Tampere University

Tellervo Kettunen

REACT SERVER COMPONENTS RENDERING PATTERNS IN NEXT.JS

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
June 2024

ABSTRACT

Tellervo Kettunen: React Server Components Rendering Patterns in Next.js
M.Sc. Thesis
Tampere University
Master’s Degree Programme in Computer Science
June 2024

This thesis focuses on rendering patterns, React Server Components (RSC), and how React Server Components are rendered in Next.js framework. The literature parts introduce main concepts of web application architecture and the rendering patterns Static Site Generation (SSG), Client-Side Rendering (CSR), Server-Side Rendering (SSR), Incremental Static Regeneration (ISR), and Streaming Server-Side Rendering (Streaming SSR) are described.

React version 18 introduced new experimental way of building applications with Server Components that run only on server, with the traditional now named as Client Components that can run on the client. In 2023 Next.js with App Router was the first to implement production-ready version of the React Server Components architecture.

Static Rendering, Dynamic Rendering, and Streaming are server rendering options in Next.js when using React Server Components. Next.js Static Rendering is similar to SSG or ISR, Dynamic Rendering is similar to SSR, Streaming is similar to Streaming SSR. The main difference is that in rendering phase also React Server Component Payload is created. RSC Payload contains rendered Server Components, and props that need to be passed later to the Client Components, and placeholders for Client Components with references to the JavaScript files. The patterns mentioned in the study with Next.js also include RSC Payload creation, even though the pattern might be only referred as SSG, ISR, SSR, or Streaming SSR.

For the case study part, a web application demo was developed using Next.js and React Server Components. Production versions, one with a static home route that uses SSG, and other with a dynamic home route that uses combination of SSR and Streaming SSR, were then tested with Lighthouse performance tests in desktop and mobile emulations. Lighthouse performance tests were chosen to gather information about how rendering performance differ between the static and dynamic route versions.

In performance tests where the initial page load is tested, static route version rendered slightly faster the first content. In overall the dynamic route version performed better in the tests, by executing tasks faster, populating page faster, and painting largest content to screen faster. Test results show that the combination of SSR and Streaming SSR in the dynamic route version produced slightly better performance than the used SSG in the static route version, but the performance differences were quite minimal.

Key words and terms: React, React Server Components, Rendering patterns, Next.js

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Tellervo Kettunen: React palvelinkomponenttien renderöintimallit Next.js:ssä
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Kesäkuu 2024

Tämä opinnäytetyö keskittyy renderöintimalleihin, React palvelinkomponentteihin (RSC) ja siihen kuinka React palvelinkomponentit renderöidään Next.js-ohjelmistokehyksessä. Kirjallisuusosissa esitellään verkkosovellusarkkitehtuurin käsitteitä ja renderöintimallit staattisten sivujen generointi (SSG), asiakaspuolen renderöinti (CSR), palvelinpuolen renderöinti (SSR), asteittainen staattisten sivujen generointi (ISR) ja suoratoistettu palvelinpuolen renderöinti (Streaming SSR).

Reactin versio 18 esitteli uuden kokeellisen tavan rakentaa sovelluksia palvelinkomponenteilla, joita ajetaan vain palvelimella, perinteisten asiakaspuolella ajettavien asiakaskomponenttien kanssa. Vuonna 2023 Next.js-ohjelmistokehys otti ensimmäisenä käyttöön tuotantovalmiin version React palvelinkomponentti -arkkitehtuurista.

Staattinen renderöinti, dynaaminen renderöinti ja suoratoisto ovat palvelinpuolen renderöintivaihtoehdot Next.js:ssä käytettäessä React palvelinkomponentteja. Next.js:n staattinen renderöinti on samanlainen kuin SSG tai ISR, dynaaminen renderöinti on samanlainen kuin SSR, suoratoisto on samanlainen kuin Streaming SSR. Suurin ero on, että renderöintivaiheessa luodaan myös RSC-hyötykuorma, joka sisältää renderöidyt palvelinkomponentit, ja myöhemmin asiakaskomponenteille siirrettävät tiedot, sekä asiakaskomponenttien paikkamerkit viittauksilla JavaScript-tiedostoihin. Opinnäytetyössä mainitut mallit Next.js:n yhteydessä sisältävät myös RSC-hyötykuorman luomisen, vaikka mallia kutsuttaisiin nimellä SSG, ISR, SSR tai Streaming SSR.

Tapaustutkimusosuutta varten kehitettiin verkkosovellus käyttämällä Next.js-ohjelmistokehystä ja React palvelinkomponentteja. Sovelluksesta toteutettiin tuotantoversiot, joista toisessa on staattinen kotireitti, joka käyttää SSG:tä, ja toinen versio dynaamisella kotireitillä, joka käyttää SSR:n ja Streaming SSR:n yhdistelmää. Versioita testattiin Lighthouse-suorituskykytesteillä työpöytä- ja mobiiliympäristöjä emuloiden. Suorituskykytesteillä kerättiin tietoa siitä, kuinka renderöinnin suorituskyky eroaa staattisen ja dynaamisen reittiversion välillä.

Sivun alkulatauksen suorituskykytesteissä, staattinen reittiversio hahmotti ensimmäisen sisällön hieman nopeammin. Muuten dynaaminen reittiversio suoriutui testeissä paremmin suorittamalla tehtäviä nopeammin, täyttämällä sivua nopeammin ja maalaamalla suurimman sisällön näytölle nopeammin. Testituloksissa SSR:n ja Streaming SSR:n yhdistelmä dynaamisessa reittiversiossa tuotti hieman paremman suorituskyvyn kuin staattisen reittiversion käyttämä SSG, mutta erot olivat melko vähäisiä.

Avainsanat: React, React Server Components, renderöintimallit, Next.js

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin Originality Check -ohjelmalla.

Contents

1 Introduction 1

1.1 Background 1

1.2 Research questions and methodologies 1

1.3 Thesis structure 3

2 Web Application Architecture 4

2.1 Client-server communication 4

2.2 HTML 5

2.3 CSS 5

2.4 DOM 5

2.5 Rendering in a browser 6

2.6 JavaScript 8

2.7 Libraries and frameworks 9

3 Rendering patterns 10

3.1 Static Site Generation 10

3.2 Client-Side Rendering 11

3.3 Server-Side Rendering 12

3.4 Hydration 13

3.5 Incremental Static Regeneration 13

3.6 Streaming Server-Side Rendering 14

4 React Server Components 16

4.1 React 16

4.2 React Server Components development 17

4.3 Server Components 17

4.4 Client Components 19

4.5 Rendering React Server Components 20

5 Next.js Framework 24

5.1 Next.js 24

5.2 Server Components rendering in Next.js 24

6 Case study 29

6.1 Web application prototype 29

6.2 Streaming SSR example 30
6.3 Experiment 31
6.4 Results and analysis 33

7 Conclusion 37

References 40

LIST OF ABBREVIATIONS

API	Application Programming Interface
CLS	Cumulative Layout Shift
CSR	Client-Side Rendering
CSS	Cascading Style Sheets
CSSOM	CSS Object Model
DOM	Document Object Model
FCP	First Contentful Paint
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
ISR	Incremental Static Regeneration
JS	JavaScript
LCP	Largest Contentful Paint
RSC	React Server Components
RSC Payload	React Server Components Payload
SEO	Search Engine Optimization
SI	Speed Index
SSG	Static Site Generation
SSR	Server-Side Rendering
Streaming SSR	Streaming Server-Side Rendering
TBT	Total Blocking Time
TS	TypeScript
TTI	Time to Interactive
UI	User Interface
VDOM	Virtual DOM

1 Introduction

1.1 Background

Rendering, in the context of web development, is the process of generating the content of a page and user interfaces. This is done by converting source code and data into a format that can then be used in a browser to show content. Web application architecture contains client, server, and database environments. The rendering process can happen either on the server or on the client. The final painting process happens in the browser. This study goes through some of the rendering patterns that are used with React and React Server Components, and how React Server Components are rendered in Next.js.

