two principal components. A component contains charts that can be hidden and another component which is

include all other charts to be displayed on the page.
Using “useState” hook it manages charts visibility, initialized its value to 'false' *const [showAll, setShowAll]= React.useState(false)* By clicking on a button which is located at the top of the page the visibility of Report will be change.

Another required feature to developing web version was the implementation of a Landing Page. This feature allows users of application to create static html page within texts, images, and information for publishing purpose. This kind of pages usually aims to attract new clients, improve user experience, and achieve better results for business. Creating landing pages have an important impact on the usability of the application. To develop a landing page, the first step is to define the goals and know the target audience. For this application the goal was giving the possibility to creating advertisements for specific services or products to Nuage users. They can represent special promotion and events without needing go thorough external landing page builder applications. It allows users to create their own pages without requiring advanced computer skills. We've implemented this functionality within two components, imported them into the main component where the 'useState' hook manage the visibility of components within an onClick function assigned to a button.

- LandingPageList: This component displays a list of all the created landing pages, where they are archived for a specified period. is developed using the Formik. Users have the option to edit, delete, or copy the page's link and can access the main "Creator" page (AddLandingPage) through a Plus button form this page.
- AddLandingPage: this component is where the development phase for creating individual landing pages takes place. The page consist of three sections: a Header with navigation buttons for going back to the previous page, a Form completion area that forms the core of the page, and a Preview section that updates in real-time as the user fills out into the forms. In the form section, a limited number of blocks are provided for creating a landing page, with the possibility to leave some of the blocks unused. The Form completion and Preview are developed at separated components and imported to a main component. To developing each of them various dependencies are imported, some event handler functions, such as 'handleClick' and states are defined to manage various aspects of the landing page.

An object ‘staticMarkup’ is created that contains key-value pairs and include a code generating a static HTML markup. The values are derived from the fields object that receives as a Prop and other variables representing data that to be sending in the Post request. An asynchronous function handles the form submission. It constructs a request object, specifying the HTTP method as "POST," setting the

"Content-Type" header to "application/json," and include the staticMarkup object as the request body. The code then makes an HTTP POST request using the api.post method (provided by Axios). when it receives the response, it checks the HTTP status code. If the status code is 201 or 200, it sets the link state variable with the value of the "Link" property from the response data. If the status code is different, it logs an error message.

Overall, the application's development has been significantly aided by the integration of libraries like Formik and the utilization of HeadlessUI components. These tools have not only saved the team valuable time but also contributed to the creation of a visually appealing user interface.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor professor Luca Ardito for the trust and the opportunity I was given, and for his support during the whole time.
I would also like to thank X-New for providing me with the opportunity to work with their development team. I extend my appreciation to my colleagues, for their guidance and support. I would also like to express my gratitude to Alessandro Abbate, The company supervisor, for his trust throughout this journey.

A very special thank goes to my fiancé Walter for all the encouragement and lovely support during the whole period of my study.
Last but not least, I would like to thank my family who believed in me and giving their unconditional support during the whole period of my study.

Table of Contents

List of Figures

Acronyms

SPA
Single Page Application

MPA
Multi Page Application

REST
REpresentational State Transfer

JSON
JavaScript Object Notation

SOAP
Simple Object Access Protocol

NPM
Node Package Manager

CLI
Command-Line Interface

DOM
Document Object Model

JSX
JavaScript XML

DRY
Don't Repeat Yourself

Chapter 1

Introduction

1.1 Motivation

From e-commerce platforms to social media networks and productivity tools, web applications are transforming the way we interact with technology. As the demand for feature-rich and user-friendly web applications continues to grow, there is a pressing need for skilled professionals who can develop modern and intuitive user interfaces.
The user interface (UI) of a web application plays a crucial role in determining its success. A well-designed and responsive UI not only enhances the user experience but also fosters user engagement, satisfaction, and ultimately, business success. However, designing and developing a modern and effective UI for web applications is a complex task that requires a deep understanding of user behavior, interaction design principles, and emerging technologies.
This thesis aims to explore the advantages of using a modern frontend library 'React.js' that can be used to create a user-friendly web application. React.js was initially released in 2013, created by *"Jordan Walke"*, Facebook's developer and now it has becomes one of the most popular JavaScript libraries. Many big companies apply React.js such as Facebook, Instagram, Firefox, PayPal, Netflix, etc.
This thesis studies the integration of React.js with the Tailwind CSS framework along with the utilization of libraries such as *Formik, Redux, Axios and React-chartjs2*. I briefly explained the theoretical background and implementation of them at the corresponding sections.
Modern frameworks and libraries such as React assist web engineers in building sophisticated applications using high-quality solutions. Using react we can create reusable UI components which help developer to save time, didn't repeat to writing code. It will efficiently update and render just the right components when the data changes[1]. Although UI design and definition of technologies to use was

changed after 3 months from the start of my collaboration with the company and the definition version of application is started, but it was a very useful period of my study to work with a company and being familiar with the process of web development to implementation of an application.

1.2 Thesis objective

Current work took place at Xnew srl, which is a company located at Turin, Italy. The subject of thesis was developing the interface part of a management software web application. The goal was the implementation of a modern web applications following the Agile methodology with usage of innovative standards available at the time in the technological and IT fields and leveraging various libraries and work with APIs, utilizing the RestAPI for seamless integration and functionality.
The main method of development applied at this project was Agile methodology as it is suitable for the changing requirements of the project. It is an iterative approach to project management and software development that helps teams deliver value to their customers faster. At agile methodology work is delivered in small and consumable. There are possibility to evaluate continuously the requirements and results, and team can respond quickly to changes[2]. The technologies used to development of this application are: html, css, JavaScript, React, Redux and Tailwind css framework. This thesis elaborates on the development process of creating an application, which was an extension of an existing desktop version. It also provides explanations of various theoretical concepts.

1.3 Thesis structure

The rest of thesis consists of 5 chapters:
Chapter 2 , provides an overview of some theorical concepts of web application, include: Single Page Application (SPA), differences between SPA and Traditional webpages, Dynamic versus static code, Webpack, Fetching Data and RestAPI. The second part briefly explains the libraries and framework used for modern web app and some differences between them. furthermore it explains about the React Library and some of its principal concepts. **Chapter 3**, is dedicated to explaining about the development tools and technologies used. At the third section of this chapter, there are explanation about external libraries implementation used at this application. **Chapter 4**, delves into the implementation phase, commencing with an introduction to the application. It encompasses an exploration of the user interface, followed by a demonstration of the application structure, including its

files and folders created during the development phase. The remainder of this chapter is dedicated to explaining the development phase of the pages which is done by collaborating with other members of the development team. These tasks include the Theme Selector, Reports, Landing Page, and Setting Pages. **Chapter 5**, concludes the thesis work by summarizing the outcomes. It also provides a brief overview of upcoming improvements and future developments that were studied and planned for implementation in the next phase.

Chapter 2

Overview Of Web Concepts

2.1 Theoretical Concepts of Web Application

This chapter is dedicated to some theoretical concept, which is divided at two categories. The following section provides a brief explanation of some Web applications concepts and technologies used at this application to gives a better comprehension of the context of this working thesis.

2.1.1 Single Page Application

A single page application is the implementation of web app that loads just a single page (web document) and then via a JavaScript APIs such as: *XMLHttpRequest* and *Fetch*, updates the body content of that single document. This feature allows users to use a website without loading whole new pages from the server [3].
Html, Css and JavaScrip are building blocks for SPA, which will provide a fluid and responsive web page. The SPA web applications or websites interact with the user, rewriting the current page dynamically. The most popular SPA frameworks are: *'React', 'Angular', 'Vue JS', 'Ember JS', 'Backbone'.*
At this project, we got benefits of React which is one of the most popular SPA framework to developing our web application. Lots of popular applications use this technology, such as: Google Maps, Airbnb, Netflix, Paypal and many more.

Single Page Application vs Multi Page Application

Lets see some differences between SPA and traditional web page to understand why SPA goes so popular. Multi-page application (MPA), consists of multiple interconnected pages, each one represents a separate HTML document. Each page has its own URL that used by the client to request that page and each request gives a new version of whole page. This means for navigating from one page to

another, the entire page loads, the server re-renders a full page and sends it to our browser. As shown in Fig 2.1, all the required components of the web page are loaded when initial request.

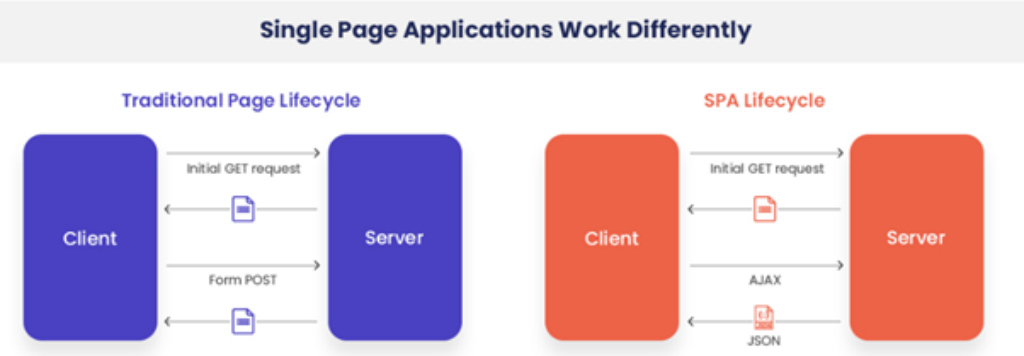


Figure 2.1: Single page application VS Multiple page application

In SPA, page components are updated without reloading entire page. After browser makes the first request to the server, the app will load with all its relevant assets which can takes some time, then for all subsequent calls an API is called to fetch new data. The data usually return in JSON format, the browser receives this data and updates the app view. In this way, the page never reloads, the application stays in the same page and there will be a single URL [4].
Both of these two architectures have benefits and disadvantages. Following are listed some advantages and disadvantages of SPA.

SPA Advantages:

- Quick load full page regarding, as it only load a page at the first request.
- consume less bandwidth since they load web pages and main resources *(html,css,js)* just once.
- Caching works better, even with a poor internet connection with SPAs, because it request data from server one time.

SPA Disadvantages:

- No good performance regarding to SEO; one of the metrics search engine use is the *number of pages* of a website and SPAs load only a single page.

- It require a lot of web browser resources since the browser do most of tasks for the SPAs. So often we need to use the latest browsers with support of some modern features.

2.1.2 Dynamic vs Static web page

At the **"Static"** web page after a server receives a request for a web page, sends the response to the client and do not any additional process and the pages will remain the same until will changes manually by someone. They usually written in programming languages such as *'JavaScript', 'html', 'css', etc.*

Figure 2.2: Static web application's approach

The **"Dynamic"** web page use where need changing frequently information, like: stock prices, weather information, etc. They are written in languages such as *'Ajax', 'Asp', 'Asp.Net', etc.* They update the display of a web page/app to show different page, generating new content as required. So, there are different content of pages for different visitors and it takes more time to load than the static web page. The meaning of dynamic at server-side and client-side is slightly different but related and both approaches usually work together.
'Client-side' js dynamically generates new content inside the browser on the client, example create a new html table, filling it with data requested from the server and display the table in a web page. Whereas *'Server-side'* code dynamically generates new content on the server, example pulling data from a database [5].
The Fig 2.3 demonstrates the differences of this approach with the static web application.

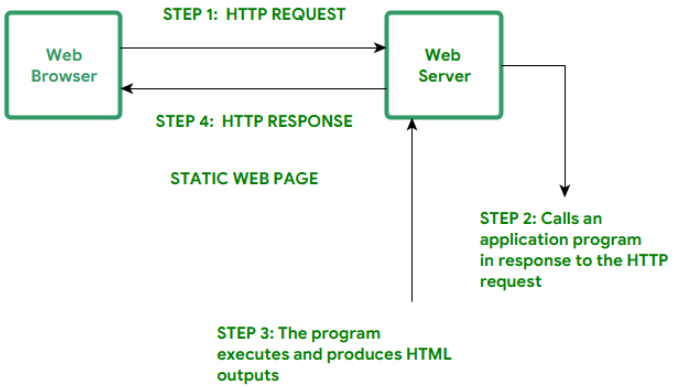


Figure 2.3: Dynamic web application's approach

2.1.3 RestAPI

Data is often stored in database on the server for permanent data storage.
For the client-side application to receive and update data from this database an API is used. We can think of it as a mediator between the users or clients and the resources or web services they want to get and uses the *http request* to communication with the client. It can be design in several ways, one of the most used are 'RESTful API'.
A RestAPI, is an api that follows the principles of REST architecture, allowing interaction with Restfull web services.
Api developers implement REST in various ways. When a client requests a resource through a Restfull api for a resource, the server responds by providing the current state of the resources in a standardized representation. This representation can be delivered in several formats like *Json , Html, XLT, python, php or plain text.* JSON is the most popular format, because it is readable by humans as well as machines. Additionally it provides Http methods such as: Get, Post, Put, and Delete.

RestAPI is faster, more lightweight and easier to use, compared to a prescribed protocol like *'Soap'* which has specific requirements such as *XML messaging, built-in security*, and *transaction compliance* that make it slower and heavier. Rest api offered increased scalability and is perfect for IoT and mobile app development [6]. Fig 2.4 illustrates how Rest architecture work[7]:

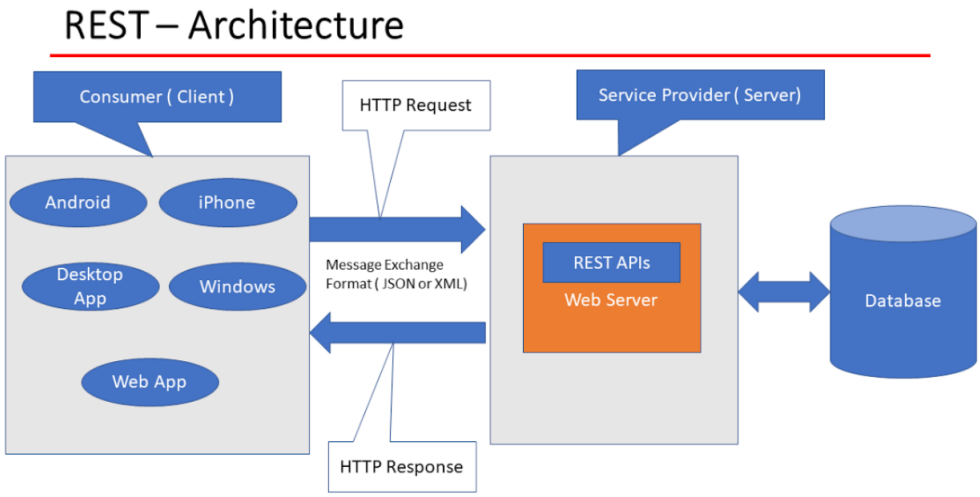


Figure 2.4: Rest Architecture

2.1.4 Agile methodology

Agile is an iterative approach to project management and software development that helps developer to deliver value faster to their customer. In such a way instead of delivering a big project, an agile team delivers work in small, but consumable, increments. So, they can evaluate continuously requirements, plans and results and the team have a natural mechanism for responding to change quickly[8].

2.1.5 Webpack

For modern JavaScript applications Webpack is a *static module bundler*. Its main purpose is to bundle js files for usage in a browser. Using webpack developing web application becomes faster and more efficient.
'Bundlers' in React are tool that can take all the files we have written and combine/bundle them into a single js file, while keeping track of their dependencies. They internally builds a dependency graph from one or more entry points of your appliction, combines into one or more bundles, modules your project needs, they figure out modules and libraries this entry point depends on. The entry point by default is: *./src/index.js*, but there can be different or multiple entry points, setting up an entry property in the webpack configuration.[9]

2.1.6 Node.js

It is a server-side JavaScript runtime environment. Java script can be used as a client-side either a server-side language. It allows developers to run JS code outside of a web browser, enabling them to build scalable and high-performance web applications. JS is single-threaded, and is not very suitable for multi-threaded tasks. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, this means it can handle multiple requests simultaneously without blocking the execution of other code.

NPM (Node Package Manager)

One of the key features of Node.js is its *package manager*. It is one of the largest software registry and command-line tool. NPM is commonly used in the React development for managing front-end dependencies regardless of the back-end technology we choose.
The CLI tool enables developers to download third-party open-source packages and apply them, which makes the development faster. It is used to interact with the npm registry and manage packages in a Node.js project. It provides commands to install and publish packages, update dependencies, manage versioning, and more.

Node Module

In the context of React, it refers to reusable block of code that is packaged and distributed through NPM (or Yarn) or available as a built-in module in the Node.js runtime environment. These modules are JS library that provide specific functionalities. They can be created by developers themselves or obtained from the NPM registry, which hosts a vast collection of open-source modules.

