POLITECNICO DI TORINO

Master's Degree Course
in Computer Engineering

Master Degree Thesis

Micro Frontends, Server Components and how these technologies can provide a paradigm shift with architectural changes in modern enterprise web app development

Supervisor
prof. Luca Ardito

Candidate
Davide Borello

Internship Tutor
Luca Corsilli

Accademic Year 2023-2024

Abstract

In the realm of modern web application development, the microservices architecture has significantly transformed back-end systems, offering scalability and maintainability. Correspondingly, the emergence of micro-frontends extends this paradigm to the front-end, catering to the demands of enterprise-scale applications striving for seamless User Experience (UX) while handling extensive data.

React, a prominent JavaScript framework, introduces server-side rendering (SSR) and, more recently, React server components (RSC), combining server-centric approaches with client-centric interactivity.

The thesis conducts a thorough review of the current state of literature of Micro Frontend Architecture and React Server Components, aiming to integrate these technologies into an already-deployed enterprise web application. The purpose is to provide valuable insights offering guidance for efficient and effective implementation in large-scale web applications.

The study begins from a comprehensive analysis of Micro Frontend architectural patterns the work focused on the study of the concept of Module Federation. Module Federation is a key requirement for adopting Micro-Frontends, as it allows for seamless integration and communication between different micro-frontends. To implement Module Federation, the adoption of Vite.js and Webpack is necessary. These tools provide the infrastructure and configuration to enable module sharing and dynamic micro-frontend loading.

Additionally, the thesis delves into the complexities of utilizing React Server Components without native framework support, emphasizing the importance of frameworks like Next.js, Remix.run, or Modern.js, which offer built-in mechanisms such as Server Side Rendering.

The usage of Micro Frontend emerged as a useful solution for a gradual adoption of server rendering practices, modularizing the application without the need for a complete rewrite and reducing challenges and risks associated with a full-scale implementation.

The thesis illustrates a demo application implementing Micro Frontend Architecture showing the most used patterns involving the usage of Vite bundler and its module federation plugin, exposing and sharing components between React micro-apps. Considerations for styling using libraries like Carbon and Tailwind, as well as state management with tools like Recoil, are discussed. Furthermore the demo shows how to integrate server rendered components using the Remix framework.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

In recent years, the microservices concept has revolutionised back-end web application development, providing a scalable and maintainable solution for managing the growing complexity of modern systems. In this context, the micro-frontends concept emerges as a natural extension of the microservices architecture for the front-end side.

Enterprise-scale web applications are required to handle vast amounts of data while maintaining high performance and providing a seamless User Experience (*UX*). React, one of the most popular JavaScript frameworks, recently introduced React Server Components (*RSC*), a technology designed to enhance page load speed and data handling efficiency in large-scale applications that can be considered as a technical evolution of server-side rendering (*SSR*).

This introduction sets the stage for a comprehensive exploration of Micro Frontends and React Server Components and their integration to address the challenges of modern web application development.

1.2 Objectives and purposes

The primary objective of this thesis is to analyze and study in depth the theoretical foundations and practical aspects of two pivotal concepts: Micro Frontends and React Server Components. By thoroughly analyzing their advantages and challenges, the aim is to provide valuable insights into their implications for performance, data management, and user experience. This research further seeks to establish a set of best practices for designing and developing modular, scalable solutions in the front-end realm. As part of this exploration, a critical focus will be placed on understanding the functionality of React Server Components and examining their integration within widely adopted frameworks such as *Next.js* and *Remix.run*. Simultaneously, Micro Frontends will be studied and tested, exploring key implementation tools like the Module Federation plugin offered by Webpack and Vite, enabling the creation of independent builds without interdependencies.

1.3 Thesis Structure

To achieve a comprehensive understanding and effective application of these concepts, this thesis work is structured into three main parts.

The first part provides an in-depth exploration of Micro Frontends, elucidating fundamental concepts, approaches, and practical examples of frameworks and supporting tools. The second part is dedicated to React Server Components, offering insights into their role within web application architectures and practical demonstrations within popular frameworks. The third and final part bridges theory with application, studying an architectural solution capable of integrating Micro Frontends and React Server Components, implementing a demonstration application to provide insights and guidance to adopt the studied concepts within an already existing enterprise web application developed by the AizoOn frontend developer team.

The subsequent chapters unfold this structure, providing a comprehensive journey through theoretical foundations, practical examples, and real-world implementation, contributing valuable knowledge to the field of large-scale web application development.

1.4 The Company

aizoOn is an independent, innovation technology consulting firm that operates globally by adopting a digital-oriented operating model based on trans-disciplinary, agile and iterative, collaborative and open logics, enabling applied innovation.

The thesis work was done by taking some company apps as reference so as to have a practical view of the aspects studied, with the aim of providing guidance for actual implementation in such applications. The adoption of these technologies and practices in the corporate context can bring benefits in terms of performance and operation of the applications developed by aizoOn. Some applications experience slow loading of certain pages due to heavy data loading. This issue can be addressed by implementing appropriate rendering strategies, which are the subject of study in this thesis. Furthermore, the modular design of certain applications, which provide separate or partially separate functions, makes them suitable for implementing the Micro Frontends architecture, also due to the existence of microservices in the backend.

Figure 1.1. aizoOn

Part I

Micro Frontend Architecture

Chapter 2

Introduction to the concept

2.1 Web Architectures

In the realm of web applications, evolution has traced a significant trajectory, transitioning from monolithic architectures to more flexible and scalable solutions, a journey in which the concept of Micro Frontends plays a crucial role. To grasp the full context of this innovation, a thorough examination of diverse web architectures and their evolution is essential. [7]

2.1.1 Monolithic Architecture

Initially, the predominant and widely adopted paradigm in web application development was the monolithic architecture. This architecture involves developing, implementing, and deploying the entire application as a cohesive unit. The primary advantage of this approach lies in its simplicity, featuring a single source code, a unified database, and an integrated structure where all functionalities and components are tightly integrated.

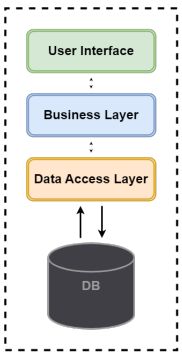

Figure 2.1. Monolith Architecture

However, as web applications grew in size and complexity, the drawbacks of this monolithic approach became increasingly apparent, rendering it inefficient. Managing and expanding a monolithic application became challenging, leading to scalability issues and potential unintended side effects or debugging problems. Consequently, the development of web applications evolved toward more efficient solutions.

2.1.2 Microservices in the Backend

The microservices architecture emerged from the notion that, unlike desktop applications, web applications are not constrained to distribute various components and functionalities as a single entity. Since users receive only the final HTML (what is displayed on the page), the origin that generated the page remains transparent to the user. Consequently, the backend of an application can be divided into multiple fragments, each providing distinct functionalities.

A microservice is an independently developed and deployable unit within the backend architecture of an application. It operates as a standalone software component fulfilling a specific function, based on the concept of *separation of concern*, which involves partitioning the functionalities of the web application into independent, separate, and specialized modules. These modules can be managed and scaled independently, ensuring a high level of resilience and reliability. Additionally, this approach prevents errors in one service from compromising the entire application.

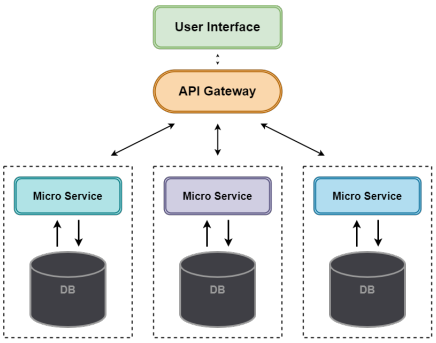

Figure 2.2. Microservices Architecture

Microservices communicate with each other through well-defined interfaces, typically exposing an API. This allows the application to be composed of different programming languages and frameworks, depending on the specific requirements.

Structuring the backend with microservices enables the delegation of implementation, management, and maintenance tasks to independent teams, potentially characterized by technological agnosticism. However, the microservices approach introduces challenges, particularly in managing application complexity, requiring the implementation of coordination, monitoring, and traffic management tools to ensure seamless cooperation among all services.

2.1.3 Micro Frontend Architecture

Concurrently with the evolution in the backend, attention has shifted towards the frontend component of applications, giving rise to the micro frontend architecture as a natural extension of microservices. Similar to microservices, micro frontends are developed, distributed, and executed independently and isolated from the rest of the application.

In contrast to monolithic frontends, where a single team is responsible for the entire frontend development, micro frontends allow for the division of the frontend into microcomponents under the responsibility of different independent development teams. This division can also occur vertically, meaning it encompasses not only the frontend but also a specific feature of the application. In this vertical approach, the responsible team handles the entire structure and layers of that feature, reducing communication overhead between teams and increasing efficiency.

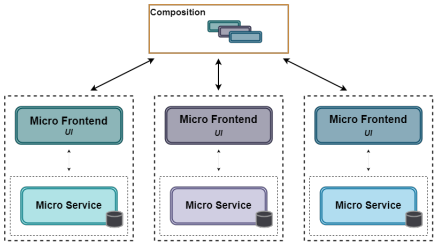

Figure 2.3. Micro Frontends Architecture

The adoption of this architecture brings forth several advantages, yet it also introduces certain challenges, which will be elaborated upon in detail later.

2.2 Advantages and Challenges in Micro Frontend Architecture

In this section, we will elucidate the primary advantages and benefits derived from the adoption of a Micro Frontend architecture, along with the operational complexities and challenges that arise [4]. This analysis is the culmination of a study involving publications, online articles, and evaluations from direct implementations.

2.2.1 Benefits of Micro Frontends

The incorporation of Micro Frontends yields a multitude of significant advantages that revolutionize traditional approaches to web application development and design.

Separation of concerns

The separation of codebases, facilitated by decoupled codebases, promotes feature independence. Development teams can work in isolation, allowing for incremental and isolated

progress. This results in greater error isolation, ensuring robustness, it promotes modularity, maintainability, and easier code management. When combined with automated processes, it accelerates development timelines.

Enhanced Scalability

Scalability becomes a more accessible goal with Micro Frontends, as they enable the management of growth and expansion without the constraints often associated with monolithic architectures. Higher scalability is achieved by allowing independent scaling of specific features or components as needed. This allows for better resource allocation and improved performance under heavy load.

Flexibility in Design and Development

One of the key strengths lies in the flexibility that Micro Frontend approach provides in both design and development. Allowing each development team to adopt diversified approaches, without being constrained by a monolithic structure, fostering innovation and adaptability, leads to a more *framework agnostic* approach, eventually characterized by the use of different tools and frameworks in separated micro frontends.

Orientation Toward Native Browser APIs

Micro Frontends' orientation towards native browser APIs, as opposed to custom APIs, enhances adaptability and integration within the standard web environment. This approach not only ensures compatibility but also simplifies the integration of new features and technologies.

Independent Deployment Approach

The ability for independent deployment allows different parts of the application to evolve autonomously. This facilitates version management and promotes greater flexibility in release strategies. Consequently, there is a reduction in compilation times, simplifying maintenance and fostering the implementation of autonomous teams.

Code Reusability for Efficiency

Micro Frontends architecture facilitates code sharing and re-usage. Standard functionality or components can be developed once and shared across multiple micro frontends, reducing duplication of effort and improving overall development efficiency.

New technologies incremental adoption

Micro Frontends provides the flexibility to adopt new technologies incrementally. Different micro frontends can be built using different technologies, allowing teams to experiment with new frameworks, libraries, or programming languages without impacting the entire application.

2.2.2 Drawbacks of Micro Frontends

The implementation of Micro Frontends, despite the numerous advantages, is not free from challenges and critical issues that require careful evaluation.

Operational Complexity

It emerges as one of the primary challenges. Effectively coordinating diverse components can become intricate, necessitating meticulous planning and management.

User Experience Inconsistencies

The user experience may be susceptible to inconsistencies when different parts of the application operate independently. It can be due to the usage of different technologies, such as libraries or even frameworks, or simply to an incremental update of different independent features. This has the potential to create fractures in the user interface, impacting the overall user experience.

Communication Challenges

Communication, as well as state management, between various components may pose challenges, resulting in response delays and difficulties in maintaining a harmonious data flow. This can lead to performance overheads, as communication management, coupled with fragmented user interfaces, may translate to longer loading times and increased resource requirements.

Development and Testing Complexity

Coordinating Micro Frontends into a cohesive ecosystem introduces additional complexity in development and testing. This complexity highlights concerns related to security and maintainability, urging a thorough evaluation during the implementation phase.

Chapter 3

Approaches and Tools for Development

3.1 Strategies and Usage Patterns

To implement a web application adopting a micro frontends architecture, various approaches are possible. The following sections illustrate the main implementation patterns and the key frameworks and tools studied and used during the thesis work. [4]

3.1.1 Composition

Micro frontends composition can occur at different stages during the application's lifecycle, influencing overall design and performance. The composition process is crucial in the functioning of a micro frontends system, therefore choosing a composition strategy that best suits the needs, using appropriate tools, libraries, and development patterns, is essential. Below, the main composition strategies are presented.

Server Side Composition

Micro frontends composition takes place on the server side before the page is sent to the client. The server is responsible for composing the view, retrieving various micro frontends, and organizing the page by creating a single response sent to the user's browser. Composing the main content on the server reduces initial loading times by avoiding displaying a white screen to the user. However, this approach puts more load on the server, which needs to handle the composition, and provides less flexibility during runtime.

Build Time Integration

Micro frontends composition occurs during the application's build phase. Each micro frontend is developed and deployed independently, but during the build process, modules are combined to form a single package. This approach reduces data transfer to the browser during the build phase and improves distribution ease since the application is

delivered as a single unit. This leads to improved dependency and version management during distribution but increases the difficulty in keeping the update pipelines of different micro frontends separate, as it requires the reconstruction and redistribution of the entire application.

Run Time Composition

Micro frontends composition occurs dynamically on the client side during the application's execution. Following the principle that all micro frontends must be delivered to the browser independently, this provides maximum flexibility during runtime, allowing dynamic updates and loading only when necessary. While this pattern results in longer initial loading times and increased complexity in managing dependencies and interactions between micro frontends, it offers maximum flexibility during execution, allowing dynamic updates. Additionally, the system becomes highly scalable as micro frontends can be distributed and updated independently.

