

Wrocław University of Science and Technology
Faculty of Information and Communication Technology

Field of study: **Computer Engineering (ITE)**
Speciality: **Internet Engineering (INE)**

MASTER THESIS

**Comparative Analysis of React Server Components and
Client Components in Modern Web Applications**

Murad Shahbazov 251213

Supervisor

prof. dr hab. inż. Czesław Smutnicki

Keywords: React Server Components, Client Components, Web Performance, SSR, SEO, Next.js, Frontend Optimization, Vite

WROCLAW (2025)

Abstract

This thesis is a comparison of React Server Components (RSC) with traditional Client Components (RCC) regarding contemporary web application development. Two functionally identical applications were created using different rendering approaches: client-rendered React application using SWR and Axios, and server-rendered Next.js app with the use of React Server Components and Prisma. Performance, bundle size, memory use, and SEO support were quantified with the assistance of tools such as Google Lighthouse and bundlers. The results are that the RSC-based app outperforms the RCC counterpart in page load time, visual stability, and SEO with the benefit of reducing client-side JavaScript overhead. The study concludes that React Server Components offer a good solution for building scalable, high-performance, and SEO-amenable web apps.

Po Polsku:

Niniejsza praca stanowi porównanie komponentów serwerowych React (RSC) z tradycyjnymi komponentami klienckimi (RCC) w kontekście współczesnego tworzenia aplikacji webowych. W tym celu stworzono dwie funkcjonalnie identyczne aplikacje, wykorzystujące różne podejścia do renderowania: aplikację renderowaną po stronie klienta z użyciem SWR i Axios oraz aplikację serwerową opartą na Next.js, wykorzystującą komponenty serwerowe React oraz Prisma. Wydajność, rozmiar paczki, zużycie pamięci i wsparcie dla SEO zostały zmierzone przy pomocy narzędzi takich jak Google Lighthouse oraz bundlery. Wyniki wykazały, że aplikacja oparta na RSC przewyższa swoją odpowiedniczkę opartą na RCC pod względem czasu ładowania strony, stabilności wizualnej i wsparcia dla SEO, jednocześnie zmniejszając obciążenie JavaScriptu po stronie klienta. Badanie kończy się wnioskiem, że komponenty serwerowe React stanowią dobre rozwiązanie do budowy skalowalnych, wydajnych i przyjaznych SEO aplikacji internetowych.

Table of Contents

Abstract	2
Table of Contents	3
List of Abbreviations and Symbols	4
1. Thesis Objective and Scope	5
2. Introduction	5
2.1 Background: Introduction to React	6
2.2 React Components: Client and Server	6
2.3 Server and Client Environments	7
2.4 Server Components	8
2.5 Server Components in Next.js	8
4. Application Architecture & Data Handling	10
4.1 Overview of the Application and Shared Technical Stack	10
4.2 Traditional Client Component Architecture	12
4.3 Server Component Architecture	13
5. General Look of the Application	16
6. Performance Benchmarking	17
6.1 Testing Setup and Conditions	18
6.2 Metric-by-Metric Comparison and Interpretation	19
6.3 SEO	21
6.4 Overview	22
7. Bundle Size Analysis	22
7.1 Methodology and Tooling	22
7.2 Client Component App (RCC) Analysis	23
7.3 Server Component App (RSC) Analysis	23
7.4 Comparative Breakdown	24
7.5 Performance Implications	25
7.6 Overview	25
8. Developer Experience	26
8.1 Overview	27
9. Conclusion	27
References	28
List of Objects	29
List of Figures	29
List of Tables	29
List of Code Listings	29
Appendix A — Project Repository and Setup Instructions	30

List of Abbreviations and Symbols

API – Application Programming Interface

HTTP – Hypertext Transfer Protocol

CSS – Cascading Style Sheets

HTML – HyperText Markup Language

JS – JavaScript

RCC – React Client Components

RSC – React Server Components

SSR – Server-Side Rendering

CSR – Client-Side Rendering

SEO – Search Engine Optimization

FCP – First Contentful Paint

LCP – Largest Contentful Paint

TBT – Total Blocking Time

CLS – Cumulative Layout Shift

SWR – Stale-While-Revalidate (React data fetching library)

CDN – Content Delivery Network

DOM – Document Object Model

UI – User Interface

UX – User Experience

TTFB – Time To First Byte

ORM – Object-Relational Mapping

CRUD – Create, Read, Update, and Delete

1. Thesis Objective and Scope

The objective of this thesis is to analyze and compare two contemporary frontend rendering models in React: Client Components (RCC) and Server Components (RSC). The objective is to see how each model affects application performance, developer experience, architectural complexity, and SEO in practical scenarios.

From an engineering perspective, this project involves implementing and designing two functionally identical web applications based on modern frameworks and tooling. One is built with React Client Components and traditional frontend-backend architecture, and the other is using React Server Components and server-side data fetching. Both solutions implement the same CRUD features and use an identical UI component library to ensure a fair comparison

From a research perspective, the thesis investigates how rendering strategy affects measurable metrics such as load time, JavaScript bundle size, network usage, and developer productivity. It seeks to identify pragmatic trade-offs and to establish whether React Server Components represent a paradigm shift in frontend development or an situational optimization.

2. Introduction

The world of contemporary web development is changing at a very fast pace, fueled by demands for improved page load, enhanced SEO, reduced bundle sizes, and manageable architecture. Web applications must be virtually instantly interactive without sacrificing maintainability or performance as expectations of users grow.

React, one of the most widely used frontend libraries, has long relied on Client Components—interactive UI blocks that are rendered entirely in the browser. While this model is powerful and flexible, it comes up short in some ways, specifically initial load performance and search engine discoverability. To reduce these constraints, the React team introduced React Server Components (RSC), a novel rendering mode whereby part of the UI can be server-side rendered and streamed to the client.

This thesis compares Client Components and Server Components in contemporary React applications. It juxtaposes their effects on performance, scalability, SEO, developer experience, and architectural complexity. The comparison is verified by building two parallel web applications—one with Client Components and a conventional backend, and the other with React Server Components and direct server-side data fetching.

The aim is to bring out the compromises involved in client-heavy versus server-heavy rendering approaches and determine if React Server Components are a paradigm change in frontend architecture for practical applications.

2.1 Background: Introduction to React

React [1] is a popular open-source JavaScript library for building interactive user interfaces. React is maintained by Facebook and allows developers to compose UI components into a hierarchy of reusable components that efficiently update and render when data changes. React's primary goal is to simplify the construction of complex user interfaces by breaking them down into simple, reusable components.