2.1.7 DOM element

The Document Object Model is a programming interface for web documents which represents the structure and content of a web page. A web page is a document that can be either displayed in the browser window or as the Html source. In both cases, it is the same document, but the dom representation allows it to be manipulated. As an object-oriented representation of the web page, it can be modified with a scripting language such as JavaScript. DOM was developed to deal with this issue of communicating between the JS and html as they can't communicate directly.

2.2 Libraries and frameworks

Frameworks and libraries provide solutions that speed up development process. There are different types of libraries and framework for all kinds of applications which help programmers to implement applications in front-end, back-end and more. At this section provide some differences between libraries and framework.

2.2.1 Compare between libraries and framework

Library and framework both give us an excellent approach to write DRY code. Although they sound similar, but they are different.

Framework is often more complex than a library. The architecture of application is predefined, and the developers only must be concerned with implementing domain-specific functionality. A disadvantages of using a framework is strict rules and patterns that must be adhered to. This requirement often leads to a steep learning curve. Additionally, the enforced rules and patterns can have a negative impact on the size of the application, increasing its complexity and making it more challenging to maintain and modify in the long run.
Libraries empower developers by providing them with the flexibility to incorporate specific functionalities as required, enabling them to have complete control over the architecture of their applications. Consequently, applications can be tailored to include precisely the necessary functionality.

A disadvantage of using libraries is that if the application is not designed properly, it can result in less maintainable software.
There are a lot of libraries and frameworks available to assist the developers to implementing complex web applications. The Fig 2.5 demonstrates a comparison between the most framework and libraries used from 2016 since 2021 which as a result the most usage are: *'React'* then *'Angular'* and *'Vue'*[10].

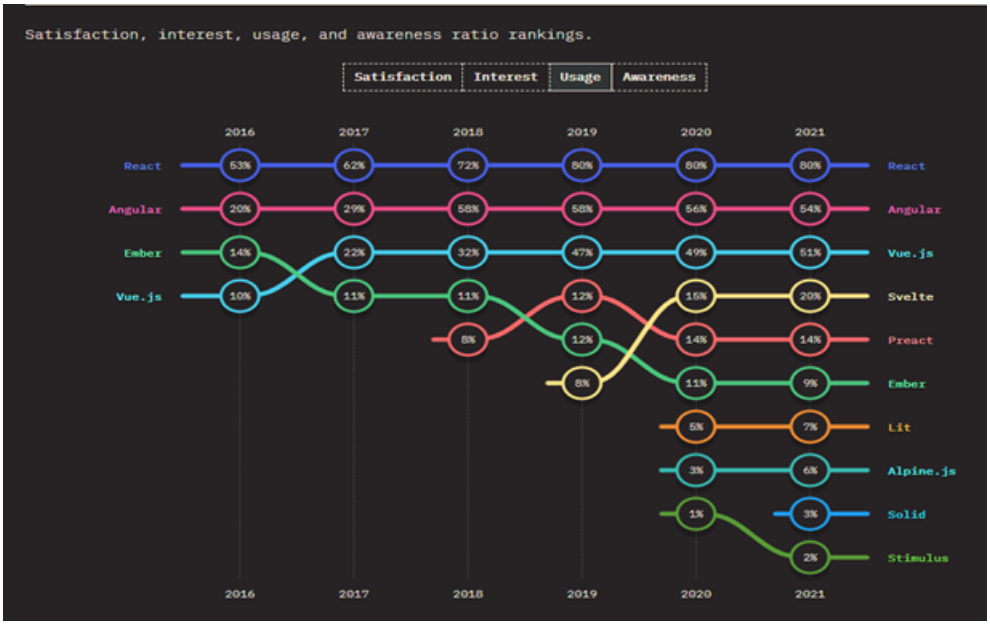


Figure 2.5: Ranking of most libraries, satisfaction, interests and usage

This result is confirmed also by a comparison made on npm trends which demonstrates React had most download during the 2021 [11].

2.2.2 React.js

React is an open source JS library to building user interfaces and is usually use for large-scale application. It is designed based on components which are self-contained module that renders some output. These components are reusable and extendable. The components are usually small building blocks because they can be combined and to easily create complex and more feature-rich components. Every component can have an internal state and manage its own state. React updates and render just the right components when the data changes. Components take input data and return what to display [12].
We can say we write react components that correspond to various interface elements, then organize these components inside higher-level components which define the

structure of our application [13].
Let's review some **Principal Concepts of React**, which is necessary to know when speak about react application.

JSX

JSX defined by React document. It is a xml-like syntax extension to ECMAScript without any defined semantics which helps developers to write React.js components. It combines syntax between Html and JavaScript, allowing developers to write the html code and embed JavaScript expression inside. It gives us the ability to write html elements in JS and place them into DOM by converting Html tags into React elements without the need of methods like *createElement()* [14].

React DOM

As we know the DOM was developed to solve the issue of directly communicate between the JavaScript and HTML. The react-dom package provides dom-specific methods that can be used at the top level of app to enable an efficient way of managing DOM elements of the web page [15].
React uses *'Virtual DOM'* which is an in-memory and lightweight representation of the dom. It means in React, for every dom object, there is a corresponding virtual dom object, which has the same properties as a real dom object.

Changing the virtual dom is 'faster' and more 'efficient' than doing the same operations on the real DOM. It renders components or jsx elements to the dom. To use the react dom in any React web application we have to import ReactDOM from the react-dom package. When we try to update the dom in React, the entire virtual dom gets updated, at the end on the real dom only the changed objects get update and changes on the real dom cause the change screen. [16]

React Router

React Router is a popular routing library for building single-page applications with React. It allows to handle routing and navigation within the React application. It keeps the UI in sync with the URL, enable navigation among views of various components and allows changing the browsers url. React Router provides a set of components and APIs that make it easy to define and manage routes in application. To use React Router at our application, it need to install as a dependency using *'npm' or 'Yarn'* and import the necessary components from the *'react-router-dom'*, then define our routes and use the provided components to handle navigation and rendering of different views based on the current url.
Components of React Router are divided to tree primary categories which are:

1. **Router:** at the core of every app must be a router.
   Two kinds of Routers usually used for web projects are the 'BrowserRouter' which uses *History API* to handle URL. It uses regular URL paths, and required the server to be configured correctly and the 'HashRouter' which stores the current location in the Hash portion of URL. It changing URLs and navigating between pages won't make any request of server configuration and has a simpler setup. We use hash and the url looks something like this: *https://www.nuageweb.it/#/cassa*
   It must render at the root of element hierarchy, usually top-level App element in a router.

2. **Route Matchers:** they refers to the mechanism used to match the current URL path to the defined routes in our application.
   The two components of Rout Matching are the 'Route' which It is responsible to render some UI when its path matches the current url and the 'Switch' which is used to wrap a collection of *Route* components. It iterates through its child Route and displays the first one that matches the url is rendered. After the version 6 of react router, the route is replaced effectively the switch component.

3. **Navigation (Route Changers):** react router provides various mechanisms for navigation and route changing.
   The most common ways are *Link, NavLink, useNavigate*. The 'Link' component is used to create link that navigate to different rout. An anchor <a> will be rendered in Html doc, wherever render a 'Link'.
   The 'NavLink is a special type of Link that can style itself as 'active' when it's to prop matches the current location.
   The 'useNavigate' hook is an add to react router v6. It provides a function to navigate to different routes programmatically.[17]

Props vs State

Two important concepts at react are 'Props' and 'State'. Both hold information relating to the components, but they are used differently.

Props

Short for Properties, are objects that stores the values of attributes in react. They are used to pass data between react components . The process involves defining or retrieving data from the parent, attributes and their values are defined or obtained, then assign it to child component's "props" attribute (similar o arguments of function in JS) and finally with dot notation we access and render the props data props. Props facilitate a unidirectional data flow, means the data can be passed

just from parent to child component and they are read-only.

State
React provides another feature for data manipulating known as "State". This concept addresses the limitation of read-only props. State holds initial data, which can be modified using a special method called *setState()*. State changes based on user actions or certain events, so components that use state rendered based on the data in the state. When the state changes, react immediately re-renders the DOM using *setState*. It doesn't re-render the entire dom, but only the components whose state has been updated. State is initialized inside its component, and state updates should indeed be made using setState().

About states the important notes should be consider are: They modify by using *setState()*, not directly and they affects the performance of app, so shouldn't be used unnecessarily.

The important **differences between props and state** are:

1. using 'Props', components receive data from outside, whereas with 'State' they can create and manage their own data.
2. data from 'Props' is read-only and cannot be modified by received component (it is immutable), whereas 'State' can modify its own component, but cannot be accessed from outside (it is private).
3. in 'Props', data is passed from one component to another, while the 'State' is a local data storage and cannot be passed to other components. [18]

React Classes and Functions Component

components are *core building blocks* of React applications. They are like JavaScript functions but work in isolation which returns HTML. They are independent and reusable block of code. They can be created in two ways in React:

Class Component are regular ES6 class that extends the component class of the React library. They are stateful and have access to all different phases of a React life cycle method. Before development of React hooks, the class components was the only option to create a dynamic and reusable component, as they give us access to life-cycle methods and all React functionalities. In React, before rendering, the components go through a life cycle of events as follows

1. *Mounting*, means adding nodes to the dom
2. *Updating*, altering existing nodes in the dom
3. *Unmounting*, removing nodes from the dom

For example, if a component need to be updated, with a change in state or props, after the 'mounting', they go through 'updating' phase, and at the end to final phase which is 'unmounting'. Is possible they don't go through every phase[19].

Functional Component are basically JS/Es6 functions. They receive props as a parameter and return the JSX to render. Such as classes, they can use into another components.
React Hooks revolutionized the usage of functional components by enabling them to handles state and ustilization of lifecycle methods, making them even more powerful and widely used in modern React development.

React Hooks

Hooks are functions that allow us to use *'state'* and other React features in functional components, without need to writing a class. They are added to React at the version 16.8 as a way to write reusable and more readable code, especially for managing state and side effects in functional components.
In other word, they are functions that let us "hook into" React state and lifecycle features from function components. Hooks don't work inside classes and they can be call only from react functions. The other important note about hooks is, we must call hooks at top level of react functions[20].
There are many number of Hooks used in react such as: *useState, useEffect, useContext, useCallback, useRef*, etc. I provide a brief explanation about two most important of them *'useState', 'useEffect'* which are most used at this project. Some details is presented about usage of hooks whenever we represent the usages of them inside code.

State Hook, allows functional components to have their own internal state. It uses *useState()* which allows setting and retrieving state. We pass the initial state to this function, and it returns a variable with the current state value and another function to update this value. [21] ***useState*** is a new way to use the exact same capabilities that *'this.state'* provides in a class. It should be first imported from react and call directly inside our component.
As an example at the following line *useState* hook is used to declare a variable named *'isNewPage'* and its corresponding setter function *'setNewPage'* which updates the value of state variable: *'const [isNewPage, setNewPage] = useState(false)'*

Effect Hook, It is the lifecycle hook which carries out an effect each time there is a state change. It allows you to run code in response to certain events, such as component *'mounting', 'updating'*, or *'unmounting'*. It does same work as life-cycle

method used in React classes, but unified into a single api.
The motivation behind the introduction of ***useEffect*** is to eliminate the *side-effects* of using class-based components. By default, it runs after the first render and every time the state is updated.[22]
'useEffect' is a function that would be called whenever the page re-renders. To create useEffect hook, we must import it from react, then go through the function and write useEffect, and then pass a function. This piece of code (useEffect) will be called after the page has been rendered:

useEffect(()=> { ..here we can write what we want happen when the page render..})

Numerous hooks are frequently employed in react projects, offering versatility in managing various facets of component behavior. Among the pivotal React hooks are *useContext, useReducer, useCallback, useRef*, and several more.

Chapter 3

Development Tools and Technologies

This chapter is devoted to explaining the technologies and tools employed in the implementation of the application, particularly focusing on the React implementation phase. In the third section, a concise overview of external libraries utilized in this project is provided.

3.1 Technologies and Tools

We capitalize on the advantages offered by Visual Studio Code, leveraging a suite of carefully selected extensions to augment our coding experience throughout the development phase. Our browser of choice is Google Chrome, equipped with a comprehensive set of developer tools crucial for debugging. To streamline the management of state changes, we integrate the Redux DevTools extension.
Postman is used as a tool for interacting with APIs. It enables us to seamlessly create and dispatch a variety of HTTP requests, including Get, Post, Put, and Delete, to our API endpoints.
Setting up a React application necessitated the installation of Node on our machines, ensuring a minimum version of 10. Alongside node, the installation automatically includes NPM, the package manager, with a minimum required version of 5.2. The Git used as version control system to track the changes.

3.2 Implementation of react app

To implement the application, the initial step was to install Node and NPM, which are necessary for creating a React app. Additionally, we needed to set up build

tools like Babel and Webpack. However, by installing CRA (Create React App), we can bypass the need to install separate build tools. It automatically generates the required files and folders to kickstart the React application. Babel and Webpack are essential because web browsers do not understand the Jsx syntax used in React. They act as translators, converting JSX code into JavaScript that browsers can interpret. Install CRA done via this instruction: - *npm install -g create-react-app* To create a react application we can use the following command: - *npx create-react-app [app-name].* Alternatively, Yarn can also be utilized for this purpose.

3.3 Implementation of external libraries

The implementation of external libraries plays a crucial role in the development of an application. Among many reasons that encourage developers to use libraries, the most important ones include:

- They often provide pre-built, optimized solutions for common tasks, significantly speeding up the development process.
- Libraries are typically well-tested and maintained by a community of developers, offering a wealth of documentation and forums to support users.
- They are regularly updated to address security vulnerabilities.

To facilitate the development of web applications, numerous external libraries exist as reusable components that have been published by other developers. These libraries can be easily installed and utilized. In a react project, the file named 'package.json' located in the root folder contains a list of all installed libraries. The most important libraries used at this project are:

- Tailwind
- HeadlessUI
- Formik and Yup
- Redux
- Axios
- Chart.js and react-chart-js2

In the following section, I provided a brief explanation of the listed libraries and their implementation, which play a crucial role in the developing of this application.

3.3.1 Tailwind

Tailwind is the most popular Utility-First CSS Framework to build Custom UI in the fastest and easiest way. It means, unlike UI kits such as Bootstrap, tailwind

doesn't provide pre-styled components. It doesn't have a default theme. With tailwind we style elements by applying pre-defined classes directly in the Html.
We defined it to write inline styling to achieve an awesome interface without write css in the different files. Tailwind allows for rapid development of modern websites by enabling us to write CSS directly in our markup. It offers high customization capabilities to tailor the styles according to our specific needs. It scan all the Html files, JS components, and any other templates for class names, generate the corresponding styles and then write them to a static CSS file [23]. The only issue with that is our markup maybe looks a lot longer.
For using and running tailwind we follow instruction of tailwind documentation for install CLI tool which is the fastest way and recommended by tailwind. [24]. The steps are as follows:

- First we installed tailwind via npm and run the init command to generate files: *'tailwind.config.js'* and *'postcss.config.js'*.

  - postcss.config.js is a single export JavaScript object that will instruct to run tailwind first. By installing PostCSS and its associated tools such as 'Autoprefixer,' we can leverage the framework's capabilities to optimize our development process.

```
module.exports = {
        plugins: {
                tailwindcss: {},
                autoprefixer: {},
        },
}
```

  This configuration file is used to customize the default theme and properties of Tailwind. such as: custom fonts, color, line spacing, etc.
  Utilizing 'PostCSS' significantly enhances the build speed of our projects, leading to improved performance.

  - tailwind.configure.js, at this file we added the path to all template files.

- Then at main css file **index.css** add *'@tailwind'* directives for each of Tailwind's layers: *@tailwind base; @tailwind components; and @tailwind utilities;*
  This is like importing tailwind styles to be recognized as real css syntax. Finally import this css file in the entry file of react which is *'index.js'*.