3.1.2 Tools and Patterns

Choosing a composition strategy depends on the specific project requirements and technological constraints. Different strategies are linked to certain technologies, tools, and patterns. During the thesis work, only some of the possible implementation solutions were addressed, considering the known technological stack and prevalent patterns [19, 2]; there are actually multiple solutions and technology models, but the choice to focus on the most diffused ones was made to provide consistency and more orientated outcomes. The main technologies are presented below, maintaining the subdivision into different composition strategies.

Server Side Composition

Opting for Server Side Composition often involves using *NGINX* to manage the composition of micro frontends on the server side.

NGINX is a high-performance, widely-use, open-source web server, reverse proxy server, and load balancer. It is known for its efficiency in handling concurrent connections and managing web traffic. This process involves the orchestration of different micro frontends on the server before the final web page is delivered to the client's browser, including the following steps:

- Request Handling: a user makes a request to the application, NGINX intercepts and handles the request on the server.
- Micro Frontend Retrieval: NGINX communicates with the backend or other relevant services to fetch the necessary micro frontends corresponding to the requested features or components.
- Composition: NGINX combines or composes the retrieved micro frontends into a cohesive view. This can be based on predefined rules, user preferences, or the specific requirements of the application.

- Single Response to Client: Once the composition is complete, NGINX sends a single, fully composed response to the client's browser. From the client's perspective, it receives a unified and fully rendered web page.

This approach provides several advantages such as reduced initial loading times, fewer client-side requests and simplified initial rendering. These benefits need to be carefully weighed against trade-off coming from an increased load on the server, as it takes on the responsibility of composing and delivering fully-rendered pages, or from a reduction of runtime flexibility.

Build Time Integration

Build Time Integration offers various strategies for incorporating micro frontends into an application during the build phase. Let's explore two primary solutions in detail and discuss their practical implementation, advantages, and drawbacks.

- The first one is the Web Component Approach. Web Components are a set of web platform APIs that allow for the creation of custom, reusable elements with encapsulated functionality. Developers leverage frameworks like React to build independent modules during the build phase. These modules can be either third-party component libraries or custom components developed in-house. Additionally, tools like Bit facilitate sharing and managing components across projects, allowing teams to create, organize, and share components independently, thereby streamlining development workflows. This provides modularity and reusability along with framework agnosticism. Web Components are not without disadvantages, in fact although cross-browser compatibility is offered, it potentially adds complexity to the development process. Furthermore, seamless integration with certain frameworks may require additional configuration and setup.

- The other solution is the Monorepo Approach. A monorepo is a single repository that contains multiple projects or components, each managed separately and combined during the build phase. In this approach, all micro frontend projects or components reside within a single repository, allowing for centralized code management and version control. During the build phase, these projects are combined to form a cohesive application. Having all micro frontend projects in a single repository simplifies code management, version control, and collaboration among developers. Combining them during the build phase reduces complexity and dependencies, resulting in faster build times and smoother deployment workflows. Moreover the projects can leverage shared configurations, dependencies, and tooling, promoting consistency and standardization. Nevertheless, as the application grows, managing multiple projects within a single repository can lead to increased complexity in code organization, dependency management, and build configurations. The centralization of code management can limit the independence of individual micro frontends and provide scaling challenges.

Run Time Composition

Run Time Composition stands out as the most commonly employed strategy, valued for its flexibility and effectiveness, particularly in Single Page Applications (SPAs), which dominate the contemporary web landscape.

- One prevalent approach involves integrating micro frontends using *iFrames*, HTML documents that can be seamlessly embedded within another document using the `<iframe>` tag. This method facilitates the creation of composite pages by combining independent components, ensuring a proper level of isolation in terms of global variables and styling without interference. Communication between iFrames can be established through appropriate event bus configuration, allowing different parts of an application to communicate without being aware of which component is listening. Despite its historical use and acceptance, iFrames have notable limitations and drawbacks, especially concerning routing, history, and complex navigation. However, they have served as a traditional solution for micro frontends implementation until the emergence of more modern technologies.
- A widely adopted pattern for runtime composition involves implementing the concept of Module Federation. Module Federation is a technology enabling a JavaScript web application to dynamically load code from another application with shared dependencies. Shared modules across applications can range from simple components to entire features, interpreted as dependencies by the receiving application. Module Federation integrates seamlessly with the build process, organizing and optimizing code bundles while managing module dependencies. Given its efficiency and the provided advantages, this approach has been studied in depth and is explained in detail in the following sections.

3.2 Exploring Technologies for Runtime Composition

As mentioned in the previous sections, among the various patterns and technologies available to implement the micro frontend architecture, only a subset of the illustrated tools have been used, in relation to the known technology stack and the subsequent needs in the company environment.

In particular, priority has been given to run-time composition, as it offers greater flexibility and it currently is the most widely used strategy in the web applications area.

3.2.1 Module Federation

As it was previously anticipated, Module Federation is the main technology used to implement Micro Frontend Architecture.

Module Federation is a cutting-edge technology that revolutionizes code sharing and resource management in web development. It allows to seamlessly integrate multiple JavaScript applications, or micro-frontends, by sharing code and resources dynamically at runtime. This concept of decoupled and independent modules enhances modularity, scalability, and maintainability in complex web projects.

The architecture of a project implementing Module Federation concept typically revolves around a main application, serving as the host and container application, used to aggregate a set of remote independent micro-applications based on requirements. Each remote application can share multiple modules that can include React components, custom elements, library components, or API functions.

The concept of Module Federation was introduced with *Webpack* bundler and recently, also *Vite* build tool started offering the possibility to incorporating this architectural concept.

The following sections will illustrate the main tools used in implementations, referring to a common technology stack that is based on Javascript (or TypeScript) and React framework with related libraries.

3.2.2 Webpack

Webpack is a powerful and versatile module bundler, used in web development to manage and bundle various assets such as JavaScript files, stylesheets, images, and more. It's fundamental for optimizing the delivery of web applications by efficiently bundling and processing these assets for efficient loading and execution in the browser.

At its core, Webpack operates on the principle of dependency graph analysis. This means it analyzes the relationships between different modules in the application and bundles them together into a single output bundle. This process not only helps reduce the number of HTTP requests required to load a webpage but also optimizes the loading process by asynchronously loading modules when needed. [25]

Key features of Webpack include support for loaders and plugins. Loaders allow to process different types of files, transforming them as needed before adding them to the bundle. For example, loaders can compile TypeScript to JavaScript, convert SASS to CSS, or optimize image files. On the other hand, plugins extend Webpack's functionality by performing tasks such as code splitting, minification, and scope hoisting, enabling developers to customize the bundling process according to their specific requirements.

Module Federation was initially introduced with Webpack 5 in 2020. The idea of sharing code between multiple applications was not a new one, but the new feature released with Webpack 5 was a significant step forward in the evolution of code sharing in web development. Initially limited to sharing modules between applications built with Webpack, newer versions expanded support to different module bundlers, as well as tools and plugins to extend its capabilities, including the "Module Federation Plugin" for Next.js. [10]

The implementation of Module Federation extends the Webpack module bundler to support remote loading of modules, using a runtime called the *Module Federation Runtime* to handle the loading and dependency resolution of remote modules.

Developers configure Webpack builds to expose and consume modules across applications through the `webpack.config.js` file, leveraging the `ModuleFederationPlugin` and providing necessary configuration.

Configuration

After having imported it, the Module Federation plugin is added inside the array of plugins and it needs some specific configuration, depending on whether it is the host or remote configuration file.

```
plugins: [
  new ModuleFederationPlugin({
    name: "container",
    remotes: {
      counter: "counter@http://localhost:8081/remoteEntry.js",
    },
    exposes: {},
    shared: {
      ...deps,
      react: {
        singleton: true,
        requiredVersion: deps.react,
      },
      "react-dom": {
        singleton: true,
        requiredVersion: deps["react-dom"],
      },
      recoil: {
        singleton: true,
        requiredVersion: deps["recoil"],
      },
    },
  }),
  new HtmlWebPackPlugin({
    template: "./src/index.html",
  }),
],
```

Figure 3.1. Host App Configuration

```
plugins: [
  new ModuleFederationPlugin({
    name: "counter",
    filename: "remoteEntry.js",
    remotes: {},
    exposes: {
      "./Counter": "./src/components/Counter",
      "./atoms": "./src/atom/atoms",
      "./selectors": "./src/atom/selectors",
    },
    shared: {
      ...deps,
      react: {
        singleton: true,
        requiredVersion: deps.react,
      },
      "react-dom": {
        singleton: true,
        requiredVersion: deps["react-dom"],
      },
      recoil: {
        singleton: true,
        requiredVersion: deps["recoil"],
      },
    },
  }),
  new HtmlWebPackPlugin({
    template: "./src/index.html",
  }),
],
```

Figure 3.2. Remote App Configuration

In both host and remote configuration, the first two properties are used to identify the application after the bundling. The `name` specifies the unique name of the application and the `filename` specifies the filename for the generated bundle; this file acts as the entry point for remote modules.

As seen in the configuration of the host application on the left, figure 3.1, the property `remotes` is used to specify the name of the remote applications to load and their corresponding URLs from which to fetch the entry file. While on the right, figure 3.2, for the configuration of a remote application is necessary to specify inside the `exposes` property, the names, with the relative path to the files, of the modules to expose to others applications.

In the shown example, the host application doesn't expose any modules and the remote one doesn't take modules from other applications; but it's important to consider that an application can be configured to take on both the role of main application and the role of federated micro-app, even at the same time.

Lastly, the `shared` object contains all the libraries and the relative dependencies that are shared among the project, in order to prevent the creation of duplicate instance of the same libraries when loading a remote application. It's possible that a remote application

uses a different version of a library respect to the one used by the host application, this is managed by the federation plugin and it won't cause issue as long as there aren't incompatibilities between newer and older features in a library, in this case it's responsibility of the developer to manually handle the shared libraries.

Usage

```
import { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import { useRecoilState } from "recoil";

import { counterAtom, nameAtom } from "counter/atoms";
import { Counter } from "counter/Counter";

export const Home = () => {
  const [count, setCount] = useRecoilState(counterAtom);
  const [name, setName] = useRecoilState(nameAtom);
  const [newName, setNewName] = useState("");

  const navigate = useNavigate();

  return (
    <>
      <Counter />
      <h1>HOME PAGE</h1>
      <div className="mt-2 ml-2">
        {!name && (
          <>
            <p>Insert your name</p>
            <p className="text-xs">
              It will be saved in a{" "}
              <span className="italic">remote recoil atom</span>
            </p>
            <div className="mt-2 mb-4">
              <p>Name: </p>
```

Figure 3.3. Webpack imports

After the proper configuration, the processes to consume the federated component is straightforward. As shown in the code snippet above, figure 3.3, the modules are imported as traditional imports using the name of the remote assigned in the configuration and the name of the module. Then each module can be used without differences from locally imported components.

In the example above the `Counter` element is imported from the remote and directly used to render the component, similarly to what happen with elements of different types.

3.2.3 Webpack Dynamic Remotes

A more advanced approach allows to declare remotes in a more dynamic style, allowing for a higher flexibility and customization of the whole project and there are several different strategies available [14]:

- Environment variables: it's the most elementary approach and it involves the substitution of `localhost` port (or other address used in production) from the remote URL configuration with environment variables, enabling the possibility to define the remote application's URL as a local or hosted production deployment at build time,

as shown in figure 3.4. This provide flexibility without compromising the simplicity, but it requires to build a new version for each environment to update the URLs.

```
module.exports = (env) => ({
  // ...
  plugins: [
    new ModuleFederationPlugin({
      name: "Host",
      remotes: {
        RemoteA: `RemoteA@${env.A_URL}/remoteEntry.js`,
        RemoteB: `RemoteB@${env.B_URL}/remoteEntry.js`,
      },
    }),
  ],
});
```

Figure 3.4. Configuration with environment variables

- Webpack plugin external-remotes-plugin: Module Federation allows to load remote containers dynamically and this can be done directly or using a plugin. The plugin `external-remotes-plugin`, developed by one of the creators of Module Federation, allows to resolve the URLs at runtime using templating. The URL can be defined inside the `window` object within the application before loading any code from the remote applications, as shown in figure 3.5. This provide the flexibility to define the URLs however wanted, but still it doesn't give complete control over the loading lifecycle.

```
module.exports = () => ({
  // ...
  plugins: [
    new ModuleFederationPlugin({
      name: "Host",
      remotes: {
        RemoteA: "RemoteA@[window.appAUrl]/remoteEntry.js",
        RemoteB: "RemoteB@[window.appBUrl]/remoteEntry.js",
      },
    }),
    new ExternalTemplateRemotesPlugin(),
  ],
});
```

Figure 3.5. Configuration with Plugin

- Promise Based Dynamic: Module Federation allows also to define the remote URLs as promises instead of URL strings, it's possible to use any promise as long as it fits the get/init interface defined by Module Federation. However it's required to have

the remote configuration in string format. Within the promise, a new script tag is created and injected into the DOM to fetch the remote JavaScript file. The figure 3.6 shows an example of using this solution.

```
// Webpack config:
module.exports = {
  // ...
  plugins: [
    new ModuleFederationPlugin({
      name: "Host",
      remotes: {
        RemoteA: `promise new Promise(${fetchRemoteA.toString()})`,
      },
    }),
  ],
};

// Fetch Remote A dynamically:
const fetchRemoteA = (resolve) => {
  // Define a script tag to use the browser for fetching the remoteEntry.js file
  const script = document.createElement("script");
  script.src = window.appAUrl; // This could be defined anywhere
  // When the script is loaded, resolve the promise back to Module Federation
  script.onload = () => {
    // The script is now loaded on window using the name defined within the remote
    const module = {
      get: (request) => window.RemoteA.get(request),
      init: (arg) => {
        try {
          return window.RemoteA.init(arg);
        } catch (e) {
          console.log("Remote A has already been loaded");
        }
      },
    };
    resolve(module);
  }
  // Lastly inject the script tag into the document's head to trigger the script load
  document.head.appendChild(script);
}
```

Figure 3.6. Promise Based Configuration

- Dynamic Remote Containers: the most flexible solution allows to load remote applications programmatically without needing to define any URLs in the Webpack configuration. With this approach a remote module is fetched using a dynamic script tag and then the remote container can be manually initialized. This enable the possibility to add new remote without modifying the configuration on the host application and to inject a module in the deployment of the application. Figure 3.7 shows that this approach does not require a static configuration at all, while an example of dynamic usage is provided in figure 3.8

```
module.exports = (env) => ({
  // ...
  plugins: [
    new ModuleFederationPlugin({
      name: "Host",
      remotes: {},
    }),
  ],
});
```

Figure 3.7. No Static Configuration

```
export const loadComponent =
  (remoteName, remoteUrl, moduleName, scope = "default") =>
  async () => {
    // Check if this remote has already been loaded
    if (!(remoteName in window)) {
      // Initializes the shared scope. Fills it with known provided modules from this build and all remotes
      await __webpack_init_sharing__(scope);
      // Fetch the remote app. We assume our remote app is exposing a `remoteEntry.js` file.
      const fetchedContainer = await fetchRemote(
        `${remoteUrl}/remoteEntry.js`,
        remoteName
      );
      // initialize the remote app
      await fetchedContainer.init(__webpack_share_scopes__[scope]);
    }
    // 'container' is the remote app
    const container = window[remoteName];
    // The module pass to get() must match the "exposes" item in our remote app exactly
    const factory = await container.get(`./${moduleName}`);
    // 'Module' is the React Component from our remote app's "exposes" configuration
    const Module = factory();
    return Module;
  };
```

Figure 3.8. Dynamic Configuration of Remote

3.2.4 Vite

Vite is a build tool that aims to provide a faster and leaner development experience for modern web projects that uses modern JavaScript frameworks [22]. It consist of two major parts:

- A local development server that provides rich feature enhancements over native ES modules, such as extremely fast Hot Module Replacement (HMR), NPM Dependency Resolving and Pre-Bundling, as well as TypeScript support.
- A build command that bundles the code with *Rollup*, pre-configured to output highly optimized static assets for production. Rollup is a module bundler for JavaScript which compiles small pieces of code into a larger and more complex block, such as a library or application. The optimized output includes techniques such as minification, tree-shaking, and other optimizations to ensure that the final output is lightweight and optimized for performance when deployed.