A user interface (UI) is the part of an application that users see and interact with — buttons, forms, navigation bars, text, and so on. React provides a declarative syntax that makes it easier to design and manage these pieces and keep the UI in sync with your application's state.

Of course, React is a library, not a framework. It specifically targets rendering user interfaces, leaving other problems like routing, state management, and data fetching up to the developer or third-party libraries.

React renders UI components into the Document Object Model (DOM) through a virtual DOM strategy, which enables small and efficient updates whenever the state of the application changes. It has support for native HTML and SVG components, and also permits developers to define their own custom components using JavaScript and JSX (JavaScript XML).

2.2 React Components: Client and Server

In React, the whole world is about the concept of components. A component is a reusable piece of code that defines some part of the UI. There are two general types of components in modern React architecture:

- Client Components
- Server Components

By convention, React apps have been very Client Component-heavy, with rendering and logic taking place solely in the browser. Such components handle state, user interaction, and sometimes even API calls for data in JavaScript. That said, as applications increase in size and complexity, client-only rendering can lead to performance issues, namely with initial page loads and search engine optimization (SEO).

To solve these issues, the React team released React Server Components (RSC) — an exciting new feature that makes it possible to render some of the UI on the server and then send it to the browser. Server Components are meant to be used together with Client Components in a hybrid

architecture, and they give developers the ability to decide where to run their code based on what it's doing.

2.3 Server and Client Environments

In order to study Server and Client Components, one needs to understand the two environments in which code may be run in a web application, as illustrated in Figure 2.1:

- The client: This refers to the user's browser, which downloads and runs JavaScript, draws the interface, and manages user interactions.
- The server: This is a backend server (usually in a data center or cloud setting) that holds the application's source code, manages database interaction, executes API calls, and creates responses to return to the client.

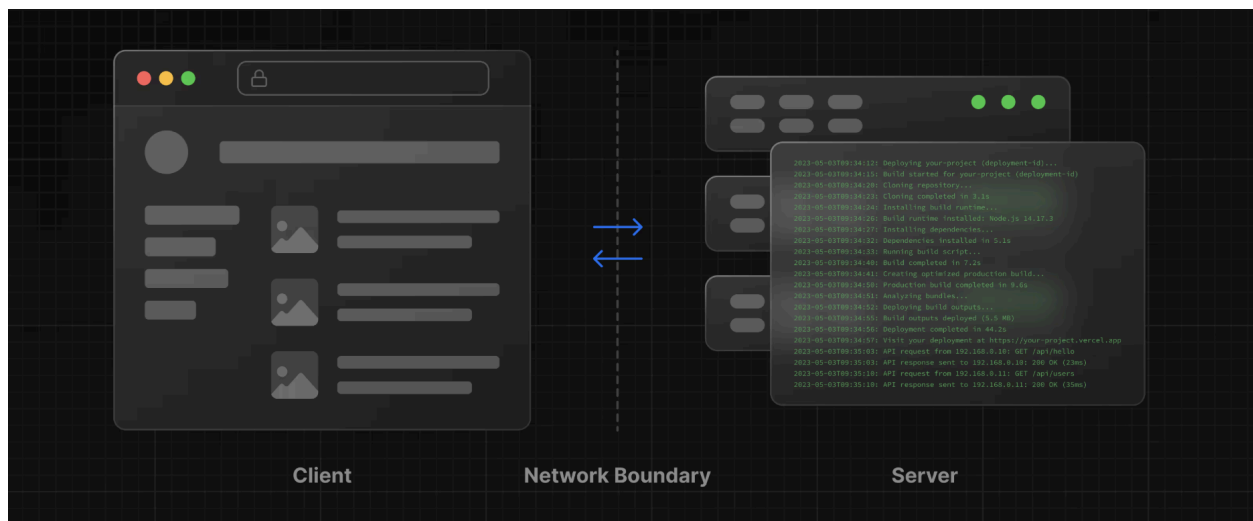


Figure 2.1 Server and Client Environments
Source: Next.js official documentation (Vercel, 2024)

Each environment has its respective weaknesses and strengths. For instance, the server can more efficiently and safely retrieve sensitive data (API keys or environment secrets, for instance) and perform computationally expensive tasks. The client, by contrast, is responsible for dynamic interactivity — i.e., responding to button clicks, handling form input, or accessing browser-specific APIs like `localStorage`.

2.4 Server Components

React Server Components are intended to run on the server, prior to sending the JavaScript bundle to the client. This step of pre-rendering enables the application to display fully rendered HTML to the browser, which effectively minimizes the time to first paint and also makes the application more SEO-friendly.

Server Components can:

- Fetch data directly from databases or APIs, close to the source
- Access secure tokens, API keys, or backend services without exposing them to the client
- Reduce the amount of JavaScript sent to the browser, which improves load times
- Be cached or streamed incrementally to the browser for fast content rendering

Client Components, on the contrary, are required when the application needs:

- Event handlers (like `onClick`, `onChange`)
- Lifecycle methods (`useEffect`, etc.)
- Browser-specific APIs (like `window`, `navigator`, or `localStorage`)
- Custom hooks that depend on the browser environment

2.5 Server Components in Next.js

Although React Server Components (RSC) are a fundamental idea presented by the React team, they are not supported out of the box across all environments. To enable them in a production-ready environment, one needs a supporting framework. At the time of writing, Next.js [2] 13+, with its App Router architecture, supports the most mature and officially supported RSC implementation.

Here, Server Components are run on the server and produce a React Server Component Payload (RSC Payload)—a serialized form that contains the output of the server-rendered component tree along with pointers to where Client Components should be hydrated on the page. When a user accesses the application:

- The server pre-renders the Server Components and generates the RSC Payload.
- The HTML and a static representation of the UI are rendered immediately (non-interactive) sent to the browser.

- JavaScript is then loaded to hydrate Client Components — attaching event handlers and making the interface interactive.
- On subsequent navigations, cached RSC Payloads allow for fast, seamless page transitions without needing full re-rendering.

This allows React applications to render fast, content-rich pages that are search engine and user-friendly, while enabling rich interactivity with Client Components where it is required.

3. Study Design and Objectives

In order to comparatively examine and test the practical differences between React Server Components and React Client Components, two functionally equivalent web applications were implemented. The goal of the research is to analyze how each approach to rendering affects performance, efficiency, SEO, and developer experience within the context of a modern web application.