- Finally, execute the CLI tool to scan the template files for classes and build the CSS: *-npm run start*

Tailwind Customization

Tailwind is designed to be customizable, which is one of its key advantages. It allows to break out the constraints when needed, enabling us to add our own custom CSS and extend various plugins. Tailwind's customization advantages empower developers to create unique, efficient, and consistent user interfaces while adapting to specific project needs and maintaining a scalable development workflow.
Fig 3.1 illustrates the **'tailwind.config.js'** configuration file, which exports a *'module'* object. This configuration object contains various properties, including *content, theme, plugins.*
In 'Content' added the path to all template files. The 'plugins' property specifies additional Tailwind CSS plugins to enable. An advantage of tailwind is its capability to integrate reusable third-party plugins, expanding the range of styles and functionality available. By simply installing the desired plugins through npm and importing them, we can easily incorporate them into our project. We can then call the specific plugins we want to utilize within the 'plugins' array.

```
module.exports = {
        content: ['./src/**/*.{js,jsx,ts,tsx}', './public/index.html'],
        theme: {
                screens: { ...
                },
                 Extend: { ...
                },
        },
        plugins: [
                require('tailwind-scrollbar'),
                require('tailwindcss-textshadow')
        ],
}
```

Figure 3.1: Add reusable third-party plugin to tailwind

'Theme' is contains various configuration options. It includes customizations such as defining additional 'keyframes' for animations, adding new colors, custom font families. Inside screens we defines the breakpoints for responsive design. At Fig 3.2 we can see more details of developing of this part:

Figure 3.2: Customization at tailwind.config.js

3.3.2 HeadlessUI

HeadlessUI is a set of accessible UI components, designed specially to work with Tailwind CSS. This library provided components such as: dropdown menu, lightbox, Switch (Toggle), Transition, Tabs, ecc for adding to our project [25].
To using these components, after installing headless UI library via npm, we import them from *'@headlessui/react'* and write the code with the customization we need to apply to our components. We got advantages of this library at different parts of application utilized components like Listbox, Tab pages, Switch, and more. further details about implementing these components at this application explained at chapter 4, Requirements of application.

3.3.3 Formik and yup

Formik is a third-party library that used to implement and manage forms in this application together with *Yup*. It is one of the most popular open-source libraries for building forms in React and React Native. Using Formik repetitive actions are reduce and we can save time. It helps with 'Taking the user input and change the current state', 'Form submission' and 'Validation'.
It sets the state for the form's value and exposes it to form via props. It provides reusable methods and event handlers, helps to greatly facilitate the development

process. These include handleChange, handleBlur, and handleSubmit, that streamline the handling of form input changes, blur events, or form submissions. Using these built-in functionalities, developers can efficiently manage form interactions and improve the overall user experience.

development Skeleton of Formik components

Formik is implemented into separate components, making them easily reusable across different sections of application. This approach allows us to integrate Formik's form management capabilities efficiently and promotes code reusability. These components handle different types of input *TextArea, SelectField, TextField*, placed in a unic folder called *formik*. These components makes use of ***useField*** hook, which is provided by Formik and returns an array with two elements: *'field'* and *'meta'*. The Fig 3.3 is demonstrating 'TextArea' component as an example, the two other components are also implemented at these way:

```
import { ErrorMessage, useField } from "formik"
import { ExclamationCircleIcon } from "@heroicons/react/solid"

const TextArea = ({ textareaStyle, ...props }) => {
  const [field, meta] = useField(props)

  return (
    <>
      <div className={`${props.inline && "flex items-center"} ${props.checkbox && 'justify-between'}`}>
        <div className="relative flex flex-col">
          <textarea className={textareaStyle} {...field} {...props} placeholder={props.placeholder} style={{ border: meta.touched && meta.error && '1px
          {meta.touched && meta.error && <ExclamationCircleIcon className="h-5 absolute right-2 top-6 text-state-error-light" />}
          <ErrorMessage component="span" name={field.name} className="text-state-error text-xs font-semibold ml-0.5 absolute top-full" />
        </div>
      </div>
    </>
  )
}

export default TextArea
```

Figure 3.3: Formik, developing TextArea component

This component renders a custom textarea input element. It takes props 'textareaStyle' and '...props' which is a "spread" operator that is used to pass all the properties of the props object to this element. The *'field'* object contains properties and event handlers that need to be attached to the 'textarea' element for proper form handling. The spread operator *(...field)* is used to pass these properties and event handlers to the 'textarea' element. The *'meta'* object contains additional metadata about the field, such as whether it has been touched or has an error. This information is used to apply certain styles and display error messages.

The style attribute is set dynamically based on the meta object and finally the *'ErrorMessage'* component is used to display the error message. This component as an example imported into the ModalPromo component which is a child component of landing page, where the CSS style of the textarea element defined and assigned to textareaStyle var. Chapter 4 provides more detailed examples of how we used Formik components in different parts of application.

A brief explanation about Yup

Another component implemented at formik folder is *Validation.js*. This file is contains data validation rules for forms and it is imported to the components where there are need data validations. At this file we get advantages of **Yup**. It is a schema-builder that allows us to define validation rules and schemas for data validation. For using Yup we should:

- first defined object schema and its validation.
- then define all the criteria for objects[26]

Yup can test whether a value is an email address with one method call, the same for date, time and etc. The Fig 3.4 is demonstrates *'newCustomerSchema'* as an example. The schema is defined using the *object().shape()* method provided by Yup, which allows creating a schema with multiple validation rules for different fields. for example the telephone number should have a length of exactly 10 digits and should only contain numeric characters.

```
export const newCustomerSchema = yup.object().shape({
    Anagrafica: yup.string().min(3, "Il nome deve avere almeno 3 caratteri").required("Obbligatorio"),
    Email: yup.string().email("Email non valida!"),
    Telefono: yup.string().length(10, "Deve avere 10 cifre").matches(/^[0-9]+$/, "Sono ammessi solo numeri")
    ....
});
```

Figure 3.4: Utilization of Yup library, creating schema

The *required()* forces user to insert a value and it can't be left. This schema then imported to the components wher utilize formik and inputs need to be validate, assigning the name of schema to *ValidationSchema* which is a special config option/prop for Yup object schema.

3.3.4 Redux

Redux is a library for managing and updating the state of application. It is suitable when having large amounts of application state that need be use in many parts of application and might be worked on by many people. As we worked with a large and complex application which needed frequently update of state over time, we got advantages of this library. It can be use with any other JS framework or library. It allows to manage the application's state in a single place, serving a centralized store for state that need to be used across our entire application[27].

To utilize Redux, we need to install the 'redux' package. Additionally, we also require another dependency called *'react-redux'* to integrate redux into our react application. 'Redux Thunk' is imported which is used to handle action. It is a midleware commonly used for data fetching and help to handle asynchronous operations, such as API calls. Redux has tree building parts:

- **Actions:** They are events and a way to communicate with the redux store, triggering state updates. Internal actions are JS objects that have a *'type'* property describing the type of action and a *'payload'* property containing additional data associated with the action, which is send to the store.
- **Store:** There is a central store that holds the application state. each component in the application can access the stored state directly, so they don't have to rely on passing data (props) from one component to another through a chain[28].
- **Reducers:** They are pure function that takes the current state and an action, and returns the new state. Reducers are used to specify how the state of the store should change for dispatched actions. The new state is then sent back to the view components of the react to make the necessary changes[29].

These tree parts have *'one way data flow'* which is demonstrates at the Fig 3.5.

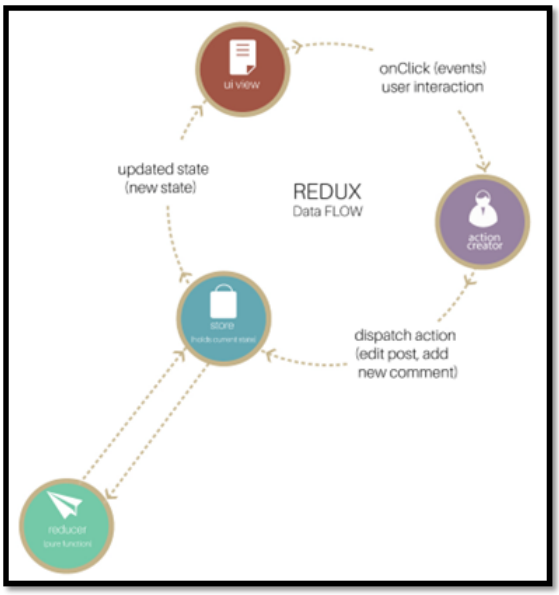


Figure 3.5: Redux data flow and main components

Implementation of Redux components

A file *'store.js'* is created and located in the redux folder, where all redux-related files are kept. This file serves as a global store. After importing statements, the store is created using *createStore* function and the 'rootReducer' is passed as an argument, which is the combination of multiple reducers. The *thunk* middleware is applied by applyMiddleware to handle asynchronous actions in Redux.

The integration of the Redux DevTools extension is accomplished through the application of the *compose* function. This function is employed to facilitate debugging within the browser's Redux DevTools extension, which must be added to the browser. In essence, it provides a way to enhance and inspect the state management of a Redux application in a browser environment.
Finally the *'persistStore'* function from redux-persist is called with the redux store as an argument to create a persistor. The persistor in Redux enables the storing and retrieval of the application state. It automatically stores the state whenever it changes and retrieves the stored state when the application starts or refreshes, ensuring the Redux store is rehydrated with the saved state.

Figure 3.6: Store.js, redux global store file

There are more than one *'reducers'* created at reducer folder imported into index file presented at this directory. The *'CombineReducers'* function from Redux lets us merge several reducers. We use it to create a 'rootReducer,' which is then passed as an argument to the 'createStore' function.

'actions' in our project are written in separate files and imported where they are needed for defining actions.
The types for these actions are defined in the *'type.js'* file, found in the *constants* folder of redux directory. These action types are subsequently imported into both the 'actions' and 'reducer' files, ensuring consistency across the project.

3.3.5 Axios

Dealing with web applications, one of the fundamental tasks is communicate with back-end servers. The client-side application for receive and update data from the database use the API and communication with api is done through Http request, which can be achieved using *'Fetch' or 'Axios'*[30]. While both options serve similar purposes, Axios is chosen for this project for its ease-to-use and extensive browser support. Some of the features of this JS library are:

- It allows making HTTP requests from both the browser using XMLHttpRequest, and from Node.js.
- Supports for the Promise api, which handle asynchronous operations and manage the resulting values or errors.
- Ability to intercept and transform both request and response data.
- It also can cancel requests or automatic transforms for Json data, so unlike FetchAPI we don't need converting our request body to a Json[31]

It need to be installed by npm in react application, then added to the project. At this project, it is implemented by creating two components 'axios.js' and 'requests.js' both placed at api folder. Fig 3.7 demonstrates *'axios'.js* component.

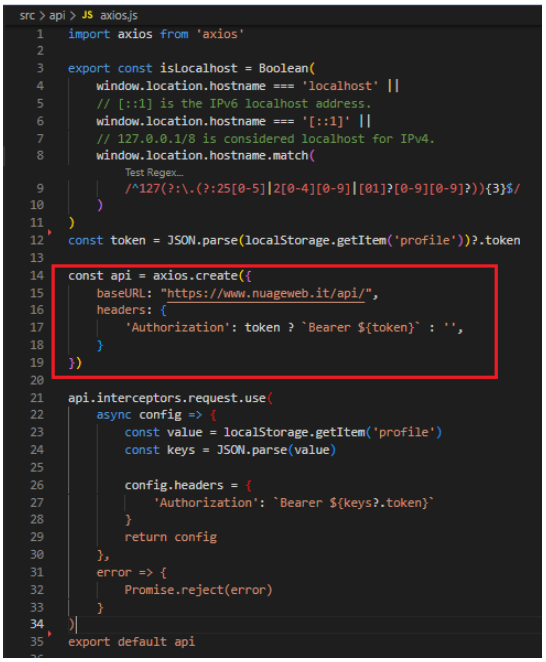


Figure 3.7: Axios implementation and creating its instance

This component make HTTP requests to an API. Firstly, it checks if the application

is running on the localhost, saved result in a constant. Then retrieves the 'token' from local storage if it exists, and parses it as Json. Then it creat an instance of Axios with 'axios.create()' function. This instance can be used to make Http requests to the specified 'baseURL' which is including the Authorization header with the token value.
The code examines if token is exists, the authorization header will be set as Bearer token, otherwise it set an empty string. Bearer token is a type of access token commonly used for authentication and authorization in Api requests.
The last part of this component, *'Request interceptor'*, is an error callback function which is called on this instance to add a request interceptor. This interceptor modifies the request configuration before it is sent.
At the *'requests.js'* component an object is defied that holds various api endpoint URLs. Each key in the object represents a specific request, associated a name to each of them, such as: *fetchServices: "/get-servizi.php"* or *postService: "/post-servizi.php",...*
These requests imported at 'App.js' file where handles routing and api calls and renders the appropriate content based on the authentication status.
Fig 3.8 demonstrates a part of *'App.js'* where after import 'request.js' component from api directory, perform api calls and fetch data.

```
// requests
import requests from "./api/requests";
// redux
import { useDispatch } from "react-redux";
import { /*clearFrequenti,*/ setData } from "./redux/actions/dataActions";
// router
import { Routes, Route, HashRouter } from "react-router-dom";
import { useSelector } from "react-redux";
import { isUserLoggedIn } from "./redux/actions/authActions";
import { getCommooditiesSectors, getCustomers, getData, getServices, getOperators, getProducts, getPromotions, getSuppliers,
import { circuitTypes, /* customerInfoTypes,*/ dataTypes, promotionsTypes, settingTypes, } from "./redux/constants/types";
import { applyTheme } from "./themes/utils";
import { greenLanternTheme, nuageTheme, pruneTheme, corporateTheme, grayScaleTheme, CMYKTheme, bumblebeeTheme, nordTheme, dr

function App() {
    const dispatch = useDispatch();
    const { token, user, authenticated, authenticating } = useSelector(
        (state) => state.auth
    );
    const localToken = JSON.parse(localStorage.getItem("profile"))?.token;

    // api calls
    useEffect(() => {
        if (!authenticated) {
            dispatch(isUserLoggedIn());
        } else {
            (async () => {
                await dispatch(setData(requests.fetchCustomers, dataTypes.SET_CUSTOMERS));
                dispatch(getCustomers());
                await dispatch(setData(requests.fetchData, dataTypes.SET_DATA));
                dispatch(getData());
                await dispatch(setData(requests.fetchCategories, dataTypes.SET_CATEGORIES));
                dispatch(getCategories());
                await dispatch(setData(requests.fetchFamilies, dataTypes.SET_FAMILIES));
                dispatch(getFamilies());
                await dispatch(setData(requests.fetchServices, dataTypes.SET_SERVICES));
                dispatch(getServices());
                await dispatch(setData(requests.fetchOperators, dataTypes.SET_OPERATORS));
                dispatch(getOperators());
                await dispatch(setData(requests.fetchProducts, dataTypes.SET_PRODUCTS));
                dispatch(getProducts());
                dispatch(setData(requests.fetchParkout, circuitTypes.GET_PARKOUT));
```

Figure 3.8: importing apis call in main App component

The *useEffect* hook is used to call api and fetch data. It has dependencies [authenticated, dispatch], meaning it will re-run when either of these dependencies changes. It retrieve authentication State from redux store.
The *useDispath* hook from redux is used to dispatch actions to the Redux store, triggering state changes. If the user is authenticated, a series of API calls is initiated. The API calls are wrapped in an asynchronous function, and await is used to ensure that each API call completes before moving on to the next one. This ensure the correct order of fetching data.
The *dispatch* function used to send actions to the redux store.

3.3.6 Chart.js and React Chart js2

Chart.js is a popular open-source JavaScript library for creating interactive and customizable charts and graphs on web pages. It provides a wide range of chart types, including line charts, bar charts, pie charts, scatter plots, and more.
'React Chart.js2' is a React wrapper for chart.js, specifically designed to integrate chart.js functionality into react applications. It simplifies the process of integrating chart.js into react applications and follows the react component-based architecture, allowing us to create reusable chart.
To implementing charts, after install the necessary packages, we should import required components at the our file. A simple syntax of implementing the charts is demonstrates at the Fig 3.9:

```
import { Doughnut } from 'react-chartjs-2';

<Doughnut data={...} />
```

Figure 3.9: Rendering a *react chart.js* components

Once imported desired charts, we define data object of chart, customize them, and render the chart component.
We got advantages of these libraries for creating reports . The more details about implementation of charts finds at chapter 4, at the *Report* section.

There are other external libraries were utilized during the implementation of this application, such as: 'react-icons', 'react-barcode', 'react-progressbar', 'react-slick', 'Jshint', cloudinary' and etc.
Most of these libraries are relatively easy to integrate, and comprehensive guides are often available to assist users with the implementation process. In this documentation, i focused on providing detailed insights into the most important of them.