While Module Federation is not natively supported in Vite, there are third-party plugins available that enables the use of it. In particular there is the `vite-plugin-federation` from *OriginJs* [21] that enables Vite projects to utilize Module Federation, allowing for the dynamic loading of remote modules and sharing of dependencies between independently deployed applications. It is a Vite-Rollup plugin inspired by Webpack and also compatible with Webpack.

The configuration of the plugin is similar to Webpack and it's performed in the `vite.config.js` file. In this case it's necessary to manually install the plugin within each project applications, using npm, yarn or other tools; then it can be imported as `@originjs/vite-plugin-federation`.

Configuration

As stated before, the configuration does not have large differences from Webpack Module Federation plugin. For the host application, figure 3.9, is necessary to provide the `name` and add the remotes name specifying the remote name and the URL from which to load its entry file. While for the remote application, figure 3.10, the property `name` specifies

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";
import federation from "@originjs/vite-plugin-federation";

export default defineConfig({
  plugins: [
    react(),
    federation({
      name: "host-app",
      remotes: {
        remote_components: "http://localhost:8081/assets/remoteEntry.js",
      },
      shared: ["react", "recoil"],
    }),
  ],
  build: {
    modulePreload: false,
    target: "esnext",

    minify: false,
    cssCodeSplit: false,
  },
});
```

Figure 3.9. Host App Configuration

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";
import federation from "@originjs/vite-plugin-federation";

export default defineConfig({
  plugins: [
    react(),
    federation({
      name: "counter-components",
      filename: "remoteEntry.js",
      exposes: {
        "./Counter": "./src/components/Counter.jsx",
        "./atoms": "./src/recoil/atoms.js",
        "./selectors": "./src/recoil/selectors.js",
      },
      shared: ["react", "recoil"],
    }),
  ],
  build: {
    modulePreload: false,
    target: "esnext",
    minify: false,
    cssCodeSplit: false,
  },
});
```

Figure 3.10. Remote App Configuration

the name of the application and the property `filename` specifies the name of the remote entry; then the exposed modules are indicated inside `exposes` property, specifying the name and the path to the file in the project.

Additionally, as in Webpack, inside the property `shared` are indicated the libraries shared among the applications to avoid duplications.

Usage

```
import { useState } from "react";
import { useNavigate } from "react-router-dom";
import { useRecoilState } from "recoil";

import { counterAtom, nameAtom } from "remote_components/atoms";
import { Counter } from "remote_components/Counter";

export const Home = () => {
  const [count, setCount] = useRecoilState(counterAtom);
  const [name, setName] = useRecoilState(nameAtom);
  const [newName, setNewName] = useState("");

  const navigate = useNavigate();

  return (
    <>
    <Counter/>
      <h1>HOME PAGE</h1>
      <div className="mt-2 ml-2">
        {!name && (
          <>
            <p>Insert your name</p>
            <p className="text-xs">
              It will be saved in a{" "}
              <span className="italic">remote recoil atom</span>
            </p>
            <div className="mt-2 mb-4">
              <p>Name: </p>
```

Figure 3.11. Vite Imports

After having completed the configuration, the imports of remote modules inside the

host application is corresponding to Webpack. In fact, as shown in figure 3.11, static imports follow the same syntax of traditional imports of components and the modules can be used straight away.

3.2.5 Vite Dynamic Remotes

With the Module Federation plugin of Vite, there are two possibilities to implement dynamic import and provide an higher degree of flexibility:

- The first possibility simply involves the usage of React Lazy, that is a feature of React library that allows for dynamic code splitting, as shown in figure 3.12. With `React.lazy`, components can be loaded asynchronously, improving the performance of React applications by only loading the code for components when they are actually needed. This helps reduce the initial bundle size of the application, leading to faster load times and better user experience.

```
const myComponent = React.lazy(() => import('remote/myComponent'))

return (
  <>
    <myComponent/>
```

Figure 3.12. Vite Lazy Import

- The other solution provides a more dynamic and flexibile approach, similar to what can be achieved in Webpack with Dynamic Remote Containers. Following this approach the remote is loaded programmatically without needing to provide a configuration in the Vite configuration file. The image 3.13 shows a possible implementation of this solution, providing a methods that uses parametric variables to configure the remote and then retrieve the desired module, as well as handling possible errors. For example if the desired remote doesn't exists, a specific error component is returned without affecting the operation of the application. As it can be seen from the example, this solution involves using `__federation_method_setRemote` and `__federation_method_getRemote` which respectively create and save the configuration of the remote and retrieve the desired module from the remote. These two methods are imported from `__federation__`, a virtual module that provides a useful scheme for allowing build time information to be passed to the source files using normal ESM import syntax.

  In order to use a component with this solution, it must be wrapped inside `React.Suspense` as shown in figure 3.14, that allows components to suspend rendering while waiting for some asynchronous data to load.

```
import {
  __federation_method_setRemote,
  __federation_method_getRemote,
  // eslint-disable-next-line @typescript-eslint/ban-ts-comment
  // @ts-ignore
} from "__federation__";

export const getFederatedComponent = ( name: string, module: string, address: string, fileName: string ) => {
  return React.lazy(async () => {
    try {
      __federation_method_setRemote(name, {
        url: `${address}${fileName}`,
        format: "esm",
        from: "vite",
      });
      const ret = await __federation_method_getRemote(name, `./${module}`);

      if (ret.default === undefined) return { default: ret };

      return ret;
    } catch (error) {
      console.log(error);
      return { default: ErrorFederatedComponent };
    }
  });
};
```

Figure 3.13. Vite Dynamic Configuration

```
const RemoteTable = getFederatedComponent(name, module, address, fileName);

return (
  <div>
    <p>{module}</p>
    <div>
      <React.Suspense fallback={<>Loading ...</>}>
        <RemoteTable />
      </React.Suspense>
    </div>
```

Figure 3.14. Vite Dynamic Usage

3.2.6 Frameworks

The examples shown above are implemented in React projects, given that the popularity of the framework and its tools allows for large possibilities of usage. Additionally, Module Federation can be implemented also using other frameworks.

During the study phase of federation, other frameworks were considered with the purpose of testing the integration possibilities with the other main topic of the thesis: server rendering.

Next.js

Next.js is a React framework for building full-stack web applications. It's possible to use React Components to build user interfaces, and Next.js for additional features and

optimizations [11].

The framework will be discussed in detail later, studying its functioning and performance with server rendering. In this part, the focus is on the implementation of micro frontends architecture and the usage of Module Federation. This can be done partially, in fact Next.js relies on Webpack allowing for the usage of *Webpack Module Federation Plugin*. The limitation is that, at the time of writing this thesis, the plugin doesn't support the latest version of the framework (Next.js v14) and in particular it has no support for the App Router, but only for the Pages Router. The first one is a newer router that allows to use React's latest features while the latter is the original Next.js router, currently still supported but not recommended as the default choice for new applications.

The usage in Next.js with Pages Router is exactly equivalent to what shown for React, adding the opportunity to provide server side rendered pages and data for a federated component from a remote application.

Remix.run

The other framework considered for implementing server rendering is Remix.run, a full stack web framework built on top of React Router [17]. Also for this case, a complete overview and technical explanation of the framework usage and implementations is provided in the following chapters. Now, it can be highlighted the non compatibility with Webpack and the not yet stable support of Vite [1]. Using Vite as an alternative compiler allows for the integration of Vite Federation plugin but since it's a new technology, the implementation doesn't provide support for server rendering for the remote applications, removing the possible advantages of choosing Remix.run.

Vue

This framework, differently from the previous ones, is not React-based and it was considered due to its integration with Vite. The creator of Vite is also the author of Vue and that is why they can be used together providing a seamless development experience; Vue provides as well the integration of the Vite federation plugin.

Vue is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model [23]. Its component-based architecture is one of the key features of the framework, which promotes reusability and maintainability. Components encapsulate both the structure and behavior of UI elements, allowing developers to create modular and self-contained pieces of code. This is extremely useful due to the extreme diversity of the web; in fact Vue is designed to be flexible and incrementally adoptable so it can be used in different ways depending on the use case, such as:

- Standalone Script: Vue can be used as a standalone script file, without any build step required. This is the easiest way to integrate Vite in case there is a backend

[1] Stable from Remix v.2.7.0, February 2024

framework already rendering most of the HTML, or the frontend logic is limited and undemanding

- Embedded Web Components: Vue is used to build standard Web Components that can be embedded in any HTML page, regardless of how they are rendered. This option allows to leverage the framework in a completely consumer-agnostic fashion.

- Single-Page Application (SPA): Vue provides core libraries and comprehensive tooling support with great developer experience for building modern SPAs, providing rich interactivity, deep session depth, and non-trivial stateful logic on the frontend. This approach allows to build applications where Vue not only controls the entire page, but also handles data updates and navigation without having to reload the page.

- Fullstack / SSR: Vue implement SSR providing APIs to render the app into HTLM strings on the server, so that users can see already-rendered HTML while JavaScript is being downloaded; then Vue manages the hydratation phase to make the screen interactive. SSR can be implemented manually modifying the Vue configuration and its server or by using the community plugin of Vite that abstracts away challenges related to configuration.

- JAMStack / SSG: Static-Site Generation is a semplification of SSR that can be implemented when the only data required are static. Hence it's possibile to pre-render an entire application into HTML and serve them as static files, improving site performance and making deployment a lot simpler, since it's no longer needed to dynamically render pages on each request. Vue can still hydrate such applications to provide rich interactivity on the client.

- Beyond the Web: Although Vue is primarily designed for building web applications, it is not limited to just the browser. Actually it's possible to build desktop or mobile apps, WebGL experiences or even to build custom renderers, like those for the terminal.

When compared to React, Vue shares some similarities but also has distinct differences. Both frameworks employ a component-based architecture and utilize a virtual DOM for efficient rendering. However, Vue is often praised for its simplicity and ease of learning, thanks to its clear and concise API. While, in terms of performance Vue and React are comparable, with both frameworks offering efficient rendering and updates. However, Vue's smaller size and simpler API may lead to faster development times and reduced overhead in certain scenarios.

Regarding Micro Frontends architecture, given the perfect integration of Vue with Vite, it's possible to use the Module Federation plugin configuring it as previously seen inside the `vite.config.js` file and the usage of remote modules doesn't differ from what seen with React. Potentially it is also feasible to integrate a Vue remote micro application within a React-based project, using appropriate libraries. This possibility is explored further as the demo project unfolds

3.3 Challenges and Solutions Encountered

In this section, the focus shifts to the challenges and solutions encountered during a more practical phase of this thesis. As the transition from theoretical exploration to hands-on implementation occurs, different approaches, frameworks, and tools are engaged.

Throughout this phase, a process of experimentation is undertaken, involving the implementation of tutorial projects from documentations and the development of simple demos to test various concepts. However, it is important to recognize that not all approaches have produced satisfactory results; also because of how recent some aspects related to this technology are. In facts, the scarcity of comprehensive documentation or practical examples often posed challenges, leading to the necessity for innovative solutions and workarounds and despite efforts made, some problems remained unresolved, resulting in limitations to implementations.

One significant aspect of this phase is the alignment of the study with practical considerations, particularly regarding technology stacks. As shown, the usage of frameworks like Next.js or Remix.run paired with micro frontend architecture is limited, given several compatibility problems. To this end, the focus of implementations is primarily on Vite and React, rather than traditional options like Webpack. This decision comes from aligning with the AizoOn's technology stack, as the goal is to integrate these concepts into the company's existing applications to improve performance and other crucial aspects.

Thus, the subsequent sections delve into the specific challenges encountered, solutions devised, and lessons learned during the practical application of Micro Frontend concepts, with a particular emphasis on the utilization of Vite and React.

3.3.1 TypeScript

Most of the examples in the previous sections are taken from demo project implemented with JavaScript, but the usage of TypeScript provide several benefits that can improve development performances.

TypeScript is a statically typed superset of JavaScript that adds optional static typing to the language, allowing to define types for variables, parameters, and return values. Additionally TypeScript performs type checking at compile time, providing early detection of potential errors; whereas JavaScript performs type checking at runtime, which may lead to runtime errors. The combination of type system and compile time checking provides type safety, leading to more robust code and reducing the likelihood of runtime errors.

TypeScript's static typing enables better IDE support that lends also to better code maintainability.

Having considered TypeScript's advantages over JavaScript, it can be useful to use it also in Micro Frontends projects. However, in this situation, the usage of TypeScript can rise some errors and warnings. Furthermore the static typing may suffer from this, without being able to infer types. In fact, the strict type safety can't be granted when a remote component is imported in the main application, both in case of static and dynamic configuration of remotes (hence also with dynamic imports).

