



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BACHELOR'S DEGREE PROGRAM IN

COMPUTER ENGINEERING

Analysis of Client/Server Rendering Technologies in Web Applications

Thesis in
Web Technologies

Supervisor
Prof. Paolo Bellavista

Submitted by
Jacopo Jop

A.A. 2023/2024

Table of Contents

Table of Contents	3
Abstract	4
Introduction	5
1. Fundamentals of Rendering in Web Development	7
1.1 Client-Side Rendering of Websites	7
1.2 Server-Side Rendering of Websites	11
1.3 Evolution and Trends in the Field	21
2. Design and Development of Test Web Applications	24
2.1 Requirements Analysis	24
2.2 System Design	29
2.3 Conclusion of System Design	35
2.4 Implementation and Development	36
2.5 Final Considerations	45
3. Evaluation and Analysis of Results	46
3.1 Introduction to Performance Testing	46
3.2 Performance Comparisons Between React.js and Next.js	47
3.3 Page Weight Comparison Between Next.js and React.js	50
3.4 Overall Comparison	52
3.5 Future Improvements	53
3.6 Conclusion of Performance Analysis	54
Conclusions	55
Bibliography	56

Abstract

The purpose of this thesis is to compare the performance of Next.js and React.js in the development of web applications, with a particular focus on search engine optimization, loading speed, and resource efficiency. The study analyzes the differences between Client-Side Rendering (CSR) and Server-Side Rendering (SSR), highlighting how Next.js, with its native support for server-side rendering, significantly enhances the user experience compared to React.js. Through practical tests conducted on a real web application, it is demonstrated that Next.js offers faster loading times, better search engine ranking, and more efficient use of network resources, making it a preferable choice for applications where performance and SEO are critical factors. However, React.js remains effective in contexts requiring high interactivity and responsiveness on the client side, such as Single-Page Applications (SPAs).

Introduction

In the context of modern web development, the internship undertaken at Mumble S.R.L. provided an exceptional learning opportunity to deepen expertise in frontend technologies, with a particular focus on React.js. This experience was pivotal in understanding the differences between client-side and server-side rendering methods, topics that will be explored in detail in this thesis.

Mumble S.R.L., based in Modena, specializes in developing cloud-based web and mobile applications for an international clientele, offering innovative solutions for both external clients and internal use. This dynamic and growing environment, driven by young entrepreneurs, provided an ideal setting to develop advanced technical skills and actively participate in practically significant projects.

The internship's primary goal was to develop a web application for analyzing website performance, using React.js for the frontend. However, the project's final form was defined only after an initial training and learning phase, during which familiarity with the necessary technologies was gained. This phase included using Visual Studio Code as the development environment, integrating with Git, deploying on Google Firebase, and utilizing MongoDB as the database. Notably, there was an opportunity to marginally explore Next.js, a framework that extends React's capabilities by enabling server-side rendering, a topic that will be further examined in this thesis.

The main internship project, named *PageSpeed Dashboard*, was developed in collaboration with another intern, sharing responsibilities and tasks. The company provided the necessary tools to work in a shared environment, using Docker to ensure compatibility across different platforms. The frontend was entirely developed in React.js, while the backend was built with Laravel, a PHP framework. For communication between frontend and backend, the Axios library was used, essential for efficiently sending and receiving data, also managing authentication through Laravel Sanctum, an optimal solution for Single-Page Applications (SPAs).

During the internship, in-depth knowledge of React.js was acquired, a library that facilitates client-side rendering, where all page content is processed directly in the user's browser. While this approach offers immediate interactivity and significant flexibility in updating content, it also presents challenges, such as managing the user's device resources and difficulties in search engine optimization (SEO). A training course on Udemy provided a solid foundation, enabling mastery of advanced concepts like React Hooks (useState, useEffect, useRef, useContext) and routing, key elements for developing modern and responsive applications.

The experience gained with React.js allowed the development of practical skills in using Material UI and Tailwind for creating user interfaces, employing advanced methodologies like BEM (Block Element Modifier) and SASS (Syntactically Awesome Style Sheets) for CSS management. These tools proved essential for ensuring a responsive and adaptable design across different platforms, significantly enhancing the user experience.

The core of the project was integrating with Google's PageSpeed Insights API, a service that analyzes a webpage's performance and returns a detailed JSON report. During the internship, advanced features were implemented to monitor performance trends over time, using charts generated with Chart.js and react-chartjs-2. This integration required a deep understanding of both frontend and backend, highlighting the importance of close collaboration between the two for optimal results. PageSpeed forms the basis for the tests conducted in this thesis.

Beyond technical implementation, the internship also involved managing deployment on AWS using Terraform, an infrastructure automation tool that enabled experimentation with Amazon EC2 instances. This step was crucial for understanding the challenges of releasing applications in a production environment and ensuring the project was not only functional but also scalable and secure.

Finally, the experience underscored the importance of a hybrid rendering approach, combining the advantages of client-side and server-side rendering to achieve an optimal balance between performance, interactivity, and compatibility. This theme is central to the thesis, which aims to critically analyze both methodologies, examining real-world application scenarios and proposing solutions to enhance the user experience in diverse contexts.

The understanding gained during the internship, combined with the theoretical analysis that follows in this thesis, laid the groundwork for a broader reflection on web development technologies, their practical applications, and their impact on user experience. Thus, this thesis aims to provide a practical and theoretical guide, based on real-world experiences, to assist developers in choosing the most suitable rendering method for their projects' specific needs, ensuring websites that are not only functional but also highly performant and user-friendly.

1. Fundamentals of Rendering in Web Development

1.1 Client-Side Rendering of Websites

Concept of Client-Side Rendering

Client-Side Rendering (CSR) has become a fundamental technique in modern web development, used to create interactive and dynamic applications^[1]. This approach leverages the power of the browser and the JavaScript language to dynamically generate and manage the content of the user interface, enabling a fluid and responsive experience. The adoption of CSR has been accelerated by the growth of advanced JavaScript frameworks such as React.js^{[4][19]}, Angular, and Vue.js^{[5][20]}, which facilitate the creation of Single-Page Applications (SPAs), where the entire process of navigation and interaction takes place within a single webpage without the need to reload the entire HTML document^[15].

Functioning of CSR^{[2][7]}:

In CSR, when a user visits a webpage, the server sends the browser a basic HTML document that contains only a minimal page structure, along with references to the JavaScript and CSS files necessary to build and manage the user interface. This process primarily involves four phases:

1. **Initial Request:** As mentioned, the browser sends an HTTP request to the server, which responds with a minimal HTML document. This document contains only the basic page structure and includes references to the JavaScript and CSS files necessary to build the interface.
2. **JavaScript Loading:** Then, the browser downloads the JavaScript files specified in the HTML. These files contain the logic necessary to build and manage the dynamic DOM (Document Object Model), which represents the structure of the user interface.
3. **Dynamic Rendering:** Using JavaScript, the browser now dynamically builds the DOM, generating the user interface elements based on the application's state and user interactions. In this process, the page content can be updated without reloading the entire page, significantly improving the user experience.
4. **Event Handling:** Once the DOM has been built, JavaScript continues to handle user events, such as clicks, inputs, and mouse movements. These events are managed directly in the browser, updating the DOM in real-time in response to the user's actions.

Advantages and disadvantages of Client-Side Rendering

Advantages of CSR^{[2][4][5][11][19][20][21]}:

CSR enables the creation of applications with immediate interactivity, where content updates occur without reloading the entire page, enhancing the user experience, particularly in applications where the speed and fluidity of interaction are critical, such as social networks and analytical dashboards. This approach is ideal for applications requiring frequent and real-time content updates, such as financial trading platforms or real-time monitoring systems.

Furthermore, CSR applications can dynamically update specific parts of the page in real-time, reducing the server load. This is particularly useful for applications that handle large volumes of data or require continuous user interaction, such as online collaboration platforms and project management tools.

Finally, CSR allows developers to create modular user interfaces with reusable and independent components. This facilitates code maintenance and makes it easier to add new functionalities or adapt the application to new requirements. The modular approach also simplifies collaboration among development teams, enabling multiple developers to work simultaneously on different parts of the user interface without conflicts.

Disadvantages of CSR^{[2][9][10][11][12][15]}:

One of the main disadvantages of CSR is the difficulty for search engines to correctly index pages. Since the content is dynamically generated in the browser, search engine bots may not execute the JavaScript necessary to view the content, leading to poor visibility in search results. To mitigate this issue, many CSR applications implement partial Server-Side Rendering techniques or use tools like Prerender.io to generate static versions of pages for search engine crawlers.

It should also be considered that the browser must load and execute JavaScript files before it can render the user interface, so the initial loading time can be significantly longer compared to applications that use Server-Side Rendering. This is particularly problematic on slow connections or mobile devices, where processing capacity is limited. Techniques such as lazy loading and resource optimization can be used to improve performance, but they require careful management.

Lastly, managing an entirely client-side application can introduce additional complexities. The management of security, performance, and compatibility across various browsers becomes more complex, requiring greater attention and the implementation of best practices to ensure that the application is secure and performant. For example, protection against attacks such as Cross-Site Scripting (XSS) becomes more complex to manage, as much of the code is executed on the client.

Practical applications of CSR [^4]:

The practical applications of CSR are numerous and cover a wide range of sectors. Applications such as Facebook, Instagram, and Twitter are emblematic examples of CSR. These social platforms leverage CSR to provide seamless user interaction. On Facebook, for example, the timeline dynamically updates with new posts, notifications, and comments without needing to reload the entire page. This is made possible by advanced CSR techniques that efficiently manage user interaction and dynamic content rendering. Similarly, e-commerce applications use CSR to enable users to browse products, add items to the cart, and complete the purchase process without interruptions.

Differences between SPA and MPA[^1][^15][^17]:

Single-Page Applications (SPAs) represent the most common application of CSR. In an SPA, the user interface is entirely managed in the browser, with a single HTML document that is dynamically updated as the user interacts with the application. This contrasts with Multi-Page Applications (MPAs), where each interaction requiring new content involves loading a new HTML page from the server. SPAs offer a smoother and more responsive user experience, but they can be more complex to develop and maintain compared to MPAs.

Client-Side Rendering Process with React.js or Vue.js

React.js and Vue.js are two of the most popular frameworks for implementing CSR, each with distinct approaches to managing rendering and DOM updates.

React.js[^3][^4][^19]:

React.js was developed by Facebook to facilitate the creation of dynamic and modular user interfaces. One of the distinctive aspects of React is the use of the Virtual DOM. The Virtual DOM is an in-memory representation of the real DOM that React uses to optimize rendering operations. When the application's state changes, React updates the Virtual DOM and compares the new version with the previous one. This process, known as "reconciliation," allows React to determine exactly which parts of the real DOM need to be updated, minimizing the number of DOM operations and thereby improving performance.

Example of React.js code:

```
//javascript
function Counter() {
  const [count, setCount] = React.useState(0);

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>Incrementa</button>
    </div>
  );
}
```

In this example, React updates only the text displaying the counter value each time the user clicks the button, without reloading the entire page^[19]. This approach optimizes user interaction, ensuring that only the necessary parts of the interface are updated.