Chapter 4

Implementation Phase

4.1 Introduction of the Desktop version

The Xtouch application is a dedicated desktop management solution designed exclusively for hairdressers and beauty centers, addressing a wide range of user needs within this industry. Some of the most important features of this application are as follows:

- **Payment system:** customer payments is very easy as this application gives possibility to payment with mix modality. They can pay with cash and with credit card half/half, or the payment can be proceeded by mix of gift/promotion cards and other payment type at the same time.
- **Managing Promotions:** the other aspect of this application is to manages promotions (gift cards, subscriptions, discount cards, vouchers, temporary promotions, accumulation services, point management).
- **Manage appointments:** managing appointments of salons respecting the real timing of the services and the availability of the collaborators, print or send (either, email) to the customer a detailed reminder.
- **Agenda:** it is a multi-platform app created to keep in touch with its customer and provides them with the possibility of online booking appointments thanks to having the "Agenda on the Web".
- **Warehouse management:** the possibility to know in real time the status of the warehouse and each product is an important feature of this application. The functionality is managed with a simple Barcode Reader When users make a sale or use a product intended for the salon. The barcode speeds up the operations of app which is integrated with all types of barcode readers.

- **Email and SMS:** this software integrates a system that allows use to send mail and sms directly to his/her customers. It also gives the opportunity to make a specific selection for the strategies the user is creating. Birthday wishes, lost customers, customers who only get one kind of treat, ecc.

The Figure 4.1 illustrates the schematic layout of the 'Cash Desk' page, which serves as the entry point of the desktop application due to the absence of a dashboard.

This screenshot showcases the working process of the application, where demonstrates by selecting a 'Service' from the available 'Categories', followed by choosing an 'Operators' subsequently, the right side of the page dynamically displays relevant information regarding the chosen operation. Additionally, the bottom section presents detailed Payment information related to this process.

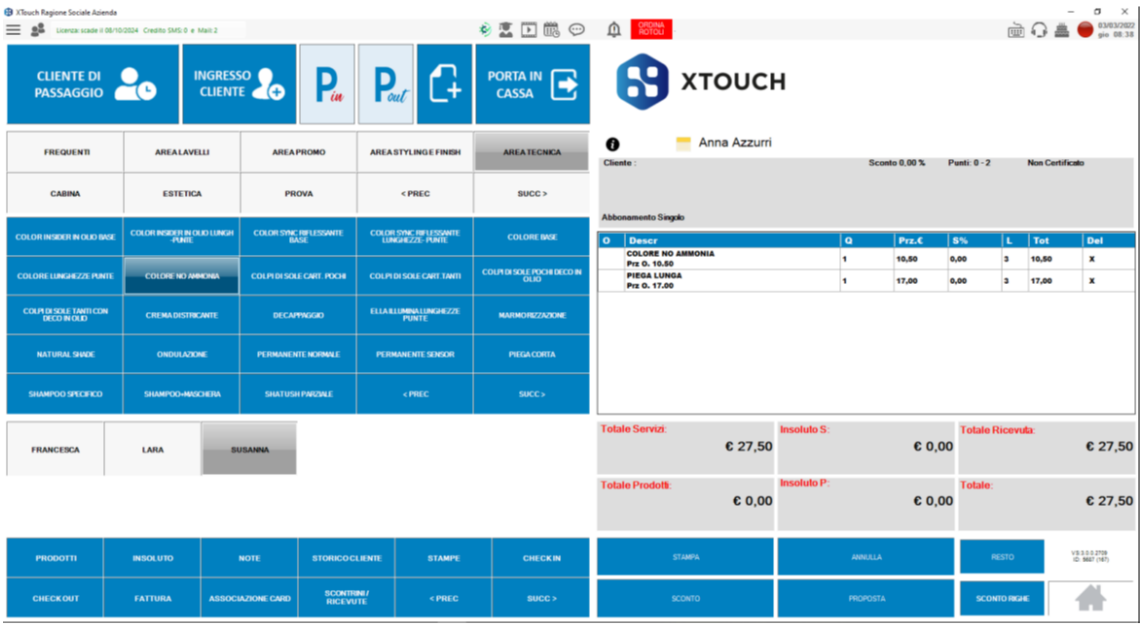


Figure 4.1: Scheme of desktop application, Cash Desk page

In the process of implementing the web version, the team thoroughly analyzed the requirements with the conclusion of including most of the features already present in the desktop application at web version and introduce some additional features which represents a modern web application.

4.2 Web version development

The Fig 4.2, demonstrates the 'Cash Desk' user interface of web version, as the development process initiated with the implementation of this page, recognizing its pivotal role as the cornerstone of this application and representing a critical development effort. It serves as a crucial navigational hub, directing users to other pages within the application in the absence of a dashboard.
The web version is named 'Nuage'. The structure of this page was defined, comprising the principal components: Category, Services, Operators, Client area, and Payment sections. Furthermore, there was plan to introduce a dashboard page in the future, intended to serve as the application's entry point, after its structure and user interface is defined completely.

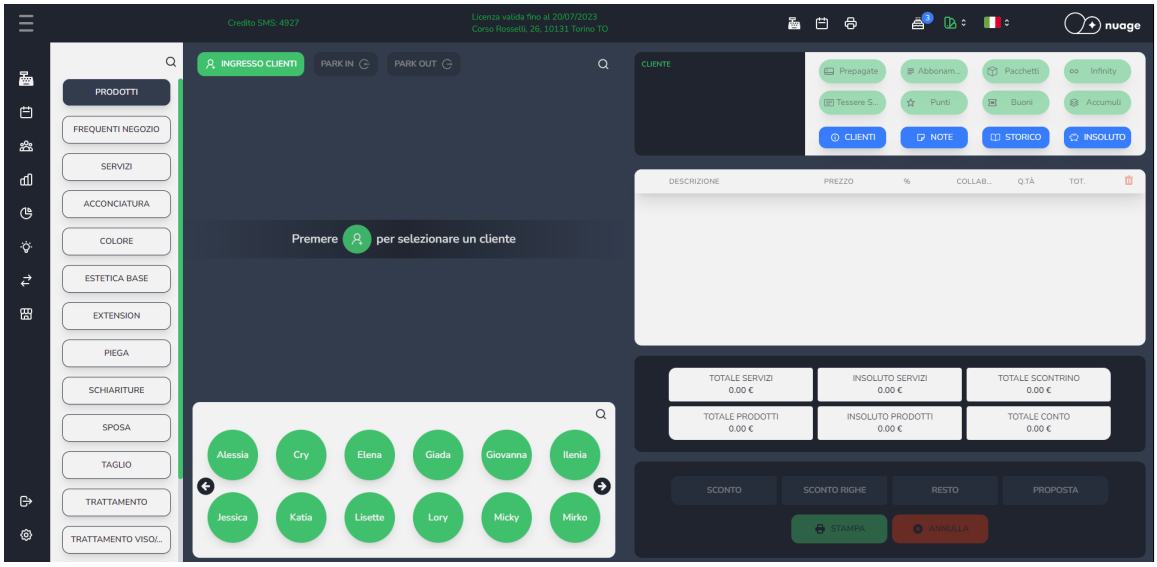


Figure 4.2: Scheme of Web application, Cash Desk page

4.2.1 Files and Folders created by CRA and their structure

There is not a proper way to structure a react application, we can decide structured the files and folders as it will be more significant for our project.
A popular way is to group similar files together [32]. We follow this approach at our project, as an example we put all api files in a folder, all pages we go around by rout placed in a unique folder and etc.

After installing CRA and created the project using the command *npx create-react-app Project-Name*, react generates a folder named *"Project-Name"*. This folder encompasses various files and sub-folders that holds the project code. Some important files and folders are: 'node-module', 'public', 'src' and 'build folders, and 'readme.md', 'package.json', 'package-lock.js', '.gitignore' files.
The remaining files and folders are added to the project during the development phase. In the following section, I provide a brief explanation of some of these files and folders.

- package.json This file is a configuration file that provides information or meta-data about the project and specifies its dependencies. Dependencies are external libraries or packages that the project relies on to function properly. The *'dependencies'* field list all the dependencies of project that are available on npm. The npm install command reads this package, retrieves the specified dependencies from the npm registry, and installs them locally in the node-modules directory which helps npm to setup same environment on different machine for our project.[33]

- package-lock.json This file is generated automatically when create a react app and provides a low-level, detailed snapshot of the exact versions of packages installed, ensuring consistency across different installations and environments.

- Readme.md This file is containing basic information about user's project. It can be used to define summary of project, build instructions, etc. It uses markup language to create content.

- .gitignore This file specifies intentionally untracked files that git should ignore.

build folder To publish our application run the following command: *'npm run build'*, it creates this folder inside the root directory.

- node module folder This folder is containing all the node packages that our application requires. In other word all dependencies that are need for initial working of react application are placed at this folder. [34]

PUBLIC folder
This folder is containing three files created automatically by *create-react-app*:
- A **Manifest.json** is metadata file which provide information about application such as its name, version, etc. It is primarily associated with PWAs and is used to define how the application should behave when installed on a user's device and utilized by modern web browsers.

- An **Index.html** is the main Html file that serves as the entry point for the application. All dynamic content of the app will be injected in the root div of this file. React components are typically defined in separate JS files and rendered into the this file using the ReactDOM library.
- A **Favicon.ico** which is logo of application and referenced in the index.html.

SRC folder
This folder is a crucial directory in a react application and typically contains the *source code* of the app, including the main JS files, components, stylesheets, and other related assets. Usually some files and folder found within the Src, which the two most important files are:

An **App.js** file, which represents the main component of the application, we can say the application content is coming from this file.

An **Index.js** file which serves as the entry point of the application. It is responsible for rendering the root component and initiating the React rendering process.
The folder structure can vary based on the project's setup. Often there are presented folders such as 'component', 'pages', 'utils', etc.
Here's a brief explanation of the files and folders created at our application, organized based on grouping approach to bring similar files and folders together:

➢ **components folder:** this folder contains all the components we have created and assets. Each component is organized within hierarchical sub-folders and has a main file which by assembling them in the main file, we can construct different parts of the application. The benefit is that these components can be imported and reused to creating other components. The most important sub-folder are:
- *data:* all pages presented at *'Anagrafiche'* (placed at Sidebar) find here.
- *home:* the main components for creating the *CashDesk* page are located in this folder. Each component related to a different part is grouped in sub-folders like 'modals', 'families', 'work'. The Families folder contains the Category development code, while the Work folder includes the development files for the Services and Operators section. By combining these components in the main file, the Cash Desk page is created.
The modals created for different activities, such as adding, deleting, etc placed at Modal folder and imported in various parts of the application to use.
- *report:* this folder contains the main page and sub-folders with all the developed code for creating reports. Development details find at section 4.3.
- *setting:* at this folder we placed all the work related to creating the Setting page. Development details find at section 4.3.

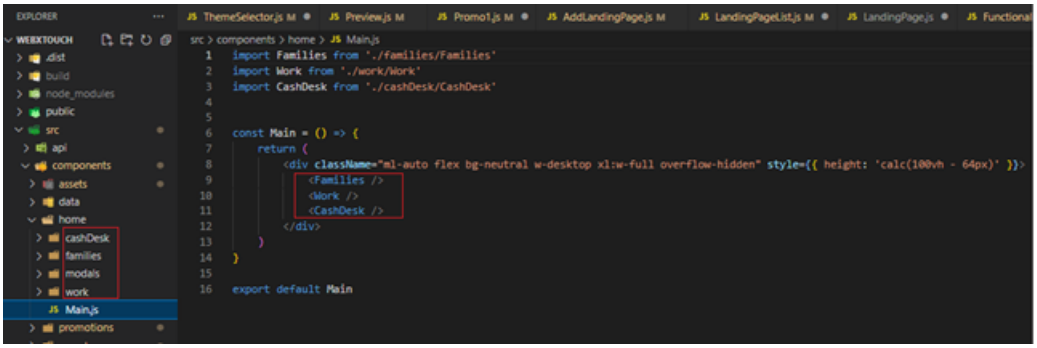


Figure 4.3: Folder structure home main page

- *promotion:* this folder contains files and folders related to the different type of promotion, such as gift cards, packets, subscriptions, landing page. Development details about *Landing Page* find at section 4.3 .
- *assets:* this folder is contains files relate to header and some other components provided to use in different parts of application.

➢ **api folder:** two files placed at this folder are: *'request.js'* which defines an object with various properties that represent Api endpoints. A *'axios.js'* file where placed a function to determine localhost, created new instance of axios and authorization checks.

➢ **formik:** this folder contains files and components created related to Formik, which are written in four separate files. By importing these components when needed, we can easily customize them based on our requirements instead of writing them from scratch every time.
These four components are: *'TextField','TextArea','SelectField','Validation'*. At *chapter 3* presented some details about the implementation of Formik.

➢ **image:** as the name says, all the images and logo placed at this folder.

➢ **pages:** such as the most large applications, the pages folder is created to organize components that represent individual pages. Each page consists of a combination of components (usually a *Navbar* and *Main* and handles specific user interactions or displays specific content.

➢ **redux:** all redux-related files are located in this folder. We import these components wherever needed to avoid repetitive code writing. There are presented a *central store* file and sub folders: *'Action', 'Constants', 'Reducer'*. At *chapter 3* presented some details about the implementation of Redux.

➢ **themes:** there find two files related to implementation of themes of application find here *util.js* and *themes.js*. Detail explained at the section 4.3.

Additional files and folders presented such as: *'utilities'* folder, which contains the 'function.js' file with various functions like Truncate and Capitalize, as well as custom hooks. The *'tailwind.config.js'* and *'postcss.config.js'* files are also present, where the configurations related to the Tailwind library are located. Further details about the Tailwind library can be found in the 'Requirements of Application' section.

4.3 Required Pages and Their Implementation

The development process involved implementing several components, each assigned to different team members using a collaborative approach to handle various parts of the project. I actively participated in the development of the following sections in collaboration with other team members: ***'Theme Selector', 'Report pages', 'Landing page'*** and ***'Impostazioni'***. In the following section, I provided a detailed explanation of the development and integration process for each of these components within the application.

4.3.1 Theme Selector

One of the requirements of this application was inclusion of a theme selector, allowing users to choose their desired color palette from a range of pre-created themes. The desired theme can be selected through a dropdown list, enabling easy switching between available options.
A theme selector enhances the design of a modern web application by prioritizing user preferences, accessibility and brand consistency. It's a powerful tool for creating an appealing, customizable, and user-centric application that meets the demands of today's users. To implementation of this part, we created two files at *themes* folder:

- *theme.js:* at this file, we defined all color palettes constants.
- *utils.js:* it is contains an exportable function *'applyTheme'*.

The Fig.4.4 demonstrates *theme.js* where defined varius exported objects such as *'greenLanternTheme', 'PruneTheme'*, which represented pre-defined pallets. Each object defines various CSS custom properties (css variables) and each property is represented by a name and its associated color value. The colors are defined for various subjects, such as 'warning', 'info', 'error', and are categorized into different groups, like 'primary', 'secondary', 'neutral'.
All these constants are imported into the *'ThemeSelector'* component where the developing part of theme selector drop down is written. By importing these pallets into *'App.js'* we make the theme selector drop down available to users on other

Figure 4.4: Creating color pallet at theme.js component

pages of application as well. This enables them to conveniently switch themes from any page, ensuring a consistent visual experience throughout the entire application. The important note for implementation of pallets to having a completely different pallet is define the same number of attributes for all pallets, otherwise if one color code is missed at a pallet it takes the value of last active theme.

Theme Selector developing phase

The structure of theme selector and its functionalities implemented within the *'ThemeSelector.js'* component. This component is responsible for rendering the dropdown list, providing users with the ability to customize and choose their preferred color theme effortlessly.

The Fig 4.5 illustrates a part of implementation of this component. The component imports necessary dependencies from react and headlessUI libraries, along with color palettes defined in 'theme.js' component and the *'applyTheme'* function from 'utils.js'.
There are also prepared an array of available themes at the begining which contains objects representing various color theme.