When building web applications, source code is turned into HTML, CSS, and JavaScript, which are the building blocks that can be used in browsers. HTML is the structure, CSS brings the style, and JavaScript adds interactivity; browsers use those to build object models which are then combined into a render tree with all the information needed to paint a page to a screen. Web application architecture chapter will describe how browsers handle rendering process after receiving HTML, CSS, and JavaScript. The rest of the study focuses more on the rendering process happening before browser rendering.

Previously, when React was used to build user interfaces for web applications, all React components were client-side components, and the component tree would only consist of client-side components. Meaning that all JavaScript for the React components needs to be downloaded to the client when a page is requested. When using React Server Components, the JavaScript used in Server Components stays on a server and is only rendered on a server [React RFCs 2024a]. This reduces the JavaScript bundle size because only JavaScript in Client Components needs to be shipped to the client. A new trend of shipping zero or minimal amount of JavaScript to the client has been noticed in web development [Vepsäläinen et al. 2023b]. React Server Components are a feature introduced more broadly first through the Next.js framework in version 13 [React 2024a, Next.js 2022]. In May 2023 it was announced that production ready version is in Next.js version 13.4 [Next.js 2023].

1.2 Research questions and methodologies

This study aims to attain an understanding of some of the rendering patterns used with React and React Server Components architecture. When starting the thesis process, the initial research question was: How are React Server Components rendered in Next.js.

During the study, research performed a comparison of two rendering options by conducting a performance audit. In this study, the research questions are as followed:

- RQ1: How to use React Server Components in Next.js?
- RQ2: How are React Server Components rendered in Next.js?
- RQ3: How does a static route with Static Site Generation rendering pattern compare to a dynamic route with a combination of Server-Side Rendering and Streaming Server-Side Rendering patterns? Comparison is done by evaluating performance metric results.

Thesis process started with identifying research area, which followed the process of studying the literature of the subject matters. After the main concepts of React Server Components, rendering patterns, and Next.js frameworks were studied, a web application development started. Performance and analysis parts of the research were conducted with two implementations with different rendering patterns.

To support the literature, a research method of comparative studies with a case study method was chosen. Comparative study is a research method used to analyse differences and/or similarities of the tested phenomenon [Coccia & Benati 2022]. One case represents one instance of a phenomenon, and in a case study, these cases or their sub-cases are researched and compared [Dumez 2015].

A simple demo web application, that uses Next.js and React Server Components, is developed during the research. Case study is conducted with the web application's two production versions, one with a static home route and the other with a dynamic home route, that have different rendering patterns. These two case study objects, with the used rendering patterns, are illustrated in Figure 1. Rendering pattern used in the static route is Static Site Generation (SSG). Rendering patterns used in the dynamic route are Server-Side Rendering (SSR) and Streaming Server-Side Rendering (Streaming SSR).

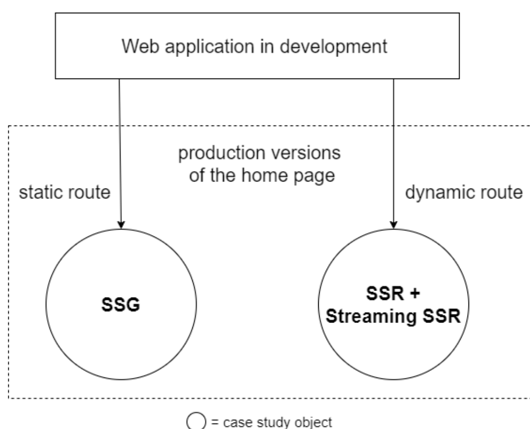


Figure 1. Case study objects.

The case study object's rendering patterns are compared by evaluating performance metric results that are gathered with Lighthouse performance tests. The comparison gives information about how the two rendering options differ from each other. The initial load of the home page is analysed with Lighthouse tests. Lighthouse performance tests were chosen to gather information about how static route with SSG rendering pattern, and dynamic route with a combination of SSR and Streaming SSR rendering patterns differ.

1.3 Thesis structure

Chapter 2 gives background information about web application architecture. Chapter 3 introduces Hydration process and the following rendering patterns: Static Site Generation (SSG), Client-Side Rendering (CSR), Server-Side Rendering (SSR), Incremental Static Regeneration (ISR), Streaming Server-Side Rendering (Streaming SSR). Chapter 4 is about React Server Components, the architecture, how the new Server Component differs from the Client Component, and how a combined component tree is rendered. Chapter 5 provides information about the Next.js framework and the rendering process in Next.js. Chapter 6 shows the created demo web application, describes how the research case study was conducted, and showcases performance evaluation results and analysis. Finally, Chapter 7 contains a conclusion.

2 Web Application Architecture

This chapter gives background information about web application architecture and introduces the steps of the process of rendering in a browser. In web development, the DOM-based strategy for rendering in a browser is the most used [Taivalsaari et al. 2019]. On the rendering path in a browser, DOM is built with HTML, CSSOM is built with CSS, a render tree is built by combining DOM and CSSOM, and finally web page can be painted on screen using render tree and layout [Ollila et al. 2022].

2.1 Client-server communication

Web application architecture contains client and server layers, and often also a database layer is added. Client and server are two different roles that software or device can have. In modern web applications, the client is a web browser, and the server is the place where a web application is hosted. The client-server architecture defines the structure of how different tasks are divided, often client-side (frontend) code handles application presentation and user interface, and server-side (backend) code is responsible for data management and logic of services [Sommerville 2007]. The client requests to use the application's services and the server serves the resources to the client.

Communication between client and server is often done using HyperText Transfer Protocol (HTTP) with a request-response model over the internet. When a user interacts with a web application for example inputs text to a search field, the client side sends an HTTP request with the input text to the server. The server responds to the request with a response that contains among other things a status code and search result data if it was found. Figure 2 illustrates a basic HTTP request-response communication model in a client-server architecture. When a user requests a page of a web application, the server sends HTTP responses that contain HTML, CSS, and JavaScript for the page.

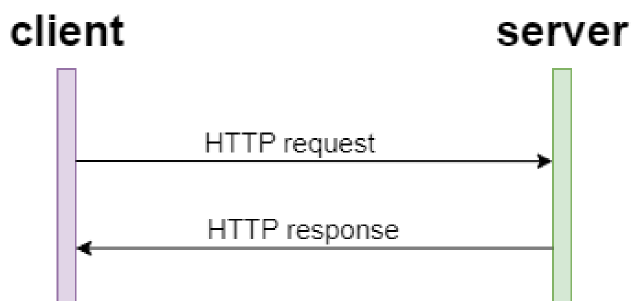


Figure 2. HTTP request-response communication model in a client-server architecture.