In certain situations, the developer can take responsibility for ensuring proper type management by suppressing the warnings. TypeScript provides mechanisms to suppress

errors or warnings using specific comment directives that tells the TypeScript compiler to ignore any type checking errors or warnings within that specific code block or file.

If there is a need for a higher level of type safety or more robust code when using remote modules, it is possible to take advantage of the very concept of module federation by exporting as a remote component also a defined type or an interface used to define the property types of another remote component.

3.3.2 CSS Styling

There are various approaches to styling in frontend development, including CSS, CSS preprocessors like Sass, CSS-in-JS solutions, and utility-first frameworks like Tailwind CSS. During the development of different projects implementing the Module Federation pattern, certain challenges emerged, particularly related to the use of Tailwind CSS.

Tailwind CSS is a utility-first CSS framework that provides pre-designed utility classes for styling elements. Rather than writing custom CSS rules, developers can apply these utility classes directly to HTML elements to achieve the desired styles. Tailwind CSS operates on static analysis of the source code during the compilation process, generating a stylesheet containing only the necessary classes based on those actually used.

However, when using Module Federation plugins (both in Webpack and Vite), where components are loaded dynamically and shared among modules, this loading mechanism compromises Tailwind's ability to detect and include requested styles. Consequently, this behavior led to the incorrect display of Tailwind definitions for remote components when loaded and rendered in the main host application.

Initially, basic solutions involved defining CSS rules for each element or passing Tailwind rules as parameters to the remote element when rendered from the host. However, these workarounds were not acceptable in the context of developing more complex components and rich and flexible UIs.

An effective solution emerged after an in-depth analysis of behavior and structures: the utilization of PostCss and Autoprefixer.

PostCSS is a tool for transforming CSS with JavaScript plugins. It parses CSS and applies transformations specified by plugins, allowing developers to write CSS using modern syntax and features that may not be supported by all browsers.

Autoprefixer, a popular PostCSS plugin, automatically adds vendor prefixes to CSS rules based on the specified browser compatibility settings. It analyzes CSS and adds the necessary prefixes to ensure that styles work correctly across different browsers.

By configuring these tools appropriately, consistent style processing through Webpack (or Vite) modules was ensured, preserving Tailwind styles' integrity when remote components are loaded. This approach resolved the issue and facilitated the correct rendering of styles for remote components within the main host application.

3.3.3 State Management and Communication

In the context of Micro Frontends architecture, state management and communication between different Micro Apps emerge as critical aspects. Separating functionality into

distinct modules can present unique challenges in terms of consistency and synchronization of information between various parts of the application.

It's important to keep in mind that the concept of micro frontends revolves around the fact of having separated modules as isolated as possible, to avoid increasing the level of complexity, provide fault tolerance and prevent possible inconsistent usage of shared data. Quoting single-spa documentations [20]: *If two microfrontends are frequently passing state between each other, consider merging them. The disadvantages of microfrontends are enhanced when your microfrontends are not isolated modules.*

However, certain situations require an implementation of communication mechanism between different modules or components and this often relies on the state managed within those modules. The main strategy [3] are illustrated below:

- Web Workers: Web Workers are scripts that run in the background, independently of the main thread, enabling concurrent execution. They can be used to exchange data for communication or synchronization in a separated thread allowing the main (UI) thread to run without being blocked/slowed down. This can improve general performances and fluidity of the entire application, especially when many remote applications run together. Moreover, Web Workers are not used much with Micro Frontends due to their complex configuration and API usage.

- Props and Callbacks: They are a fundamental mechanism for passing data and triggering actions between components. It's possible to pass state management variables from a parent component of main app to a child element of a remote component and even callback function to be invoked as a response to specific actions. This method is intuitive and straightforward when components maintain a direct parent-child relationship but it may become cumbersome in more complex scenarios involving deep nesting or sibling components.

- Custom Events: This approach is highly scalable and aligns with event-driven architectures common in microservices and it is used to allow components within a micro frontends application to communicate asynchronously. Components can dispatch custom events, and other components can listen for these events and respond accordingly. While this offers a robust solution for state sharing, it comes with a set of challenges related to dependency on event subscription timing and coordination overhead.

- Platform Storage APIs: localStorage, sessionStorage, or IndexedDB are examples of platform storage APIs, this method allows micro frontends to set and read data independently directly, minimizing dependency on the container app. It is versatile and applicable in web and mobile contexts, with Local Storage for browsers and Async Storage for mobile apps. Nevertheless it has some drawbacks as limited scalability, debugging difficulty during development and security concern.

- Libraries: There are several libraries that can be used for state management in micro frontends. These libraries are well-suited for complex applications, providing tools for organizing, accessing, and updating state in a predictable and scalable manner. *Redux* is a predictable state container for JavaScript applications that provides a

centralized store and pure reducer functions for managing application state. *Recoil* is a React-specific state management library that introduces atoms for decentralized state management within components, offering a simpler API and optimized performance. Redux follows a centralized approach with actions and reducers, whereas Recoil introduces atoms for decentralized state management within components. Redux can be more suitable for large-scale applications with complex state management needs, while Recoil offers a simpler and more intuitive API with optimized performance for React applications.

3.3.4 Vite Dynamic Configuration

As seen in previous sections and images 3.8 and 3.13, with module federation it is possible to programmatically configure a remote application in the host application code. With Webpack plugin this implementation has been tested and used successfully, while with Vite plugin some problems and errors emerged during development. In fact it has been noted that the use of methods from the `__federation__` virtual module, to set and get the remote components, can generate errors at runtime, when using the application. This situation occurs in a particular condition, specifically in the case in which the remote application exposes multiple modules to be shared with the host application: the federation method used to retrieve one of the remote modules simply returns the component as a plain object and it is not recognized as a module, causing the *React Lazy* to fail at runtime since the component can not be parsed as a React module. After an analysis of the internal representation and process behind the federation methods from the Vite plugin, a workaround was found, although it cannot be considered as a permanent solution since the problem lies in the plugin itself and will have to be resolved by the responsible developers. The said workaround involves a straightforward additional step to be performed after the federation method returns the remote module: the returned object value should be enclosed in another object as value of a field named *default*. At runtime, this is interpreted as a default export for the given component, allowing for the module loading system implemented by the browser or the JavaScript engine to treat the object as a module and consequently resolve the dynamic loading returning a React component. Figure 3.13 already shows this solution.

As mentioned, this is an acceptable solution until the plugin is updated, even taking into account that the dynamic features of this have only been introduced recently [26].

Part II

React Server Components

Chapter 4

Introduction to the concept

4.1 Rendering

Rendering, in the context of web development, refers to the process of generating and displaying the final output of a web application or website to the user's browser. It involves taking raw data, typically stored in databases or received from external sources and transforming it into a visual representation that users can interact with. This includes organizing the data into structured layouts, applying styles and formatting, and integrating interactive elements.

Rendering is a fundamental aspect of web development, crucial for creating engaging and user-friendly digital experiences across a wide range of devices and platforms. Different rendering strategies were introduced to improve performances and also different frameworks contributed to provide alternative strategies.

In recent years, React has emerged as a prominent tool in modern web development. However, the rendering strategies adopted by this framework have consistently sparked concern and debate among developers, primarily due to issues such as slow data fetching, large bundle sizes, and complex business logic.

To address these challenges, alternative rendering strategies have emerged and other technologies as Angular, Vue.js, and Svelte also contribute to this ecosystem, each with its own rendering approaches and considerations. Understanding the nuances of different rendering strategies is essential to navigating the diverse landscape of modern web development.

This chapter will delve into the complexities of rendering within React, the benefits and tradeoffs of various approaches [1], and examine the role of React server components in shaping the future of web application development, considering React as the core technology as well as other related frameworks such as Next.js, Remix.run, Modern.js and RazzleJs.

Web Vitals Metrics

Web Vitals is a Google initiative to provide unified guidance for web page quality signals that are essential to delivering a great user experience on the web [24].

The most important metrics are called Core Web Vitals, a subset of Web Vitals that apply to all web pages and they may evolve over time. The current set focuses on three aspects of the user experience: loading, interactivity, and visual stability. The three metrics are illustrated below and they will be used to provide a contextual comparison between the different rendering techniques.

- **Largest Contentful Paint** (*LCP*): measures loading performance. It is used to track the time it takes for the largest content element, such as an image or text block, to become visible to the user. A fast LCP ensures that users perceive the page as loading quickly and efficiently.
- **First Input Delay** (*FID*): measures interactivity. FID tracks the time between when a user interacts with the page (for example, clicks a button or taps a link) and when the browser responds to that interaction. A low FID ensures that users can interact with the page smoothly without delays.
- **Cumulative Layout Shift** (*CLS*): measures visual stability. CLS tracks the amount of unexpected layout shifts that occur during page load. A layout shift is possibly disruptive, especially when elements move unexpectedly, causing users to click on the wrong link or button. A low CLS ensures that the page remains visually stable and predictable.

4.1.1 Client-Side Rendering

Client-Side rendering (*CSR*) is the simplest rendering in a traditional Single-Page Application (SPA). The server delivers a minimal HTML shell to the client, and the content is generated in his browser using JavaScript. Using React, an empty `<div>` element is sent to the user's browser along with a large bundle of JavaScript containing React and the application code. Subsequently, the client is responsible for making data requests, such as to a database, processing the user interface, and rendering the resulting HTML interactively.

CSR is optimal for single-page applications and web applications that require a high degree of interactivity and real-time updates, providing a seamless and dynamic user experience. In several situations this can result in unacceptably slow websites due to heavy work performed by the client, particularly for initial page loads, leading to high *LCP* and *FID* values. Additionally, *CLS* can be variable, especially if content is loaded asynchronously or if layout changes occur after initial render

In the case of the company with which this thesis work was conducted, having an application that requires loading a large amount of data makes this aspect extremely relevant. Moreover, in the commercial realm, Core Web Vitals for SPAs are also prone to falling below standards, leading to poor SEO performance.

With the evolution of React over time, some improvements have been introduced, and the framework has received significant contributions from its community to enhance rendering performance. In particular, the core React team and engineers working on frameworks like Gatsby and Next.js have devised solutions to render pages on the server, aiming to reduce the client-side workload.

4.1.2 Static Site Generation

The first solution to improve rendering performance is Static Site Generation (*SSG*). In this approach, all the website's pages are generated at build time, when the application is initially deployed to the server. During this process, routes and pages are pre-rendered on the server, and the resulting static HTML is then sent to the client. This means that when users access the pages, there is already have static content to view, thereby reducing the initial loading time and consequently providing low *LCP* time, improving overall performance. Page interactivity is subsequently added through client-side JavaScript execution, resulting in acceptable and relatively low *FID* time. Since pages are pre-rendered with stable layouts, values of *CLS* tend to be considerably low.

This approach is particularly well-suited for content-heavy websites, blogs, e-commerce sites, and landing pages. However, *SSG* may not be ideal for websites with a vast number of pages, as build times may take longer, impacting development efficiency. Additionally, *SSG* is not suitable for websites with frequently changing content, especially if real-time updates are a priority.

Considering this approach in relation to the corporate scope of the company, *SSG* can bring benefits in the case of data-intensive pages that do not require frequent updates.

4.1.3 Server Side Rendering

Another approach is Server-Side Rendering (*SSR*), where routes are rendered at the time of request. Unlike *SSG*, which pre-renders pages during compilation time, with *SSR*, the server dynamically processes content each time a user requests a page. This means that the client receives HTML already compiled directly from the server, ensuring that users quickly see the content, resulting in a lower *LCP* time. Once again, interactivity is added through client-side JavaScript execution with acceptable values of *FID*. Both *SSG* and *SSG* aim to enhance overall performance and user experience but differ in when content generation occurs.

SSR can be resource-intensive given that the server must regenerate the HTML for each request, which can strain the server's resources. Additionally, it may not be the most suitable choice for content that rarely changes, as the benefits of pre-rendering may not outweigh the server computation costs.

Streaming SSR

Streaming SSR is an extension of Server-Side Rendering. It allows the server to start sending chunks of HTML to the client as soon as they are generated, rather than waiting for the entire page to be processed. This enables the client to start rendering and displaying content progressively as it arrives, providing a smoother and faster user experience, especially for pages with large amounts of data or complex layouts.

This approach improves *SSR* efficiency, in particular related to other important Web Vitals metrics that are Time to First Byte (*TTFB*) and Time to Interactive (*TTI*). *TTFB* measures the time it takes for the browser to receive the first byte of data from the server after making a request, reflecting server's responsiveness; while *TTI* measures the time

it takes for a web page to become fully interactive, meaning that all content has loaded, and the page is responsive to user input.

Therefore, sending content to the client in chunks as it becomes available can significantly improve the overall time it takes to receive the first byte and, as a result, can result in a faster and more responsive user experience.

Streaming SSR is not without disadvantages. In fact, it can introduce additional complexity in the development process as well as requiring careful management of server-side rendering and data streaming. Additionally, streaming SSR may have higher resource requirements compared to traditional *SSR*, as it involves maintaining open connections and managing data streams in real-time.

4.1.4 Incremental Static Regeneration

The Incremental Static Regeneration (*ISR*) strategy was recently introduced by Next.js, it combines elements of both *SSG* and *SSR*. *ISR* bridges the gap between these two approaches by allowing specific pages or sections of pages to be regenerated on-demand, combining the efficiency of static site generation with the flexibility of server-side rendering. The regeneration, or update, can be obtained specifying a revalidation time for each page. This feature determines how often a page is regenerated and updated.

When a user visits a page, *ISR* checks the current time against the revalidation time to decide whether to serve the cached static page or re-render it: in case the current time exceeds the revalidation time for a specific page, *ISR* invalidates the cache for that page and generates a fresh version of it.

LCP and *FID* performances can vary depending on the caching and regeneration settings, but can typically still be optimized for freshness, speed and responsiveness. Also *CLS* can be optimized depending on the caching and regeneration settings.

This approach ensures to always receive the most up-to-date content, maintaining a balance between performance and real-time updates. On the other hand, it requires a more complex configuration than *SSR* and *SSG*, potentially introducing complexity in managing cache and revalidation, along with increased server load and resource consumption.