Each object has properties such as id, name, class, palette and type, which are required for implementation of theme selector, then using Usestate hook we defined the state variable 'theme' and its corresponding set function 'setTheme', which

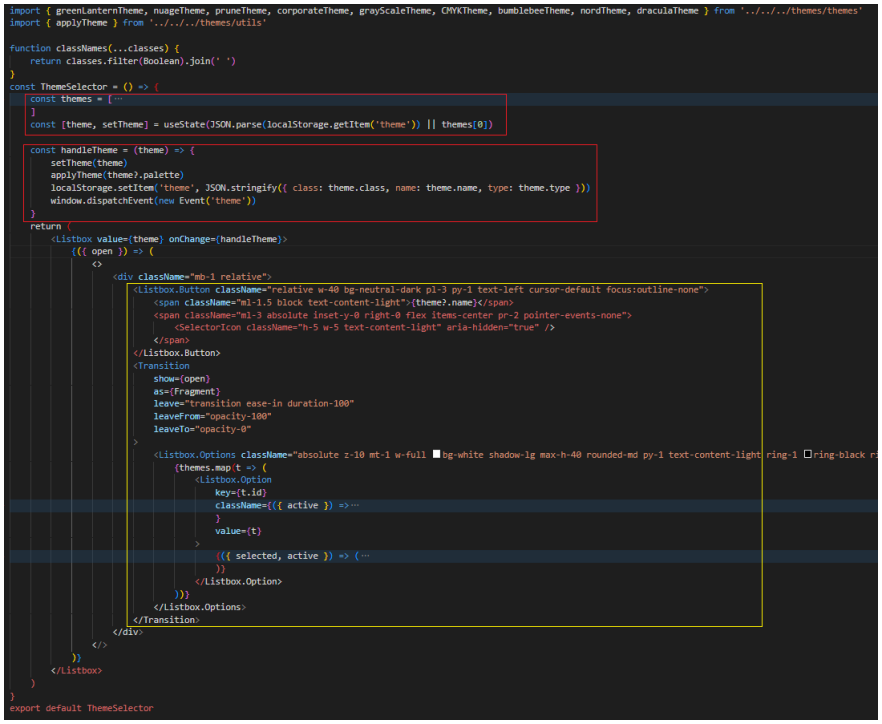


Figure 4.5: development of Theme Selector component

has responsibility of manage the selected theme, allowing for theme switching and rendering the appropriate color palette based on the user's preference or the default theme. The initial state of 'theme' is being set using the value returned by the expression.
const [theme, setTheme] = useState(JSON.parse(localStorage.getItem('theme')) || themes[0])

It attempting to retrieve a 'theme' value from the browser's localStorage object using the *'getItem'* method. If successful, the value is parsed as JSON. If this fails or if there is no 'theme' value stored in localStorage, then the default value of theme is set to an array containing a single element of 0.

At the following of code function *'handleTheme'* manages changing theme. This function is placed at Listbox tag as an 'onChange' function. The 'theme' passes as value and every time an option is selected from the dropdown list, this value passes as props to the function handleTheme, then this function will update the theme with the new one, setting all its item.
The *'applyTheme'* function imported to this component from *'utils.js'* allows to change the appearance of the user interface by applying the custom theme.
It takes a theme object as an argument, which contains CSS variables and their

```
export function applyTheme(theme) {
        const th = theme ? theme : {}
        const root = document.documentElement;
        Object.keys(th).forEach((cssVar) => {
                root.style.setProperty(cssVar, th[cssVar]);
        });
}
```

Figure 4.6: applyTheme function imported into Theme selector component

corresponding values. Then, it sets these css variables on the root element.
To create the selector dropdown in this component, we utilize the 'Listbox' component from the *HeadlessUI* library, which provides the possibility of having a custom and beautiful select menu for our application.

By importing these constants into 'App.js' we make the theme selector dropdown available on other pages of application as well. At App component, the useEffect hook is used to set the theme of the application based on a value stored in the browser's local storage. This hook runs once, when the component is mounted and sets the initial theme based on the stored preference in local storage.
A Listbox is renders with options for selecting a theme. Every time a theme is selected, the handleTheme function is called which updates the state with the new theme. wrapping the 'Listbox.Options' in a transition tag, the 'Transition' is applied to this component to have animate open/close panel.
The ThemeSelector component is imported to the 'Navbar.js', where it put together all the components related to header.

4.3.2 Reports

Nowadays data visualization is one of the most required features for implementing applications. Reports play a crucial role in modern web applications. They provide data transparency, which is particularly important for businesses and organizations that require accountability and regulatory compliance. Reports enable users to gain a better understanding of ongoing activities, and through charts, anyone can easily analyze multiple sets of data on a regular basis. This is essential for making informed decisions and evaluating the success of strategies.
There are many other advantages to having a reporting page on websites that I won't cover in detail here. There are numerous libraries available for creating

charts in React applications, one of the most used library is ***'react-chartjs-2'*** which we get advantages of that for creating reports of this application. It is an easy-to-use library for creating different kinds of charts, like *Bar, Pie, Line*. We installed another package ***'chart.js'*** that is like a core library used for any kind of application not just React app. The *react-chart-js-2* is used as a wrapper for *chart.js*, which let use chart.js elements as React components.

In the initial phase, we prepared three report pages, with the expectation of adding more reports, as soon as all the related APIs become available and their requirements definitions are finalized. These three report pages are: *'Fatturato Generale', 'Riepilogo Giornalieri', 'Produttività Collaboratori'*.
All the report pages are developed as individual components that are include child components imported to the main component of each report page. Additionally three common components are created and imported into all report pages. A 'Date Picker' for selecting a date range to visualizing reports, a 'Gender' drop-down for visualizing data, based on gender, and 'Sector' drop-down for data filtering based on sectors.

Developing 'Riepilogo Giornalieri' reports

Figure 4.7 depicts a schematic representation of the 'Riepilogo Giornalieri' report, with the date picker and drop-downs selectors positioned at the top of page.

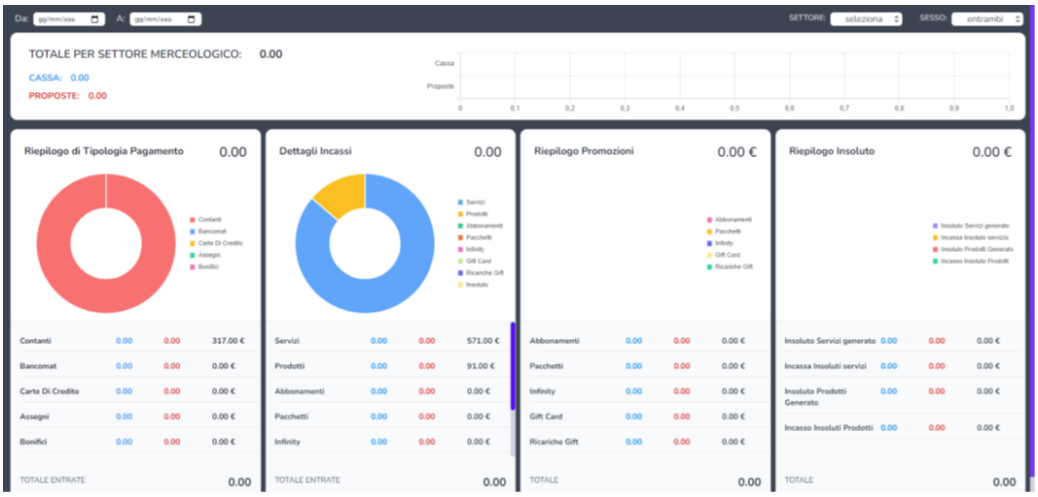


Figure 4.7: User Interface of report 'Riepilogo Giornalieri'

This report page is created within five individual components, each represents a different report. The reports within the Doughnut chart, is divided into segments. Each segment represents the proportional value of individual data on the chart.

Below the charts, there are additional information related to these datasets, which are based on three categories: *'Cassa,' 'Proposte'*, and *'Totale'* highlighted with tree different color Red, Blue and Black. The total amount for these categories displayed at the top of the page within a Bar chart, reporting the total amount earned individually for the two categories, 'Cassa' and 'Proposte'.

HeadlessUI utilization with Report pages

To create drop-down of 'Selector' components, *ListBox* is imported from 'HeadlessUI' library. The Fig 4.8 demonstrates a part of developing of *'SelectSector'* component.

```
import { Listbox, Transition } from "@headlessui/react";
import { useSelector } from "react-redux";
import { SelectorIcon } from "@heroicons/react/solid";

function classNames(...classes) {
  return classes.filter(Boolean).join(" ");
}

const SelectSector = () => {
  const Sectors = useSelector(
    (state) => state.allData.reports.SettoriMerceologici
  );
  const [SelectedSector, setSelectedSector] = useState("Seleziona");

  return (
    <Listbox value={SelectedSector} onChange={setSelectedSector}>
      <label className="pr-2 font-semibold text-sm lg:text-xs"> SETTORE: </label>
      <div className="relative">
        <Listbox.Button className="relative w-32 pr-2 justify-center rounded-md bg-white text-neutral cursor-default focus:outline-none lowercase">
          {SelectedSector.Descrizione || SelectedSector}
          <div className="flex items-center justify-start">
            <span className="absolute inset-y-0 right-0 flex items-center pointer-events-none"> <SelectorIcon className="h-4 w-5 text-neutral" aria-hidde
          </div>
        </Listbox.Button>
        <Transition as={Fragment} leave="transition ease-in duration-100" leaveFrom="opacity-100" leaveTo="opacity-0" >
          <Listbox.Options className="fixed z-10 mt-0.5 w-32 lowercase bg-white shadow-lg rounded-md text-xs px-1 py-1 ring-1 ring-black ring-opacity
            {Sectors?.map((o) => (
              <Listbox.Option key={o.Id} className={({ active }) => classNames( active ? "text-content-light bg-primary-light" : "text-neutral", "cursor-
                {({ selected, active }) => (
                  <div className="flex items-center">
                    <span className={classNames( selected ? "font-semibold" : "font-normal", "ml-1 block truncate" )} > {o.Descrizione} </span>
                  </div>
                )}
              </Listbox.Option>
            ))}
          </Listbox.Options>
        </Transition>
      </div>
    </Listbox>
  );
};

export default SelectSector;
```

Figure 4.8: itergration headlessUI to developing dropdown

Initially it fetch data from redux by utilizing the 'useSelector' hook enabling the extraction of data from the redux store and saved it to Sectors array. Later in jsx, a mapping function iterates over each sector object of this array, which conditionally rendered if the Sectors array exists. Subsequently, it employs the 'useState' hook to manage the currently selected sector, initializing its default value with 'seleziona'.

In jsx, the *value* prop is set to *SelectSector*, that holding the current value of the state and the callback function 'onChange' set to the *'setSelectedSector'* that when a new value is selected from the dropdown, this function will be invoked with the updated value and displays the new value.

The *'Listbox.Button'* is rendered which represents the button that triggers the opening and closing of the dropdown list with a dynamic expression *{SelectedSector.Descrizione || SelectedSector}*. It displays the description of the selected sector if it exists, otherwise default value 'Seleziona' will be appear. A Transition effect is applied to dropdown to animate the appearance and disappearance of elements.

The *'className'* prop defined at the top of this page, is set using a function that takes an object with an *'active'* property and returns a string of classes using the *'classNames'* utility function. This function commonly used in react projects to conditionally apply CSS classes to element. At this code it is applied based on whether the option is 'active' or not.
There's also a nested function that takes the *selected* and *active* properties as arguments and it renders the *sector description*. The option's appearance and behavior are determined by the applied classes and the provided description.

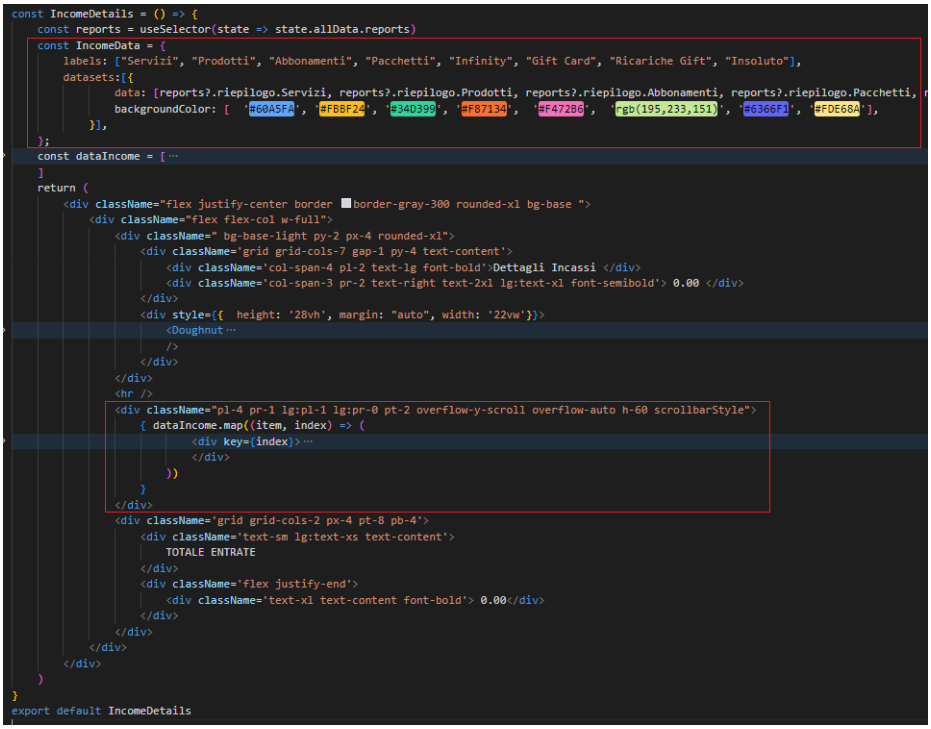
Implementing individual charts

Each segment or chart is implemented individually. Figure 4.9 illustrates one chart development the 'Dettagli Incassi,' as an example.

After importing the chart from the *react-chart-js2*, we proceeded to develop this part by creating data object. The data first retrieve from the redux store using the 'useSelector' hook and saving into "reports".
Then these data assigned to the *'datasets'* property of *'IncomeData'* object which is contains the actual data and background colors for each category.

To development the second part below the chart which displays information related to these datasets, a *'dataIncome'* array is defined, containing details for each income category, including properties such as category name, total cash, total proposals, and a mapping function is employed to iterate through the data set *'dataIncome'* rendered these items.

Inside the JSX code the chart is rendered within a div with inline styles to control its height, margin and width. It receives also the *IncomeData* object as the data prop mapping through this array.



```
const IncomeDetails = () => {
    const reports = useSelector(state => state.allData.reports)
    const IncomeData = {
        labels: ["Servizi", "Prodotti", "Abbonamenti", "Pacchetti", "Infinity", "Gift Card", "Ricariche Gift", "Insoluto"],
        datasets:[{
                data: [reports?.riepilogo.Servizi, reports?.riepilogo.Prodotti, reports?.riepilogo.Abbonamenti, reports?.riepilogo.Pacchetti,
                backgroundColor: [  '#60A5FA', '#FBBF24', '#34D399', '#F87134',  '#F472B6',  'rgb(195,233,151)', '#6366F1', '#FDE68A'],
            }],
    };
    const dataIncome = [···
    ]
    return (
        <div className="flex justify-center border border-gray-300 rounded-xl bg-base ">
            <div className="flex flex-col w-full">
                <div className=" bg-base-light py-2 px-4 rounded-xl">
                    <div className='grid grid-cols-7 gap-1 py-4 text-content'>
                        <div className='col-span-4 pl-2 text-lg font-bold'>Dettagli Incassi </div>
                        <div className='col-span-3 pr-2 text-right text-2xl lg:text-xl font-semibold'> 0.00 </div>
                    </div>
                    <div style={{  height: '28vh', margin: "auto", width: '22vw'}}>
                        <Doughnut ···
                        />
                    </div>
                </div>
                <hr />
                <div className="pl-4 pr-1 lg:pl-1 lg:pr-0 pt-2 overflow-y-scroll overflow-auto h-60 scrollbarStyle">
                    { dataIncome.map((item, index) => (
                            <div key={index}>···
                            </div>
                        ))
                    }
                </div>
                <div className='grid grid-cols-2 px-4 pt-8 pb-4'>
                    <div className='text-sm lg:text-xs text-content'>
                        TOTALE ENTRATE
                    </div>
                    <div className='flex justify-end'>
                        <div className='text-xl text-content font-bold'> 0.00</div>
                    </div>
                </div>
            </div>
        </div>
    )
}
export default IncomeDetails
```

Figure 4.9: developing a single chart *'IncomeData'*

Fig 4.10 displays more details of this chart component Doughnut configuration.

```
<Doughnut
        data={IncomeData}
        options={{
                maintainAspectRatio: false,
                layout: {
                        padding: { top: 5, left: 5,right: 5, bottom: 5 },
                },
                plugins: {
                        legend: {
                        display: true,
                        position: 'right',
                        labels: {
                                boxWidth: 10,
                                font: { size: 11 }
                        }
                }
          },
        }}
      />
```

Figure 4.10: Doughnut chart configuration

It is important ensure that the data points are numerical values, this allows Chart.js

to sum all the numbers and calculate the relative proportions accordingly.
The report page *'Fatturato Generale'* follows the same approach, however to developing reports in *'Produttività Collaborati'*, we employ different chart type.

Developing 'Produttività Collaborati' report page

This page is developed using Bar charts, which excel at precise value measurement and are well-suited for comparing multiple data sets across different categories compared to doughnut charts. This chart like the other charts, offers various customizable properties, like *'backgroundColor'*, *'borderRadius'*.
This report page features also a doughnut chart, to displaying the percentage of total sales by each colleague.
An additional requested feature for this report page, was the ability to hide two specific charts by clicking a button located at the top of the page. This functionality is particularly useful for data privacy, allowing users to hide specific charts in the presence of colleagues or customers.
The charts created for this report page are categorized and placed into two components. The charts that will be hidden are written within the *'HideReports'* component, while the other charts are written in individual components and then imported into *'Productivity'* component as their main components.