Vue.js^{[5][20]}:

Vue.js is known for its simplicity and flexibility. Like React, Vue uses the concept of a Virtual DOM to optimize rendering, but it also offers a reactive template system that simplifies dynamic data binding to the DOM. Vue allows the user interface to be defined using standard HTML templates, which are automatically updated when the underlying data changes. This reactivity makes Vue particularly suitable for applications requiring dynamic and complex data management.

Example of Vue.js code:

In the following example, Vue automatically updates the paragraph content when the user clicks the button, without requiring a full page reload. This simplicity in implementation is one of the reasons why Vue is widely used in modern web application development.

```
<!--html-->
<div id="app">
  <p>{{ message }}</p>
  <button @click="updateMessage">Cambia Messaggio</button>
</div>

<script>
  new Vue({
    el: '#app',
    data: {
      message: 'Ciao, Vue!'
    },
    methods: {
      updateMessage() {
        this.message = 'Messaggio aggiornato!';
      }
    }
  });
</script>
```

State Management in CSR Applications^{[13][14]}:

State management is a crucial element for the success of CSR applications. In React, state management can be decentralized within components, but it is common to use tools like Redux to centralize the application's state. Redux provides a single global store that maintains the state of the entire application, facilitating the management of complex states and data synchronization between components. Vue.js, on the other hand, uses Vuex, a state management system that seamlessly integrates with Vue's reactivity, offering a robust solution for state management in large-scale applications.

1.2 Server-Side Rendering of Websites

Concept of Server-Side Rendering^{[1][22]}

Server-Side Rendering (SSR) represents one of the most established techniques in the field of web development, enabling the generation of complete HTML content on the server before sending it to the user's browser. This approach has been widely used for many years, especially in the early days of the web, when most pages were dynamically generated using server-side scripting languages such as PHP, ASP, and JSP.

In particular, SSR was introduced to address issues related to the generation of dynamic content on web servers. Technologies like CGI (Common Gateway Interface) allowed the execution of server-side programs that generated dynamic HTML in response to user requests. With the evolution of scripting languages like PHP and ASP, building interactive web applications with dynamically generated and personalized content for each user became simpler. Frameworks such as Ruby on Rails and Django further streamlined this process, integrating business logic directly into the rendering phase.

Functioning of SSR^{[7][22]}:

The SSR process involves several key phases:

1. **Initial Request:** When a user requests a page, the browser sends an HTTP request to the server.
2. **Content Generation:** The server executes the necessary code to generate the page content, which may include accessing a database, processing data, and dynamically generating content. The result is a complete HTML page, ready to be sent to the browser.
3. **Response Delivery:** Once the HTML is generated, the server sends it to the browser, which can display it immediately without requiring further processing.
4. **Loading Additional Resources:** The browser downloads additional resources such as images, CSS files, and JavaScript to complete the page display.

This approach ensures that the user receives a fully formatted page ready for viewing, improving perceived speed and user experience.

Advantages and Disadvantages of Server-Side Rendering

Advantages of SSR^{[1][9][15][22]}:

One of the main advantages of SSR is the ability to generate complete HTML pages that can be easily indexed by search engines. This significantly improves the website's visibility in search results, making it particularly suitable for sites that rely on SEO to attract traffic, such as blogs, e-commerce platforms, and news sites.

Equally important is the user's perception of faster loading times compared to CSR, as the HTML is generated on the server and sent to the browser ready for display. This is

particularly advantageous for users with slow connections or less powerful devices that may struggle to perform client-side rendering quickly.

Finally, SSR ensures greater consistency in the user experience across different devices and browsers, as the rendering occurs on the server and does not depend on the client's capabilities. This reduces the likelihood of compatibility issues across browsers or devices, enhancing the site's accessibility.

Disadvantages of SSR^{[6][9][11][22]}:

Generating complete HTML on the server for each request requires significant resources, especially for high-traffic sites. This can lead to scalability issues, necessitating more powerful servers or load balancing solutions. The infrastructure required to support SSR can be more complex and costly compared to CSR.

Secondly, although the content is immediately visible, the full interactivity of the page depends on the hydration of client-side JavaScript code. This can introduce a brief delay before the user can fully interact with the page, especially if the required JavaScript is complex or takes time to download and execute.

In conclusion, implementing and maintaining an SSR system requires a more complex infrastructure, with careful management of server resources and concurrent requests. It is necessary to balance the load and optimize performance to avoid bottlenecks and ensure rapid responses to users. This may require implementing advanced caching and load balancing techniques, increasing management complexity.

For example, caching rendered pages can reduce server load and improve response times. Additionally, using Content Delivery Networks (CDNs) allows static content to be distributed geographically closer to end users, reducing latency.

Practical Applications of SSR^[22]:

The practical applications of Server-Side Rendering (SSR) are extensive and span various sectors. Websites such as eBay, LinkedIn, and certain sections of Amazon use SSR to ensure faster loading times and better search engine optimization. For example, eBay leverages SSR to make product pages immediately available to users and facilitate indexing by search engines. This is made possible by server-side rendering, which generates complete HTML pages ready for display in the browser without further processing. Similarly, news sites and blogs like **The New York Times** use SSR to provide quickly accessible and easily indexable content, improving both user experience and online visibility. This technique is particularly effective for platforms that heavily rely on organic traffic and SEO, as it ensures content is immediately available to both users and search engine crawlers.

Server-Side Rendering Process with Next.js or Nuxt.js^{[6][24][26]}

Next.js is a React-based framework that extends React's capabilities, providing native support for Server-Side Rendering (SSR). Similarly, Nuxt.js is a Vue.js-based framework that offers comparable features, facilitating SSR implementation in Vue.js applications. Both frameworks simplify the creation of universal (isomorphic) applications, where JavaScript code can run on both the server and the client.

When a user requests a page built with Next.js or Nuxt.js, the server executes the associated React or Vue.js code, building the Virtual DOM and generating the necessary HTML. This HTML is then sent to the browser, which can immediately display the page. Once the HTML is loaded, React or Vue.js "hydrates" the user interface, attaching JavaScript code to existing components to make them interactive.

Next.js^{[24][26]}:

As mentioned, Next.js is an open-source framework based on React, designed to facilitate the development of web applications with advanced features such as Server-Side Rendering (SSR), Static Site Generation (SSG), and automatic resource optimization. Created by Guillermo Rauch and developed by Vercel (formerly Zeit), Next.js has quickly become one of the most popular tools for building isomorphic applications (i.e., those that perform part of the rendering on both the server and the client).

The need for Next.js arose from the increasing complexity of development with React, which, while extremely powerful for client-side rendering, lacked infrastructure for server-side rendering and managing critical aspects like SEO and initial page performance. Before Next.js, optimizing React applications for SEO was a challenge, as Client-Side Rendering (CSR) typically delayed full page loading and could hinder search engine indexing. Next.js addresses these issues by providing an architecture that natively integrates Server-Side Rendering and static pre-rendering, significantly improving SEO visibility and reducing perceived loading times for users.

Key Features of Next.js:

1. One of the most significant features of Next.js is its support for Server-Side Rendering (SSR), which enables page content to be generated directly on the server rather than in the browser. This approach is advantageous for SEO and initial performance optimization. During SSR, the server generates the complete HTML for a given page and sends it to the client, which displays it immediately. Subsequently, **JavaScript** is executed to make the page interactive, in a process known as **hydration**. This means the user sees a fully rendered site right away, improving the browsing experience. In detail, Next.js performs rendering on the server for each request, allowing for dynamic content management. However, Next.js is smart about resource management: it does not generate everything from scratch for each request but can combine SSR with **caching** and **CDN** usage to reduce server load, making the solution scalable.

2. In addition to server-side rendering, Next.js offers a **Static Site Generation (SSG)** system. This approach involves pre-rendering page content at the application's build time, creating static HTML files that can be served very quickly by a server or **Content Delivery Network (CDN)**. This feature is particularly useful for sites that do not require constant dynamic content, such as blogs, documentation, or landing pages. With Next.js, **Static Site Generation** can be combined with dynamic updates through **Incremental Static Regeneration (ISR)**, which allows static pages to be incrementally regenerated when new requests are made. This ensures that static pages remain up-to-date without needing to rebuild the entire site.
3. Next.js introduces an extremely flexible architecture that allows defining **API Routes** within the application. This means the same project can host both the frontend and a series of **serverless functions** acting as a lightweight backend. This integration greatly simplifies the development of complete applications, eliminating the need for dedicated servers or complex backends, as APIs are managed directly by the framework and hosted by services like **Vercel**. Thanks to API Routes, HTTP calls from React to the backend can be handled, seamlessly integrating business logic or database access in a scalable manner.
4. Another innovative aspect of Next.js is **file-based routing**. Instead of manually defining all application routes, Next.js automatically creates routes based on the folder structure in the `/pages` directory. For example, creating an `about.js` file in `/pages` automatically generates the `/about` route. This eliminates complexity in routing management and makes it easy to add or modify pages. Additionally, Next.js supports **Dynamic Routing**, allowing dynamic paths to be created using **parameters** in file names. This is particularly useful for creating pages that depend on dynamic data, such as user profiles or product details in e-commerce.
5. One of Next.js's primary goals is to improve web application performance through various automatic optimizations. One of these is integrated **code splitting**, which automatically divides code into smaller bundles loaded only when needed. Combined with **lazy loading** for deferred resource loading, Next.js significantly optimizes page load times. Moreover, Next.js automatically handles page **prefetching**, making resources for subsequent navigation available in advance. This improves the user experience by reducing wait times when moving between pages.
6. Another fundamental Next.js tool is the **<Image> component**, designed to automatically optimize image loading. This component efficiently handles resizing, compression, and **lazy loading** of images, helping to reduce bandwidth usage and improve page load times.
7. Finally, Vercel, the hosting platform developed by the creators of Next.js, is optimized for Next.js applications, ensuring simplified **deployment** and optimal performance. Vercel supports **automatic scaling**, balancing workloads, and dynamically distributing resources. Thanks to native integration, deploying a Next.js project on Vercel takes just a few clicks, and the infrastructure is automatically optimized to handle traffic spikes and deliver content efficiently.

Next.js is built to be isomorphic, meaning it can execute part of the code on both the server and the client. This hybrid model offers the best of both worlds: the high performance of **Server-Side Rendering** combined with the interactivity of **Client-Side Rendering**. This architecture enables a fast **time-to-first-byte (TTFB)**, improving perceived user speed and increasing the **Page Speed Score**, which is crucial for good search engine rankings.

Improved SEO and UX:

Next.js addresses many of the **SEO** issues that plagued CSR-based applications. With SSR and SSG, pages are pre-rendered and ready for search engine indexing. This improves site ranking and visibility. Combined with tools like React Helmet and Next.js APIs, page metadata can be effectively managed, which is essential for advanced SEO optimization.