Two applications were implemented:

- The React Client Components application (RCC) is built with React and Vite [4]. It's the old frontend-backend model, calling a REST API [12] written in TypeScript [8] with an SQLite [7] database. Data fetching on the client side is handled with Axios and SWR.
- The Server Components app (RSC) is built using Next.js 13+ and the App Router. It uses React Server Components and talks to the database directly using Prisma [3] without a separate backend. Server-side rendering is used by default, with interactive Client Components embedded where needed.

Both applications share the same features a CRUD-based event management system—and the same UI component library. This parity ensures that any differences in performance or development experience that are noticed are attributable to the rendering model, rather than to extraneous differences.

The comparison focuses on four key areas:

- Performance: Load time, JavaScript bundle size, and memory usage.
- Efficiency: Data-fetching behavior, network overhead, and hydration cost.
- SEO: Server-rendered content visibility.
- Developer Experience: Project architecture, maintainability, and ease of implementation.

The remainder of this document details the structure of both applications, presents performance benchmarking results, and discusses development-related lessons learned from the implementation process.

4. Application Architecture & Data Handling

In order to fairly compare React Server Components (RSC) and regular Client Components (CC), two functionally identical event management applications were developed. Their sole distinction lies in their architecture and data-fetching approaches, which are two paradigms of building modern web applications.

4.1 Overview of the Application and Shared Technical Stack

To enable a useful and equitable comparison between the Client Component (CC) and React Server Component (RSC) strategies, both applications were designed to accomplish the same functional demands with a shared UI organization and styling scheme. The live application is a basic event management tool that allows users to CRUD (create, read, update, delete) events. This ensures that both frontend implementations control the same data structures, processes, and user interactions.

Core Features

Both versions of the application possess the following features:

- Event Listing: A list of upcoming events in date order, sorted by time.
- Event Creation: Input form for data such as event name, location, date, time, image URL, and contact information.
- Event Editing: Edit previously created events with updated details.
- Event Deletion: Permanently delete an event from the database.

Interface & Styling Tools

The user interface in both versions benefits from the same styling foundation:

- Tailwind CSS [10]: Utility-first CSS framework that allows for rapid styling via pre-defined classes. Tailwind offers flexibility in design while maintaining consistency across the application.
- shadcn/ui [6]: Library of UI components made up of Tailwind and Radix UI primitives. It offers accessible, pre-styled React components such as modals, inputs, buttons, and alerts. Utilization of this library reduced boilerplate UI overhead and allowed functional differences between architectures to be prioritized.

All shared components (e.g., buttons, dialogs, form controls, tables, etc.) were built with the shadcn/ui system and integrated inside Tailwind-styled wrappers. This gave visual and functional parity between the two projects.

Database and Data Model

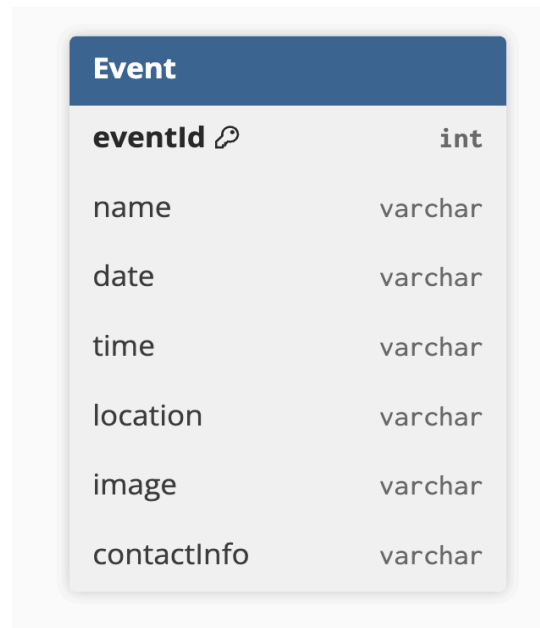
Both projects use SQLite as the underlying database. It is a light, file-based relational database that is ideal for small programs and testing. One of the major reasons SQLite was selected is that it is easy to install and does not require a separate database server, which is easier to develop and faster to develop.

The data model is made simple intentionally. There is only one table, which is called Event, and it contains all the data that is needed in dealing with events. The fields are eventId, name, date, time, location, image, and contactInfo.

Having the same database schema for both client and server component versions of the app makes things simple. That way, if there's any difference in behavior or performance between the two, it's purely because of how the frontend is put together — not because of the database.

The intention here is not to create an elaborate system with numerous features or tables, but simply to contrast how the same data is processed in two different frontend designs.

Figure 4.1 shows the detailed schema of the database table relevant to the application.

A diagram of a database table named 'Event'. The table has a dark blue header with the name 'Event' in white. Below the header, the table structure is shown with columns and their data types. The first column, 'eventId', is the primary key, indicated by a key icon, and has a data type of 'int'. The other columns are 'name', 'date', 'time', 'location', 'image', and 'contactInfo', all with a data type of 'varchar'.

Event	
eventId 🔑	int
name	varchar
date	varchar
time	varchar
location	varchar
image	varchar
contactInfo	varchar

Figure 4.1 Database structure

4.2 Traditional Client Component Architecture

In the first approach to the application's implementation, the architecture shares a standard client-server layout that most traditional web applications possess. The backend is made by Express.js and TypeScript, and carries out the database tasks. It uses SQLite3 for storing data, making everything simple and local as per the intention behind this comparison.

On the client side, the application is developed using React and normal Client Components. These aren't directly tied to the database — instead, they're making HTTP requests to the server. To make those requests (like creating or deleting an event), I used Axios [9], a popular JavaScript HTTP client.

For data fetching, I used SWR [5], a light data fetching library built by Vercel. It is easier to deal with remote data than to do it manually with `useEffect` and local state. With SWR, you have caching, revalidation, and loading/error state management for free out of the box — something most modern frontend applications nowadays rely on. There are other alternatives like React Query, but SWR was more than enough for my case.

For making everything reusable and clean, I created a simple custom hook that wraps the SWR logic for loading events. This allows any component within the app to simply call `useEvents()` and get all the data and states it needs.