2.2 HTML

HTML is an abbreviation for HyperText Markup Language, which was developed in the early 1990's. With HTML markup documents can be structured in a standardised way which web browsers can interpret. The structure of an HTML document is a hierarchical tree, that is built with elements and text. [HTML 2024]

Listing 1 contains a basic HTML document example, where the root element html contains head and body elements. Elements inside angle brackets are called tags, so the start of a body element is indicated with a starting tag <body> and the end of the body element with an ending tag </body>. The body element contains what will be shown in a web browser from this page.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Example Document</title>
  </head>
  <body>
      <h1>Page header</h1>
      <p>Paragraph text</p>
  </body>
</html>
```
Listing 1. Example of an HTML document.

2.3 CSS

HTML documents can look quite plain in a browser without styling. HTML documents can be styled with CSS which stands for Cascading Style Sheets. CSS language provides standardised ways to add style and structure that web browsers can understand when a web page or web application is rendered on the browser [MDN Web Docs 2024a].

There are many different libraries that provide styled components or utility classes that help with styling. These CSS frameworks can provide templates and pre-build styling classes that are quick to use in development. Eventually, in the build process, all styling is turned into CSS for the browsers.

2.4 DOM

Document Object Model (DOM) is an application programming interface (API) that is provided by a web browser. DOM is one of the Web APIs that is used when developing for the web. HTML documents are represented with DOM's tree-like data structure, that

enables browsers to internally have representation of a web page. DOM is generated by parsing HTML documents; in that process, each HTML element gets a corresponding node in the tree of objects. After the DOM is built, JavaScript can access and update the DOM tree. [DOM 2024]

Figure 3 contains an example of a tree data structure of the DOM tree, that represents the HTML document content from Listing 1. The tree has a root, that is html node, which has two children, head and body nodes. The body node contains the visible part of the page. Head node has title node as its child node. The title node has one child node, a text node that contains text: Example Document. The body node has header (h1) and paragraph (p) nodes as children. Header and paragraph nodes have text nodes as their child nodes.

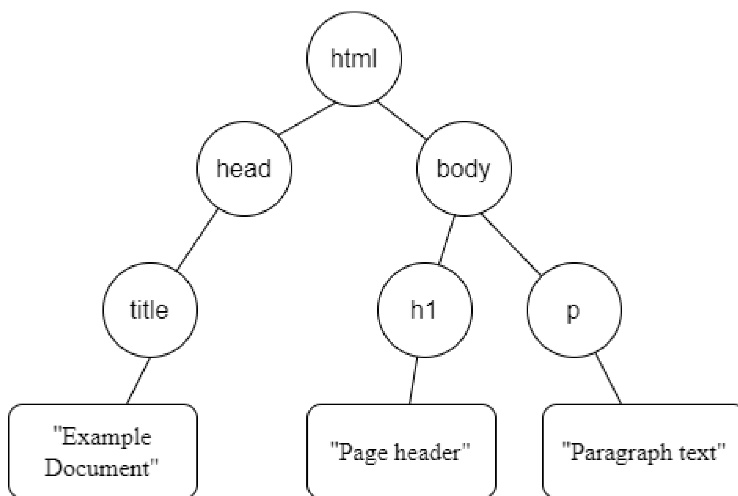


Figure 3. DOM tree example.

2.5 Rendering in a browser

When a user requests to open a page, browser's rendering process can start after browser commits to navigation [Google Developers 2024a]. At this stage, browser has received initial data from the server and can start parsing the data. HTML files are turned into DOM and CSS files into CSSOM (CSS Object Model) [Ollila et al. 2022]. If CSS styling is not added to application, browser will use default CSS [Google Developers 2024b].

Example CSSOM, shown in Figure 4, is built to be used with the DOM tree in Figure 3. CSSOM example uses Google Chrome default CSS. The example CSSOM tree contains html, body, header (h1), and paragraph (p) nodes with their corresponding style rules.

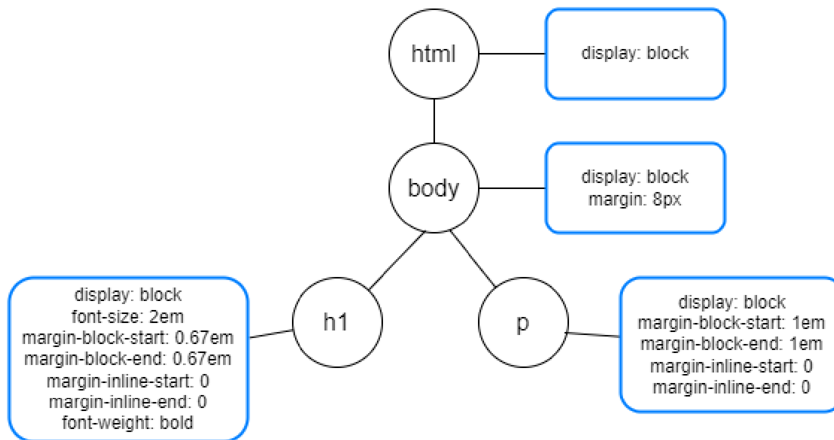


Figure 4. CSSOM example.

After building DOM and CSSOM trees, browser will combine those trees to create a render tree [Ollila et al. 2022]. The render tree will only contain nodes that are relevant for the rendering. For example, elements that have CSS properties with display none, are not added to the render tree. After style and layout steps, the render tree is painted on the browser's screen. When parsed JavaScript is added and executed, the page is interactive for the user. [MDN Web Docs 2024c, Google Developers 2024b]

Developers cannot directly control the painting process, meaning the final rendering of a page to a browser's screen. All needed alterations have to be made indirectly, by modifying HTML, CSS, or by using JavaScript. [Taivalsaari et al. 2019]

A render tree example is shown in Figure 5. It is formed by combining the visible parts of the DOM tree in Figure 3 with the CSSOM in Figure 4. All visible nodes have their corresponding style rules applied to them. With a render tree and layout, a page can be painted on a browser's screen [Ollila et al. 2022].

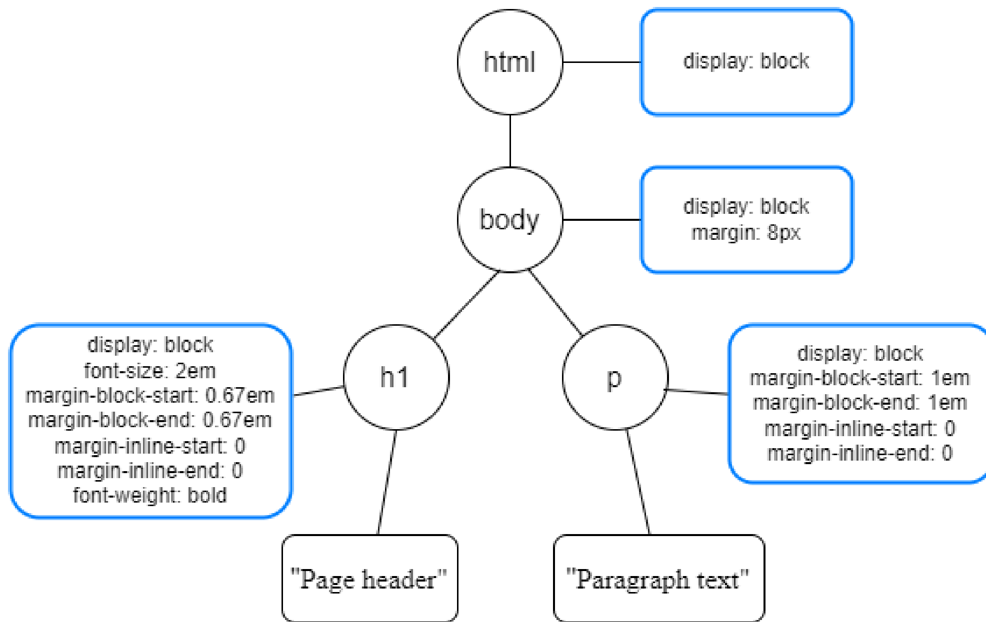


Figure 5. Render tree example.

2.6 JavaScript

JavaScript (JS) is a dynamic programming language that is used, among other things, to make static HTML elements interactive. JavaScript is single-threaded and supports both imperative and declarative programming styles. [MDN Web Docs 2024b]. After a user has requested a page of a web application, and the browser has created a DOM presentation of it, JavaScript can add interactivity to the page with hydration. Hydration process is introduced in Chapter 3.

JavaScript was created in 1995, and in 1997 ECMA standardised JavaScript [Thakkar 2020]. ECMA (European Computer Manufacturers Association) changed its name in 1994 to Ecma International [2024]. ECMA-262 [2024] standard, better known as ECMAScript (ES), defines language specifications for scripting languages like JavaScript. Standards first version release ECMAScript 1 (ES 1) was in 1997, and it based specifications on JavaScript language, the latest version was released in 2023 which was the 14th version [Web Reference 2024].

Dynamic typing in JavaScript means that type checking is done at runtime. This can lead to errors being found when users try to use web application on browser. TypeScript (TS) was developed to add stronger types to JavaScript code. TypeScript compiler checks possible errors during compilation. If TypeScript code is compiled without errors, it then behaves like regular JavaScript code at runtime [TypeScript 2024].