Example Implementation with Next.js:

In this example, the **getServerSideProps** function is executed on the server, fetching the necessary data before the page is rendered and sent to the client.

```
//javascript
import React from 'react';

function HomePage({ data }) {
  return (
    <div>
      <h1>Benvenuto nella Home Page</h1>
      <p>{data.message}</p>
    </div>
  );
}

// Funzione di fetching dati che verrà eseguita sul server
export async function getServerSideProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return { props: { data } };
}

export default HomePage;
```


Nuxt.js[^20][^26][^28]:

Nuxt.js is an open-source framework based on Vue.js, created in 2016 by **Sébastien Chopin** and **Alexandre Chopin**, known as the Chopin brothers. The project was born from the idea of extending Vue.js capabilities to facilitate the development of universal (isomorphic) applications, offering an experience similar to that of Next.js but for the Vue.js ecosystem. Nuxt.js was designed to simplify the creation of Vue.js applications with advanced features such as Server-Side Rendering (SSR), static site generation, and hybrid rendering.

The need for Nuxt.js arose from the growing complexity of developing performant and SEO-optimized Vue.js applications. While Vue.js provided a powerful framework for building reactive user interfaces, it lacked an integrated architecture for routing, SSR, and resource management. Nuxt.js was developed to fill these gaps, offering a modular and opinionated structure that allows developers to focus on application logic without worrying about underlying configuration. The framework introduces features such as filesystem-based routing, async data fetching, advanced state management with Vuex, and automatic performance optimization. Nuxt.js's philosophy is based on simplicity and convention over configuration, in line with Vue.js principles, providing an improved development experience and accelerating the time-to-market for modern web applications.

Nuxt.js quickly gained popularity within the Vue.js developer community, becoming one of the primary tools for developing advanced Vue.js web applications. Its ability to simplify the implementation of complex techniques like SSR and static site generation has made Nuxt.js a preferred choice for many projects, from building corporate websites to creating high-performance web applications.

Example Implementation with Nuxt.js:

```

<!-- html pages/index.vue -->
<template>
  <div>
    <h1>Benvenuto nella Home Page</h1>
    <p>{{ data.message }}</p>
  </div>
</template>

<script>
export default {
  async asyncData({ $axios }) {
    const res = await $axios.get('https://api.example.com/data');
    return {
      data: res.data
    };
  }
};
</script>
```

In this example, the **asyncData** function is executed on the server during the rendering process. This function retrieves the necessary data for the page, which is then incorporated into the generated HTML and sent to the client. Upon loading in the browser, Nuxt.js hydrates the components with the provided data, making the application interactive without additional initial API calls.

State Management in SSR Applications^{[13][14]}:

State management is a crucial element in server-side rendered (SSR) applications as well. In this context, the additional challenge is maintaining synchronization between the server and client states. Frameworks like Next.js for React and Nuxt.js for Vue.js provide tools to effectively manage state in SSR applications, ensuring a seamless transition from the server to the client environment.

Example Implementation with Next.js and Redux:

In **React** with Next.js, **Redux** can be used to centralize the application's state, even in an SSR environment. During server-side rendering, the Redux store is populated with the necessary data, and its initial state is embedded in the HTML sent to the client. Once the page is loaded in the browser, Redux hydrates the store on the client using the initial state provided by the server, ensuring state consistency between server and client.

```
// javascript store.js
import { createStore } from 'redux';
import rootReducer from './reducers';

export function initializeStore(preloadedState) {
  return createStore(rootReducer, preloadedState);
}
```

During server-side rendering, the store is initialized and pre-populated with data:

```
/// javascript pages/_app.js
import App from 'next/app';
import { initializeStore } from '../store';

class MyApp extends App {
  static async getInitialProps(appContext) {
    const reduxStore = initializeStore();
    appContext.ctx.reduxStore = reduxStore;
    // Eseguire azioni asincrone per popolare lo store
    await reduxStore.dispatch(fetchInitialData());
    const appProps = await App.getInitialProps(appContext);
    return {...appProps, initialReduxState: reduxStore.getState()};
  }
  render() {
    const { Component, pageProps, initialReduxState } = this.props;
    const store = initializeStore(initialReduxState);
    return (
      <Provider store={store}>
        <Component {...pageProps} />
      </Provider>
    );
  }
}
export default MyApp;
```

In this example, the Redux state is initialized and populated on the server, then passed to the client for hydration. This ensures that the client starts with the same state as the server, avoiding discrepancies and improving the user experience.

In Vue.js with Nuxt.js, state management in SSR applications is handled through Vuex, Vue's official state management system. Similar to Redux, the Vuex store is populated on the server, and its state is serialized in the HTML. Upon loading on the client, the store is hydrated with the initial state, maintaining consistency between server and client.

Example Implementation with Nuxt.js and Vuex:

```
// javascript store/index.js
export const state = () => ({
  message: ''
});

export const mutations = {
  setMessage(state, message) {
    state.message = message;
  }
};

export const actions = {
  async nuxtServerInit({ commit }) {
    // Simula una chiamata API per ottenere dati dal server
    const message = await fetchMessageFromAPI();
    commit('setMessage', message);
  }
};
```

In this case, the **nuxtServerInit** action is automatically executed by Nuxt.js during server-side rendering. This action allows the store to be initialized with the necessary data before the page is sent to the client.

On the client, Nuxt.js automatically handles the hydration of the Vuex store using the initial state provided by the server, ensuring a seamless transition.

Advantages of State Management in SSR Applications:

- **State Synchronization:** Keeping the state synchronized between server and client avoids issues such as data discrepancies and improves application consistency.
- **Performance Migliorate:** Pre-populating the state on the server can reduce the number of API requests made by the client, enhancing load times and application responsiveness.
- **Optimized SEO:** With data already available during server-side rendering, search engines can correctly index the application's dynamic content.

Challenges and Considerations:

It is important to ensure that the serialized state is secure and does not contain sensitive information, as it will be visible in the HTML page source sent to the client. Additionally, in SSR applications, managing user sessions and authentication on the server may be necessary, adding complexity to state management. Furthermore, maintaining the state on the server for each user can have implications for server resources and application scalability.

In conclusion, state management in SSR applications requires a careful approach to ensure state consistency between server and client. Using tools like Redux and Vuex facilitates this process, providing mechanisms to initialize and transfer state efficiently. Properly addressing these challenges allows the full benefits of SSR to be leveraged, improving both performance and user experience.

1.3 Evolution and Trends in the Field

Evolution of Rendering Technologies^{[2][9][17]}

The field of web rendering has undergone significant evolution over the past few decades. From simple static sites with manually generated HTML, the industry moved to dynamic sites with SSR and, subsequently, to SPAs with CSR, followed by a shift with the advent of frameworks like Next.js.

The evolution of browsers has had a significant impact on rendering techniques. The introduction of advanced support for JavaScript, AJAX, and, more recently, WebAssembly has enabled browsers to perform complex operations directly on the client, driving the adoption of CSR. Modern browsers, such as Chrome, Firefox, and Edge, are capable of handling the computational load required for CSR, making it possible to create rich and interactive web applications.

Current Trends and the Future of Web Rendering^{[6][24]:}

Static Site Generation (SSG) and Incremental Static Regeneration (ISR):

SSG is a technique that combines the advantages of static and dynamic page generation. During build time, all pages are pre-rendered into static HTML, which can be served quickly to clients. This approach is particularly useful for sites with content that changes infrequently, such as blogs or technical documentation.

Example of SSG Implementation with Next.js:

```
//javascript
import React from 'react'

function BlogPost({ post }) {
  return (
    <div>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </div>
  )
}

export async function getStaticPaths() {
  const res = await fetch('https://api.example.com/posts')
  const posts = await res.json()

  const paths = posts.map((post) => ({
    params: { id: post.id.toString() },
  }))

  return { paths, fallback: false }
}

export async function getStaticProps({ params }) {
  const res = await fetch(`https://api.example.com/posts/${params.id}`)
  const post = await res.json()

  return { props: { post } }
}

export default BlogPost
```

In this example, **getStaticPaths** generates all possible blog pages during build time, while **getStaticProps** fetches the necessary data for each page, making the site extremely fast to load.

Evolution of Hybrid Rendering:

Hybrid rendering is a combination of CSR and SSR, where parts of the application are pre-rendered on the server, while others are rendered on the client. Frameworks like Next.js and Nuxt.js natively support this approach, offering flexibility to optimize SEO and performance.

React Server Components and Edge Rendering^{[9][11][18][19]}:

React Server Components is a new technology that allows part of the rendering to be performed on the server and part on the client, reducing the computational load on the client and improving performance. Similarly, Edge Rendering, which leverages CDNs to perform rendering as close as possible to the end user, is gaining popularity as a way to reduce latency and enhance user experience.

Future Challenges and Considerations

Sustainability and Rendering^{[9][11]}:

With the increasing complexity of web applications, resource consumption is also rising. Sustainability is becoming an important aspect of web application design, considering the environmental impact of server infrastructure and content delivery networks. Minimizing rendering load and optimizing energy consumption are key challenges for the future.

Development of Web Standards:

Standardization bodies, such as W3C and WHATWG, continue to develop new standards to improve the interoperability and efficiency of web technologies. For example, Web Components and Shadow DOM offer new opportunities to create modular and reusable user interfaces, reducing the need for direct DOM manipulation and improving performance.

2. Design and Development of Test Web Applications

2.1 Requirements Analysis

Website Presentation

To effectively compare the use of **Client-Side Rendering (CSR)** and **Server-Side Rendering (SSR)** technologies, it was decided to develop a website as a test platform. This website will be created in two functionally identical versions: one using **React.js** for client-side rendering and the other using **Next.js** for server-side rendering. Both versions will be hosted on **Vercel**, ensuring a consistent environment for comparison.

To maximize the project's utility, it was chosen to develop the website `lucajop.it`, belonging to my father, **Luca Jop**, an architect who needs an online presence to showcase his work and provide contact information. In 2020, I had already built a version of the site using **WordPress** and various plugins, but this solution lacked the desired flexibility, and frequent plugin updates made the site difficult to use.

Motivation for the Technologies Used:

The choice of `React.js` and `Next.js` as the primary frameworks for developing the `lucajop.it` website was not arbitrary but based on precise considerations related to the project's needs. `React.js` was selected for its **declarative** and **component-based** approach, which enables the creation of modular and easily maintainable user interfaces. This is particularly advantageous for a project that anticipates the site's evolution over time, as each component can be updated or replaced without significant impacts on the entire system.

On the other hand, the choice of `Next.js` was driven by the need to improve loading performance and search engine optimization (**SEO**). The Server-Side Rendering (SSR) offered by `Next.js` allows pages to be pre-rendered on the server before being sent to the client. This reduces the time required for initial page loading and facilitates better indexing by search engines, a critical aspect for an architect's website aiming to be visible online to attract new clients. Additionally, `Next.js` supports **Static Site Generation (SSG)** and other optimization techniques that could be leveraged in later development phases for more static content, such as the presentation of completed projects.

Purpose, Objectives, Features, and Key Pages:

The primary goal of the website is to provide an online platform that effectively and attractively presents Luca Jop's projects, facilitating contact with potential clients. The design must be minimalist and elegant, emphasizing ease of use and accessibility of information.