Listing 4.1 shows the code for this hook.

```
import useSWR from 'swr';

import { Event } from '@types/event';

const EVENTS_URL = '/events';

export const useEvents = () => {
  const { data, error, isLoading, mutate } = useSWR(EVENTS_URL);
  return {
    events: data as Event[],
    isLoading,
    error: error,
    mutate,
  };
};
```

Code Listing 4.1: Data Fetching in RCC

For stuff like adding, updating, or deleting an event, I just employed plain `axios.post`, `axios.put`, and `axios.delete` requests. It's a very ubiquitous pattern that keeps concerns separate: the frontend handles the UI and talks to the backend, who then talks to the database. This pattern is still what most production apps use today.

4.3 Server Component Architecture

For the second deployment of the application, I used the React Server Components (RSC) approach, though. The easiest and most recommended method of applying RSC nowadays is with Next.js, as this supports out-of-the-box right now. It's basically the default if you're creating something production-grade with server components these days.

All Next.js components are server components by default. That is, they get rendered on the server and shipped down as HTML to the browser. They're not interactive in themselves — they're simply plain old static markup unless we purposely make them interactive.

If we need to get something interactive (a button, form, modal, etc.), we need to make that specific component a client component. That's done by just adding "use client" at the top of the file. That tells Next.js to render it in the client, not the server, and enables things like `useState`, `useEffect`, and browser events. So in this project, I used a hybrid approach: most of the page is server components, and only the pieces that need interaction are client components.

Another key difference is that we don't need a separate backend. With server components, we can connect directly to the database from inside the Next.js application. Server code never reaches the browser — pre-rendered HTML is sent along with a very small JavaScript bundle (for the interactive bits). That's more secure and doesn't require you to set up a separate backend API.

I used a new ORM called Prisma to handle database queries. It is easier than SQL raw, and gets along great with SQLite. Being inside a full-stack Next.js application, I can just call `prisma.event.findMany()` in any server component or server function. Code Listing 4.2 shows an example of how easy it is to get data.

```
const EventsPage = async () => {  
  const events: Event[] = await prisma.event.findMany({  
    orderBy: { date: 'asc' },  
  });  
  // render events using HTML  
};
```

Code Listing 4.2: Data Fetching in RSC

No custom data fetching hooks to write, no API URL concerns, and no `useEffect` management. It's all loaded in advance of the component even being rendered. For add, update, or delete operations on existing events, I used server actions. They're "use server" marked special functions callable only on the server. They make it safe (e.g., no potential for SQL injection from the client) and neatly decoupled logic. It is shown in the Code Listing 4.3.

```

'use server';

import { prisma } from '@lib/prisma';
import { EventSchemaType } from '@schemas/event-schema';
import { format } from 'date-fns';

export async function addEvent(data: EventSchemaType) {
  await prisma.event.create({
    data: {
      ...data,
      date: format(new Date(data.date), 'yyyy-MM-dd'),
    },
  });
}

export async function updateEvent(eventId: number, data:
EventSchemaType) {
  await prisma.event.update({
    where: { eventId },
    data: {
      ...data,
      date: format(new Date(data.date), 'yyyy-MM-dd'),
    },
  });
}

export async function deleteEvent(eventId: number) {
  await prisma.event.delete({
    where: { eventId },
  });
}

```

Code Listing 4.3: Server Actions

I can import and call these actions from my client components (like forms or buttons) and know that everything is handled safely and efficiently.

Overall, this approach simplifies the whole workflow — no additional backend setup needed, fewer moving parts, and easier data handling all around.

5. General Look of the Application

Both applications share the same user interface. This is done to maintain the comparison as to how they are built, and not how they look. The UI includes a simple list of events with data like name, date, time, location, and contact information. There are the add, edit, and delete event forms. Figures 5.1 and 5.2 show how the interface looks.