2.7 Libraries and frameworks

React [2024a], also known as React.js, is a library used to build user interfaces for web applications. React is often referred to as a framework, even though it is a library. Other JavaScript frameworks include for example Vue [2024], Angular [2024], Svelte [2024], Solid [2024], and Qwik [2024]. React, Vue, and Angular have been in use for about ten years, the more recent Svelte was released in 2016, Solid was released in 2020, and Qwik was released in 2022.

The yearly State of JavaScript survey had almost 40 thousand responders in 2022 from developers all over the world. According to the results of the 2022 survey, React is still the most used front-end JavaScript framework. React was used by 82%, Angular 49%, and Vue 46% of the responders. The interest of these three has declined over the years, in this survey Vue had interest of 51%, React 47%, and Angular 20%. Survey responders were most interested about the newer technologies Svelte with 70%, Qwik with 67%, and Solid with 66%. The usage percentage of Svelte was only 21%, Solid with 6%, and Qwik with 2%. [State of JS 2024]

Frameworks that are based on other frameworks can be called as meta-frameworks. React based meta-frameworks are Next.js [2024b], Remix [2024], and Gatsby [2024]. Nuxt [2024] is based on Vue, and SvelteKit [2024] is based on Svelte. Astro [2024] supports many frameworks for example React, Svelte, Solid, and Vue.

According to the results of the State of JavaScript 2022 survey, Next.js is still the most used rendering framework as it has been for the last five years. Next.js was used by 49%, Gatsby by 23%, Nuxt by 18%, and SvelteKit by 12% of the responders. When asked about the rendering frameworks the survey responders were most interested about Astro with 67%, SvelteKit with 66%, Next.js with 65%, and Remix with 57%. [State of JS 2024]

3 Rendering patterns

Developers cannot directly control the painting process in a browser [Taivalsaari et al. 2019]. But often developers can choose from multiple rendering pattern options, which one is the most suitable for the current project. Rendering patterns describe the process of how the needed creation and delivery of HTML and other content is handled to user's browser.

In this chapter, hydration process and five rendering patterns are introduced. The selected rendering patterns can be used with React. Section 3.1. is about Static Site Generation (SSG), Section 3.2. goes through Client-Side Rendering (CSR), Section 3.3. introduces Server-Side Rendering (SSR), Section 3.4. is about Hydration, Section 3.5. about Incremental Static Regeneration (ISR), Section 3.6. goes through Streaming Server-Side Rendering (Streaming SSR).

Although rendering patterns are introduced here as separate, there are also hybrid rendering strategies that can combine parts of static, server-side, and client-side rendering.

3.1 Static Site Generation

In static rendering or Static Site Generation (SSG) process all static HTML files are generated at build time and can be hosted using for example a CDN (Content Delivery Network) [Vepsäläinen et al. 2023a, Osmani 2023]. From CDN, files can be served directly to a client without server-side processing when a user requests a web page. Static generation makes serving pages faster than rendering pages with every request with Server-side Rendering (SSR) [Osmani 2023].

When using static rendering all content including data needs to be known at build time, meaning when the production build is made. If data needs to be updated, then the whole application needs to be rebuilt to create pre-rendered pages. When an application contains lots of frequently changing data or data that is user-specific, then server-side rendering could be a more suitable pattern because all pages are rendered on every request.

Figure 6 illustrates Static Site Generation process. During the build time process, all data is queried from a database and then all static HTML files are pre-rendered with data. Now all site files are ready to be served from the server. When a request for page content comes from a client, the server can immediately respond with readymade static HTML files. On client pre-rendered HTML is shown. At this point page is viewable and interactable without dynamic content.

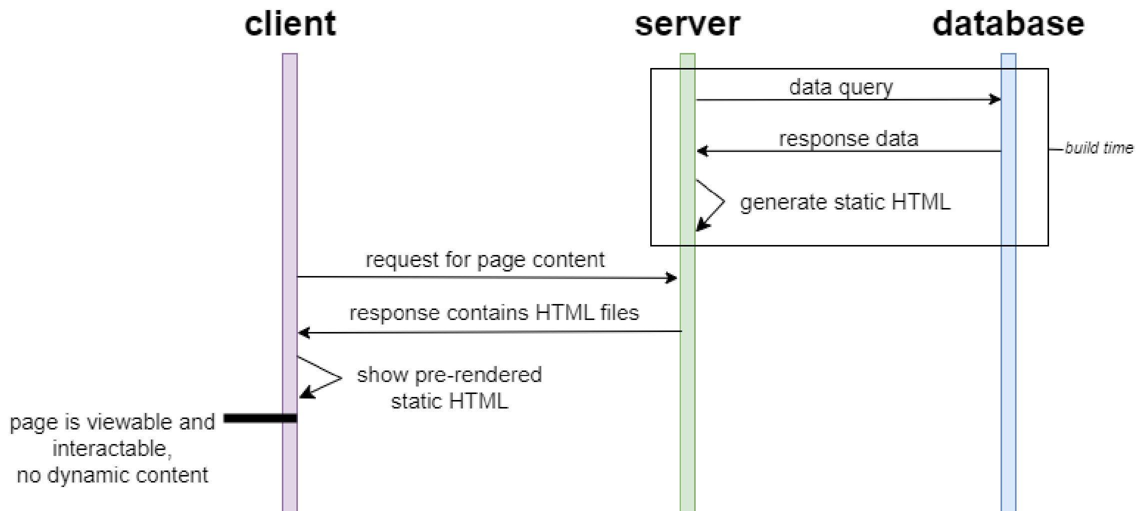


Figure 6. Static site generation process.

3.2 Client-Side Rendering

When using Client-Side Rendering (CSR) strategy, all rendering is done in a client, meaning that browser handles all rendering at runtime. When a user requests a page that uses CSR, the server will send minimal details in an HTML file, the client will then load and execute needed JavaScript and then render content on a page dynamically using JavaScript. [Osmani 2023]

After the first time-consuming rendering, client-side rendering provides fast page transitions, because no need to request new page content from a server. When using CSR with web applications that contain pages that need to request new data from a server or a database, after the client receives the requested data, the client needs to only render the new data on the client, not the whole page. Single Page Application (SPA) is a web application architecture that uses client-side rendering strategy.

Figure 7 illustrates the Client-Side Rendering process. When a request for page content comes from the client, the server responds with minimal HTML, that contains only a blank page and with information for the client to request needed JavaScript from the server. The client then downloads JavaScript from the server. The client starts parsing and executing JavaScript. If JavaScript code includes data fetching, data is requested from the database. After all initial data is inserted into the code and the rendering process is finished on the client, the page is viewable and interactable. After this initial rendering, the client contains all the code needed to re-render the application and request data from the database.