Key pages of the website:

- **Home Page:** A minimalist landing page with visually engaging elements to capture attention.
- **Works:** A section dedicated to completed projects, presented through an interactive gallery.
- **About:** A page detailing the architect's professional background, design philosophy, and skills.
- **Contacts:** Contact information for sending inquiries or scheduling appointments.
- **404 Error Page:** A customized page displayed when a user attempts to access a non-existent page or enters an incorrect URL. This page must be consistent with the site's design and guide the user back to the main sections.
-

From the **Works** section, users can access **detailed project** pages, which include:

- **Image Carousel:** A visual presentation of the project through photographs and renders.
- **In-Depth Description:** Detailed information about the project, such as objectives, solutions adopted, and results achieved.

The intent is to provide visitors with a comprehensive overview of the architect's professional capabilities, helping them evaluate the possibility of collaboration.

Additionally, including a **404 Error Page** is important to keep users engaged even in case of navigation errors, avoiding frustration and potential site abandonment.

Analysis of Constraints, Limitations, and Requirements

Technological Constraints:

- **Use of Free and Open-Source Technologies:** To contain costs and ensure maximum flexibility, it was decided to adopt free tools and platforms.
- **Development Frameworks:** Using **React.js** and **Next.js** enables leveraging modern JavaScript capabilities to create dynamic and performant user interfaces.

- **Programming Language:** All development will be done in **JavaScript**, for both front-end and back-end.

Hosting and Distribution Constraints:

- **Vercel:** Choosing Vercel as the hosting platform ensures efficient and automated distribution of both site versions. Vercel natively supports both React.js and Next.js, simplifying the deployment process.
- **Workflow Automation:** Thanks to Vercel, every change to the source code can be automatically integrated and deployed, improving the efficiency of the development cycle.

Security and Performance Constraints:

- **Integrated Security:** Vercel automatically provides SSL/TLS certificates, ensuring encrypted communications between the site and users.
- **Optimized Performance:** The goal is to ensure fast loading times, delivering a smooth user experience on both desktop and mobile devices.

Database Constraints:

- **MongoDB: MongoDB Atlas** will be used, offering a free plan suitable for the project's needs. MongoDB was chosen for its flexibility and scalability in managing non-relational data.

Compatibility Constraints:

- **Support for Modern Browsers:** The site will be optimized to work correctly on the latest versions of major browsers (Chrome, Firefox, Safari, Edge). Using modern technologies may limit support for outdated browsers.

Scalability and Future:

The choice of technologies like **Next.js** and **React.js** also offers a clear advantage in terms of project scalability. The architecture is designed to adapt to increasing workloads without compromising system performance. Thanks to React's modularity and Next.js's optimized resource management, it will be possible to add new features or pages in the future without redesigning the entire site. Additionally, using **MongoDB Atlas** as a **NoSQL** database provides flexibility and horizontal scalability that perfectly suits an evolving project.

Definition of Use Cases and Involved Actors

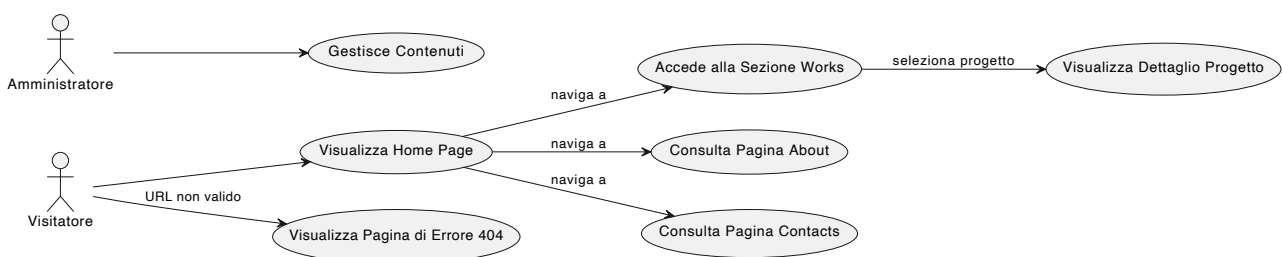
Main Actors:

- **Visitors:** Users accessing the site to learn about the architect's work and evaluate potential collaboration.
- **Administrator:** The architect or a delegated person responsible for updating the site's content.

Main Use Cases:

1. **Viewing the Home Page:** The visitor accesses the site and gets a first impression of the professional activity.
2. **Navigating the Works Section:** The visitor explores various completed projects, viewing previews and brief descriptions.
3. **Project Details:** In-depth exploration of a single project through images and detailed descriptions.
4. **Consulting the About Page:** The visitor reads information about the architect's professional background.
5. **Accessing Contact Information on the Contacts Page:** The visitor finds the details needed to request information or schedule a meeting.
6. **Viewing the 404 Error Page:** If the visitor enters an invalid URL or attempts to access a non-existent page, the system displays a customized 404 error page, offering options to return to the homepage or navigate to other sections.
7. **Content Management:** The administrator accesses a control panel (even a simple one or based on external tools) to add or edit projects and update information.

UML Diagram:



Specification of Functional and Non-Functional Requirements

Functional Requirements (FR):

- **FR1:** The system must allow visitors to view a list of available projects.
- **FR2:** The system must enable viewing detailed information for each project, including high-resolution images and extended descriptions.
- **FR3:** The system must provide contact information to allow visitors to reach the architect.
- **FR4:** The administrator must have the ability to add, edit, and remove projects from the site.
- **FR5:** The site must automatically adapt to different screen sizes (responsive design), ensuring good usability on mobile devices and tablets.
- **FR6:** The system must properly handle requests for non-existent pages, displaying a customized 404 error page and providing navigation options.

Non-Functional Requirements (NFR):

- **NFR1 - Performance:** The site's pages must load in less than 2 seconds on a standard broadband connection.
- **NFR2 - Scalability:** The site's structure must allow the addition of new features or pages without requiring a complete code overhaul.
- **NFR3 - Usability:** The interface must be intuitive, with clear and consistent navigation across different sections.
- **NFR4 - Maintainability:** The code must be written following development best practices, with a clear and documented structure to facilitate future changes or interventions by other developers.
- **NFR5 - Compatibility:** The site must be compatible with the latest versions of major browsers (Chrome, Firefox, Safari, Edge).
- **NFR6 - User Experience in Case of Error:** The 404 error page must be consistent with the site's design and provide clear navigation options to help users find desired information and reduce abandonment rates.

Conclusion of the Requirements Analysis:

The requirements analysis has clearly outlined the project's objectives, defining essential functionalities and identifying technical and operational constraints. Developing the site in two versions, one with React.js and one with Next.js, will enable a direct comparison between CSR and SSR technologies, evaluating their strengths and weaknesses in a real-world context. The next phase of the project involves designing the system architecture, considering the identified requirements, and planning development to ensure compliance with constraints and achievement of the set objectives.

2.2 System Design

Proposed System Architecture

The architectural design of the **lucajop.it** website was conceived to ensure modularity, scalability, and maintainability, leveraging the capabilities of **Client-Side Rendering (CSR)** and **Server-Side Rendering (SSR)** through **React.js** and **Next.js**. The system architecture is divided into several key components, each responsible for specific functionalities, interacting synergistically to deliver an optimal user experience.

General System Structure and Component Interactions:

The system architecture is based on a three-tier model, differentiated for the two versions of the site:

1. Front-End

- **React.js (CSR)**: Responsible for presenting the user interface and managing user interactions. Uses React.js for client-side rendering.
- **Next.js (SSR)**: Provides server-side rendering, improving initial performance and SEO optimization. Also manages API Routes for interaction with the back-end.

2. Back-End

- **Serverless Functions on Vercel (React.js CSR)**: Implementation of back-end functionalities through serverless functions provided by Vercel, ensuring a scalable and easily maintainable architecture.
- **API Routes (Next.js SSR)**: API endpoints developed with Next.js, responsible for handling client requests and interacting with the database.

3. Database

- **MongoDB Atlas**: Used for managing non-relational data, ensuring flexibility and scalability in storing information.

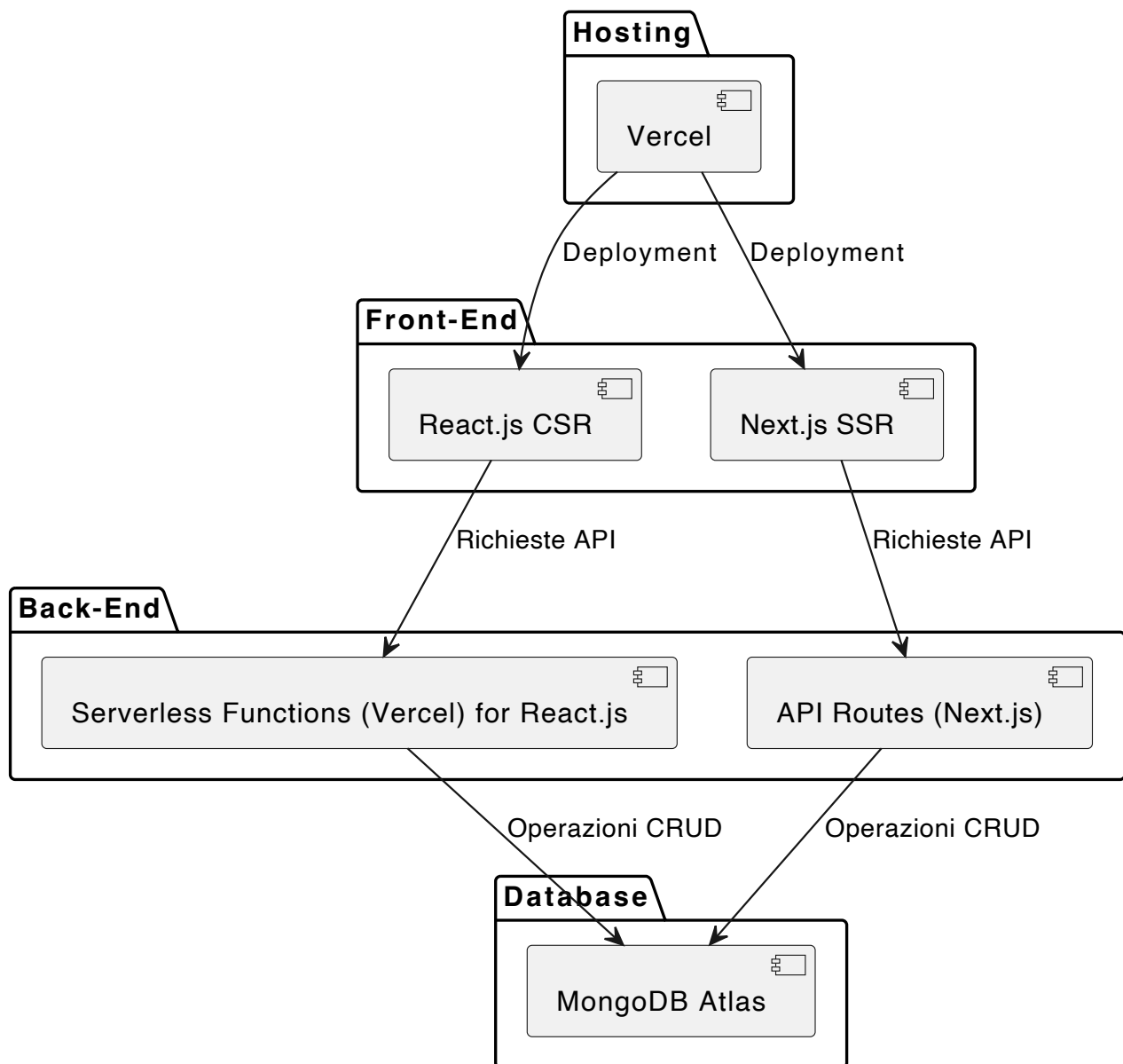
4. Hosting e Deployment

- **Vercel**: Hosting platform used to deploy both versions of the site, automatically managing deployment, scalability, and security.