4.1.5 React Server Components

React Server Components (*RSCs*), or simply Server Components, represent the latest technological innovation in web content pre-rendering, designed by the React team and introduced as a stable feature in March 2023 [16]. They introduce a new mental model to the framework, allowing the creation of components that span both the server and client worlds. With *RSCs*, server-side rendering can occur at the component level without waiting for an entire page to be rendered on the server; *RSCs* individually fetch data and render entirely on the server, and the resulting HTML is streamed into the client-side React component tree, interleaving with other Server and Client Components as necessary. This contributes to obtain low values of *LCP* and *FID* as well as *CLS*, leading to a smoother and more responsive user experience overall. As reported, Server Components seamlessly integrate also with Client Components to provide a versatile blend of server-side efficiency and dynamic client-side interactivity.

RSCs can be considered as an improvement of *SSR* (as well as Streaming SSR), providing more granular control over server-side rendering and seamlessly integrating with existing React workflows. Additionally *RSCs* provide a balance between server-side rendering and client-side interactivity.

However, *RSCs* should be considered a newborn architectural approach, highly promising but, as shown in the following sections, with significant drawbacks and challenges still to be resolved.

4.2 The Problems RSCs Solve

It is important to understand the reasons why using *RSCs* can be useful and beneficial. Bearing in mind that it adds a certain degree of complexity to the code and implementation of the project, in case you have to start from the foundations, while it represents a very complicated solution to implement in the case of refactoring an already existing application. It is therefore necessary to underline the problems that are addressed and resolved with this approach [13].

4.2.1 Data Fetching

One of the main issues addressed by React Server Components concerns data management and one-directional data flow. In the early stages of the framework competition, React gained success primarily due to its innovative idea of one-directional data flow, contrasting with frameworks like Angular. This one-directional data flow implies that data is passed only in one direction within the application (along the component tree). If there are updates in the data, React will reconcile the entire component tree, re-rendering all components with modified props and data and their respective children. With *RSCs*, the one-directional flow now involves the server as well, allowing for server-side management of changes without additional efforts to synchronize states.

To better understand this concept, a classic example of data retrieval in React can be considered. Having a root component, such as the structure of a social media app, with typical elements like a post feed and a profile button, the problem arises when data need to be retrieved, such as the profile, the feed, and other sidebar information. Traditionally, this involves combining the usage of effect and state hooks, respectively `useEffect` and `useState`. Data is fetched stored in state variables and then eventually passed down as props from a shell element to the component that need it, as shown below in image 4.1.

The problem with this approach is that developers have to think and implement the logic to trigger the fetch, how to handle the states and how to handle relative errors in a disconnected way to the components, since this is performed at a higher level in the component tree. This approach also introduces a "waterfall" where it's necessary to wait for the inner element, such as profile, to load before requesting related or outer elements, as the feed, creating a noticeable delay for users, especially those far from the server. A possible solution could be fetching all data in a single React Promise, avoiding the waterfall problem when waiting for different elements data, but still providing a delay and an initial waterfall waiting.

```jsx
import { useState, useEffect } from "react";
import Profile from "./Profile";
import Feed from "./Feed";

export default function Shell() {
  const [profile, setProfile] = useState(null);
  const [feed, setFeed] = useState(null);

  useEffect(() => {
    fetch("/api/profile")
    .then((res) => res.json())
    .then((data) => setProfile(data));
  }, []);

  useEffect(() => {
    if (profile) {
      fetch(`/api/feed/${profile.id}`)
      .then((res) => res.json())
      .then((data) => setFeed(data));
    }
  }, [profile]);

  return (
    <div>
      <Profile profile={profile} />
      <Feed feed={feed} />
    </div>
  );
}
```

Figure 4.1. Data Fetching with Waterfall

RSCs solve this problem by allowing data retrieval logic to be handled directly within the components themselves, eliminating the need for complex client-side state management. Each component (server or client) can receive data through props, allowing for a more modular display and reducing the logic in the main component. This process adds the server to the one-directional data flow, enabling data reloading and dynamic UI updates without additional efforts in state synchronization, significantly improving performance and user experience. In this way, each component can internally perform data fetch operations using an asynchronous await to request the data and potentially pass it to child components, as illustrated in the examples in images 4.2 and 4.3.

Additionally, the use of this technique within React components is possible thanks to the presence of a "fallback" mechanism, implemented through `<Suspense>` component, introduced in React 18, that enables developers to manage asynchronous operations in a declarative and efficient manner. When a React component wrapped in `<Suspense>` triggers an asynchronous operation, React suspends the rendering of that component and its subtree, providing a fallback content (such as loading spinners or placeholders) while the operation is in progress. Once the operation completes, React resumes the rendering of the suspended component and replaces the fallback content with the actual content returned.

This mechanism is integrated by default in frameworks that supports *RSCs*. This means that while using frameworks like Next.js is not necessary to directly rely on the

usage of suspendable components since `Suspense` is implemented internally and its mechanism is operated by the framework compiler.

```
export async function Profile() {
  const profileData = await fetch("/api/profile");
  return (
  <div className="flex">
      <div>{profileData.image}</div>
      <div>{profileData.name}</div>
      <div>{profileData.username}</div>
  </div>
  )
}
```

Figure 4.2. Profile Component

```
export async function Feed() {
  // the profileData fetch will automatically be deduplicated by React,
  // so we're not hitting the same endpoint multiple times per request
  const profileData = await fetch("/api/profile");
  const feedData = await fetch(`/api/feed/${profileData.id}`);
  return (
  <div className="flex flex-col">
      {feedData.map(post => (
    <div>
        <p>{post.username}</p>
        <h3>{post.title}</h3>
        <p>{post.content}</p>
      </div>
    ))}
  </div>
  )
}
```

Figure 4.3. Feed Component

4.2.2 Composable Business Logic

As explained above, it is possible to perform data retrieval directly within the components without having to use effect hooks. However, this does not have an immediate utility, considering that for some frameworks there are already similar solutions such as `getServerSideProps` for Next.js, loaders from React Router, or `.server` files for Astro.

The problem with such solutions concerns how the requests are handled. No framework natively addresses data loading based directly on components. Even with Remix's nested routing, developers are still forced to write non-composable route-based logic, which means that data retrieval and revalidation can only occur based on routing path, not on the component. Furthermore, it is not easy to reuse this logic by importing it from a `*.jsx` file or from NPM.

This means that there is no framework-agnostic way to write business logic in a composable manner in React, it is not possible to directly import a database query from the file containing the logic. Additionally, the solutions present in frameworks often have to use routing-based logic to obtain data in the application, which can lead to logic duplication, or they have to use nested routing, an excellent solution but one that still does not support "composition" of logic in a modular way.

One of the key aspects of React Server Components is the ability to "componentize" business logic so that it can be composed in different areas of the application without having to copy/paste only certain parts of logic from a specific path.

A great example is implementing Stripe in an application to collect subscription revenues. Before *RSCs*, it was necessary to consult the Stripe documentation, search for React integrations, check if there were specific guides for the framework used, whether it was Remix, Next.js, or Astro, etc., and finally deal directly with integrating with Stripe. Now, thanks to *RSCs* allowing the distribution of React components whose logic is executed only on the server, a developer can publish a Stripe integration component to a

private NPM registry, for example. This way, it becomes feasible to use and reuse server-side business logic as you would with a UI library, allowing the entire team to use the same logic to integrate with Stripe by simply importing the library.

4.2.3 Bundle Sizes

A significant benefit often mentioned by developers regarding *RSCs* is the lack of impact on bundle sizes. The bundle size refers to the total size of all the JavaScript files that are required to run the application in the browser. This includes not only the written code, but also any third-party libraries or dependencies that the application relies on. Bundle size is an important metric to consider because larger bundle sizes can lead to slower load times for your application, especially on slower internet connections or devices.

Actually, it is important to note that *RSCs* themselves do not have zero bundle sizes. Even if a static server component such as a simple `div`, is sended, React and ReactDOM would still be sent to the client, making the bundle size not zero.

What is meant by "0 impact on bundle sizes" is that the libraries and dependencies used by the server components will not be shipped to the browser.

For example, in case of having an element that does not rely on interactivity, thus server-renderable, even though itself has other libraries dependencies, it is possible to send it to the browser as if it had no dependencies, without impacting the bundle size at all. This proves to be very useful as it reduces the bandwidth used by the application, since less JavaScript is sent over the network, and leads to faster loading times, since fewer bytes are sent in the client-side bundle.

4.3 State of Literature

Given all the advantages and benefits illustrated in the previous sections, it would be natural to ask why *RSCs* are not the most used technology currently and are not adopted by default by the main frameworks in the world of web development.

As stated in the official React documentations [16], currently the only framework that implements *RSCs* is Next.js. This framework showcases a deep integration of a router that really buys into RSC as a primitive, but it's not the only way to build a RSC-compatible router and framework, even if integration with other tools an libraries is limited. In fact, reporting the React documentation again:

We generally recommend using an existing framework [...] Building your own RSC-compatible framework is not as easy as we'd like it to be, mainly due to the deep bundler integration needed.

The reason behind this statement and the present situation is that *RSCs* are not stably supported in the current version of React (18), probably they'll be included as part of the mainline, stable release in React 19, even if the React development team has not yet released information on when exactly it will arrive. Actually, the only method to implement *RSCs* outside Next.js is using React Canaries.

4.3.1 React Canaries Releases

The Canary channel is a prerelease channel that tracks the main branch of the React repository. This allows to start using individual new React features before they land in the semver-stable releases (the released versions considered stable for production use).

Canaries differ from Experimental releases, because experimental APIs can undergo significant breaking changes on their way to stabilization, or can even be removed entirely. While Canaries can also contain errors, any significant breaking changes is planned and announced in the React Canary blog. So the usage of Canary workflow must be careful, it's important to always pin the exact version of the Canary in use.

As said Canary can be used to implement *RSCs*, since the React team established that React Server Components conventions have been finalized and they re ready to be adopted by frameworks while still having issues with some related features that come in the way and prevent the release of React Server component support in a stable version of React [15]. This is mainly due to he deep integration with the bundler required for operation. The current generation of bundlers are great for client use, but they were not designed for primary support that involves splitting a single module graph between server and client. For this reason the React team is currently working directly with bundler developers to get primitives for *RSCs* integrated directly. Therefore a technological leap can be expected in the near future, facilitating increased integration and subsequently wider dissemination of this technology.

During the thesis work, a study was carried out on how it was possible to implement *RSCs* without the support of a framework like Next.js. The main outcome is that implementing a Canary version of React to use *RSCs* is not a feasible solution in an enterprise context. Basing the architecture of a corporate project on characteristics that are not yet stable requires a cumbersome and complex configuration, not suitable in an environment that requires a high level of security and reliability, as well as flexibility in configuration to ensure code maintainability and clear separation of concerns.

In the next chapter the study carried out on the different frameworks is illustrated, also in relation to React Canary and it is explained in depth why the *RSCs* technology is, to date, still at a too experimental level to be implemented.

Chapter 5

Approaches and Tools for Development

This chapter presents the main frameworks for implementing server rendering as well as *RSCs*. These technologies were explored through an initial documentation phase, followed by the practical creation of demo applications to test and validate the concepts.

5.1 Next.js

Next.js is a React framework for building full-stack web applications [12]. It automatically abstracts and configures the tools needed for React, such as bundling, compilation, and others, allowing to focus on building the application instead of spending time on configuration.
The main features of this framework includes:

- Routing: A file-system based router built on top of Server Components that supports functionalities such as nested routing, layouts, loading states, error handling.
- Rendering: Client and Server Components that provides Client-side and Server-side Rendering, further optimized with Static and Dynamic Rendering on the server with Next.js, Streaming on Edge and Node.js runtimes.
- Data Fetching: Simplified data fetching with async/await mechanisms in Server Components, and an extended *fetch* API for request memoization, data caching and re-validation.
- TypeScript: Improved support for TypeScript, with better type checking and more efficient compilation, as well as custom TypeScript Plugin and type checker.
- Styling: Support for different styling methods, including CSS Modules, Tailwind CSS, and CSS-in-JS.
- Optimizations: Image, Fonts, and Script Optimizations to improve the application's Core Web Vitals and User Experience.

Routing is a critical aspect of web applications that enables navigation between various pages, allowing users to access different parts of an application. Next.js offers two different routers, to implement a different routing system.

5.1.1 Pages Router

It is the original Next.js router. The Pages Router has a file-system based router built on concepts of pages. When a file is added to the `pages` directory it's automatically available as a route and the page element is treated as a React Component.

For example, a file named `info` is routed to `/info` route; while files named `index` are always routed to the root directory. This mechanism supports nested files, allowing for the creation of nested folder structures.

```
pages/index.js            → /
pages/info.js             → /info
pages/blog/index.js       → /blog
pages/blog/first-post.js  → /blog/first-post
```

With this system it is possible to create not only static but also dynamic routes, following the convention of placing the filename between square brakets, for example `[name]`. It allows to create routes from dynamic data, Dynamic Segments are filled in at request time or prerendered at build time and the dynamic value of the segment can be accessed from `useRouter`.

```
pages/users/[name].js  → /users/John     params: {name: 'John'}
pages/users/[name].js  → /users/Marc     params: {name: 'Marc'}
```

Others more advanced patterns are available such as Catch-All Segments or Optional Catch-all Segments. They are an extension of Dynamic Segments, the first one allows to catch-all subsequent segments by adding an ellipsis inside the brackets `[...segmentName]`; the latter make the catch-all optional by including the parameter in double square brackets: `[[...segmentName]]`.

```
pages/shop/[...id].js   → /shop/a      params: {id: ['a']}
pages/shop/[...id].js   → /shop/a/b    params: {id: ['a','b']}
pages/shop/[[...id]].js → /shop        params: { id: [] }
```

Additionally, it allows the reuse of components between pages, implementing layout patterns.

This routing system provides automatic code-splitting, each page is a separate bundle, thus improving loading times.

It is therefore an ideal solution for simple projects, its structure works right from creation without any need for configuration, creating less complexity in navigation and page creation. It is also suitable for situations where speed is preferred to flexibility, those

cases that require rapid development and implementation with a minimal configuration; suitable for websites and applications with simple navigation and no need for complex routing scenarios.

In the case of more complex projects, this approach has some disadvantages, as flexibility and support for shared layouts and nested routing are limited. Additionally, with large applications, maintenance can be complex and additional configuration steps may be required to handle complex routing needs. Furthermore, the advantage brought by code-splitting and separation into separate bundles can turn into a disadvantage as it is more difficult to share components between different pages.