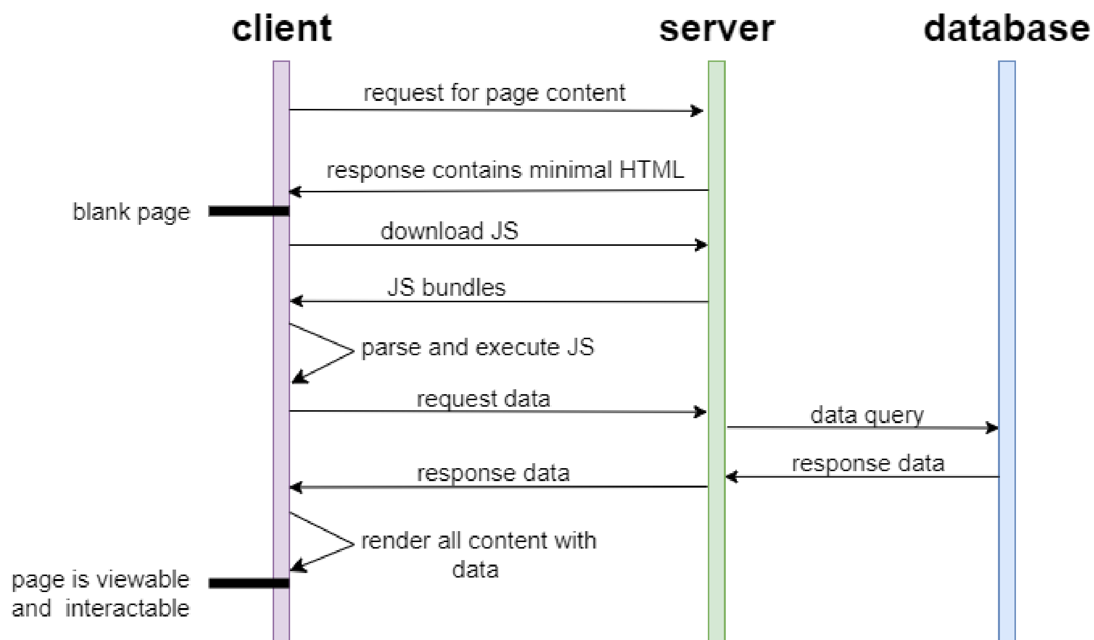


Figure 7. Client-side rendering process.

3.3 Server-Side Rendering

Server-Side Rendering (SSR) is a rendering strategy where all page content is rendered on a server and full HTML is sent to a client after the client has sent a request. In SSR also data fetching operations are done on the server and data is inserted to HTML before it is sent to the client. [Osmani 2023]

With SSR first load of the web application will be faster than with CSR, because with SSR server produces fully loaded HTML, and no need for extra data fetching from the client. This is also beneficial for search engine optimization (SEO) because search engines can examine the initial fully rendered HTML and index the content, which can lead to better findability in searches. [Osmani & Miller 2024]

Figure 8 illustrates the Server-side rendering process and how hydration is done after SSR. When a request for page content comes from a client, the server queries initial data from the database, and then static HTML files are pre-rendered with initial data. The server can then send a response that contains pre-rendered HTML and JS bundles. On the client, the pre-rendered HTML is shown and now the page is viewable to the user, but the page is not yet interactable. A hydration process needs to happen after this so that the page is interactable. The hydration process is the part where the client parses and executes JavaScript and attaches needed event handlers to the HTML. After the hydration process is ended the page is interactable.

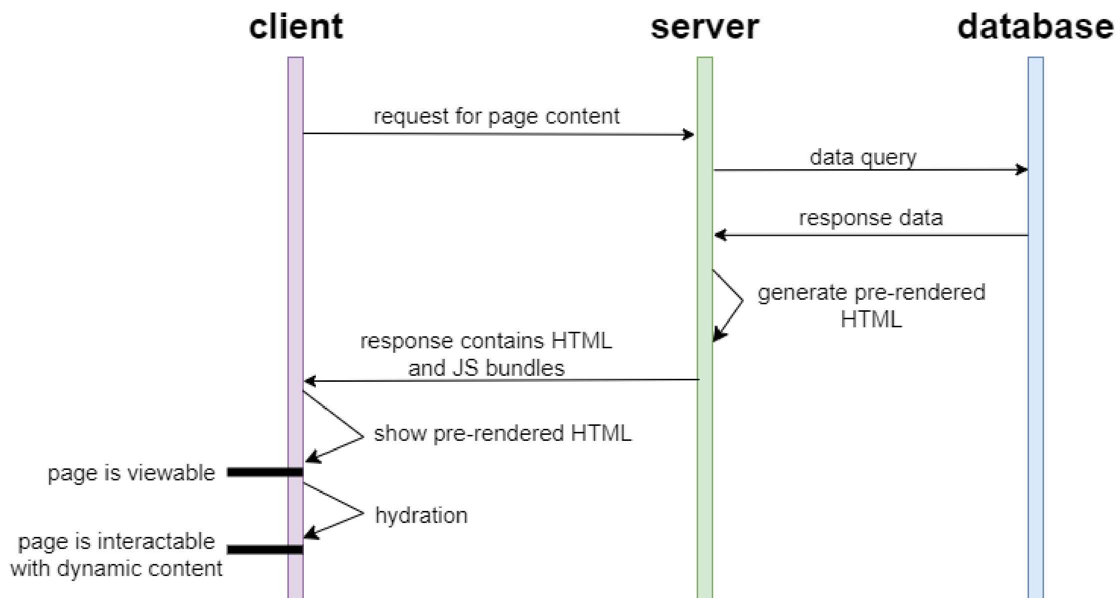


Figure 8. Server-side rendering and hydration process.

3.4 Hydration

Web application needs to be hydrated before users can interact with it. Server-side rendered, or static rendered HTML pages that use JS for interaction need hydration before a user can interact with the page. Hydration is a process, where the client (browser) loads JavaScript and attaches all needed event handlers. [Osmani 2023] The client can have a page visible on a screen, but dynamic content needs to be hydrated before user interaction. For users this can be confusing, to see the page but buttons, etc. are not clickable.

With SSR and SSG rendering strategies the First Contentful Paint (FCP), the time between the initial request from a browser to seeing the first content on screen, can be quicker than with CSR, but Time to Interactive (TTI) is not always quicker. [Patterns.dev 2024]

3.5 Incremental Static Regeneration

Incremental Static Regeneration (ISR) is a rendering pattern that combines dynamic data content with Static Site Generation (SSG). The downside of SSG is that it generates all content at build time, so data content can't be updated without rebuilding the whole application. With ISR data can be updated with re-rendering pages when revalidation is needed. Initially application is pre-rendered at build time, so users are served a stale version of the site, but if data is changed revalidation happens and causes the re-rendering of the page. [Patterns.dev 2024, Next.js 2024a]

The Incremental Static Regeneration process is visualised in Figure 9. All static HTML files are pre-rendered with data during the build time. These files can be cached and served when a user sends a request for page content. After the initial request, a revalidation is triggered in the background when previously determined time limit has been passed. Revalidation means that a page that contains data fetches new data, and new HTML files are pre-rendered and cached. The new HTML is sent to the client and the page is updated to correspond with the new data.

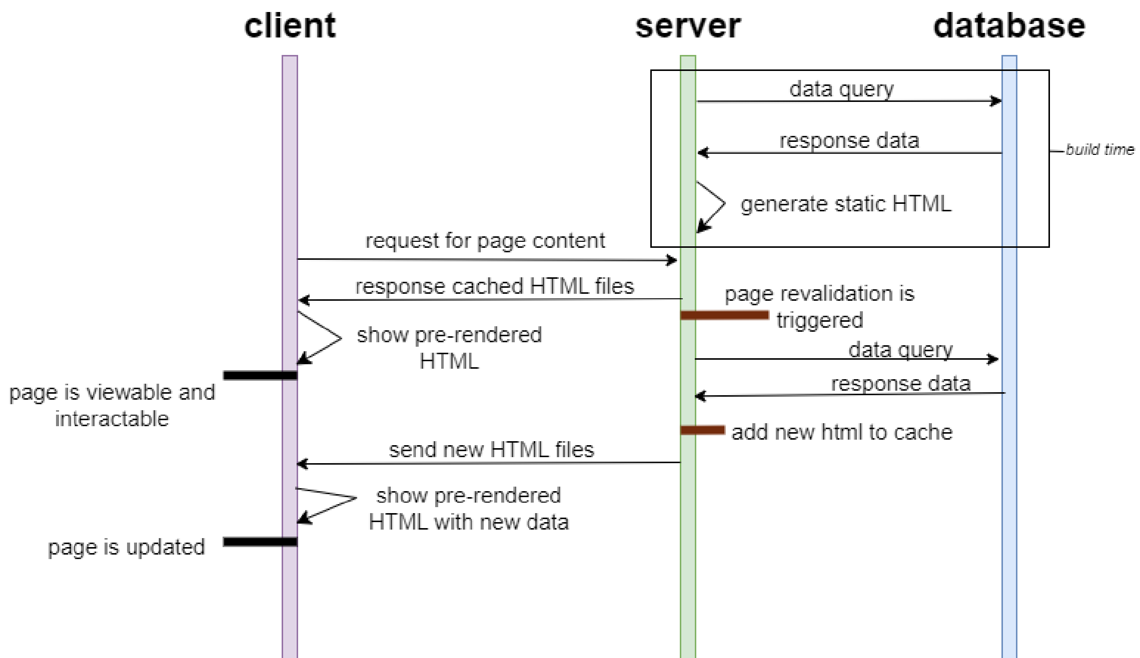


Figure 9. Incremental Static Regeneration process.

3.6 Streaming Server-Side Rendering

Streaming Server-Side Rendering (Streaming SSR) is a technique where a server can stream rendered parts in chunks to the client. When regular SSR is used, the server must wait that all needed parts are server rendered before sending a whole response to the client [Abramov 2021]. With Streaming SSR the client can start rendering and hydrating already the first chunks it gets, so this way user can see and interact with content earlier and Time to Interactivity (TTI) reduces compared to regular SSR [Patterns.dev 2024].

Figure 10 shows the Streaming Server-Side Rendering process with hydration. When a user sends a request for page content, first HTML chunks start rendering. Pre-rendered HTML chunks are then streamed to the client. On the client pre-rendered HTML is shown and that part is hydrated. Now the first part of the page is viewable and interactable. On the server, data is queried from the database, and HTML chunks are rendered and

streamed to the client. After everything is streamed to the client and pre-rendered HTML is shown on the client and hydrated, the page is fully viewable and interactable.

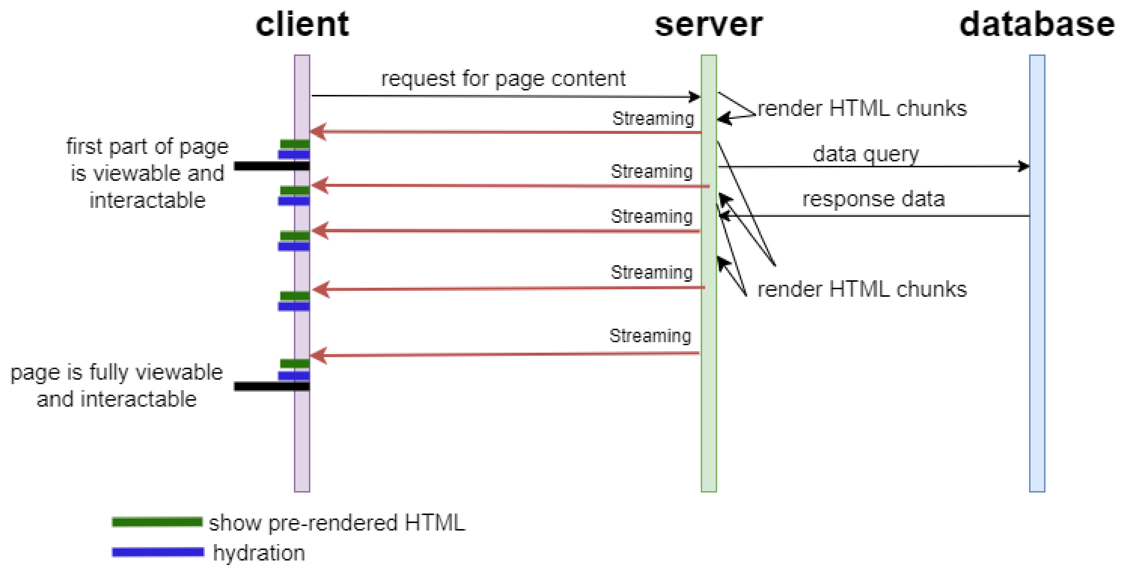


Figure 10. Streaming Server-Side Rendering and hydration process.

4 React Server Components

Chapter 4 is about React Server Components, the architecture, how the new Server Component differs from the Client Component, and how a combined component tree is rendered. Section 4.1. introduces React, Section 4.2. goes through the React Server Components development. Section 4.3. is about Server Components, Section 4.4. about Client Components. Section 4.5. goes through the rendering of the React Server Components and the combined component tree.

4.1 React

React is a library used to build user interfaces (UI) for web applications. Facebook company, now named Meta [2021], created React and open-sourced it in May 2013 [Thakkar 2020]. In 2017 React was relicensed with MIT Licence [Engineering at Meta 2017]. MIT Licence is a permissive software licence [The Open Source Initiative 2024].