Interactions between these components are orchestrated through HTTP requests and RESTful APIs, with Vercel serving as the hosting platform, simplifying deployment and management of the site versions.

High-Level Diagram:

Below is a high-level UML diagram illustrating the main system components and their interactions for both versions of the site:



System Breakdown into Modules and Layers:

The system architecture is organized into distinct modules, each with specific responsibilities, further divided into layers to facilitate management and scalability.

1. Presentation Layer (Front-End)

- **UI Components:** User interface elements built with **React.js** and managed through **Next.js** for server-side rendering.

2. Business Logic Layer (Back-End):

- **Serverless Functions (Vercel):** Serverless functions for the React.js CSR version, handling operations such as sending emails or accessing data.
- **API Routes (Next.js):** API endpoints developed with **Next.js**, responsible for handling client requests and interacting with the database.
- **Services:** Modules encapsulating business logic, such as project management and authentication (if needed).

3. Persistence Layer (Database):

- **Database NoSQL: MongoDB Atlas** is used to store data related to projects, contact messages, and other information necessary for the site's operation.

4. Infrastructure Layer (Hosting and Deployment)

- **Vercel:** Hosting platform used to deploy the **React.js CSR** and **Next.js SSR** applications, automatically managing deployment, scalability, and security.

Detailed Description of Modules, Architectural Layers, and Technology Integration in the Overall Architecture:

The system architecture for **lucajop.it** is organized into various modules and layers, each with specific responsibilities, collaborating to ensure optimal performance, scalability, and maintainability. Below is an integrated description of the main modules and architectural layers, along with considerations on technology integration and performance.

1. Front-End

- **React.js CSR:** Manages client-side rendering, offering an interactive and responsive user interface. Uses modular components to facilitate code reusability and maintenance.
- **Next.js SSR:** Provides server-side rendering, improving initial performance and SEO optimization. Also manages API Routes for back-end interaction.
- **Performance Optimization:**
 - **Code Splitting and Lazy Loading:** Implemented through Next.js, allowing only necessary resources to be loaded when needed, improving page load times.
 - **Image Optimization:** Use of Next.js Image Optimization to resize, lazy-load, and compress images automatically, further enhancing performance and user experience.

2. Back-End

- **Serverless Functions (Vercel) for React.js CSR:** Serverless functions handling operations such as sending emails, accessing data, and other back-end logic specific to the React.js CSR version.

- **API Routes (Next.js SSR):** API endpoints managing requests from the front-end, performing read and write operations on the database, and applying necessary business logic.
- **Business Logic Services:** Modules implementing specific functionalities, such as processing project data.
- **Performance Optimization:**
 - **Caching:** Implementation of caching strategies at the Vercel and browser levels to reduce server requests, improving overall performance and reducing back-end load.

3. Database

- **MongoDB Atlas:** Managed NoSQL database offering high availability, scalability, and security for storing application data. Facilitates efficient and flexible CRUD operations.
- **Scalability:** The choice of MongoDB Atlas ensures the database can scale horizontally to handle increased data volumes and requests without compromising performance.

4. Hosting

- **Vercel:** Provides an optimized environment for deploying web applications, natively supporting React.js and Next.js. Automatically manages resource provisioning, load balancing, and application security.
- **Scalability:** Vercel automatically handles infrastructure scalability, ensuring the site can manage traffic spikes without performance issues.
- **Security:**
 - **Data Protection:** Implementation of SSL/TLS certificates through Vercel to encrypt communications between the site and users.
 - **API Management:** Use of secure endpoints to prevent attacks such as Cross-Site Scripting (XSS) and code injection.

5. Manutenibilità

- The modularity offered by React.js and Next.js facilitates code maintenance, allowing individual components to be updated or replaced without affecting the entire application.
- The use of tools like Babel ensures an efficient and easily configurable build process, simplifying dependency and resource management.

Development of Main Components

The development phase of the system's main components required careful planning and implementation to ensure each component fulfills its role within the overall architecture. Below is a detailed description of the key components developed and their relation to the initial design.

Detailed Description of Main Components and Their Functionalities:

1. Header

- **Functionality:** Provides the site's main navigation, including the logo and links to different sections.
- **Implementation:**
 - **React.js:** Creation of a reusable component managing the navigation menu and adapting to different screen sizes.
 - **Next.js:** Use of internal links for optimized routing and support for SSR to improve performance.

3. Works

- **Functionality:** Displays an interactive gallery of the architect's completed projects.
- **Implementation:**
 - **React.js:** Dynamic components managing image display and user interactions.
 - **Lazy Loading:** Loading images only when visible to the user, improving performance.

4. Project Detail

- **Functionality:** Provides in-depth information about a single project, including images and detailed descriptions.
- **Implementation:**
 - **Image Carousel:** Use of libraries like Swiper to create responsive and customizable carousels.
 - **Next.js:** Optimization of image rendering to improve load times.

5. About

- **Functionality:** Presents the architect's professional profile, including education, experience, and design philosophy.
- **Implementation:**
 - **React.js:** Modular components for biography, timeline, and skills.
 - **CSS Modules:** Consistent styles to maintain a professional appearance.

6. Contacts

- **Functionality:** Displays the architect's contact information without including a contact form.
- **Implementation:**
 - **React.js:** Component displaying details such as email, phone number, and address.
 - **Consistent Design:** Maintains the same visual style as the rest of the site for a seamless transition.
 - **Integration with Serverless Functions (React.js CSR):** Even without a form, additional functionalities like automatic email generation or messaging service integrations can be implemented.

7. 404 Error Page

- **Functionality:** Handles navigation errors, informing the user that the requested page does not exist and offering alternative navigation options.
- **Implementation:**
 - **React.js:** Customized component maintaining the site's style and providing useful navigation links.
 - **Next.js:** Configuration of a customized 404 error page seamlessly integrated with the application's routing.
 - **Consistent Design:** Use of the same style and layout components as other pages, ensuring the 404 page integrates perfectly with the site's overall aesthetic.

Relationship Between Design and Component Implementation:

The architectural design defined a clear division of modules and layers, facilitating the development of individual components. For example:

- **Modularity:** Each component was developed independently, following the single responsibility principle. This simplified maintenance and extension of the site's functionalities.
- **Reusability:** Components like the Header were designed to be reused across all pages, ensuring consistency and reducing code duplication.
- **Error Handling:** The design of a 404 Error Page consistent with the rest of the site ensures effective error management, improving the user experience.

2.3 Conclusion of System Design

The system design for lucajop.it followed a modular and scalable approach, leveraging the capabilities of React.js and Next.js to create a performant, SEO-optimized, and easily maintainable website. The design choices ensured seamless integration between front-end and back-end, while the selection of technologies guaranteed that the system could grow and adapt to future needs without compromising performance.

The development phase translated the design into functional and interactive components, addressing and resolving technical challenges that arose during the process. The inclusion of elements such as the 404 Error Page further enhanced the user experience, demonstrating particular attention to the site's quality and robustness.

The next phase of the project will focus on the detailed implementation of components, followed by testing and production deployment, ensuring that all functionalities meet the defined requirements and that the site effectively responds to user expectations.

2.4 Implementation and Development

This chapter examines the details of the implementation of the **lucajop.it** website using React.js and Next.js. It analyzes the key steps in front-end development, page and routing implementation, API management, the adopted development environment, and the challenges faced during the process. It is worth noting that Redux and Webpack were not used in the project, and to facilitate future comparison tests, neither lazy loading nor caching was implemented. The administrative part for creating, editing, and deleting projects was also not implemented for the same reason.

Tools Used for Development:

1. Visual Studio Code (VS Code)

- Motivation: VS Code was chosen as the primary development environment for its integration capabilities with modern development tools. VS Code supports a wide ecosystem of extensions that streamline web development. Extensions like **Prettier** automate code formatting, ensuring a consistent and readable style, while **MongoDB for VS Code** allows direct interaction with the database during development, making it easier to test queries and verify data.
- Extensions Used:
 - Prettier: For automatic code formatting.
 - MongoDB for VS Code: For direct interaction with the MongoDB database.

2. GitHub

- Motivation: Choosing GitHub as the code management platform facilitated versioning and tracking of changes, ensuring that every modification was documented and easily reversible in case of errors, thus guaranteeing a smooth and collaborative development process.
- Versioning and Commit Management: A specification for adding human- and machine-readable meaning to commit messages following the structure `<type>(<optional scope>) : <subject>`
(e.g. `"feat(shopping cart): add the amazing button"`)
- Frequent Commits: Performing frequent and descriptive commits to track changes and facilitate rollback in case of issues.

Details of React.js Implementation

The front-end implementation using React.js followed a modular approach, focusing on creating reusable components and effectively managing local state. Below are the key steps taken during development.

```
//javascript
import { useEffect, useState } from 'react';
import { useNavigate, useParams } from 'react-router-dom';
import WorkDetail from '../components/Works/WorkDetail.js';
import { Helmet } from 'react-helmet';

function WorkDetails() {
  const { workId } = useParams();
  const [workData, setWorkData] = useState(null);
  const navigate = useNavigate();

  useEffect(() => {
    async function fetchWork() {
      try {
        const response = await fetch(`/api/works/${workId}`);
        if (!response.ok) {
          navigate("/404");
          throw new Error('Work not found or server error');
        }
        const data = await response.json();
        setWorkData(data);
      } catch (error) {
        console.error('Error fetching work:', error);
      }
    }
    fetchWork();
  }, [workId, navigate]);


  if (!workData) return <p>Loading...</p>;

  return (
    <>
      <Helmet>
        <title>{workData.title}</title>
        <meta name="description" content={workData.description} />
      </Helmet>
      <WorkDetail
        id={workData.id}
        images={workData.images}
        title={workData.title}
        shortDescription={workData.shortDescription}
        description={workData.description}
        role={workData.role}
      />
    </>
  );
}

export default WorkDetails;
```

Project Structure:

Logical organization of folders, dividing components into specific directories (components/Works/WorkDetail.js, etc.) to improve code maintainability and scalability. In the React.js version, pages are directly contained in src/pages.



```
/api
  /works
    - [workId].js
    - works.js
/public
/src
  /components
    /About
      - About.js
      - Timeline.js
    /Contacts
      - Contacts.js
    /Layout
      - Layout.js
      - MainNavigation.js
    /UI
      - Card.js
      - Carousel.js
      - Image.js
      - ScrollToTop.js
    /Works
      - WorkDetail.js
      - WorkItem.js
      - WorkList.js
  /pages
    - About.js
    - Contacts.js
    - Home.js
    - NotFound.js
    - WorkId.js
    - Works.js
  /styles
    - App.css
    - index.css
- App.js
- index.js
```

Creation of Main Components (Example: WorkDetail Component):

- WorkDetail: A component dedicated to displaying project details, managing the presentation of information such as title, description, images, and role.
- Helmet: Use of the React Helmet library to dynamically manage page metadata, improving SEO optimization.
- Route Management: Implementation of routes using React Router, enabling navigation between different pages, such as the project detail page.