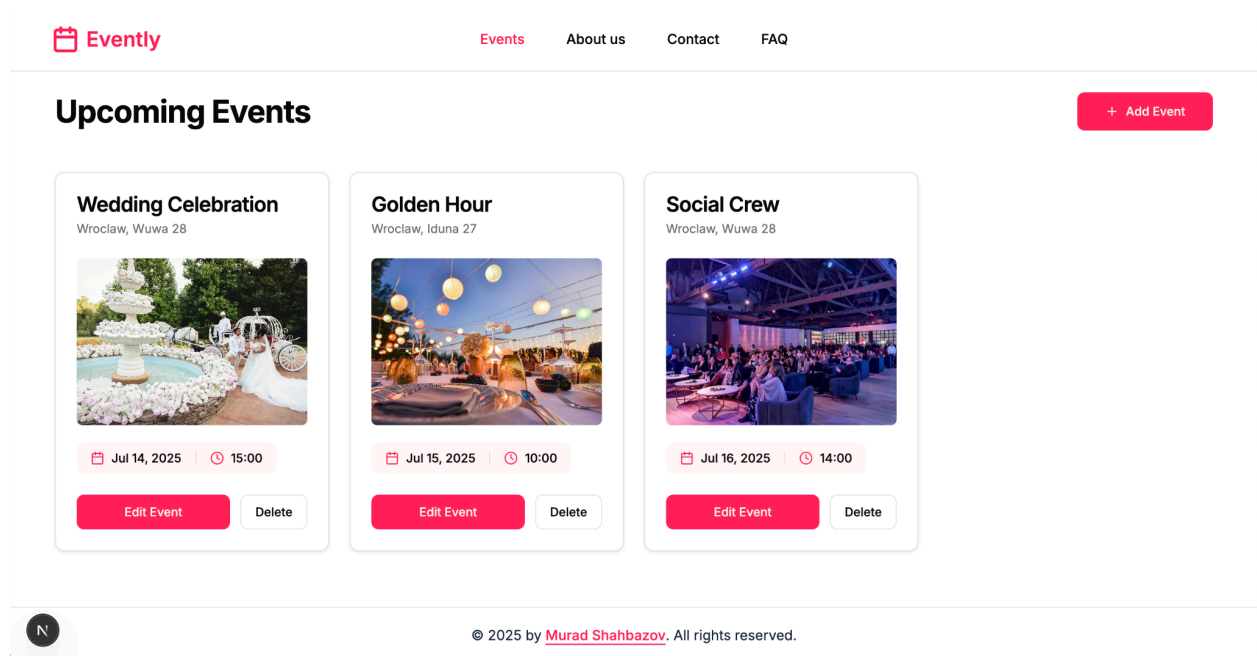


Figure 5.1 UI for displaying upcoming events

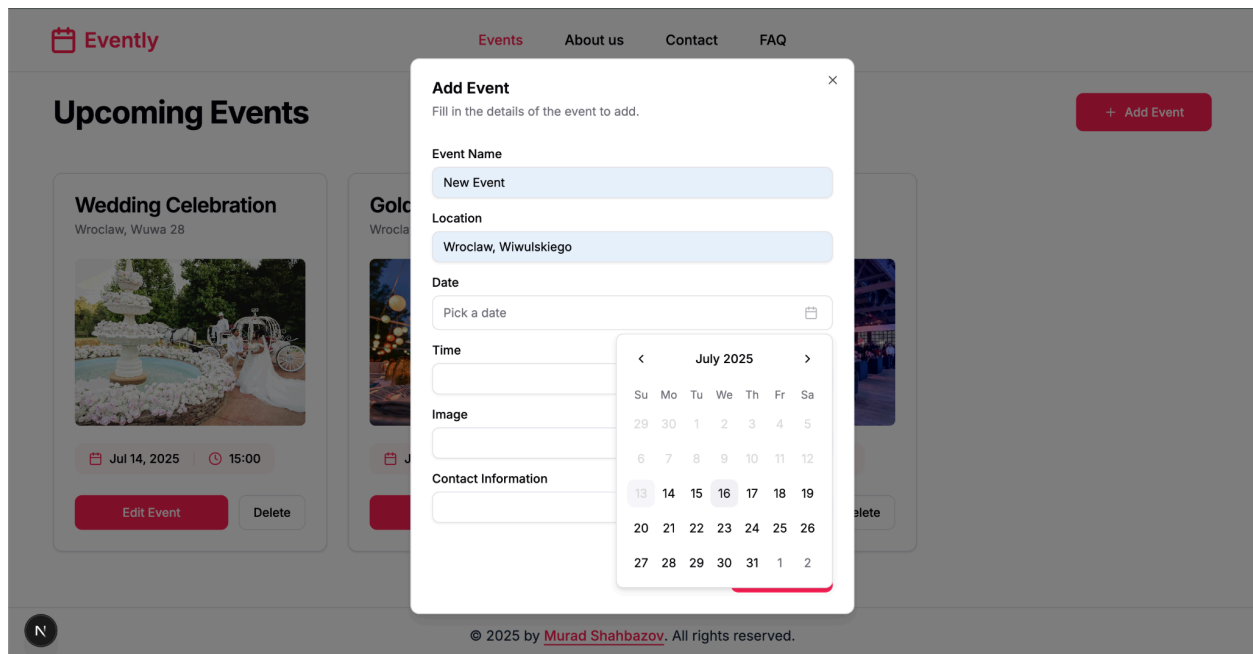


Figure 5.2 UI for adding new events

6. Performance Benchmarking

In the contemporary web development environment, performance is one of the central drivers of user experience, engagement, and overall satisfaction. As frontend applications have grown increasingly complex, it has become critical to optimize how fast and efficiently web pages load. React applications, client-rendered and server-rendered, each have different performance profiles depending on where rendering and data loading occur — in the browser or on the server.

To experiment the performance of two patterns, React Client Components (RCC) and React Server Components (RSC), in an organized fashion, the research utilizes Google Lighthouse, which is a familiar, standardized, and widely adopted tool to conduct web performance tests. Lighthouse gives us an estimate of various metrics that represent real-world load behavior as well as user interactivity experience.

6.1 Testing Setup and Conditions

Both applications were built and executed in a production environment to simulate actual conditions. The RCC app was locally hosted using Vite optimized production build (vite preview), and the RSC app was executed using Next.js production mode (next build && next start). Lighthouse audits were run in Chrome DevTools, emulated on a desktop device profile with custom throttling on so that timing was uniform for all of the important metrics such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), and Total Blocking Time (TBT).

The specific version utilized was Lighthouse 12.6.0, and all tests were performed following cold load with cache disabled to mimic initial visits — a critical scenario where the performance differences are most noticeable. The results of the reports are shown in Figures 6.1 and 6.2.