5.1.2 App Router

Modern routing system, designed to overcome the limitations imposed by the Pages directory approach using the latest React features. The App Router works in a new directory named `app` that operates alongside the `pages` directory to allow for incremental adoption, with the App Router taking priority over the Pages Router.

By default the components inside the directory are *RSCs*. The reason behind this choice is a performance optimization and a strategy to allow developers to easily adopt them. Additionally is possible to use Client Components.

This routing system implements a hierarchical structure that can be visualized as a tree, in which the root represents the `app` directory. Nested directory can be represented as subtree, a portion of the main three, with each root element serving as a layout component for its corresponding route.

Within this structure, folders are utilized to define routes. A route is essentially a single path of nested folders, mirroring the file-system hierarchy from the root folder down to a terminal leaf folder that includes a `page.js` file. These `page.js` files define the content to be rendered for each specific route.

```
app/dashboard/settings/page.js   →   /dashboard/settings
```

Also with App Router is possible to define dynamic routes using the same convention implemented with Pages Router, as well as Catch-all Segments and Optional Catch-all Segments.

```
app/users/[name]/page.js   → /users/John   params: {name: 'John'}
app/shop/[...id]/page.js   → /shop/a       params: {id: ['a']}
app/shop/[[...id]]/page.js → /shop         params: { id: [] }
```

As said, nested folders are normally mapped to URL paths. However, it is possible to mark a folder as a Route Group to prevent the folder from being included in the route's URL path. This allows to organize route segments and project files into logical groups without affecting the URL path structure. A route group can be created by wrapping a folder's name in parenthesis.

```
app/(shop)/account/page.js   →    /account
```

Additionally it is possible to intercept routes. This feature allows to load a route from another part of the application within the current layout. This routing paradigm can be useful when you want to display the content of a route without the user switching to a different context. Intercepting routes can be defined with the `(..)` convention, which is similar to relative path convention `../` but for segments. This approach also allows to define special files at leaf level by naming them with specific names. For example it's possible to define a layout by default exporting a React component from a `layout.js` file, so that every segment (or subtree) can have a custom layout. Other special files are `loading.js`, that helps creating meaningful Loading UI with React Suspense, `error.js`, which allows to gracefully handle unexpected runtime errors in nested routes using React Error Boundary.

This hierarchical approach offers, along with the related features, flexibility and scalability, allowing for the organization of routes in a logical and intuitive manner. It also enables the creation of complex routing structures that can accommodate various navigation scenarios within the application. So it is more suitable for large applications and for scenarios in which more control over the routing logic is needed.

5.1.3 Rendering and Server Components

By default, Next.js uses Server Components within App Router. This allows to automatically implement server rendering with no additional configuration.

Server Components do not support client-side actions, such as click events, and React hooks, but a Server Component can be converted into a Client Component marking it with the "`use client`" directive at the beginning of the file.

Client Components are used to write interactive UI that is pre-rendered on the server and can use client JavaScript to run in the browser. In fact, doing the rendering work on the client allows Client Components to use state, effects, and event listeners, as well as have access to browser APIs, like geolocation or localStorage.

Server Components are therefore ideal for use when it is necessary to perform a data fetch process, access backend resources directly, maintain sensitive information (such as tokens or keys) on the server, maintain large dependencies on the server by reducing the size of the JavaScript code sent to the client. While Client Components are used there is the need to add interactivity, event listeners, handle hooks and integrate React components.

On the server, Next.js utilizes React's APIs to handle rendering. The rendering work is divided into chunks that are rendered in two steps: React renders the server components into a special data format known as the React Server Component Payload (*RSC Payload*). Next.js then utilizes the *RSC Payload* and the JavaScript instructions of any Client Components for rendering HTML on the server. At this point, on the client-side, the HTML is used to immediately display a non-interactive preview of the route (this occurs only during the initial page load); the *RSC Payload* is used to reconcile the tree of Client and Server components and update the DOM. Finally, the JavaScript instructions are used for the process of hydrating the Client Components and making the application interactive. Rendering on the server can be done by following three different possible strategies.

Static Rendering (Default)

With static rendering, route rendering occurs at build time, or in the background if the data has been re-validated. The result is cached and can be sent to a Content Delivery Network. This optimization allows the result of the rendering process to be shared between user and server requests. This type of rendering is useful when a route contains data that is not personalized to the user and known at build time, such as a product page in an e-commerce site or a post in a static blog.

Dynamic Rendering

With dynamic rendering, routes are rendered for each user request at request time. this strategy is useful when the route contains data personalized for the user or information known only at request time, such as cookies or search parameters in the URL.

It is important to note that in most websites, routes are not fully static or fully dynamic. For example, an e-commerce page may use cached product data that's re-validated at an interval, but also has uncached, personalized customer data. Next.js allows having dynamically rendered routes that have both cached and uncached data. This is because the *RSC Payload* and data are cached separately, allowing to opt into dynamic rendering without worrying about the impact on performance by fetching all data at request time. During rendering, Next automatically switches from static to dynamic rendering if dynamic functions (functions that rely on information known only at request time) or non-cached data are detected.

Developers don't have to choose between static and dynamic rendering as Next.js will automatically choose the best rendering strategy for each route based on the features and APIs used. Instead, it is possible to choose when to cache or re-validate specific data, specifying cache properties within data fetches and selecting data re-validadion strategy.

Streaming

The streaming technique enables to progressively load the UI from the server, implementing the *Streaming SSR* pattern explained before. The rendering process is divided into chunks and each portion is sent to the client as soon as it is ready, without waiting for all the components of the page to be ready. This allows the user to see parts of the page immediately, before the entire content is rendered. The approach works well with React's component model because each component can be considered a chunk.

Streaming is implemented by default within the App Router structure and this improves the performance of both the initial loading of the pages (reducing the Time To First Byte) and the UI which depends on slower data retrieval, which would otherwise block the entire route. For example in the case of reviews on a product page on an e-commerce site.

5.2 Remix.run

Remix represents a solid evolution in the landscape of full-stack web frameworks. Built upon the power of React, it offers a fast, smooth, and resilient user experience. Since its official launch in October 2021, Remix has gained significant popularity, establishing itself as one of the reference frameworks for the React community. The widespread adoption of this framework is driven by the numerous advantages it offers, which, despite being relatively recent and continuously expanding, make it reliable and high-performing [18].

Remix is characterized by its extremely fast bundling achieved through *esbuild*, a JavaScript/CSS bundler and minifier, ensuring efficient code compilation. Recently, with the release in February 2024, is it possible to use Vite instead, providing all the advantages that come with this tool.

On the server side, it adopts a progressive enhancement approach by sending only the necessary resources—JavaScript, JSON, and CSS to the browser, thus ensuring faster initial loading and improved performance. Moreover, server-side dynamic rendering, similar to Next.js, allows the server to render React components and send pre-rendered markup to the client, enhancing SEO, reducing Time To First Paint, and improving overall user experience. Automatic data re-fetching, when modified or updated, is handled by the framework, eliminating the need for manual re-validation. Furthermore, Remix integrates key tools such as React Router, a production server, and backend optimizations, providing a comprehensive solution without the need for separate configurations and integrations.

5.2.1 Routing

As mentioned, routing is built on top of React Router, since React Router is developed by the Remix team itself. Remix adopt the file-based routing system, similar to Next.js Pages Router routing system with more advanced features.

Routes are associated with individual files in the project structure, this means that when a file is introduces in the *routes* folder, Remix inherently understands it as a route. Nested routing can be implemented, other than with basic file-based hierarchy, separating route segments with dots within the file name.

```
app/routes/profile.tsx            →   /profile
app/routes/profile.settings.tsx   →   /profile/settings
```

This modular design supports also dynamic segments to match segments of non-static URL and use that value in the code. The convention requires prefixing the file name with the symbol `$`. This can be combined, similarly to Next.js, with Optional Segments by wrapping a route segment in parenthesis. This capability enables developers to handle optional parameters within the URL structure, accommodating various use cases without the need for complex routing configurations.

```
app/routes/users.$name.tsx          →   /users/John
app/routes/($lang).$category.tsx    →   /en/technology
app/routes/($lang).$category.tsx    →   /technology
```

The filename maps to the route's URL pathname except for `_index.tsx` file that is used as the entry point for the corresponding segment. The index route is essential for the root route, it define the root layout shared among the application. Additionally is possible to define an index route for each route segment, providing nested layouts to obtain higher modularity.

This approach enables the ability for several routes in the nested route tree to match a single URL. This granularity ensures that each route is primarily focused on its specific URL segment and related slice of UI. This approach champions the principles of modularity and separation of concerns, ensuring each route remains focused on its core responsibilities.

Thus Remix's routing system not only simplifies the organization of project files but also enhances modularity and code maintainability. By associating routes with individual files, developers can easily locate and manage different sections of the application, promoting a clear separation of concerns, as each route remains focused on its specific functionality, allowing for the creation of user-friendly navigation experiences while ensuring consistency and scalability across the application.

Remix also offers other features that simplify the development of a web application, such as simplified state management, which reduces the need for tools like Redux and React Context and providing a large and fast data flow, easily transmitting information between frontend and backend, leveraging the full-stack architecture to seamlessly synchronize state between the client and server. This ensure consistency and enable developers to easily share data between components and routes, reducing the need for complex data sharing mechanisms. It also facilitates the creation of smooth transitions between pages with the use of Optimistic UI, providing instant feedback to users while asynchronously processing data updates in the background. This approach enhances user experience by reducing perceived latency and improving responsiveness. As for error handling, it is built in natively, allowing each route to define an error handling function.

5.2.2 Rendering and Server Components

Remix, unlike Next, does not implement React Server Components. This does not mean that what we have seen so far is not inherent to the topic that guided the development of this thesis. Indeed the approach of Remix, although the first versions in 2018 implement experimental versions of *RSCs*, is to fully exploit the functionality of technologies such as Suspense, *RSCs*, and *SSR Streaming* through a "Full-Stack Framework" structure that emphasizes and maximize the functionality of these existing technologies. In this way you benefit from co-locality of the code, which from the developer's point of view, allows you to consider the route modules as server components.

This means that the operation is very similar to what happens in Next or internally to the *RSCs*. This is possible thanks to Fullstack Data Flow, a set of features that allows to automatically keep the UI synchronized with the persistent state of the server. This process unfolds in three stages:

- Route Loader: The route loader provides data to the UI, specifically the route component. This function executes automatically when a user navigates to a particular route. Then, after loading the data, rendering takes place.

- Action Routes: If the route component involves data that can be modified, such as via a form submission, the corresponding action route, a server function, is responsible for managing mutations and the resulting actions. These functions are defined as HTTP requests (GET, POST, PUT, DELETE or PATCH) as required, following the same loader structure. This setup enables co-location of a data set within a route module where data reading, component rendering, and data writing occur in the same location. While there are alternative methods to respond to data modifications, such as event listeners or file loading, actions form the foundational mechanism.

- Page Loader Revalidation: Following any data modification, the page loader is automatically re-validated, ensuring that updated values are seamlessly reflected.

In addition, studies have shown that the current server rendering performance of Remix is comparable to, if not better than, the results obtained by using frameworks such as Next through the use of *RSCs* [5].

Remix developers express confidence that when *RSCs* become viable for integration into Remix, migration will be as straightforward as renaming a file for a route. This indicates a forward-looking approach, anticipating future advancements in React's ecosystem and their seamless integration into the Remix framework.

5.3 Other Frameworks

During the development of this thesis work, other frameworks were studied to explore the implications of Server Components in the current state of literature.

The frameworks shown in this section have been considered less thoroughly due to their recent introduction and evolution, as well as the technologies they implement, which deviate from the enterprise application stack. However, it is important to have a complete overview of the frameworks and tools that can be used to implement server rendering and in particular *RSCs*.

5.3.1 Modern.js

Modern.js is a progressive web framework based on React from ByteDance [9].

This framework supports all configurations and tools needed by React applications and has built-in additional features and optimizations. Developers can use React to build the UI of the application, and then gradually adopt the features of Modern.js to solve common application requirements, such as routing, data acquisition, and state management.

Modern.js is based on Webpack and provides out-of-the-box Rspack support, a high performance JavaScript bundler based on Rust.

Modern.js was heavily inspired by Remix.js in routing and SSR and, at the same time, Modern.js' routing solution is based on react-router, which is created by the Remix team.

Additionally it implements React 18 to build user interfaces and it can be used with any community state management library, such as Redux or Recoil.

Modern.js also provides Micro Frontends support through the use of the Garfish tool, but this approach is still in the early stages of development and cannot be integrated with other technologies, such as Module Federation discussed above.

As for rendering strategies, Modern.js provides support for Client Side Rendering, Server Side Rendering, and Static Site Generation, allowing developers to choose the rendering mode they need. The benefits of using *SSR* in Modern.js are ease of use, developers do not have to write complex configurations or server side logic, nor do they have to worry about the operation and maintenance of *SSR*. Additionally, it implements a complete *SSR* downgrade strategy to ensure that the page can run safely and a built-in caching system to solve the problem of high server side load.

Thanks to these aspects, Modern.js can be considered a valid alternative to a framework like Remix.run, although it is relatively recent and still in the expansion phase. Therefore we are not yet inclined towards its use in a corporate environment.

5.3.2 Vue.js

Vue framework was previously illustrated in section 3.2.6 and as said, it is not a React-based framas well as its potential for integration with React. Regarding server rendering concepts, it is interesting since it is possible to directly implement the server request handler through the declaration of a Node.js server for basic solutions, but even more so thanks to the built-in support for Vue Server Side Rendering provided by Vite. Vite offer `vite-plugin-ssr` that abstracts away the challenging of configuring the server responsible for handling server rendering requests.

Additionally from Vue also came Nuxt, a higher-level framework built on top of the Vue ecosystem which provides a streamlined development experience for writing universal Vue applications. This framework has built-in SSR capabilities by default, that do not require a manual server configuration.

Although this framework's approach is interesting, it has not been explored in depth as it does not fit into the main technology stack. However, given its compatibility with Module Federation plugin, it should be possible to integrate it within a React project, leaning on additional libraries to perform the implementation.

5.4 RSCs from Foundations

An important feature of React Server Components is the opportunity to use them without the need of framework support or external systems and technologies for deployment and use. As mentioned above, this approach is discouraged by the developer team behind *RSCs*. Therefore the following is the result of a study and research phase for educational purposes rather than an implementation of production-ready tools. In fact, this phase served to understand more in depth the functioning of the *RSCs*, rather than to understand how to use them.