Example of Route Implementation:

```
//javascript
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import WorkDetails from '../components/Works/WorkDetails';
import NotFound from '../components/NotFound';

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/works/:workId" element={<WorkDetails />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}
```

- API Interaction: Creation of a function to fetch project data via API calls, handling errors and loading states to ensure a smooth user experience.
- Styling: Use of CSS Modules to apply modular styles and prevent class conflicts, keeping CSS code organized and easily maintainable.

Details of Next.js Implementation

Next.js offers a range of built-in features that simplify the development of performant and SEO-optimized web applications. Below are the main aspects of the implementation:

Project Structure:

Logical organization of folders, dividing components into specific directories (components/Works/WorkDetail.js, etc.) to improve code maintainability and scalability. In the Next.js version, the absence of APIs is notable, and pages in the /pages folder are encapsulated according to Next.js routing specifications.

```
/components
  /About
    - About.js
    - Timeline.js
  /Contacts
    - Contacts.js
  /Layout
    - Layout.js
    - MainNavigation.js
  /UI
    - Card.js
    - Carousel.js
    - Image.js
    - ScrollToTop.js
  /Works
    - WorkDetail.js
    - WorkItem.js
    - WorkList.js
/pages
  /[workId]
    - index.js
  /about
    - index.js
  /contacts
  /new-work
    - index.js
  /works
    - index.js
  - _app.js
  - 404.js
  - index.js
/public
/styles
```


Page Creation:

- SEO Optimization with Head Component: Use of Next.js's `<Head>` component to dynamically insert titles and metadata into pages, improving search engine optimization.
- Local State Management: Use of React's local state with `useState` and `useEffect` to manage data and user interactions.
- Styling with CSS Modules: Application of modular styles to keep CSS code organized and prevent class conflicts.

The use of `getStaticProps` and `getStaticPaths` to generate static pages during the build phase improves performance and SEO optimization.

Here, the insertion of the `WorkDetail` component into the page is shown.

```
//javascript
import { MongoClient, ObjectId } from "mongodb";
import Head from "next/head";
import { Fragment } from "react";
import WorkDetail from "../../components/Works/WorkDetail";

function WorkDetails(props) {
  return (
    <Fragment>
      <Head>
        <title>{props.workData.title}</title>
        <meta
          name="description"
          content={props.workData.description}
        />
      </Head>
      <WorkDetail
        id={props.workData.id}
        images={props.workData.images}
        title={props.workData.title}
        shortDescription={props.workData.shortDescription}
        description={props.workData.description}
        role={props.workData.role}
      />
    </Fragment>
  );
}

...
```

```

...

export async function getStaticProps(context) {
  const workId = context.params.workId;

  // Verifica se l'ID è valido
  if (!isValidObjectId(workId)) {
    return {
      notFound: true, // Restituisce una pagina 404 se l'ID non è valido
    };
  }

  const client = await MongoClient.connect(
    "mongodb+srv://*****:*****@cluster0.kajhjck.mongodb.net/works?
retryWrites=true&w=majority"
  );
  const db = client.db();

  const worksCollection = db.collection("works");

  const selectedWork = await worksCollection.findOne({
    _id: ObjectId(workId),
  });

  client.close();

  if (!selectedWork) {
    return {
      notFound: true, // Questo renderà la pagina 404
    };
  }

  return {
    props: {
      workData: {
        id: selectedWork._id.toString(),
        title: selectedWork.title,
        images: JSON.parse(JSON.stringify(selectedWork.images)),
        shortDescription: selectedWork.shortDescription,
        description: selectedWork.description,
        role: selectedWork.role
      },
    },
    revalidate: 1, // Incremental Static Regeneration ogni 1 secondo
  };
}

export default WorkDetails;

```

Here, the implementation of `getStaticPaths` is observed, where projects returned from the database are mapped.

In the following code, the implementation of `getStaticProps` is observed, where the project requested by the user is identified based on the ID and transformed into an object that populates the `WorkDetail` fields.

```
...

export async function getStaticPaths() {
  const client = await MongoClient.connect(
    "mongodb+srv://*****:*****@cluster0.kajhjck.mongodb.net/works?
retryWrites=true&w=majority"
  );
  const db = client.db();

  const worksCollection = db.collection("works");

  const works = await worksCollection.find({}, { _id: 1 }).toArray();

  client.close();
  return {
    fallback: 'blocking',
    paths: works.map((work) => ({
      params: { workId: work._id.toString() },
    })),
  };
}

function isValidObjectId(id) {
  return /^[a-fA-F0-9]{24}$/.test(id);
}

...
```

Use of Vercel for Deployment

Vercel was chosen as the hosting platform for deploying the lucajop.it project due to its powerful features integrated with Next.js and React.js, offering a simple and fast deployment process, along with advanced features such as load balancing, automatic scalability, and security management.

Deployment Process on Vercel:

1. Linking the GitHub Repository:

- Access [Vercel](https://vercel.com/) and create a new project.
- Connect to a GitHub account and select the lucajop.it project repository.
- Configure build settings, specifying the framework used (Next.js or React.js).

2. Deployment Automation:

- Every push to the main branch (`main`) automatically triggers a new deployment.
- Management of previews for pull requests, facilitating testing and review before merging.

3. Monitoring and Management:

- Vercel Dashboard: Allows monitoring of deployment status, viewing logs, and managing project settings.
- Rollback: Ability to roll back to previous versions of the site in case of issues.

2.5 Final Considerations

The implementation of the lucajop.it website benefited from the capabilities offered by React.js and Next.js, leveraging a modular and maintainability-focused approach. Despite the absence of Redux and Webpack, it was possible to effectively manage local state and resource bundling thanks to Next.js's built-in features.

The development environment configured with VS Code and GitHub facilitated source code management and collaboration, while challenges encountered during development were addressed through best practices and the adoption of appropriate tools.

The decision not to implement caching and lazy loading allowed future performance tests to remain clear and focused on the intrinsic capabilities of React.js and Next.js, facilitating subsequent comparisons and optimizations. Additionally, the Next.js-based infrastructure enables easy evolution of the site by adding new pages or functionalities without compromising performance, while the use of a scalable database like MongoDB Atlas ensures the system can handle increased data volumes without difficulty.

The next phase of the project will focus on performance analysis and continuous system monitoring to ensure that the lucajop.it website effectively meets user needs.

3. Evaluation and Analysis of Results

This chapter explores the methodologies and tools necessary to perform performance tests on the websites developed with **React.js** and **Next.js** for **lucajop.it**. Performance tests are essential to ensure that the application delivers a smooth user experience, fast loading times, and efficient resource management. Given that **caching** and **lazy loading** were not implemented in the project, the tests focus on the intrinsic capabilities of **React.js** and **Next.js** in terms of rendering, interactivity, and API management.

3.1 Introduction to Performance Testing

To evaluate the efficiency and performance of the website developed in **React.js** and **Next.js**, we used **Google Lighthouse** as the primary tool. Google Lighthouse is an open-source tool integrated into Chrome's Developer Tools that leverages the PageSpeed API, extensively discussed during the internship. It provides automated audits to improve the quality of web pages, covering categories such as Performance, Accessibility, Best Practices, SEO, and Progressive Web App.

Lighthouse was run on **Google Chrome** from a **laptop** connected to the internet via **WiFi** with a connection speed of **25 Mbps**. All tests were performed **without cache**, ensuring that each page load reflected real performance without the aid of previously stored resources. This approach ensures that the results represent the site's actual performance under standard conditions.

The key metrics analyzed for comparison are:

- **First Contentful Paint (FCP)**: Time required to display the first content.
- **Largest Contentful Paint (LCP)**: Time taken to load the largest visible element on the page.
- **Speed Index**: Measures the speed at which visible content is populated.
- **Total Blocking Time (TBT)**: Quantifies the time during which the page is "blocked" and unresponsive to user input.
- **Cumulative Layout Shift (CLS)**: Evaluates how much page elements shift during loading.
- **Time to Interactive (TTI)**: Measures how long it takes for the page to become fully interactive.

3.2 Performance Comparisons Between React.js and Next.js

Below are the detailed comparisons for each of the site's main pages, for both the **React.js** and **Next.js** versions.

All tests were performed 10 times, with the reported values being the averages and the variance as a percentage relative to the mean.

1. Home Page

Metrics	React.js	Variance (%)	Next.js	Variance (%)
First Contentful Paint (FCP)	1.8 s	0.01	1.1 s	0.04
Largest Contentful Paint (LCP)	3.6 s	0.01	3.3s	1.09
Speed Index	6.0 s	0.02	4.2 s	0.83
Total Blocking Time (TBT)	0.4s	0.03	0.0 s	0.01
Cumulative Layout Shift (CLS)	4	0.00	0	0.00
Time to Interactive (TTI)	6.2	0.02	0.0 s	0.01

Comparative Analysis:

- Next.js stands out for a better **First Contentful Paint** and a lower **Speed Index**, ensuring a faster and smoother user experience compared to React.js. The **Total Blocking Time** and **Time to Interactive** are significantly better in Next.js, suggesting that interactivity is nearly immediate thanks to **Server-Side Rendering (SSR)**.

2. Works Page

Metrics	React.js	Variance (%)	Next.js	Variance (%)
First Contentful Paint (FCP)	1.9 s	0.49	0.5 s	0.03
Largest Contentful Paint (LCP)	2.6 s	0.96	0.8 s	1.50
Speed Index	6.9 s	0.30	5.5 s	0.60
Total Blocking Time (TBT)	0.4 s	0.11	0.0 s	0.01
Cumulative Layout Shift (CLS)	6	14.44	0	0.02
Time to Interactive (TTI)	5.4 s	0.02	0.9 s	0.01

Comparative Analysis:

- Next.js loads visible content much faster than React.js. The **Total Blocking Time** of zero on Next.js demonstrates that there were no significant slowdowns during JavaScript execution, resulting in a smoother user experience. The **Time to Interactive** is also much faster in Next.js, thanks to SSR optimization.