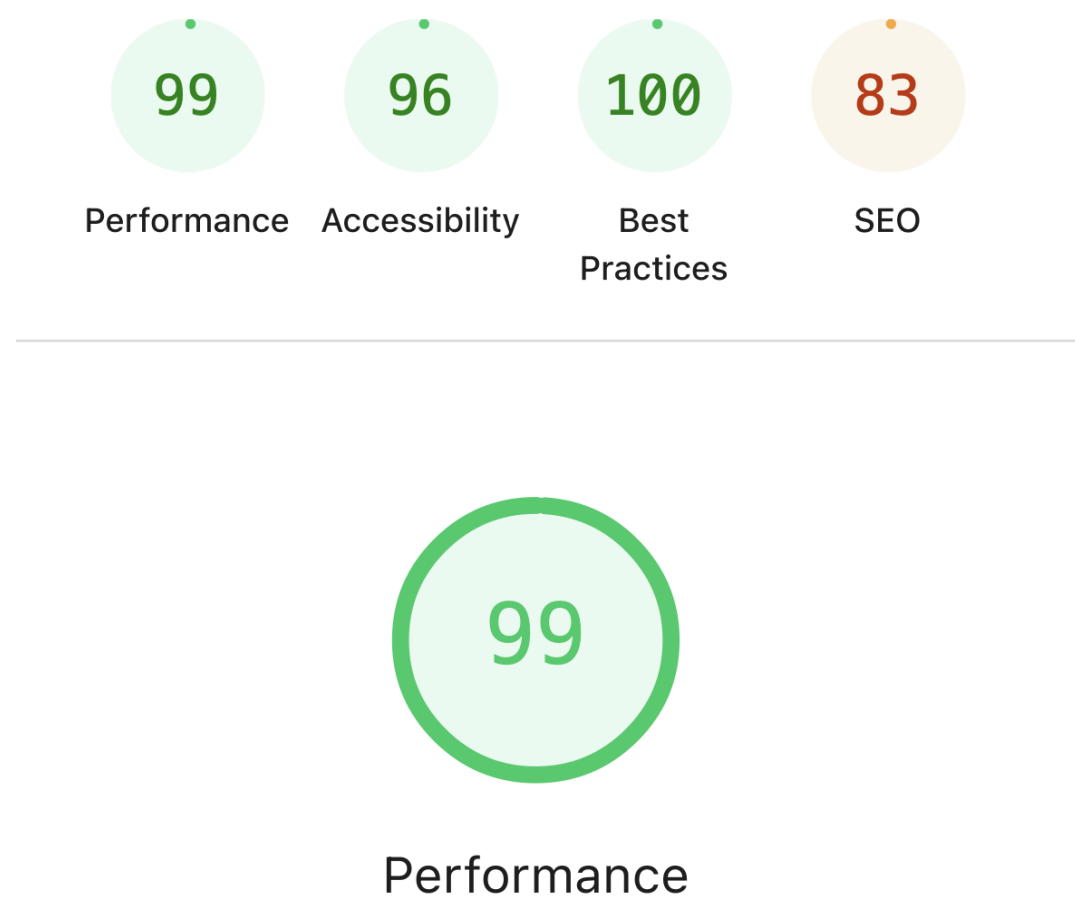


Figure 6.1 Lighthouse report for RCC

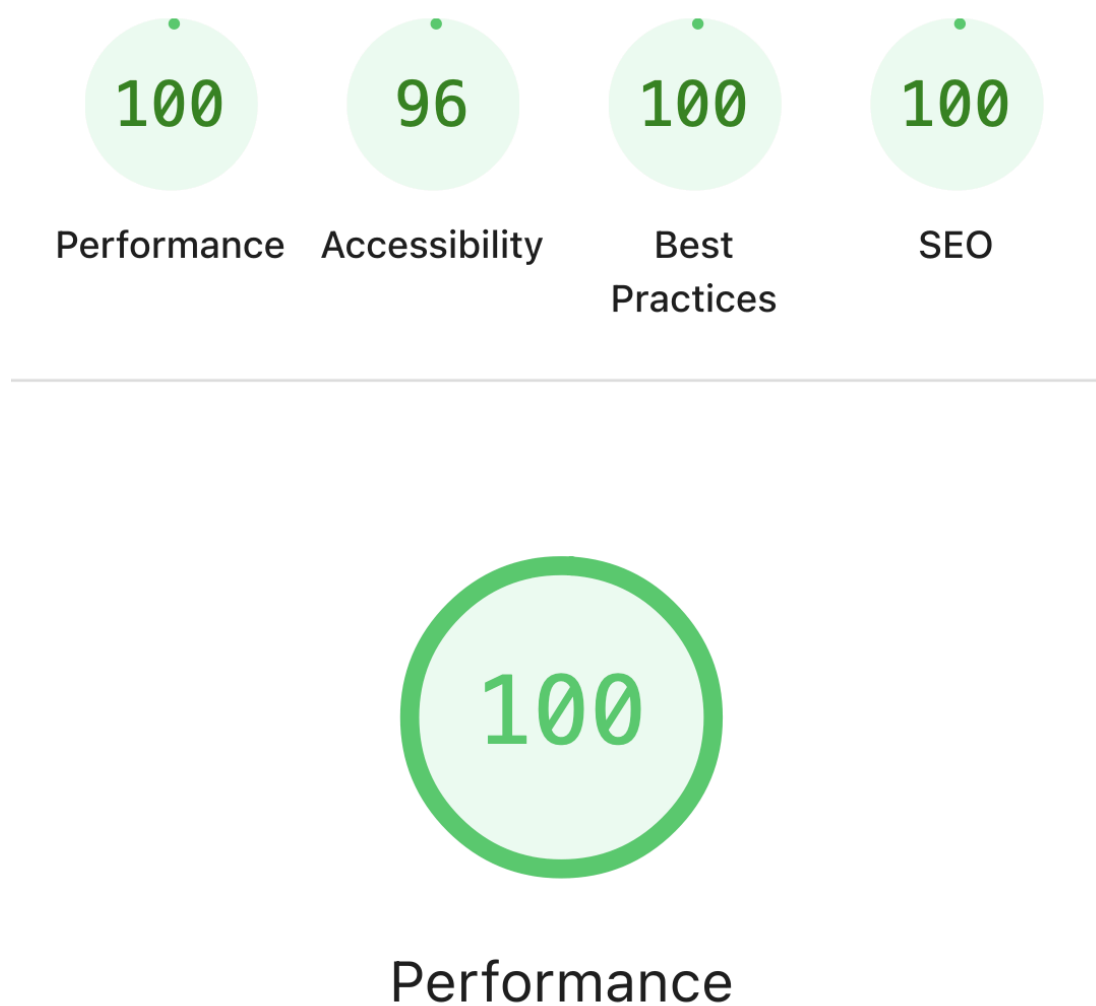


Figure 6.2 Lighthouse report for RSC

6.2 Metric-by-Metric Comparison and Interpretation

The key performance metrics for both RCC and RSC are summarized in Table 6.1.

Metric	RCC (Client Components)	RSC (Server Components)
Performance Score	99	100
First Contentful Paint (FCP)	0.4 seconds	0.2 seconds
Largest Contentful Paint	0.8 seconds	0.5 seconds
Total Blocking Time (TBT)	0 milliseconds	0 milliseconds
Cumulative Layout Shift	0.041	0.000
Speed Index	0.4 seconds	0.2 seconds

Table 6.1 RCC and RSC Performance Metrics

First Contentful Paint (FCP)

The First Contentful Paint (FCP) is a metric that indicates the time when the browser is painting the first piece of content (image or text) from the DOM. For this test, the RSC app significantly outperforms the RCC app (0.2s vs 0.4s). This improvement is mainly attributed to the fact that React Server Components render HTML on the server and send it to the browser pre-hydrated with data. This reduces the load of client-side JavaScript execution, which allows content to appear much sooner.

On the other hand, the RCC app must first load and run the JavaScript bundle, set up React, and then load data asynchronously (in this case using SWR and Axios) that adds delay before anything is rendered on screen.

Largest Contentful Paint (LCP)

LCP is the time taken by the largest visible content element (e.g., a hero image or heading) to load. The RSC app provides an LCP of 0.5 seconds compared to RCC's 0.8 seconds. Once more, this performance benefit comes from the fact that server components are available early in the client in HTML. The RCC delay comes from pre-client-side fetching of data, with page layout rendered prior to actual display of data.

Total Blocking Time (TBT)

Both the implementations took 0 ms TBT, i.e., there were no long tasks blocking the main thread upon loading. This means that neither of the apps has redundant synchronous JavaScript or long blocking tasks in the critical rendering path.

While both work well here, do remember that TBT is usually sensitive to bundle size and third-party scripts — something we'll discuss again in the next section on bundle size.

Cumulative Layout Shift (CLS)

CLS examines visual stability, i.e., how much content shifts during loading. Layout jumps enrage users and destroy user experience. The score of the RSC app is 0.000, whereas RCC delivers a non-zero shift (0.041). This is likely because of RCC's asynchronous data loading leading to delayed content injection with layout jumps.