Web application's user interfaces that are constructed with React are made with building blocks called components. React components are pieces of JavaScript code that declare how component behaves and what it looks like in UI. Components are exported as JavaScript functions that return markup, that contains the component's logic, content, and style in one place. Popular syntax for the markup is JSX (JavaScript syntax extension), which makes it possible to write markup that is similar to HTML, to a JavaScript file. [React 2024a]. React uses properties, often shortened as props, that are read-only objects to pass data from a parent component to a child component.

React uses Virtual DOM (VDOM) to keep a representation of the actual DOM in memory, and with it, React can update and render UI components more efficiently. React builds Virtual DOM by creating a component tree that represents the application and its state. When props or state change in UI, React will make a new Virtual DOM and compare it to the previous Virtual DOM, then it calculates the differences in the two tree instances. After React has calculated the most efficient way to update the actual DOM, React will sync the changes to the actual DOM in the browser. [React 2024b, Javatpoint 2024]. The use of Virtual Dom can increase rendering performance when compared with the original DOM [Taivalsaari et al. 2019].

The term rendering in React means the process of component tree rendering before it is given to browser's DOM. In this case browser rendering means the final painting process when nodes from DOM is painted to the browser screen. [React 2024a]

4.2 React Server Components development

React version 18, released in March 2022, brought new features that can add concurrency when those concurrent features are used in an application. Concurrent React means that rendering can be paused and continued later, and user interface stays consistent even though rendering may be paused. React can now create several versions of user interface elements at the same time when concurrent features are applied. In previous versions of React when rendering starts it continues synchronously until it finishes. [React 2022]

React 18 introduces concurrency for example in the Suspense component in a way that components allow streaming HTML and Selective Hydration [Abramov 2021]. Now users can interact with other components that are already loaded, even though some suspended components are still loading content.

React Server Components (RSC) are experimental feature introduced in React 18 [React release 2022]. Even though React Server Components are not tightly dependent on concurrency, React Server Components architecture benefits from features like Suspense component and streaming server-side rendering, that implement concurrency in their design [React 2022]. React Server Components architecture includes Server Components, which run only on the server, and Client Components which can run on the client [React 2023]. By using a directive ‘use client’ it states that the component is a Client Component meant to be rendered on the client [React RFCs 2024b].

Next.js version 13, beta release in October 2022, introduced new App Router in the app directory for doing routing and layout for web applications. The App Router added support for React v18 features like React Server Components. [Next.js 2022]. A stable release of the App Router was in May 2023, when App Router was announced to be production-ready [Next.js 2023].

React states that the most comprehensive implementation of React Server Components architecture is in Next.js App Router [React 2024a]. React and Next.js teams have worked together to build functionalities to the Next.js framework that leverage Reacts core components [Next.js 2023].

4.3 Server Components

In this new React Server Components feature, all components are Server Components by default, Client Components are denoted with directive. Server Components can run either at build time or on the server when requested. Data sources that are on the server side are accessible by the Server Components, these include for example database and filesystem.

When Server Components are run at build time they produce static content. When server rendered by user's request, Server Components can generate dynamic content by fetching fresh data from data sources. Server Components can give data also as props to Client Components. [React 2023]

When additional libraries or packages are added to an application in development, the code size grows. Server Component's code is not added to JavaScript bundle, which contains all the JavaScript code and dependencies to run a web application in the browser. [React RFCs 2024a]. If extra libraries and packages are only used to modify component before sending it to client, it's beneficial to use them in Server Components, to reduce JavaScript sent to client. To get the full benefits of Server Components, there needs to be used a bundler capable of managing React Server Components features [React 2023]. This is one reason why React Server Components are first introduced through Next.js framework.

Listing 2 contains a Server Component example, where Note component is a Server Component. Note component gets props, which contains an id of a note and the isEditing is a status of editing, that is Boolean value true or false. Unlike Client Components, Server Components can be asynchronous functions. And Server Components can have direct access to the database (DB). With the id value, the component can directly get the note that corresponds to that id from the database. [React RFCs 2024a]

Server Components can import Client Components. In this example, in Listing 2 the Note component is a parent component to NoteEditor, which is a Client Component imported by the Note component. Server Components can pass props to Client Components. The child component NoteEditor gets its props, a note object, from the parent component. NoteEditor is only rendered if the current Note's isEditing value is true. Conditionally rendering is done in this case with a ternary operator, only if isEditing value evaluates true the NoteEditor component is rendered.