Fig 4.11 illustrates a Bar chart with multiple data sets named *'Fatturato Collaboratori'*. The button located at the top of the page "mostra tutto" has responsability of hide/show specified charts.

Figure 4.11: Bar chart multiple datasets

To implementing this report page, a main component 'Collaborators' is created where using a useState hook manages charts visibility defining a variable 'showAll'

with initialized value to 'false', meaning do not demonstrate the charts of 'HideReport' as default. *const [showAll, setShowAll]= React.useState(false).*
The Fig 4.12 illustrate structure of main page.

```
const Collaborators = () => {
  const [showAll, setShowAll] = React.useState(false)
  return (
        <div>
              <div>
                <button className='....' onClick={() => setShowAll(!showAll)}>
                      <span className='mr-3'>mostra tutti</span>
                       {!showAll ? <AiOutlineEye className="text-lg" /> : <AiOutlineEyeInvisible className="text-lg" />}
                </button>
              </div>
              {showAll?
                (<div>
                      <HideReports/>
                        <div>
                            .... import  Date Picker and Selector components here ....
                        </div>
                      <Productivity/>

               ):(
                      <div>
                          .... import  Date Picker and Selector components here ....
                       </div>
                      <Productivity/>
              )}
        </div>
        )
}
export default Collaborators
```

Figure 4.12: Developing 'Produttività Collaboratori' main page

The return statement defines the structure and layout of the component's UI. The button 'mostra tutti' toggles the value of showAll, when clicked. The icon will changed within a conditional rendering logic based on the value of *'showAll'*. If its value is true, all report components are rendered, otherwise only the 'Productivity' component is rendered which includes a defined number of reports component.

The Fig 4.13 demonstrates implementation of Doughnut chart of this page for the report 'Fattuarto Totale'. The chart represents the report of percentage of total sales attributed by different collaborators, along with corresponding average values of each.

A function to generate *random colors* is implemented and applied to to this chart to managed to created color beacuse the number of collaborators can be vary at different beauty center.

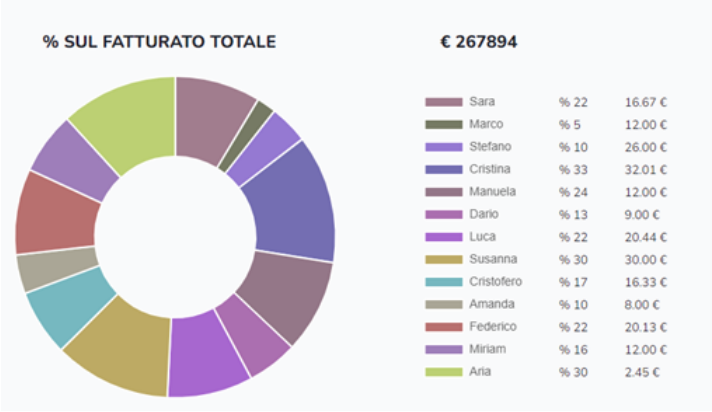


Figure 4.13: Produttività Collaboratori user interface

The Fig 4.14 demonstrates development of this chart. This part is implemented by defining an object which is contains an array of *'labels'* representing the names of collaborators and the *'dataset' TotSalesChartData*.

```
import { Doughnut } from "react-chartjs-2"

var DataTotalsale = [22, 5, 10, 33, 24, 13, 22, 30, 17, 10, 22, 16, 30];
var randomColorTotalSale = [];
var DataTotalSaleLength = DataTotalsale.length;
let i = 0;
while (i <= DataTotalSaleLength) {···
};
const TotSalesChartData = {
    labels: ["Sara", "Marco", "Stefano", "Cristina", "Manuela", "Dario", "Luca", "Susanna", "Cristofero", "Amanda", "Federico", "Miriam", "Aria"],
    datasets: [{
        data: DataTotalsale,
        backgroundColor: [...randomColorTotalSale]
    }],
};
const TotSaleDataArray = [···
]
const PercentOnTotal = () => {
    return (
        <div className='grid grid-cols-5 gap-4 bg-base-light px-4 my-4'>
            <div className="col-span-2">
                <div className='grid grid-cols-4 text-xl text-content font-extrabold'>
                    <div className='col-span-3 text-center pb-6'> % SUL FATTURATO TOTALE </div>
                    <div className='col-span-1 text-right pr-4 font-extrabold'>€ 267894 </div>
                </div>
                <div style={{ height: '40vh', margin: "auto"}}>
                    <Doughnut
                        data={TotSalesChartData}
                        options={{···
                        }}
                    />
                </div>
            </div>
            <div className="col-span-2 mt-16 pt-2"> {
                TotSaleDataArray.map((item, index) => (
                    <div className='flex' key={index}>
                        <div className=' text-content text-sm mt-1.4 mb-1 ml-4 pt-0.5 w-15'> {item.percent !== null ? item.percent : 0.00}</div>
                        <div className=' text-content text-sm mt-1.4 mb-1 ml-4 pt-0.5'> {item.media !== null ? item.media.toFixed(2) : 0.00} € </div>
                    </div>
                ))
            }
            </div>
        </div>
    )
}
export default PercentOnTotal
```

Figure 4.14: Developing 'percentage sul fatturato totale'

The dataset represnts 'data' getting from *DataTotalsale* array and the 'backgroundColor' getting from *randomColorTotalSale* array were stored the generated colors.

The spread operator (...) is employed to generate a new array by spreading the elements of the *'randomColorTotalSale'* array into the *backgroundColor* property of the datasets object. This ensures that the backgroundColor property of the datasets object is populated with a fresh array containing individual colors.
Inside the JSX the chart is rendered within a div with inline styles and receives the *'TotSalesChartData'* object as the data prop.

Random colors generation is written inside a while loop. The 'graphBackground' variable is created by concatenating the random RGB values into a string format: *"rgb(randomR, randomG, randomB)"* and then it is pushed into the 'randomColorTotalSale' array, which then assigned to 'backgroundColor' property of *TotSalesChartData* object. This object is the data assigned to the chart. The development of random color is demonstrated at the Fig 4.15.

```
var randomColorTotalSale = [];
var DataTotalSaleLength = DataTotalsale.length;
let i = 0;
while (i <= DataTotalSaleLength) {
        var randomR = Math.floor((Math.random() * 100) + 100);
        var randomG = Math.floor((Math.random() * 120) + 100);
        var randomB = Math.floor((Math.random() * 120) + 100);
        var graphBackground = "rgb("
                + randomR + ", "
                + randomG + ", "
                + randomB + ")";
        randomColorTotalSale.push(graphBackground);
        i++;
};
```

Figure 4.15: Implementation of random color generation

4.3.3 Landing page

One of the required features for the web version was the implementation of a landing page, which is an additional feature compared to the desktop version.

This feature allows users of application to create static html page within texts, images, and information for publishing, without needing go thorough external landing page builder applications. They often feature product descriptions and calls to action to encourage visitors to make a purchase. Some landing pages creates to promote and facilitate event registration. Some characteristics of effective landing pages include:

1. Simplicity: They usually very focused and have a clear, unambiguous purpose.
2. Persuasive Content: its content including headlines, images, and text, are usually highly compelling and tailored to the target audience.
3. Clear Message: visitors will immediately understand what the page is about and what action is expected from them.
4. Mobile Responsiveness: given the diverse devices visitors might use, landing pages should be responsive and display well on mobile devices.

To develop a landing page, we need:

1. Define the goal: determine the primary goal of our landing page. At this application it is implemented to create advertisements for specific services or products, representing special promotions and events.
2. Know our audience: Nuage users are the person who works at beauty center. The page should be design as clear as possible and users be able to build their own page without having a special computer skill.
3. Having a clean and visually appealing design: we should design the page to attract the visitors using high-quality images, videos, and fonts.

This feature is implemented within two components.

1. 'LandingPageList', This component displays a list of all the created landing pages, where they are archived for a specified period and users have the option to edit, delete, or copy the page's link and can access the main "Creator" page (AddLandingPage) through a Plus button form this page.
2. 'AddLandingPage', as the name suggests is where the development phase for creating each individual landing page is presented.

These two components are imported into a main component *'LandingPage'* where using a *useState* hook, it manages the visualization of these two components.

Fig 4.16 demonstrates the implementation of this part. There are a conditional rendering based on the value of *isNewPage* state. If it is true, the *AddLandingPage* component is rendered, otherwise the *LandingPageList* is rendered. The outer div contains another conditional rendering, indicating when the *isNewPage* state is False, the *LandingPageList* should be rendered within displaying the 'Plus' icon defined into the *PromotionsNavbar* component. This icon is a button which has functionality of adding a new landing page.

The landing page is located in sidebar under 'Promozioni', going through *Creator* tab, it leads user to the 'LandingPageList' as the default value of the *isNewPage*

Figure 4.16: Landing Page main component development

state sets to False in the main landing page component.

Developing Landing Page List

'LandingPageList' is developed by implementing a table allowing users to view all landing pages created and they stored for a specified period of time. Each row contains information about individual created pages, including id, link, ecc. The 'plus' button at the top of the table leads to open a new page where users can create a new landing page.

Figure 4.17 illustrates the user interface of this component, that is written as a functional component taking two props 'isNewPage' and 'setNewPage' from its parent *LandingPage* and returns a jsx element, which is it container.

It utilizes the *useSelector* hook to extract data from redux, promotions section. We make use of some functions like *useSorableData* and *ToEuropean* at this component, that are imported from utility folder and they are written for the purpose of being used in different parts of the project.

To the icons presented at each row associated an event handler 'onClick' for user interaction with the purpose of *copy* the link, *edit* or *delete* the selected page. We got advantage of *CopyToClipBoard* react component to copy the link of the created page, offering an alternative method for copying text in React. It should be installed via npm, imported into the component, and then used to wrap the copy icon.

To implement the grid, we make use of the Tailwind CSS *'grid-cols'* functionality to organize the page into columns. The container consists of two main sections

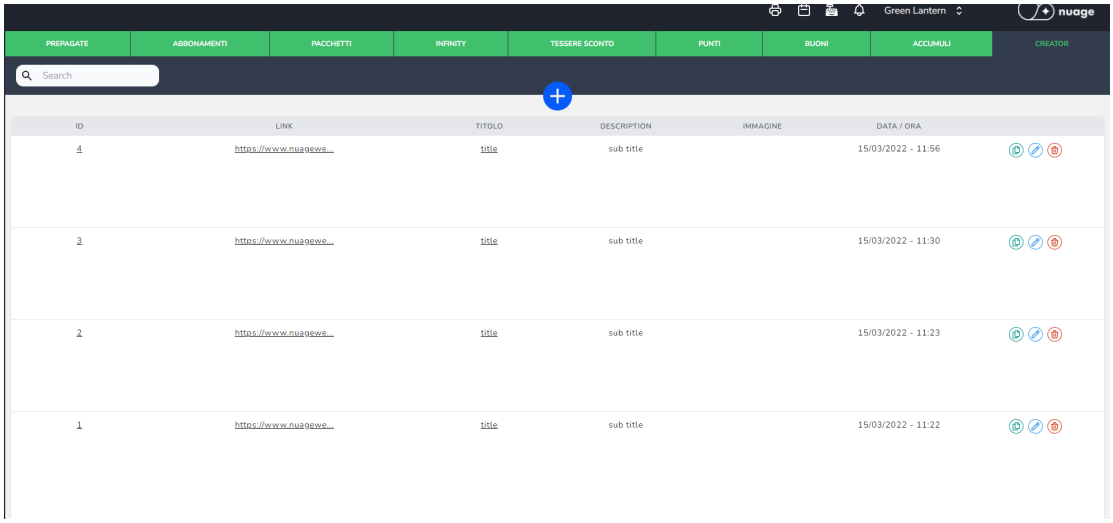


Figure 4.17: User interface of LandingpageList

header and *content*. Inside the content section, we map through an array called 'items' rendering each item from the array as a row in the grid.

Developing Creator Page

This page has evolved into the 'AddLandingPage' component, where the development phase for creating individual landing pages takes place.
The page consists of three parts: a *header* containing buttons used for navigating back to the previous page, a *form completion* section which serves as the skeleton of page, and a *preview* section that synchronizes with the form section, updating the preview of created page in real-time as the user fills out the form.

In the form section a limited number of blocks are provided for creating a landing page with the possibility to leave some of blocks unused. By default, all the blocks are in an 'active' state to make them visible to users. The visibility of each block can be managed by clicking on the eye button (nascondi/visualizza) positioned at the top of each block.

At Fig 4.18 displays the user interface of 'Creator' page, where the preview of the landing page is synchronously visible to user once he/she inserted title 'Test Create landing' and company name 'xnew srl'. This functionality is implemented for all blocks, allowing users to check the results of the created page immediately in the preview section.

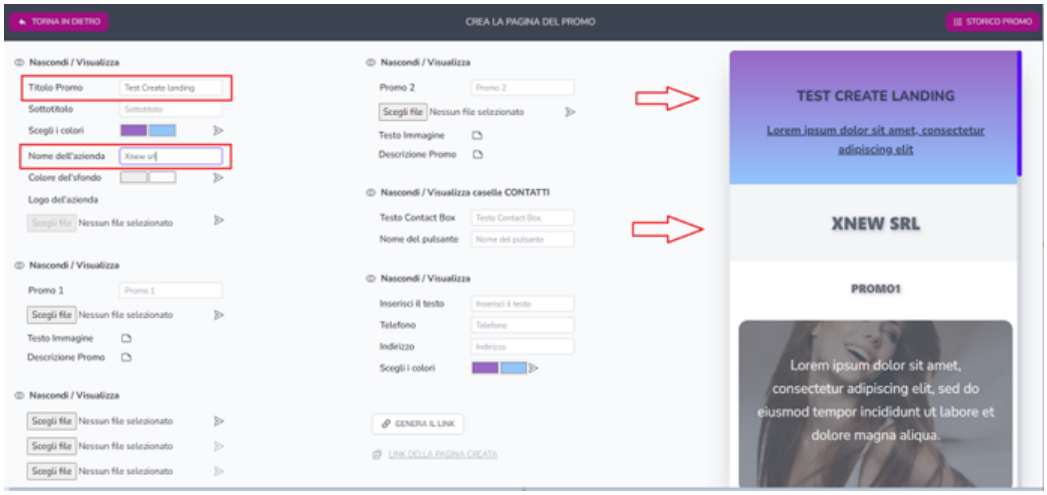


Figure 4.18: Form completion at Creator page with synch preview

The Creator's main component development is demonstrates at Fig 4.19. It imported the *GetInputAddLanding* and *Preview* components and renders the user interface.

The main content is structured within a two columns layout. The left column houses the 'GetInputAddLanding' component where developed 'form completion' and the right column features the 'Preview' component.
Props are passed to these two components to manage content and state for their respective sections. The 'GetInputAddLanding' is dedicated to the form section and is responsible for rendering input fields and handling user input. It receives various state variables and setter functions as props, since the 'Preview' component is responsible to displaying a preview of the block contents.

The *useState* hook managed various state variables such as: *carouselImg, ImgPromo1* which hold the data for different images and *fields* and *setFiels* is used to store texts data. The *showComponent* keeps track of which components to show in the preview section.

Utilization of Formik library with landing page

As mentioned previously 'GetInputAddLanding' component is dedicated to form implementation. At this component after importing various dependencies defines

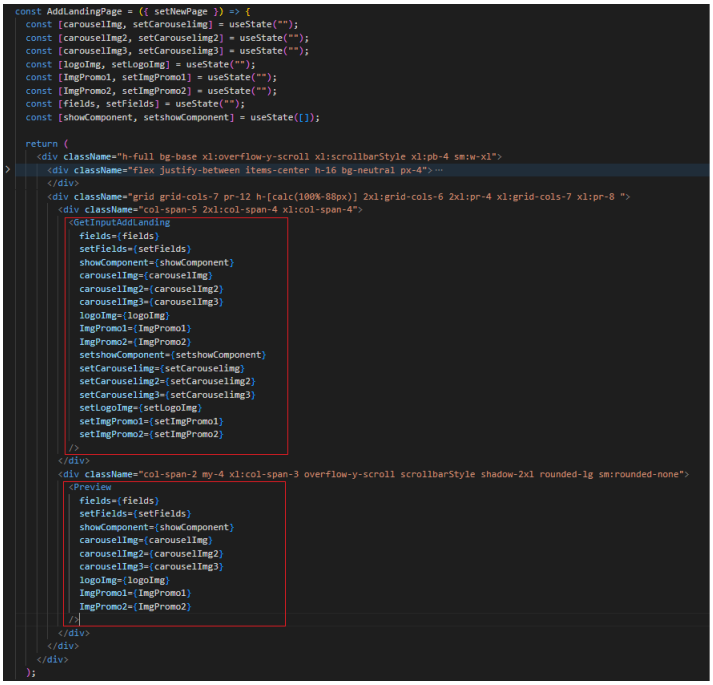


Figure 4.19: Developing Creator main page

states to manage various aspects of the landing page configuration and also some event handler functions such as *'handleClick', 'handleCarousel1'* that triggered when certain events occur. We get advantage of *Cloudinary* service to upload images, that is an image and video management services and enables users to upload, store and deliver images for websites and applications. [35].