This once more highlights RSC's architecture strength — showing the entire page with data server-side avoids content jumping in later and pushing UI components aside.

Speed Index

Speed Index shows how quickly content fills visually — not necessarily when the largest or first item is loaded. RSC app (0.2s) beats RCC (0.4s) significantly, reinforcing earlier findings: server-rendered HTML leads to faster perceived performance, and less JavaScript work delays visual updates.

6.3 SEO

The Lighthouse reports show a marked difference in SEO performance between the two approaches. The React Server Components (RSC) application had a flawless SEO score, whereas the Client Components (RCC) application was lower. This is largely because of how the content is served. In the RSC application, HTML is server-rendered and completely accessible at the time of loading, which makes it simpler for search engines to crawl. The RCC app renders most content through JavaScript after the initial load, which can limit SEO effectiveness.

6.4 Overview

The performance benchmarking clearly indicates that Next.js-powered React Server Components provide a better user experience with regards to load time, content stability, and SEO readiness. By removing the requirement for client-side data fetching and turning rendering to the server, the RSC model reduces hydration costs, decreases the bundle size, and accelerates time-to-interaction.

The RCC rendering, as effective as it is, is bottlenecked by having to wait for JavaScript parsing and running before it is able to render out any data. This creates latency and damages both SEO and perceived speed, particularly on slow devices or low networks.

These results justify the increasing popularity in the React community towards hybrid rendering, and why React Server Components are to become a pillar of contemporary, scalable, and high-performance web applications.

7. Bundle Size Analysis

In contemporary web applications, JavaScript bundle size is one of the major performance, load time, and user experience considerations. The bigger the bundle, the longer the initial render time, the higher the parsing and execution times, and the poorer the performance on less powerful devices or networks. Minifying client-side JavaScript is thus important, particularly in Single Page Applications (SPAs) developed with frameworks such as React.

This section compares the bundle sizes of two versions of the same application: one written using React Client Components (RCC), and the other using React Server Components (RSC). The comparison shows how directly architecture decisions affect the final JavaScript payload sent to the browser.

7.1 Methodology and Tooling

To measure and visualize the bundle size for both implementations, the following tools were used:

- Rollup Plugin Visualizer : For RCC application (which was built with Vite and Rollup), we used rollup-plugin-visualizer to generate an interactive HTML report. This tool visualizes the JavaScript bundle contents as a tree map in hierarchical form, sized in decreasing order.
- Next.js Bundle Analyzer: In the RSC application (which we developed using Next.js), we used the `@next/bundle-analyzer` plugin. It's a Webpack plugin that creates an equivalent HTML report to analyze the size of JavaScript chunks emitted by server-side rendering.

Both applications were compiled in production mode to make the bundle output representative. Just the sizes of the JavaScript bundle are being compared here—CSS and static files (such as fonts or images) are excluded for simplicity.

All size mentions in this section are to raw (not compressed) sizes, as determined by analysis tools.

7.2 Client Component App (RCC) Analysis

- Total JavaScript bundle size: **~183 KB** (raw)
- Key contributors:
 - react and react-dom: **~41 KB**
 - UI component library (shadcn/ui): **~26 KB**
 - Data fetching libraries (swr, axios): **~18 KB**
 - Custom UI logic and routing: **~30 KB**
 - Routing (react-router-dom): **~18 KB**
 - Miscellaneous utilities (e.g., clsx, date-fns): **~12 KB**

In the RCC architecture, almost all logic—rendering, data fetching, and routing—is executed in the browser. As a result, the final JavaScript bundle includes:

- All routes and pages
- All UI and state logic
- Third-party data-fetching libraries
- The full UI component library

This traditional client-rendered SPA model comes at the cost of heavier bundles, especially as more routes and features are added.

7.3 Server Component App (RSC) Analysis

- Total JavaScript bundle size: **~89 KB** (raw)
- Key contributors:
 - react and react-dom: **~36 KB**
 - UI component library (shadcn/ui): **~22 KB**

- Minimal interactive logic: ~**25** KB
- No client-side routing or data-fetching libraries

In contrast, the RSC app renders most data and components on the server-side. It does not include unnecessary JavaScript by:

- Eliminating swr, axios, or any client-side data-fetching utilities
- Using server-side routing (handled by Next.js)
- Sending pre-rendered HTML along with only essential JavaScript needed for interactivity (like buttons or inputs)

As a result, the client bundle is significantly smaller and simpler.

7.4 Comparative Breakdown

The bundle size comparison between Client Components (RCC) and Server Components (RSC) is shown in Table 7.1.

Metric	Client Components (RCC)	Server Components (RSC)
Total JS Bundle Size (raw)	~183 KB	~89 KB
Data Fetching Included	Yes (SWR, Axios)	No (server-side Prisma)
Routing Logic	Client-side (react-router)	Server-side (Next.js)
Hydration Load	High	Low

Table 7.1: RCC vs. RSC Bundle Sizes

7.5 Performance Implications

Small bundle size has significant real-world advantages:

- Smoother first page loads with reduced JS payload
- Better performance on low-end hardware (especially mobile)
- Improved Core Web Vitals scores, including:
 - First Contentful Paint (FCP)
 - Time to Interactive (TTI)
- Reduced memory consumption and JavaScript execution time
- Better SEO, since pages are pre-rendered and not dependent on client hydration

React Server Components provide a more scalable option by pushing complex logic and rendering to the server. This yields more optimized frontend performance with complete interactivity.

7.6 Overview

Bundle analysis categorically demonstrates that React Server Components (RSC) have a drastically lower client-side JavaScript footprint than standard React Client Components (RCC). With only ~89 KB of client-side JavaScript, the RSC application is revealed to be more scalable, with less client-side overhead, and initial load as well as runtime execution performance improvement.

This review verifies that React Server Components are a productive way ahead for the development of high-performance, SEO-optimized web applications in the current JavaScript universe.

8. Developer Experience

From the perspective of a developer, the difference between using React Client Components (RCC) and React Server Components (RSC) is apparent immediately. While both approaches utilize React, they present very distinct development patterns when it comes to complexity, performance, and overall experience.

With RCC architecture, mounting the whole stack is accomplished via a separate backend service. In our project, the RCC app was dependent on a REST API implemented using TypeScript and SQLite, where routes, controllers, CORS policies, and API validation were created. With every change to the data layer, developers would need to update the frontend logic (with Axios or SWR), manage loading states, and maintain API contracts manually. This process introduces overhead and adds more code and configuration needed.