3. About Page

Metrics	React.js	Variance (%)	Next.js	Variance (%)
First Contentful Paint (FCP)	1.6 s	0.02	0.7 s	0.01
Largest Contentful Paint (LCP)	2.5 s	0.06	1.5 s	0.01
Speed Index	5.5 s	0.07	4.3 s	0.02
Total Blocking Time (TBT)	0.3 s	0.03	0.0 s	0.04
Cumulative Layout Shift (CLS)	4	0.10	0	0.00
Time to Interactive (TTI)	5.2 s	0.02	0.4 s	0.01

Comparative Analysis:

- Next.js offers a much faster **First Contentful Paint**, reducing user wait times. The **Largest Contentful Paint** is better in Next.js, with the main element loading in shorter times.

4. Contacts Page

Metrics	React.js	Variance (%)	Next.js	Variance (%)
First Contentful Paint (FCP)	1.7 s	6.38	0.6 s	0.01
Largest Contentful Paint (LCP)	2.8 s	0.78	1.2 s	0.06
Speed Index	5.7 s	0.61	3.9 s	0.03
Total Blocking Time (TBT)	0.3 s	0.28	0.0 s	0.03
Cumulative Layout Shift (CLS)	5	0.24	0	0.03
Time to Interactive (TTI)	5.3 s	9.05	0.4 s	0.01

Comparative Analysis:

- On this page as well, Next.js outperforms React.js in all key metrics, offering faster loading and immediate interactivity times. The **Total Blocking Time** of zero in Next.js suggests much more efficient client-side resource management.

5. WorkDetail Page

Metric	React.js	Variance (%)	Next.js	Variance (%)
First Contentful Paint (FCP)	2.2 s	0.07	1.2 s	0.27
Largest Contentful Paint (LCP)	3.8 s	0.04	3.8 s	3.89
Speed Index	6.1 s	0.04	4.6 s	0.13
Total Blocking Time (TBT)	0.6 s	0.27	0.1 s	2.53
Cumulative Layout Shift (CLS)	6	0.29	0	0.01
Time to Interactive (TTI)	6.7 s	0.01	0.4 s	0.02

Comparative Analysis:

- Next.js loads visible content more quickly, with better **Speed Index** and **FCP** compared to React.js. The **Total Blocking Time** and **Time to Interactive** are also significantly better in Next.js, suggesting that users can interact with the page almost immediately.

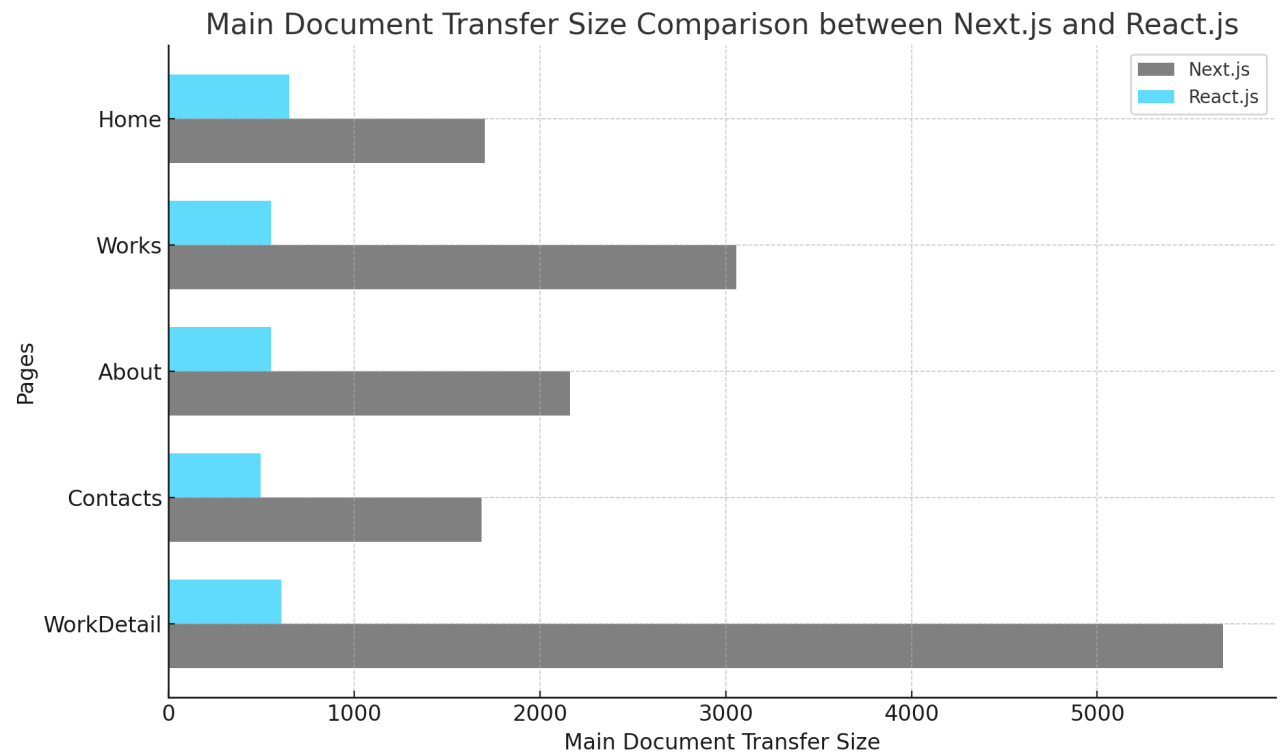
Lighthouse reports 0 seconds for **Total Blocking Time (TBT)**, **Cumulative Layout Shift (CLS)**, and **Time to Interactive (TTI)** on Next.js for several reasons related to rendering optimization. With **Server-Side Rendering (SSR)**, the HTML is pre-generated by the server, reducing TBT and TTI since the page is already interactive when loaded. The **CLS** is 0.0s due to layout optimization, managed by tools like the `<Image>` component. The TTI can be 0.0s thanks to techniques like **code splitting**, which reduce the JavaScript load executed on the client. In summary, the 0.0s on these metrics indicates excellent server-side optimization, providing a smoother user experience without blocks or layout shifts.

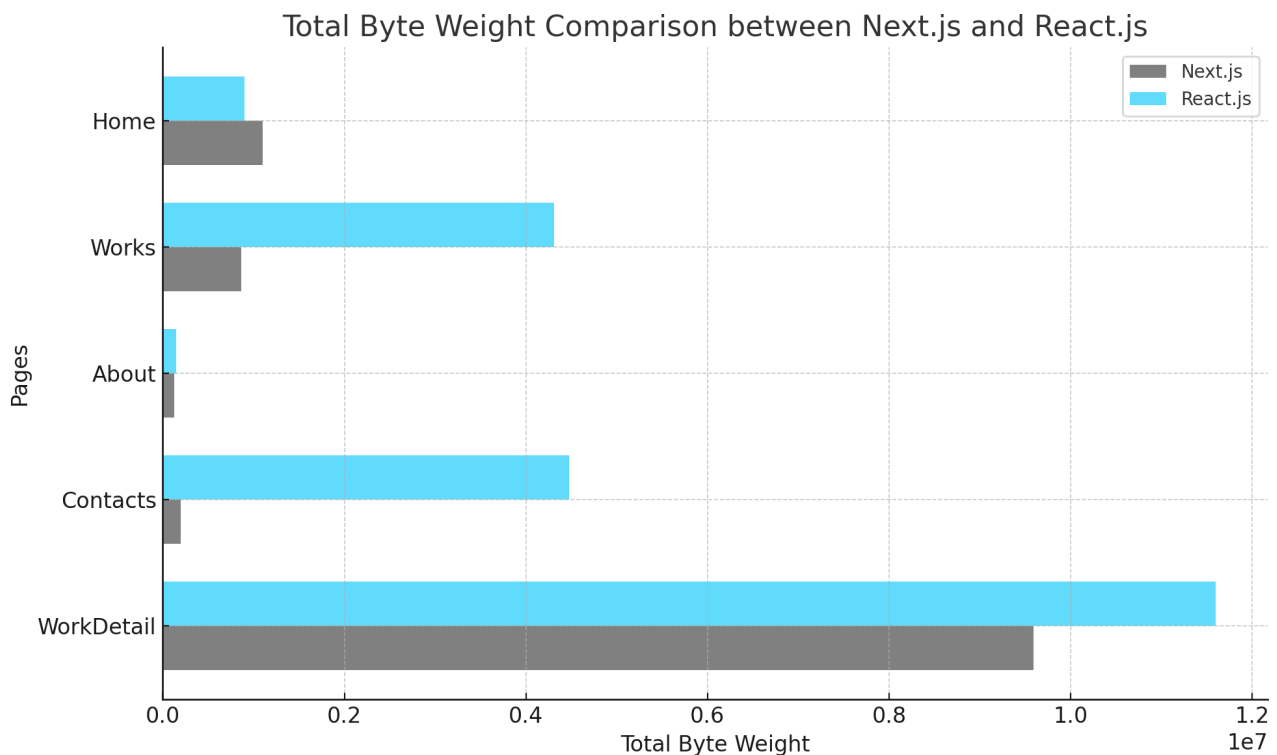
3.3 Page Weight Comparison Between Next.js and React.js

In addition to examining loading and interactivity performance, we conducted a detailed analysis of page weight, comparing the Next.js and React.js versions. Page weight significantly impacts loading times, especially on slow connections, and is a key indicator of resource efficiency.

Comparison Results:

Pagina	Next.js (totalByteWeight)	React.js (totalByteWeight)	Next.js (mainDocumentTransferSize)	React.js (mainDocumentTransferSize)
WorkDetail	9,587,519 bytes	11,593,814 bytes	5,677 bytes	606 bytes
Contacts	196,844 bytes	4,473,034 bytes	1,685 bytes	497 bytes
About	126,904 bytes	149,020 bytes	2,161 bytes	551 bytes
Works	864,675 bytes	4,308,050 bytes	3,056 bytes	552 bytes
Home	1,096,959 bytes	898,551 bytes	1,704 bytes	650 bytes





In Lighthouse, the **totalByteWeight** and **mainDocumentTransferSize** values represent important aspects of the web page's resource load:

1. Total Byte Weight:

- This value represents the total data transferred to fully load a web page. It includes all types of resources such as HTML, CSS, JavaScript, images, fonts, and other files. It is a key measure for evaluating the overall resource weight of the page, directly affecting loading times. Excessive weight can slow down page loading, especially on slow connections.

2. Main Document Transfer Size:

- Indicates the amount of data transferred to obtain the page's main HTML document. This value is part of the **totalByteWeight** and is important because it is the initial file the browser must download to begin processing and displaying the page. A smaller size can improve initial loading times.