To developing this component we utilizes the 'Formik' library to manage form state and form submission. It wraps the form fields and handles form validation. The form is divided into sections based on different components of the landing page, include *Header, Promo1 , Carousel, Promo2, Contact Box and Footer*. As I explained in chapter 3, a skeleton for each Formik type is creted. This involved importing Formik components, such as *TextField* from the Formik folder.

The Fig 4.20 demonstrates a part of the developing of this component. The *'handleSubmit'* function referenced in the onSubmit of Formik component will be call when the form is submitted, passing the form values as the argument. The *values, setFieldValue* props is defined that provide access to the current form values and update individual field values.

The *initialValues* props passes to the 'name' attribute associated to each item. The

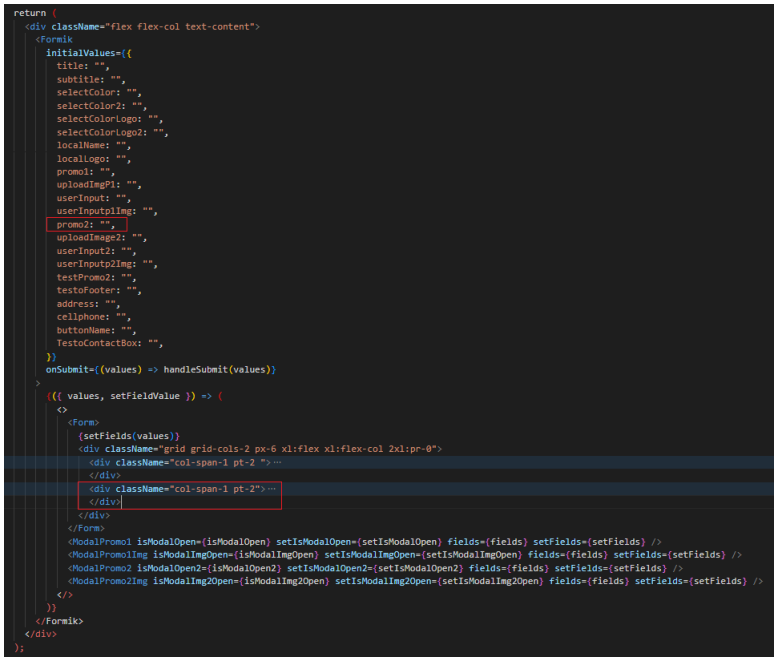


Figure 4.20: Formik implementation and importing Modals at 'Creator'

code renders components such as header, promo1, ecc, conditionally based on the value of *showComponent* props defined at the beginning of the code.

The Fig 4.21 illustrates how 'promo2' takes the initial value by name attribute. There are other attributes defined such as lable, type and a css class to be applied to the input field. The code checks if "Promo2" is included in the *showComponent* array, the section should be shown, a button with the class eyeStyle is rendered and clicking on it triggers the removal of "Promo2" from the showComponent array using the *setshowComponent* function and if it is not included section will hide with changing eye icon.

The Modal components shown at the bottom of the image are provided for writing longer texts where we need to have multiple lines of text, such as descriptions in the form section. They are written as separated components, imported to this file and have default values *false*. An onClick event handler assigning to them, set *isModalOpen* to True when clicked and triggers the opening of modals.

The Fig 4.22 demonstrates the object 'staticMarkup' is created that contains key-value pairs and include a code generating a static HTML markup. The values are

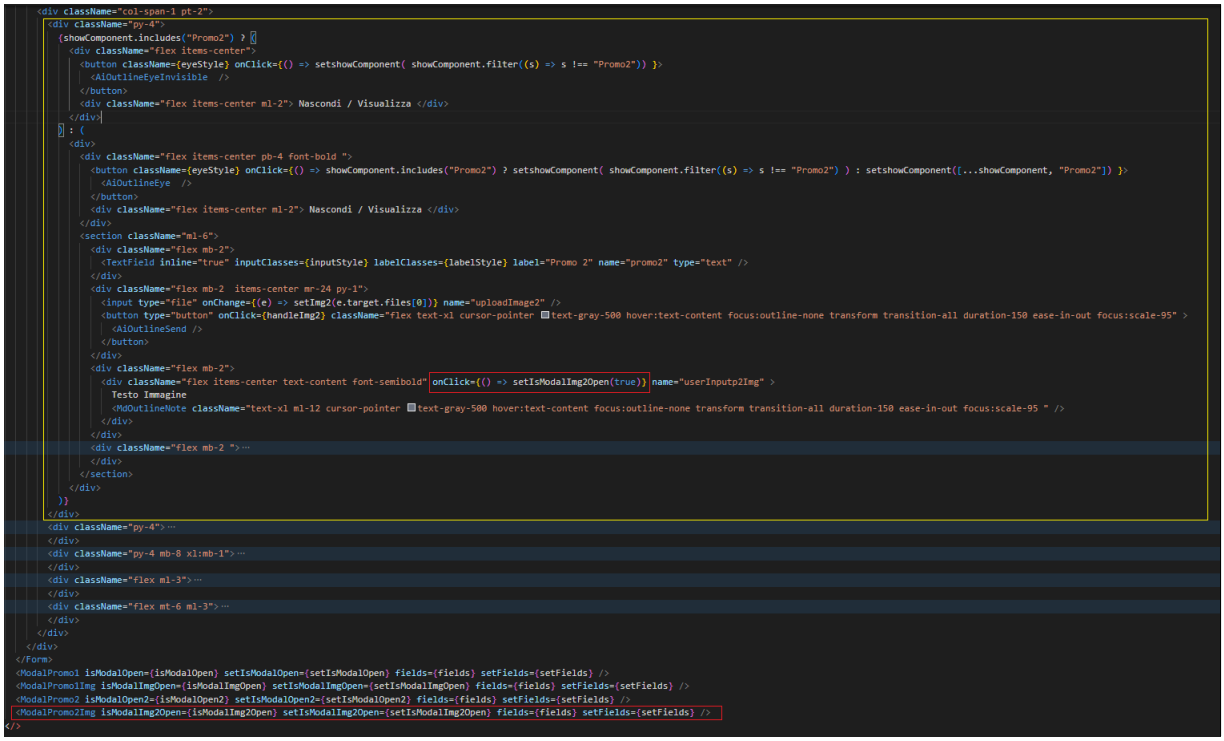


Figure 4.21: Conditional rendering logic for showing or hiding a block

derived from the fields object that receives as a prop and other variables representing data to be sending to the api within Post request.
The form submission is handled through an asynchronous function . It constructs a request object, specifying the Http method as POST setting the "content-type" header to "application/json" and includes the staticMarkup object as the request body. The code then makes an Http Post request using the api.post method (provided by Axios). When it receives the response, it checks the http status code. If the status code is 201 or 200, it sets the link state variable with the value of the "Link" property from the response data. If the status code is different, it logs an error message.

```
const staticMarkup = {
    Html: ReactDOMServer.renderToStaticMarkup(landingPage),
    Title: fields?.title,
    Subtitle: fields?.subtitle,
    ...
}
```

Figure 4.22: Static Markup object defining to hold the data

Preview component development

The 'Preview' component is responsible for displaying blocks. Separate components are created for each block and then imported into the main preview component. Preview receives several props, including *fields, setFields, showComponent* and image-related props. It conditionally renders different child components based on the values received from the *showComponent* prop, that is defined in the Creator component and passed down to control which components are displayed within preview.

In Fig 4.23 you can see the development of the 'Preview' component, where illustrates how each child component is wrapped inside a conditional statement that checks if component should rendered or not. The props are passed to each child component based on the props received by the Preview component and their appearance is determined by the provided data.

```
import ContactBox from "./ContactBox";
import FooterG from "./FooterG";
import CarouselG from "./CarouselG";
//import CarouselGtest from './CarouselGtest'
import Promo1 from "./Promo1";
import Promo2 from "./Promo2";
import HeaderG from "./HeaderG";

const Preview = ({ fields, setFields, showComponent, carouselImg, carouselImg2, carouselImg3, logoImg, ImgPromo1, ImgPromo2,}) => {
  return (
    <div className="flex justify-center text-center shadow-2xl bg-base-light rounded-lg sm:rounded-none touch-auto ">
      <div className="w-full h-full">
        {!showComponent.includes("Header") ? (
          <HeaderG fields={fields} setFields={setFields} logoImg={logoImg} />
        ) : null}
        {!showComponent.includes("Promo1") ? (
          <Promo1 fields={fields} setFields={setFields} ImgPromo1={ImgPromo1} />
        ) : null}
        {!showComponent.includes("Carusele") ? (
          <CarouselG
            carouselImg={carouselImg}
            carouselImg2={carouselImg2}
            carouselImg3={carouselImg3}
          />
        ) : null}
        {!showComponent.includes("Promo2") ? (
          <Promo2 fields={fields} setFields={setFields} ImgPromo2={ImgPromo2} />
        ) : null}
        {!showComponent.includes("Conatctbox") ? (
          <ContactBox fields={fields} setFields={setFields} />
        ) : null}

        {!showComponent.includes("Footer") ? (
          <FooterG fields={fields} setFields={setFields} />
        ) : null}
      </div>
    </div>
  );
};

export default Preview;
```

Figure 4.23: Preview main component development

The Fig 4.24 demonstrates development of one block, Promo1 component that is imported to Preview. This component after receiving props such as *fields, setFields,*

ImgPromo1 conditionally renders the user interface.
The content of the this block is dynamic and based on the props received it displays a title, an image, and a description. There are provided default value for each option, if there are not any user input the default will be displays.
For example for displaying image, the code checks If 'ImgPromo1' prop is provided, it uses the image specified in ImgPromo1, otherwise it uses the default image specified in 'ImgP1'. The the *Description* gets the value of 'userInput' associated with 'name' attributes which is defined at 'GetInputAddLanding' component. Users

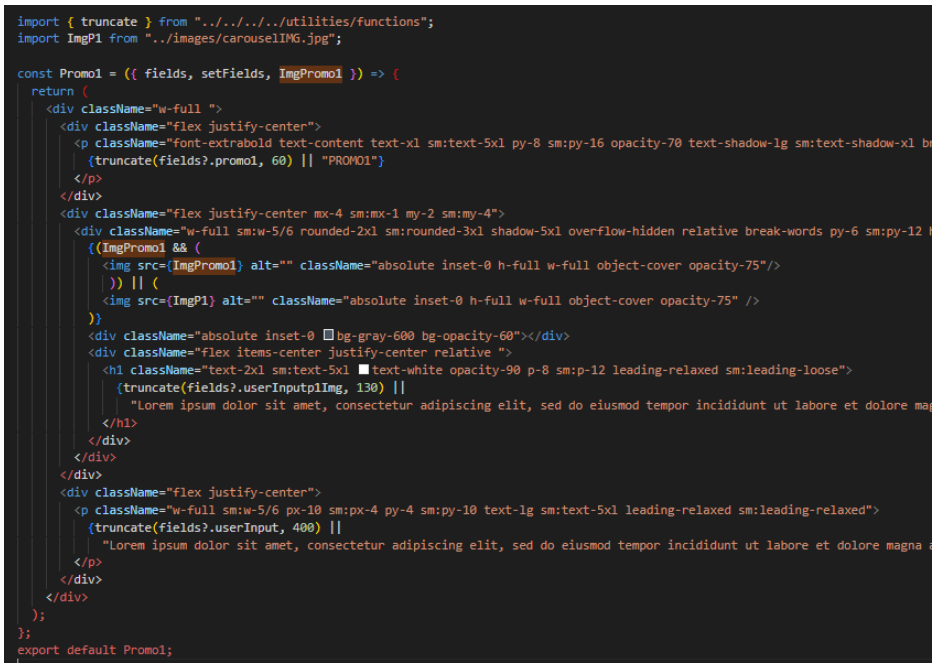


Figure 4.24: Preview implementation, one block development

have the option to toggle the visibility of different components within the interface. This can be achieved through a button located at the top of each block, which is represented by an eye icon.

The user interface for this functionality is depicted in Figure 4.25. As shown in the image, users have the ability to select the visibility option for a specific component, in this case, 'Promo 2,' while keeping all other components hidden.

An 'onClick' function is used to manage the opening of a modal for writing the text on image and the 'Description.' When a user clicks on the corresponding icons, this function is activated, enabling users to input text for these elements within the modal.

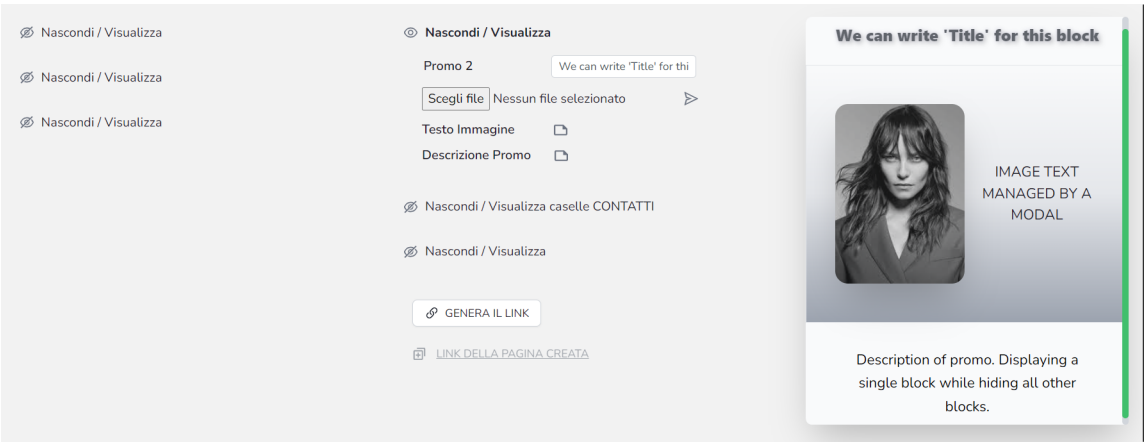
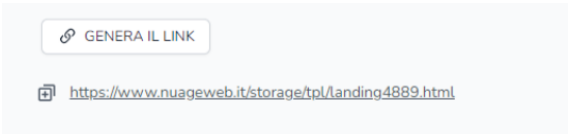


Figure 4.25: Displaying a single block and hiding all other blocks

The button is positioned at the bottom of creator page will generates the link of html page once user finished creating the desired page. In this way clicking on this button, the link of landing page will be generated.



By clicking on the link we can go to the advertise page created. The Fig 4.26 is demonstrates the user interface of generated page.

Figure 4.26: Generated html landing page

4.3.4 Impostazioni

Nuage provides a 'Settings' section which can be reach from the sidebar. Settings pages offer users with a range of customizable options to personalize their website experience. For example they can configure privacy and security settings, or accessibility options.

For the implementation of 'Impostazioni,' we leveraged the advantages of creating tabbed pages. This approach effectively organizes content into distinct sections, enabling users to seamlessly switch between different sections and significantly enhancing their overall experience.

The Fig 4.27 demonstrates a user interface of page *Impostazioni Azienda* inside the setting section.
This page includes information related to the company which can be updated by user and saved via dedicated button positioned at the end of each page. We used Formik library to developing of this part. The tabs are created at separated components and imported to the main component to build whole setting page.