Compared to that, the RSC variant streamlines the whole thing by allowing developers to directly interact with the database from the React components using Prisma. No setup or maintenance of a backend API is required whatsoever. This obviates the necessity to:

- Write and maintain backend routes or controllers
- Configure and debug CORS policies
- Create REST endpoints or handle API versioning
- Set up separate validation logic on both ends

Boilerplate code is significantly reduced, and data model or UI behavior changes can be implemented more rapidly with fewer pieces of code.

The second difference is how RSC encourages a new way of thinking about the app architecture: developers begin thinking in terms of what logic goes on the server and what can be performed on the client. For this, they have a more thoughtful and performance-aware development process. For example:

- Static or database-rendered content may stay on the server
- Only fully interactive items (like buttons or forms) must be client-rendered

This separation not only helps performance but also makes the app easier to understand and maintain.

8.1 Overview

The RSC development workflow is less heavyweight, quicker, and more seamless. It eliminates most of the usual web development pain points — like API upkeep and duplicated code — and offers a repetitive, full-stack experience that's pleasant to use. It allows one to develop rapidly without losing structure or performance.

9. Conclusion

This thesis explored and contrasted two methods of building React applications: the traditional Client Component approach and the new React Server Components (RSC) approach. The goal was to compare their performance, efficiency, and differences in developer experience through hands-on experience and comparison.

The findings registered definitive benefits to the use of RSC. In performance, the RSC version was better than the RCC version in almost all major performance metrics. It had a perfect Lighthouse performance score of 100, a quicker First Contentful Paint (0.2 seconds vs. 0.4), better Largest Contentful Paint (0.5 seconds vs. 0.8), and a lower Cumulative Layout Shift of 0.000 vs. 0.041. All of these are due to the smaller client-side bundle and less JavaScript execution — the RSC bundle was about half the size of the RCC one.

In addition to technical feature, the developer experience with RSC was also significantly better. It does away with the need for having to implement and manage a different backend, and extra routes code, APIs, validation, and CORS code. With RSC and other technology like Prisma, developers can now query the database directly in server components, which results in faster development time and less boilerplates. The bundled architecture gets the tech stack together and reduces context switching, making it straightforward for a single developer to work on both frontend and backend using only JavaScript or TypeScript.

This also has new possibilities in team organization and workflow. By combining tasks in a single project, RSC eliminates frequent need for communication between disparate frontend and backend teams. Delivery is quicker and roles are more adaptable — allowing more developers to be full-stack engineers without having to learn several frameworks or languages. RSC certainly has trade-offs, however. A significant factor of consideration is supporting several clients. Where the identical backend must support multiple kinds of clients — for example, a mobile app and a web app — a traditional API-based backend might still be ideal, with increased separation and sharing between platforms. Further, although RSC does simplify some things about backend development, it still involves working with server-side code, databases, and deployment customs. In short, Server Components in React offer an incredibly robust and cutting-edge method of building full-stack React applications. They improve performance, simplify development, and encourage a fresh way of thinking about rendering and data loading. As this project has shown, taking advantage of this architecture can lead to cleaner codebases, better user experiences, and faster delivery — making it a worthwhile web development trend of the future.

References

1. **React.** Meta. (n.d.). React – A JavaScript library for building user interfaces.
<https://react.dev/>
2. **Next.js.** Vercel. (n.d.). Next.js documentation.
<https://nextjs.org/docs>
3. **Prisma.** Prisma Data, Inc. (n.d.). Prisma – Next-generation ORM for Node.js.
<https://www.prisma.io/docs>
4. **Vite.** Evan You. (n.d.). Vite – Next Generation Frontend Tooling.
<https://vitejs.dev/>
5. **SWR.** Vercel. (n.d.). SWR: React Hooks for Data Fetching.
<https://swr.vercel.app/>
6. **Shadcn UI.** (n.d.). Shadcn UI – UI Components built with Radix UI and Tailwind CSS.
<https://ui.shadcn.com/>
7. **SQLite.** SQLite Consortium. (n.d.). SQLite Documentation.
<https://www.sqlite.org/docs.html>
8. **TypeScript.** Microsoft. (n.d.). TypeScript – JavaScript With Syntax for Types.
<https://www.typescriptlang.org/>
9. **Axios.** Axios Project. (n.d.). Axios – Promise Based HTTP Client
<https://axios-http.com/>
10. **Tailwind CSS.** Tailwind Labs. (n.d.). Tailwind CSS – A utility-first CSS framework.
<https://tailwindcss.com/>
11. **Google Lighthouse.** Google. (n.d.). Lighthouse – Performance Metrics and Best Practices for Web Applications.
<https://developer.chrome.com/docs/lighthouse/>
12. **REST API.** Mozilla Developer Network (MDN). (n.d.). REST APIs – Introduction.
<https://developer.mozilla.org/en-US/docs/Glossary/REST>

List of Objects

List of Figures

Figure 2.1 Server and Client Environments	7
Figure 4.1 Database structure	12
Figure 5.1 UI for displaying upcoming events	16
Figure 5.2 UI for adding new events	17
Figure 6.1 Lighthouse report for RCC	18
Figure 6.2 Lighthouse report for RSC	19

List of Tables

Table 6.1 RCC and RSC Performance Metrics	20
Table 7.1: RCC vs. RSC Bundle Sizes	25

List of Code Listings

Code Listing 4.1: Data Fetching in RCC	13
Code Listing 4.2: Data Fetching in RSC	14
Code Listing 4.3: Server Actions	15

Appendix A — Project Repository and Setup Instructions

The full source code of this thesis project can be found at GitHub in the below link:

GitHub Repository: <https://github.com/murad-sh/rsc-vs-rcc>

The repository contains three main folders:

- /back-end/ — Traditional backend for the RCC app
- /rcc/ — React Client Components application (Vite-based)
- /rsc/ — React Server Components application (Next.js-based)

Running the Applications Locally

1. Install Dependencies :

Make sure that you have Node.js (18 or higher) and npm installed on your machine. Then, open a terminal, navigate to the directory of the project you want to run, and install the dependencies using the command `npm install`.

2. Start the Application :

Lastly, once you have the dependencies installed, you can run each app using the following

- For the RSC (Server Components) application as well as the RCC (Client Components) application, run the project after installing dependencies via **npm run dev**.
- For the back-end service, start the server by running **npm start**.