```
// Note.js - Server Component

import db from 'db';
// (A1) We import from NoteEditor.js - a Client Component.
import NoteEditor from 'NoteEditor';

async function Note(props) {
  const {id, isEditing} = props;
  // (B) Can directly access server data sources during render, e.g. databases
  const note = await db.posts.get(id);

  return (
    <div>
      <h1>{note.title}</h1>
      <section>{note.body}</section>
      {/* (A2) Dynamically render the editor only if necessary */}
      {isEditing
        ? <NoteEditor note={note} />
        : null
      }
    </div>
  );
}
```

Listing 2. Server Component example [React RFCs 2024a].

Server Components make code splitting automatic, by viewing all Client Component imports as possible locations of code-splitting points [React RFCs 2024a]. Bundlers should detect code splitting automatically in these situations. Code splitting is optimisation that can be added by splitting application code into multiple smaller bundles. Developers can add code splitting also with lazy loading and dynamic imports.

4.4 Client Components

The naming convention “Client Components” is a way of separating the old regular React components from the Server Component. Client Components are typical React components, which can use regular React features like hooks, have event handlers, and manage state on the client side. Client Components are indicated with ‘use client’ directive as the first line of code in a file. The ‘use client’ directive tells that Client Components must be rendered on the client, bundler needs that directive information to be able to separate Client Components and Server Components. [React RFCs 2024a]

Listing 3 contains a Client Component example a NoteEditor component, which was imported in Server Component example in Listing 2. NoteEditor can use React hook named useState to add state variables, that are component’s own memory. With useState an initial value is added to the title and body. UseState returns an array with two elements, the current state and the function that can be used to update the value of the state. React will re-render the component when the state is updated. NoteEditor component contains a form that has onSubmit event handler, which will trigger when the form is submitted.