Analysis of Results:

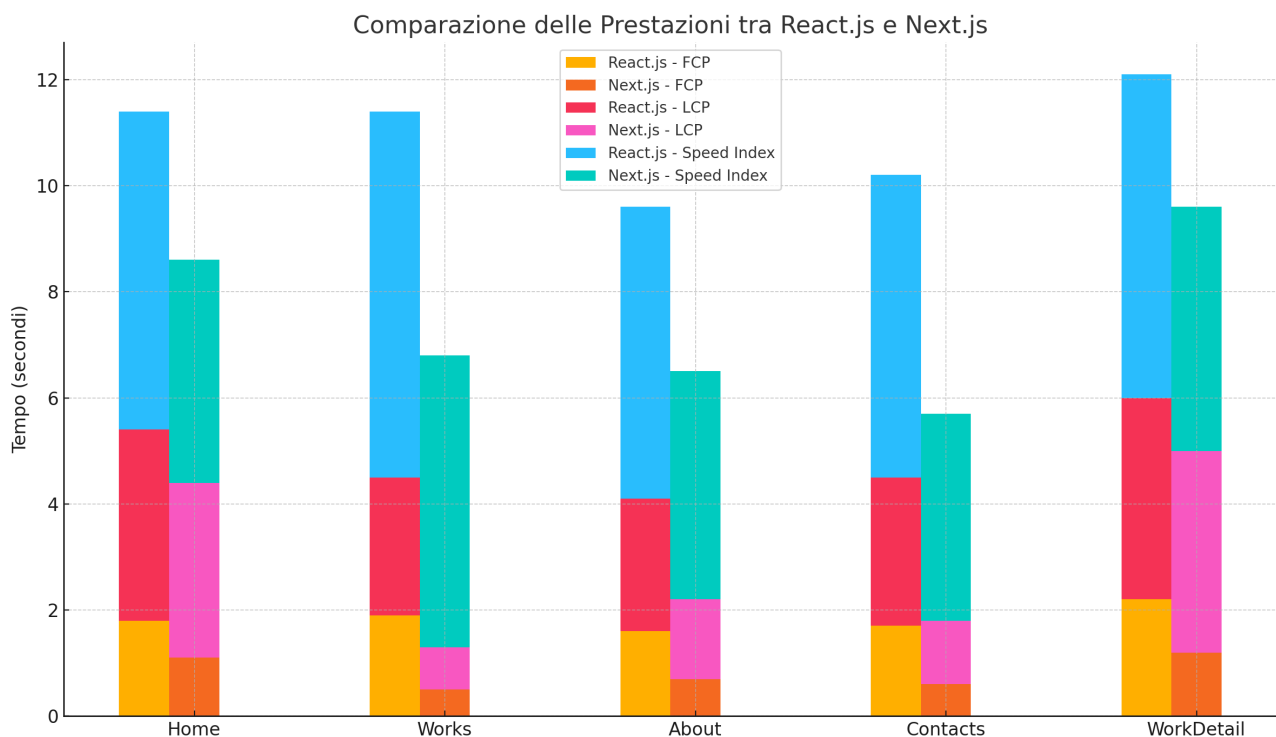
From the collected data, we can see that Next.js offers significantly lower page weight in several key areas compared to React.js, with an average improvement ranging from **10%** to **75%**, depending on the page's complexity and loaded resources, despite the main document size being consistently larger. This translates into faster loading times and a smoother user experience, especially on slow connections or less powerful devices. These results further confirm that Next.js, thanks to Server-Side Rendering (SSR) and native resource optimization, is a better choice for ensuring high performance and reduced resource consumption.

3.4 Overall Comparison

The results clearly show that Next.js is significantly superior to React.js in all performance metrics. In particular, Next.js excels in initial loading times and interactivity, with an average improvement of **25% to 30%** compared to React.js.

Next.js's native optimizations, particularly Server-Side Rendering (SSR) and efficient resource management, significantly enhance the perceived speed of the application and reduce blocking times.

Here is the graph showing the performance comparison between React.js and Next.js for the tested pages, based on metrics such as **First Contentful Paint (FCP)**, **Largest Contentful Paint (LCP)**, and **Speed Index**.



From the graph, the performance improvement of 25% to 30% when choosing Next.js can be visually appreciated.

3.5 Future Improvements

As previously explained, to facilitate the comparison tests between React.js and Next.js, caching and lazy loading techniques were not implemented. This approach ensures that performance measurements are based solely on rendering capabilities and resource management, without the influence of caching or deferred loading optimizations.

However, the user experience is not optimized, as pages may take longer to fully load.

Here's how to further optimize the **lucajop.it** website:

1. Implementation of Caching:

- **Server-Side Caching:** Use server-side caching strategies to store frequent API responses.
- **Client-Side Caching:** Implement caching of static resources via service workers or cache manifests.

Example of Custom Headers Configuration in `next.config.js`:

```
// next.config.js
module.exports = {
  async headers() {
    return [
      {
        source: '/:all*(jpg|jpeg|png|gif|svg|css|js)',
        headers: [
          {
            key: 'Cache-Control',
            value: 'public, max-age=31536000, immutable',
          },
        ],
      },
    ];
  },
};
```

2. Implementation of Lazy Loading:

- **Images:** Use Next.js's `<Image>` component to lazy-load images.
- **Components:** Load dynamic components only when needed using `next/dynamic`.

Example of Using Next.js's `<Image>` component:

```
// javascript Next.js lazy loading
import Image from 'next/image';

function WorkDetail({ images }) {
  return (
    <div>
      {images.map((img, index) => (
        <Image
          key={index}
          src={img.src}
          alt={img.alt}
          width={500}
          height={300}
          loading="lazy" // Lazy loading automatico
        />
      ))}
    </div>
  );
}

export default WorkDetail;
```

3.6 Conclusion of Performance Analysis

This chapter has illustrated the methodologies and tools essential for conducting performance tests on the React.js and Next.js projects for **lucajop.it**. Through the use of tools like Google Lighthouse, WebPageTest, and Chrome DevTools, it is possible to gain a thorough understanding of the application's performance, identify areas for improvement, and compare the capabilities of the two technologies.

Despite the absence of optimizations such as caching and lazy loading, the performance tests provide a solid foundation for evaluating rendering efficiency and resource management. Planning for future improvements will further optimize the application, enhancing the user experience and ensuring system scalability.

Conclusions

This thesis has thoroughly explored the comparison between two of the most widely used technologies in modern web application development, React.js and Next.js, focusing on the differences between Client-Side Rendering (CSR) and Server-Side Rendering (SSR). Through the implementation of real-world projects and the analysis of their performance, we have clearly outlined the strengths and weaknesses of each approach, evaluating critical aspects such as page loading speed, resource optimization, and search engine indexing.

The tests conducted using tools like Google Lighthouse confirmed that Next.js outperforms React.js in several contexts due to its ability to optimize resources, improve initial loading speed, and ensure better overall performance, particularly for websites requiring strong SEO optimization. The main advantages of Next.js emerged in the significant improvement of metrics such as First Contentful Paint (FCP) and Largest Contentful Paint (LCP), particularly noticeable on slow connections or less powerful devices, and in the native optimization of images and code through code splitting.

However, despite Next.js's benefits, the CSR offered by React.js remains a suitable solution for contexts requiring high interactivity and responsiveness on the client side, such as Single-Page Applications (SPAs).

Overall, the thesis has provided a comprehensive overview of how different rendering technologies can impact the design and performance of web applications, offering a solid starting point for future implementations and optimizations.

Bibliography

1. **W3C**, "Client-Side vs Server-Side Scripting."

Available at: [https://www.w3.org/wiki/Client-side_vs_server-side_scripting]

2. **Flanagan, D.** (2020). *JavaScript: The Definitive Guide*. 7^a edizione. O'Reilly Media. ISBN: 978-1491952023.

3. **React Documentation**, "Introducing JSX."

Available at: [<https://reactjs.org/docs/introducing-jsx.html>]

4. **Facebook Engineering**, "Virtual DOM and Internals."

Available at: [<https://engineering.fb.com/2013/06/20/web/react-a-javascript-library-for-building-user-interfaces/>]

5. **Vue.js Documentation**, "Reactivity in Depth."

Available at: [<https://vuejs.org/v2/guide/reactivity.html>]

6. **Next.js Documentation**, "Server-Side Rendering and Data Fetching."

Available at: [<https://nextjs.org/docs/basic-features/data-fetching>]

7. **Mozilla Developer Network (MDN)**, "Introduction to the DOM."

Available at: [https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction]

8. **Google Developers**, "Rendering on the Web."

Available at: [<https://developers.google.com/web/updates/2019/02/rendering-on-the-web>]

9. **Grigorik, I.** (2013). *High Performance Browser Networking*. O'Reilly Media.

ISBN: 978-1449344764.

10. **Lighthouse**, "Performance Audits."

Available at: [<https://developers.google.com/web/tools/lighthouse/audits>]

11. **Souders, S.** (2007). *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly Media.

ISBN: 978-0596529307.

12. **Prerender.io**, "Getting Started Guide."

Available at: [<https://prerender.io/documentation/getting-started>]

13. **Redux Documentation**, "Getting Started with Redux."

Available at: [<https://redux.js.org/introduction/getting-started>]

14. **Vuex Documentation**, "State Management for Vue.js."

Available at: [<https://vuex.vuejs.org/>]

15. **Google Developers**, "SEO Basics for Single Page Apps."

Available at: [<https://developers.google.com/search/docs/fundamentals/javascript-seo-basics>]

16. **Mozilla Developer Network (MDN)**, "Asynchronous JavaScript."

Available at: [<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous>]

17. **Fielding, R. T. & Taylor, R. N.** (2002). "Principled Design of the Modern Web Architecture." *ACM Transactions on Internet Technology*, 2(2), 115-150.

18. **Sharkie, P. & Brewer, M.** (2018). *Progressive Web Apps with React*. Apress.

ISBN: 978-1484240415.

19. **Eisenman, E.** (2019). *Learning React: Modern Patterns for Developing React Apps*. 2^a edizione. O'Reilly Media.

ISBN: 978-1492051725.

20. **Wang, A. & Hsu, M.** (2018). *Fullstack Vue: The Complete Guide to Vue.js*. Fullstack.io.

ISBN: 978-1987595291.

21. **Google Developers**, "Web Fundamentals: Performance Optimization."

Available at: [<https://developers.google.com/web/fundamentals/performance/>]

22. **IBM Developer**, "Understanding Server-Side Rendering and Its Benefits."

Available at: [<https://developer.ibm.com/articles/wa-understanding-server-side/>]

23. **Rauschmayer, D.** (2019). *JavaScript for Impatient Programmers*. Leanpub.

ISBN: 978-1091210098.

24. **Cormier, J. & Connors, T.** (2020). *Next.js Quick Start Guide*. Packt Publishing.

ISBN: 978-1839211560.

25. **WHATWG**, "HTML Living Standard."

Available at: [<https://html.spec.whatwg.org/>]

26. **Rauch, G.** (2016), "Introducing Next.js". *Vercel Blog*.

Available at: [<https://vercel.com/blog/next>]

27. **Nuxt.js Documentation**, "Introduction to Nuxt.js."

Available at: [<https://nuxtjs.org/docs/get-started/installation>]

28. **Chopin, S. & Chopin, A.** (2016), "Announcing Nuxt.js: A Universal Vue.js Application Framework." *Nuxt.js Blog*.

Available at: [<https://nuxtjs.org/blog/nuxt-isomorphic-application-framework-vuejs>]

29. **Vercel**, "Documentation".

Available at: [<https://vercel.com/docs/getting-started-with-vercel>]

30. Google **Lighthouse**, "Overview". *Chrome for Developer*.

Available at: [<https://developer.chrome.com/docs/lighthouse/overview?hl=it>]