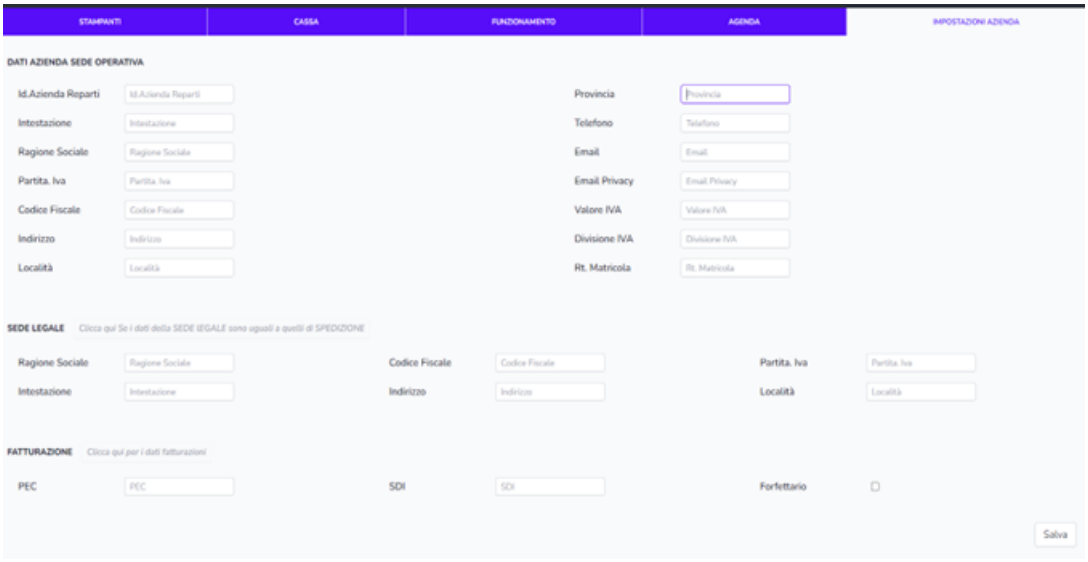


Figure 4.27: Setting page, tab Impostazioni Azienda

At the Fig 4.28 we can see a part of development of the main page where after importing all components to main page, there are conditions to setting the active tab. A *useState* is defined to manages which tab is currently open. It takes a value between 1 to 5 which means the Setting page includes 5 tabs.

When a tab is clicked, the *onClick* handler updates the 'openTab' state to switch to the selected tab's content and consequently the component renders different content based on the currently open tab.

Pages implementation and form validation

The pages are created by importing components such as 'TextField' from the Formik folder. These components are then configured with labels, classes, and names in the TextField tags. The names assigned to these tags are defined in the 'initialValues', where the values for each item are obtained via an API and assigned to them. An 'onSubmit' function presented at the Formik tags which passed the form values.

The input data entered in forms must undergo validation. To handle this validation process, we've imported the 'validations' component from the 'formik' folder. This component is developed and utilized wherever input validation rules are required. We get advantage of Yup for its implementation. It allows creating a schema with multiple validation rules for different fields. The Fig 4.27 shows the

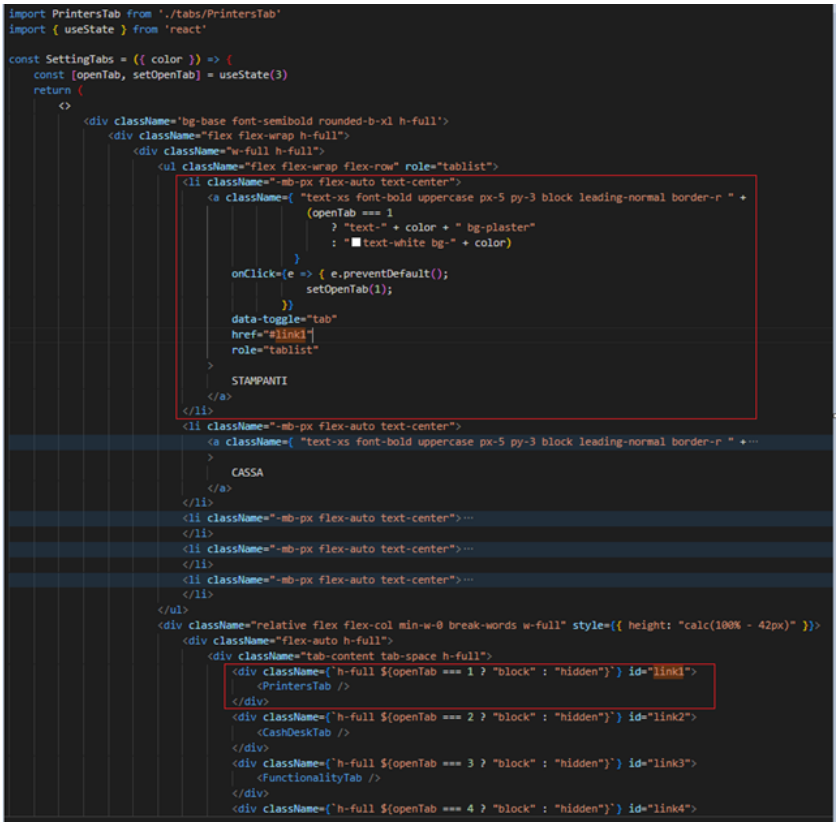


Figure 4.28: main page development of *Impostazioni*

ValidationSchema is get 'newSetting' validation rule from Formik library, where defined almost all schemas.
At chapter 3 detailed how we created schemas and defined the criteria and its requirements at validation component for each part. For example the required rule for the 'cellophane' is having the 10 characters which must be just number, etc. Then we imported these objects to the components at different part of application. The Fig 4.29 demonstrate how we import and used validation schema for setting page.

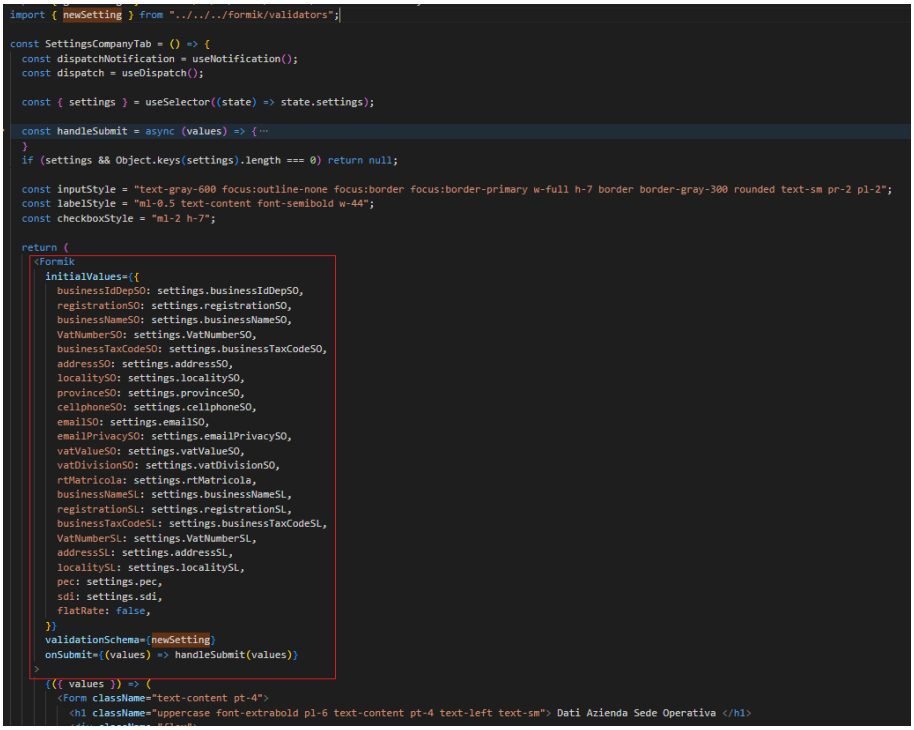


Figure 4.29: Formik, get value from validation component

Chapter 5

Conclusion and Future Works

5.1 Conclusion

In the following sections, i will highlight the most significant contributions of this thesis work and outcomes achieved during its development.

The aim of this project was to develop a web version of desktop application. While working in collaboration with the company, the ultimate design and technological choices underwent revisions and adjustments, but this gives me opportunity to study on some more technologies. This period served as a valuable learning experience and allowed me to gain insights into working with a company and become well-versed in the process of web development and application implementation.

The 'Nuage' web application is already a comprehensive platform, continually growing and evolving to meet the changing needs of its users. Utilization of *Agile* methodologies and user involvement has been crucial to the success of this application, allowing to rapidly respond to user feedback, incorporate new features, and maintain a high level of user satisfaction in the rapidly evolving landscape of modern web applications.

The incorporation of *Tailwind CSS* has significantly influenced and enhanced the implementation of this application, fostering a more efficient and maintainable codebase. The framework's customizable nature has empowered development team to swiftly adapt to evolving design requirements and deliver a visually appealing user interface.

Utilization of *External Library* significantly speed up the development process, as they provide pre-built components and functions. They often come with a wide range of features and functionalities that can enhance application.

The development of a *Landing Page* provides the opportunity to attract new clients for beauty centers. With effective search engine optimization (seo), the landing page not only becomes discoverable in pertinent search results but also serves as a potent tool for attracting new clients within the beauty sector. This functionality can serves as a powerful tool for optimizing marketing campaigns, improving user experience, and achieving better results for businesses.

Creating *Report* pages have an important impact on the usability of the application. Well-designed reports and data visualizations help users understand and interpret data. In particular, by using react-chart-js, complex data can be represented in a structured manner.

In the 'Future Work' section, I have outlined several upcoming improvements planning for the application as it keeps growing. There are many new features and developments in the works, and these upgrades will help make the application even better and more refined over time.

5.2 Future Work

Responsive page

Nowadays having a responsive web site is one of the most important requirements as smartphones and tablets catching up the lifestyle of users. Responsive design allows a website content to flow across all screen resolution and sizes.
Tailwind enables us to create responsive designs just like any other part of our design, utilizing utility classes. we experienced implementing this feature at some part of application and this approach is now set to be implemented across the entire application for a cohesive and responsive user experience.

Every utility in Tailwind is also available in screen-size specific variations. This is done using predefined screen sizes (media query breakpoints), each of them are given a unique name like sm, md, lg and xl. There are five breakpoints by default in Tailwind: 'sm', 'md', 'lg', 'xl', '2xl', which is set for the common device resolutions [35] We can define more customize break-point if we need at *tailwind.config.js*. Writing the responsive with tailwind can be done in an easy way, applying every utility class conditionally at different breakpoints to control the appearance on

specific screen sizes range. For doing that we add the break-point name followed by the ':' character and then the name of utility.
As an example at the main page of 'Creater' we give condition to span different number of columns at different screen size, enable vertical scrolling if needed and round its corners, with the option to disable corner rounding on small screens:
' '

Adding a dashboard

The development phase began with the implementation of a cash desk, primarily due to the absence of a desktop dashboard. For the web version, a dashboard needed to be created from scratch.
A dashboard serves as the primary entry point of an application, providing users with a centralized and organized interface where they can access key information, features, and functionalities. It acts as a control center, offering an overview of relevant data and allowing users to navigate to specific areas or perform various tasks within the application. They play a pivotal role in ensuring users can quickly and effectively engage with the application and its offerings.

Multi language support

The application should supports multiple languages, offering users the convenience of selecting their preferred language when using the application. To achieve this goal 'react-i18next' should be used that is a powerful internationalization framework. We need install all the required dependencies from i18n and create the i18n file for all the configurations. Then creting Json files for each language that we're gonna have in our our app inside a folder named locale. The jsons will have the following structure:

```
{
"translations": {
"key": "Translated Value"
...Other key-value pairs... }
}
```

Bibliography

[1] React documentation authors. *A JavaScript library for building user interfaces)*. https://reactjs.org/. 2022 (cit. on p. 1).

[2] Attlassian documentation authors. *The Agile Coach)*. https://www.atlassian.com/agile/ (cit. on p. 2).

[3] MDN authors. *SPA (Single-page application)*. https://developer.mozilla.org/en-US/docs/Glossary/SPA/. 2021 (cit. on p. 4).

[4] Developedia authors. *SPA (Single Page Application)*. https://devopedia.org/single-page-application. 2020 (cit. on p. 5).

[5] Maneesh Kumar Singh. *Difference between Static and Dynamic Web Pages*. https://www.geeksforgeeks.org/difference-between-static-and-dynamic-web-pages/. 15 Jun,2020 (cit. on p. 6).

[6] RedHat authors. *What is a REST API?* https://www.redhat.com/en/topics/api/what-is-a-rest-api. 8 May,2020 (cit. on p. 7).

[7] Gabriel Gitonga. *Rest Architecrure*. https://www.linkedin.com/pulse/understanding-rest-architecture-gabriel-gitonga. 5 Jun,2021 (cit. on p. 7).

[8] Atlassian authors. *What is a Agile?* https://www.atlassian.com/agile (cit. on p. 8).

[9] Maneesh Kumar Singh. *webpack concepts-webpack documentation*. https://webpack.js.org/concepts/. 2020 (cit. on p. 8).

[10] website authors. *Frontend Framework*. https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/ (cit. on p. 10).

[11] npm trend collaboratos. *Angular VS React vs Vue*. https://npmtrends.com/angular-vs-react-vs-vue (cit. on p. 10).

[12] react documentation authors. *Getting started of react*. https://reactjs.org/docs/getting-started.html. 2022 (cit. on p. 10).

[13] Ari Lerner. *30 days of React - an introduction to react*. https://www.newline.co/fullstack-react/30-days-of-react/ (cit. on p. 11).

[14] authors of W3school. *React Tutorial*. https://www.w3schools.com/react/react_jsx.asp (cit. on p. 11).

[15] react documentation authors. *ReactDOM*. https://reactjs.org/docs/react-dom.html. 2022 (cit. on p. 11).

[16] Code Academy authors. *Virtual DOM*. https://www.codecademy.com/article/react-virtual-dom (cit. on p. 11).

[17] REACT ROUTER authors. *Primary Components*. https://v5.reactrouter.com/web/guides/primary-components (cit. on p. 12).

[18] Cem Eygi. *React.js for Beginners-Props and State Explained*. https://www.freecodecamp.org/news/react-js-for-beginners-props-state-explained/. Feb, 2020 (cit. on p. 13).

[19] Ohans Emmanuel. *React lifecycle methods: An approachable tutorial with examples*. https://blog.logrocket.com/react-lifecycle-methods-tutorial-examples/. Apr, 2021 (cit. on p. 14).

[20] reactJS doc authors. *Hooks at a glance*. https://reactjs.org/docs/hooks-overview.html (cit. on p. 14).

[21] Esteban Herrera. *State in React: A complete guide*. https://blog.logrocket.com/a-guide-to-usestate-in-react-ecb9952e406c/. Dec, 2020 (cit. on p. 14).

[22] Ihechikara Vincent Abba. *React Hooks Tutorial – useState, useEffect, and How to Create Custom Hooks*. https://www.freecodecamp.org/news/introduction-to-react-hooks/. Oct, 2021 (cit. on p. 15).

[23] Obinna Ekwuno. *Tailwind CSS vs. Bootstrap: Is it time to ditch UI kits?* https://blog.logrocket.com/tailwind-css-vs-bootstrap-ui-kits//. July, 2021 (cit. on p. 18).

[24] tailwindcss authors. *Installation - integratio guides*. https://v2.tailwindcss.com/docs/installation (cit. on p. 18).

[25] tailwindLABS authors. *Completely unstyled, fully accessible UI components, designed to integrate beautifully with Tailwind CSS*. https://headlessui.com/ (cit. on p. 20).

[26] Techzaion Blog authors. *Validation with Yup*. https://www.techzaion.com/validation-with-yup (cit. on p. 22).

[27] Redux Documentation authors. *Redux Essentials, Part 1: Redux Overview and Concepts*. https://redux.js.org/tutorials/essentials/part-1-overview-concepts (cit. on p. 22).

[28] Alex Bachuk. *Redux - an introduction*. https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/ (cit. on p. 23).

[29] @archnabhardwaj. *explain Reducers in Redux*. https://www.geeksforgeeks.org/explain-reducers-in-redux/. Feb, 2022 (cit. on p. 23).

[30] verma$_a$*nushka*. *Difference between Fetch and Axios.js for making http requests*. https://www.geeksforgeeks.org/difference-between-fetch-and-axios-js-for-making-http-requests/. Jul, 2021 (cit. on p. 25).

[31] authors of Axios-Http.com. *Getting Start- what is Axios*. https://axios-http.com/docs/intro (cit. on p. 25).

[32] authors of reactJS authors of. *File structure*. https://it.reactjs.org/docs/faq-structure.html/ (cit. on p. 30).

[33] Node Js authors authors. *What is Node.js*. https://www.javascripttutorial.net/nodejs-tutorial/what-is-nodejs/ (cit. on p. 31).

[34] Abhishek Singh. *Create-react-app files/folders structure explained*. https://medium.com/@abesingh1/create-react-app-files-folders-structure-explained-df24770f8562/. 14 June, 2020 (cit. on p. 31).

[35] authors of Wikipedia. *Cloudinary*. https://en.wikipedia.org/wiki/Cloudinary/ (cit. on p. 50).