The study starts from a tutorial [6] showing how to construct React server components from scratch using TypeScript mimicking Next.js App Router structure and mechanisms. This can be considered as a cornerstone for understanding the mechanism behind React's component lifecycle, state management, and data fetching primitives.

This approach involves manually configuring and setting up a server using Express, a popular Node.js web application framework, used to define routes and handle requests for the React application.

Express server is configured using the `express.static` middleware to serve static files from the directory that contains the bundled JavaScript files generated by the React build process (the server-side and client-side code, as well as any other static assets required by the application).

5.4.1 Routing

The server defines routes to handle different types of requests and implements route handling in a manner similar to Next.js by dynamically importing React components based on the requested route. For example, the route handler for `/:page` dynamically imports the appropriate page component based on the requested page name. Additionally, the route handler for `/:dir/:email` demonstrates handling dynamic routes with multiple parameters.

5.4.2 Component Rendering and Hydratation

One of the core features of the server configuration is its ability to render React components on both the server and client sides. This process involves rendering React components into HTML markup on the server and then hydrating them on the client side to attach event handlers and state.

When a request is made to the server, the server dynamically imports the appropriate React component based on the requested route. The `createReactTree` function asynchronously constructs the React component tree. It supports rendering both functional and class-based components, allowing for a flexible and modular architecture.

The server then utilizes `renderToString` from *react-dom/server* package to convert the React component tree into HTML markup. This HTML markup includes the initial server-rendered content, ensuring that the user receives a fully populated HTML document upon the initial request. This approach exploits the benefits of *SSR* by providing modularity and componentization derived from *RSCs*.

After the initial HTML markup is delivered to the client, the client side code, typically contained in `index.tsx` file, takes over the rendering process. The HTML markup received from the server is hydrated using the `hydrateRoot` function from *react-dom/client* package. Hydration makes the components interactive and responsive. During hydration, React reconciles the server rendered HTML with the client side React components. It compares the existing HTML structure with the React component tree, preserving any client side state or user interactions that occurred on the server rendered page. This process ensures a seamless transition from server rendered content to client side interactivity, providing a consistent user experience.

Part III

Integration in Enterprise Application

Chapter 6

R&D Outcomes and Thesis Direction

6.1 Overview of Research Outcomes

6.1.1 R&D Outcomes

Based on the R&D activities, the research on adopting advanced enterprise patterns such as Micro-Frontends and Server-Side-Rendering can be summarized as follows and with the outcomes shown below, in figure 6.1.

Regarding micro frontends, the adoption of this architecture offers numerous benefits and, as explained, the most suitable approach requires Module Federation through the use of bundlers like Vite.js and Webpack.

On the other hand, *RSCs* has emerged as a promising technology, yet still in its nascent stages and undergoing rapid expansion. Consequently, the focus of the server rendering approach shifted towards *SSR* techniques. This strategic shift was driven by the need for stability and widespread support within existing frameworks, such as Remix, Next.js and Modern.js, particularly in the context of enterprise application development, with *SSR* providing several performance improvements. Notably, some frameworks like Next.js have begun integrating *RSCs* directly into their architecture, offering a glimpse into the future potential of this technology.

However, successful implementation of *SSR* depends on fulfilling framework requirements, such as Module Federation for micro frontends integration. Overall, the research underscores the importance of carefully selecting and integrating frameworks to leverage the benefits of micro frontends architecture and *SSR* effectively.

The initial finding of the research phase is that incorporating Server-Side Rendering (*SSR*) practices into an already-deployed Enterprise Web Application, without the need for a complete rewrite, may depend on adopting *Micro-Frontends*. Hydrating the application with *SSR* requires significant maintenance and development efforts, which may be

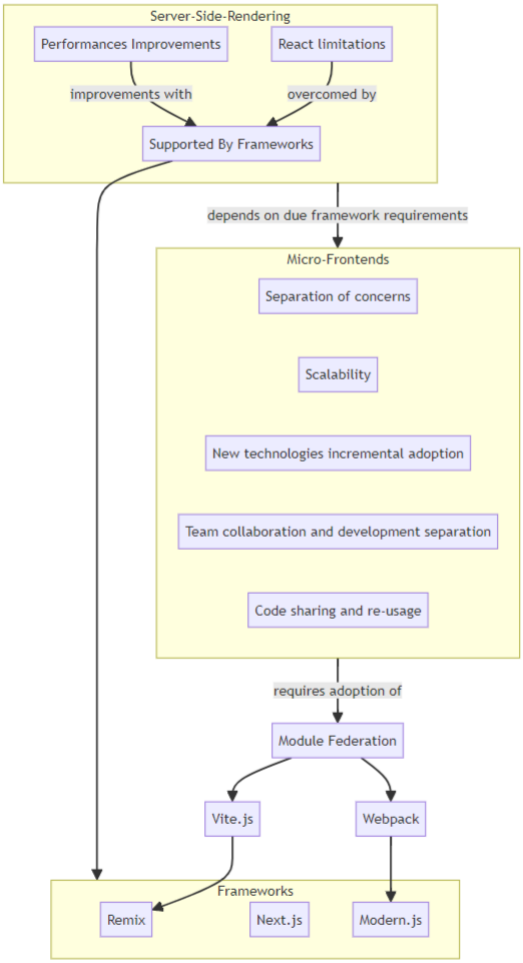

Figure 6.1. R&D Outcomes

challenging for a standard team. Therefore, it is recommended to consider adopting a framework (as suggested by the React development team) that provides a pre-packaged architecture and meets the requirements. However, adopting a framework typically involves a complete rewrite of the application. Alternatively, a more effective strategy could be to modularize the application and gradually adopt *SSR* as needed, with the assistance of external frameworks.

6.1.2 Challenges in Incorporating SSR

While incorporating Server-Side Rendering practices into an already-deployed Enterprise Web Application can bring significant benefits, there are also some challenges to consider:

- **Maintenance and Development Efforts**: Implementing *SSR* requires ongoing maintenance and development efforts. It involves managing server-side rendering

configurations, ensuring compatibility with different libraries, and addressing potential performance bottlenecks. These tasks may require additional resources and expertise.

- **Learning Curve**: Adopting *SSR* often involves learning new concepts and technologies. Teams may need to familiarize themselves with optimizing rendering performance and troubleshoot any issues arising during the implementation process.

6.1.3 The gradual Adoption of SSR

To overcome the challenges mentioned above in incorporating Server-Side Rendering, one effective approach is to leverage the benefits of micro-frontends. By adopting a micro-frontends architecture in modularized applications, teams can gradually introduce *SSR* to specific pages or components (modules). This allows for a more controlled and manageable adoption of *SSR* without the need for a complete rewrite of the application. It reduces the challenges and risks associated with a full-scale implementation and allows teams to gradually incorporate *SSR* practices as needed. By modularizing the application and introducing *SSR* to specific pages or components, teams can incrementally enhance their application's performance and user experience while minimizing the maintenance and development efforts required.

6.2 Selected Frameworks and Technologies

Module Federation is a key requirement for adopting Micro-Frontends, as it allows for seamless integration and communication between different micro-frontends. To implement Module Federation, the adoption of Vite.js and Webpack is necessary. These tools provide the infrastructure and configuration to enable module sharing and dynamic micro-frontend loading.

Regarding Server-Side Rendering, several frameworks are available that support this approach. Remix, Next.js, and Modern.js are frameworks that provide built-in support for *SSR*. These frameworks offer the necessary tools and libraries to implement *SSR* effectively, including handling server-side rendering configurations, optimizing rendering performance, and enhancing interactivity.

The choice of frameworks for adopting Micro-Frontends and Server-Side Rendering depends on the bundler used in the project. If Vite.js is utilized as the bundler, it is recommended to consider using Remix for Server-Side Rendering. Remix is a framework that provides built-in support for Server-Side Rendering and offers the necessary tools and libraries to implement *SSR* effectively. On the other hand, if Webpack is the preferred bundler, Modern.js should be considered. Modern.js is another framework that supports Server-Side Rendering and provides the required features to optimize rendering performance and enhance interactivity. Selecting the appropriate framework based on the bundler used ensures compatibility and seamless integration within the project's development environment. Next.js is not a suitable solution, due to limited compatibility with the concept of module federation, only with Pages Router is it possible to exploit the Webpack plugin.

As previously mentioned, is it possible to adopt Vite Module Federation Plugin through the usage of Vue framework and a research was conducted about the possibilities of integrating remote Vue applications within the main React Project. Although this is possible, it presents some limitations related to the mandatory use of external libraries to provide correct handling of Vue components within React. It turns out that the support of Module Federation for these libraries is limited and it was not explored further during the development of the final project.

6.2.1 Implementation

The following chapter concludes the thesis work by presenting a demo project created to explain the best strategies and the approach to integrate the concepts studied into a business application. This demo was demonstrated within the company to begin the process of gradually adopting the technologies in an application already in use by several customers, in order to bring sustainable benefits that improve its performance.

Chapter 7

Demonstration of Integration

This chapter illustrates a demonstration application, designed to blend the concepts covered during the thesis project and provide the business development team with a solid representation of the work that can be done thanks to this research.

7.1 Project Overview

7.1.1 Introduction

The project is centered around a main *React* application acting as host application. React is utilized without additional frameworks to align closely with the current initial situation of the company application.

In fact, this allows to gradually restructure the central application already distributed. This is done by modularizing the main application via micro frontends and consist of moving the modules considered at least semi-independent within separate remote applications. The process can be done one step at a time for each module, keeping the main application running; actually, it is sufficient to configure the remote modules at the same time as they are moved and import them in the main application.

The Benefit of Modularization

The approach is especially useful, given that the web application is sold to different customers (other companies), who may only need certain features offered by that application.

A monolithic application would require an effort on the part of developers to modify the code according to customer needs, increasing the complexity of project management. In fact, it would be necessary to maintain separate repositories and codebases for each customer, leading to possible problems of inconsistency, dependency management or errors related to incorrect code modifications. Instead, this would not be a problem with micro frontends, as the development team would only need to adjust the configuration of remote modules from the host application and remove remote apps that implement unwanted or unnecessary functionality.

7.1.2 Technology Stack

This section provides an overview of the main technologies and libraries used in the project, including:

- *Typescript*: A statically typed superset of JavaScript that enhances code maintainability and scalability.
- *Vite Bundler*: A next-generation frontend tooling that leverages native ES module support to provide fast build times.
- *Vite Module Federation Plugin*: A plugin from Vite that enables module federation, facilitating the integration and communication of micro frontends within the application architecture.
- *React*: JavaScript library for building user interfaces, used as the foundation for the host and remote applications.
- *Carbon Design System* for React: A React implementation of the Carbon Design System, providing consistent UI components for the application.
- *Recoil*: A state management library for React applications, used for managing application state across components.
- *Sass*: A CSS preprocessor that extends CSS with features like variables, mixins, and nesting, used for styling the application.
- *Tailwind*: A utility-first CSS framework that provides pre-built styles and encourages composing designs using small, single-purpose classes.
- *PostCSS*: A tool for transforming CSS with JavaScript plugins, enabling various tasks like autoprefixing, minification, and syntax enhancements.
- *Autoprefixer*: A PostCSS plugin that automatically adds vendor prefixes to CSS rules, ensuring compatibility across different browsers.

The decision to utilise these tools was based on the technology stack employed within aizoOn applications, with the aim of aligning with the actual use case.

7.2 Project Structure

The overall structure consists of four separate applications. As mentioned, there is a main application, host, while the other remote micro-apps offer different features that are used within the host application.

The different applications are illustrated below, starting with `config-app`, an application that offers configuration functionality for the entire project. Afterwards, the main `host-app` application and the various remote micro-apps are presented, `tables-app`, `datatable-app`, which offer simple functions to demonstrate the functioning of concepts, methodologies and tools designed to be useful for application in the company.

An important aspect is how these remote applications are imported and integrated in the host app. As discussed, it is possible to perform static and dynamic imports with Vite Module Federation Plugin and in this project both strategies are implemented to illustrate the differences.

7.2.1 config-app

It's a React application, bundled using Vite. Through module federation the application exposes elements for authentication and configuration. Below are presented the modules exposed by this remote application.

configuration

It contains a method named `getCompanyConfiguration` which reads and returns the content of the configuration file accordingly to the company name provided (defined with login). This file, called `configuration.json`, provides information about customer companies using the application; in particular it specifies, for each company, which remote applications and which modules within the remote they have access to. The information about each remote app concerns remote name, remote entry point file name, name of the different modules of that remote that the given company has access to (in fact a remote app can expose multiple modules and a company may have access only to a sub set of them), lastly also the address from which to load the entry point file is provided. The figure 7.1 shows an example of the file contents.

```
{
  "AizoOn": [
    {
      "name": "react_table",
      "filename": "remoteEntryTablesApp.js",
      "modules": ["DynamicTable", "DataTable", "ModuleName"],
      "address": "./tables_dist/assets/"
    },
    {
      "name": "react_datatable",
      "filename": "remoteEntryDataTableApp.js",
      "modules": ["DataTable"],
      "address": "http://localhost:8082/assets/"
    }
  ],
  "Politecnico": [
    {
      "name": "react_table",
      "filename": "remoteEntryTablesApp.js",
      "modules": ["DynamicTable"],
      "address": "./tables_dist/assets/"
    }
  ]
}
```

Figure 7.1. configuration.json

It is important to note that this approach is implemented only to show in a simple way how this method allows a dynamic and flexible configuration of remotes: the developer

team can modify this file by adding or removing remote applications and exposed modules without needing to manipulate the code. When this is integrated into the company project, a better solution will be implemented than using a JSON file that takes into account the security and organization aspects that have not been the subject of in-depth studies in this demo.

Remote

This is a file declaring a custom Typescript type used to provide type safety when manipulating data coming from `configuration` method. In fact, the content of the file shown in figure 7.1 can be described as a collection of keys representing companies name whose associated value is an array of `Remote` objects.

LoginComponent

This module is a React element that provide an interface, implemented using components from Carbon library, to perform authentication by selecting the company and the user role. As said selecting a company allows to have access to different modules according to the configuration file. Additionally also selecting the type of user role(for example Admin, User) allows to perform different actions inside the project.

Also for this case the authentication process is simplified, no password or verification system is implemented as it is not fundamental for the study conducted.

The interesting feature implemented in this component is that it receives as props two methods that respectively takes the values of the selected company and user role; so the logic for providing the authentication in the entire project is handled outside the remote component within the main application, providing flexibility in the implementation.

atoms and *selectors*

These two modules are two files containing, the first one the Recoil atoms for the company name, the user role and the configuration, the latter a recoil selector that brings together all the three atoms information and returns them. The company and role atoms are set during login while the configuration value is the content of the configuration JSON file, according to the selected company.

These modules are used to provide state management in the entire project, providing consistency and up to date data.

Declaring and configuring atoms in the host app and share them to other remote applications could be seen as a more simple solution, but it may lead to inconsistency problems or race conditions while accessing them. So the approach of declaring components in remote micro-app, as implemented here, has emerged as more suitable and effective.

7.2.2 tables-app

This remote application provide modules with module federation that are implemented to demonstrate the possibilities that this technology offers for sharing React components.

DynamicTable

This module is a React Component, created to represent a table using elements from the Carbon React library. It is a dynamic table in the sense that it receives as a parameter a "data" object containing the data to be represented within the table whose structure is composed of an array of objects of the same type not known a beforehand. In fact, it reads the fields key and create the table header accordingly, allowing to represent different types of data without modifying the code.

This is a simple case showcasing how it is possible to use a component from a remote application to represent data coming from a different application.

DataTable

Similar to the previous, also this module is a React Component that uses Carbon React library to represent a table. In this case the values are fetched internally from an asynchronous api within the remote application. So the remote application exposes this module while keeping the logic for data loading, providing to the host application the table element without the app being aware of the internal workings.

So differently from the previous case it demonstrates how it is possible to share a completely independent element.

api

This module is a file containing an api function used to asynchronously fetch data. The method creates and resolves a Promise to fetch data while handling errors and return the data as a JSON object.

It demonstrates how it is possible to share methods other than React Components. This function can be used in the host application providing an await mechanism as a traditional asynchronous api.

It is possible to combine the use of data loaded from this api with the element provided by the `DynamicTable` module.

7.2.3 datatable-app

This remote application is similar to the `tables-app` and it was created to demonstrate how to use and share styling properties from a remote component.

DataTable

This module is the only one exposed by the application. Basically it is a React Component copied from the `DataTable` component of the `tables-app` remote with the difference that Tailwind styling has been added.

As explained in the theoretical part, in order to use Tailwind correctly in a remote component shared with the main application, it is necessary to provide a configuration for PostCss and Autoprefixer. The advantage of this is that it is not necessary to share also the library and the host application, as in this case, may not even use it. In fact the

host app doesn't have Tailwind among its dependencies but the remote component, when imported in the host app, is correctly rendered with the applied CSS styles defined with Tailwind.

7.2.4 host-app

The React host application acts as a container, connecting together the functionalities provided by the other remote micro applications. The main features are the structure provided by React Router for navigation and how the remote modules are actually imported.

Routes and Navigation

Through the use of React Router it is possible to navigate between different pages and interact with the application and all the remotes illustrated above.

- `/`: the route shows different pages depending on the authentication. If the user is not authenticate it shows the login page, which imports the login component interface from the remote and save the selected credentials inside `localStorage` just for simplicity. If the user is authenticated it shows the home page. This page is a simple landing page with a layout structure that is common to the page of other routes. The layout consists of a top bar and a left bar menu. The top bar allows to navigate to home page and perform logout, while the menu provides navigation to the other routes.

  The interesting aspect of this page is related to the left bar menu, which items depends on the company credentials selected from the login process. In fact, after login is performed the application knows, from the configuration file, which remotes and modules the user has access to. So the navigation menu provides link items accordingly to the content of the configuration file. The figures 7.2 and 7.3 below show the differences within the menu for different companies.

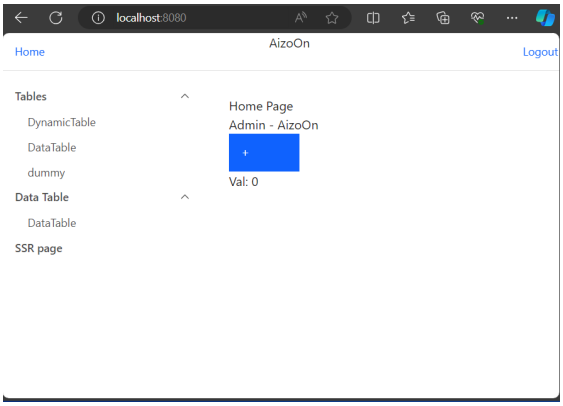

Figure 7.2. Home Page AizoOn Login

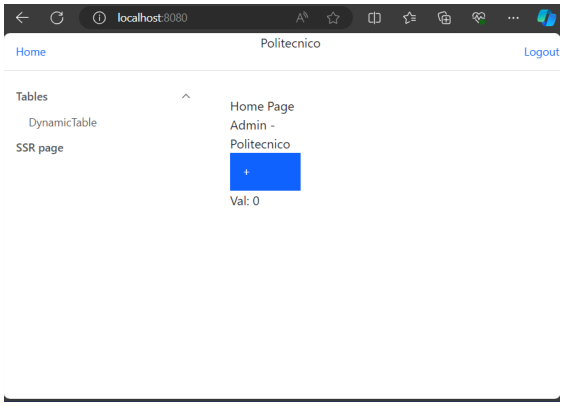

Figure 7.3. Home Page Politecnico Login

- `/tables/:module`: this route is accessible only if the user is authenticated. In fact, the user can navigate here by clicking on one of the items of the Tables drop down menu in the left panel. The parameter `module` depends on the clicked link of the menu and it is used to import and load the corresponding module that has to be shown in the page.

  So this route actually map the page to show all the remote tables exposed by `tables-app` and `datatable-app`. It also handles eventual errors when loading the remote modules without compromising the functioning of the application.

- `/ssr`: this route was implemented just to demonstrate the functionalities of React Router V6, implementing the loader for the route. Here the remote `api` module is loaded to fetch data inside the route loader; then the shown page imports the `DynamicTable` component from the remote to render the fetched data.

Remotes Imports

Previously in section 3.2.5, it is explained how dynamic imports work and how to implement this approach for remote modules. In the presented demo project, dynamic imports are an interesting concept. In fact following this approach allows to avoid writing the remotes configuration in the `vite.config.js` file of the host app: it is sufficient to modify the `configuration.json` file in the configuration-app remote application.

Another important aspect is that the host application loads the remote entry point file from a given address. During the development phase the applications run locally within `localhost`, so each remote application can expose the remote entry file on different ports. In a more complex scenario that involves multiple remote micro application, maintaining all of them running at the same time may raise performance issues, even more problematic when the project is bundled and distributed in production mode.

This aspect can be improved, thus avoiding having as many separate servers running as there are remote applications, by moving the bundled folder generated in output from the remote application build phase inside the one generated by the host application. Additionally it is necessary to modify the address from which to load the remote entry file to match the path.

As it is shown in the previous section, in figure 7.1 both approaches can be maintained concurrently for different remote micro applications, specifying in the configuration file if the remote application is running on a certain address exposing the entry file, or it can be found in the bundled folder.

7.3 Demo Project Shortcomings

Some concepts studied during the initial phases of this thesis work were not implemented within the demonstrative project. The main absence is related to a framework that provides server rendering.

The Remix framework was not integrated inside the project because, even if the implementation of Vite and module federation is feasible, the aspect related to server side

rendering, such as the loader can not be shared within the exposed module. The main problem is related to the fact that Vite Module Federation plugin works only from the client side, so it is not compatible with the server rendering provided by the framework.

This emerged during the development phase and after some studies and insights an alternative solution was thought of. A possible strategy is the one that uses Remix to implement the main host application, with this approach is possible to provide server side rendering from the main app while loading components from remote micro applications. The drawback of this approach is that it requires a complete redesign of the current application abandoning the modularization strategy, that is a fundamental aspect related to micro frontends architecture implementation. Ultimately, this solution requires an unsustainable effort and was therefore not considered valid.

The biggest limitation that emerged from the development of this study is related to the incompatibility of the Module Federation plugin with server rendering. In fact, the Vite plugin works at runtime, only on the client side. This means that `__federation__` methods are not available server-side, so a remote component cannot be exposed with a server rendering strategy. This limited even more the possibilities of integrating different rendering strategies within a micro frontends architecture.

With the current state of offered services, more priority was given to the micro frontend approach that can actively provide advantages to the development and usage of the application.

Furthermore, the integration of other frameworks, such as Vue, was tested, but the impossibility of integrating these frameworks in a "native" way emerged; it is necessary to rely on libraries for the conversion of exported components, leading to lower performance regarding loading and user experience.

Chapter 8

Conclusions

With the development of the demonstration project, this thesis work comes to an end. But the creation of the demo is only the first step of a long term path that involves the adoption of the explained concepts within company applications.

8.1 Outcomes

8.1.1 Micro Frontend Architecture

Among all the analyzed and studied approach, implementing Micro Frontend comes out as a widespread solution that pairs well with the microservices backend architecture. It is possible to follow different approaches and the use of Module Federation was found to be the most suitable and closest to company needs and requirements.

The choice between Vite and Webpack plugins for Module Federation depends on several factors, for our case the choice was guided by having to start from an already existing application, which uses Vite as a bundling tool.

Although the Vite Module Federation plugin is newer and still waiting for some improvements and expansions compared to the plugin offered by Webpack, it can be utilized in production ready application and the strategy of gradually redesign the company application to implement the architecture is the optimal approach.

8.1.2 Server Side Rendering

The initial focus was on React Server Components as it seemed they could be a revolutionary technology capable of revolutionizing the rendering paradigm. After an in-depth study, however, it emerged that this technology is poorly developed and integrated and, at present, not ready to be used in a corporate context.

The integration of frameworks that implements server rendering strategies turned out to be cumbersome and complex, while leading to limitations issues and concerns with modularity and scalability of applications. In particular, the integration of frameworks such as Next and Remix within a micro frontend architecture did not bring satisfactory results, thus leading to having to make a choice on which aspect to focus on. Thus giving

priority to the architectural aspect of micro frontend integration and the advantages that this approach provides.

8.2 Future Works

As said, the conducted research culminated with the realization of the demonstration project with the objective to show the features, the advantages and the drawbacks offered by the micro frontend approach and by the different rendering techniques, in order to proceed with the best strategy for the company application.

From this point, the work will continue by putting what has been studied into practice. We will proceed with the modularization of an already deployed application and with the gradual adoption of a micro frontend architecture.

At the same time, we stay up-to-date with advancements in server rendering technologies. We are particularly interested in the upcoming release of React 19 [8], which may introduce significant improvements in the area of React Server Components, thus providing a possibility of integrating this powerful tool at a later time.

Bibliography

[1] Gervais Yao Amoah. Exploring rendering strategies, 2023. URL https://dev.to/gervaisamoah/exploring-rendering-strategies-csr-ssr-ssg-and-isr-p5a.

[2] Bhargav Bachina. 7 different ways to implement micro-frontends with react, 2020. URL https://medium.com/bb-tutorials-and-thoughts/7-different-ways-to-implement-micro-frontends-with-react-907b5e262230.

[3] Nitsan Cohen. How to share states between react micro-frontends using module-federation?, 2023. URL https://blog.bitsrc.io/how-to-share-state-between-react-micro-frontends-using-module-federation-f3762996c208.

[4] Hiren Dhaduk. Micro frontend architecture: The newest approach to building scalable frontend, 2023. URL https://www.simform.com/blog/micro-frontend-architecture/#:~:text=Micro-frontend%20architecture%20is%20a%20strategy%20in%20which%20the,experience%20and%20is%20easy%20to%20modify%20and%20scale.

[5] Ryan Florence. React server components and remix, 2021. URL https://remix.run/blog/react-server-components.

[6] Tejas Kumar. React server components from scratch, 2023. URL https://github.com/TejasQ/react-server-components-from-scratch/tree/main.

[7] Deepak Maheshwari. Introduction to micro frontend architecture, 2021. URL https://medium.com/nerd-for-tech/introduction-to-micro-frontend-architecture-13f71f8333b.

[8] Meta. React 19.0.0, 2024. URL https://github.com/facebook/react/milestone/40.

[9] Modern.js. Modern.js documentation, 2024. URL https://modernjs.dev/en/guides/get-started/introduction.

[10] Module Federation. Webpack module federation documentation, 2023. URL https://module-federation.io/docs/en/mf-docs/0.2/getting-started/.

[11] Next.js. Next.js documentation, 2024. URL https://nextjs.org/docs.

[12] Next.js. Next.js documentation, 2024. URL https://nextjs.org/docs.

[13] Ricardo Nunez. The problems that react server components solve, 2023. URL https://servercomponents.dev/the-problems-rscs-solve.

[14] Oskari. 4 ways to use dynamic remotes in module federation, 2022. URL https://oskari.io/blog/dynamic-remotes-module-federation.

[15] React Dev Team. React canaries: Enabling incremental feature rollout, 2023. URL https://react.dev/blog/2023/05/03/react-canaries#example-react-server-components.

[16] React Dev Team. React server components, 2023. URL https://react.dev/blog/2023/03/22/react-labs-what-we-have-been-working-on-march-2023#react-server-components.

[17] Remix.run. Remix.run documentation, 2024. URL https://remix.run/docs/en/main.

[18] Remix.run. Remix.run documentation, 2024. URL https://remix.run/docs/en/main.

[19] Jonathan Saring. 4 practical ways to build micro frontends, 2020. URL https://codeburst.io/4-practical-ways-to-build-micro-frontends-4dc4f0b8a921.

[20] Single-Spa. Single-spa documentation, 2024. URL https://single-spa.js.org/docs/recommended-setup/#ui-state.

[21] Vite. Vite module federation documentation, 2023. URL https://github.com/originjs/vite-plugin-federation.

[22] Vite. Vite documentation, 2024. URL https://vitejs.dev/guide/.

[23] Vue.js. Vue.js documentation, 2024. URL https://vuejs.org/guide/introduction.html.

[24] Philip Walton. Web vitals, 2024. URL https://web.dev/articles/vitals.

[25] Webpack. Webpack documentation, 2024. URL https://webpack.js.org/concepts/.

[26] Jiannan Zhang. Feat: dynamic loading of remote support and test demo, 2023. URL https://github.com/originjs/vite-plugin-federation/pull/481.