```
// NoteEditor.js - Client Component

'use client';

import { useState } from 'react';

export default function NoteEditor(props) {
  const note = props.note;
  const [title, setTitle] = useState(note.title);
  const [body, setBody] = useState(note.body);
  const updateTitle = event => {
    setTitle(event.target.value);
  };
  const updateBody = event => {
    setBody(event.target.value);
  };
  const submit = () => {
    // ...save note...
  };
  return (
    <form action="..." method="..." onSubmit={submit}>
      <input name="title" onChange={updateTitle} value={title} />
      <textarea name="body" onChange={updateBody}>{body}</textarea>
    </form>
  );
}
```

Listing 3. Client Component example [React RFCs 2024a].

4.5 Rendering React Server Components

The composition of application components is hierarchical, the structure is a tree that starts from a root component and spreads downwards. In React Server Components architecture, the root is always a Server Component.

React Server Component rendering process is illustrated in Figures 11, 12, and 13. Figure 11 has a React tree, that showcases an application structure, that consists of Server and Client Components.

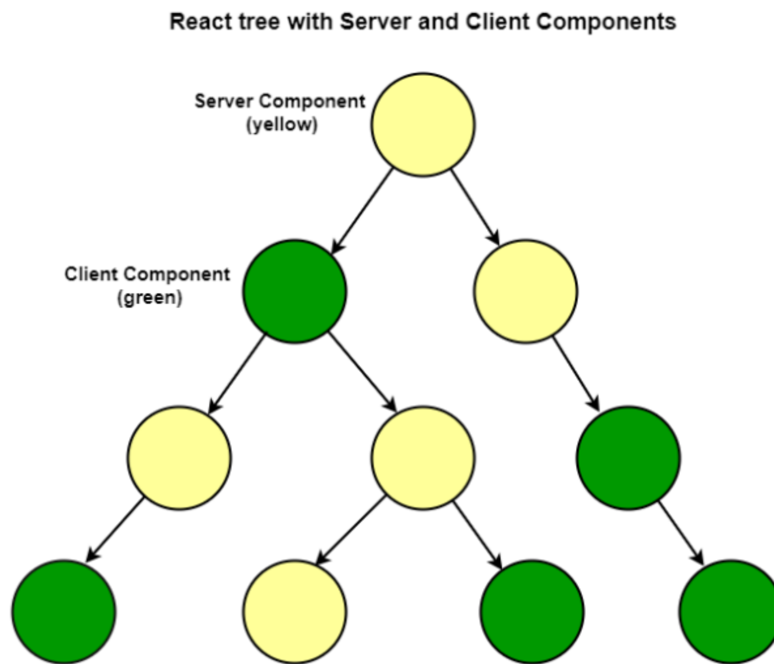


Figure 11. React tree with Server and Client Components. Adapted from [Wu 2022].

The rendering process workload is divided by React and a framework, where Next.js handles some sections for example routing and bundling areas. Next.js Framework is the one that first gets users' requests to view a given URL, the framework then passes needed properties to the starting component and requests React to start rendering. [React RFCs 2024a]

On the server, React starts rendering first the root Server Component and all other Server Components from the tree. From Server Components, React streams native HTML components (e.g. span, div) as JSON description of the user interface. Unlike Server Components, Client Components are not fully rendered on a server, Client Components are streamed with serialised properties and an indication where code is in the JS bundle. From server framework progressively streams the response from React to the client. [React RFCs 2024a]

Figure 12 portrays how React transforms component information. The response format is like JSON but with placeholders for Client Components or suspended components, which are streamed later to a browser. When the response data tree is serialisable, it can be sent to a client. [React RFCs 2024a, Wu 2022]

Figure 12. Serialisable React tree, that can be sent to a browser. Adapted from [Wu 2022].

When React Server Components are rendered on a server, they don't return HTML, by default a description of the user interface is returned. The rendered form of Server Components is outputted as data, that are "Virtual DOM" elements in JSON format. Those elements are constructing the Server Component parts of the tree. Reconciling the Virtual DOM is possible with this data format, and it makes it possible to merge with the Client Components, which are rendered only on the client and when re-rendering happens. [React RFCs 2024a]

In Next.js documentation React Server Component Payload (RSC Payload) is a term that is used for describing this data format, that is sent from server to client as response and is then used to update the DOM [Next.js 2024b]. React does not yet use this term in documentation. React documentation doesn't have comprehensive information about React Server Components, because RSCs are still experimental feature on the latest React release. At the time of writing this research, the latest React release is version 18.2.0, released in June 2022 [React releases 2024].

On the client, React reconstructs the tree with streamed data and renders Client Components. Figure 13 shows the reconstructed tree. React can render and reconcile virtual DOM in stages. Rendering can suspend and still, other parts that are already painted (attached to browsers DOM) on the screen, are interactable.

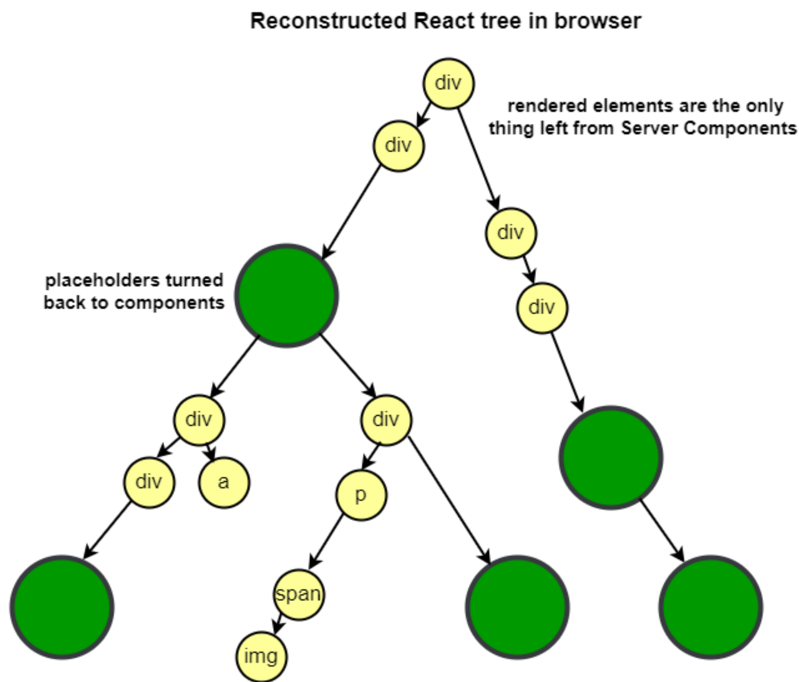


Figure 13. Reconstructed React tree in browser. Adapted from [Wu 2022].

React Server Components architecture does not replace Server-Side Rendering (SSR), they can be used together but it is not mandatory [React RFCs 2024a]. React specifies how RSC rendering works, the framework that is used with RSC can decide if it also wants to use other rendering patterns. Next.js uses Static Rendering, Dynamic Rendering, and Streaming with RSC in the App Router version [Next.js 2024b].

5 Next.js Framework

Chapter 5 is about the Next.js framework, the new App Router, and how React Server Components are rendered in Next.js. Section 5.1. introduces Next.js, Section 5.2. goes through Server Components rendering in Next.js.

5.1 Next.js

Next.js is a framework used to build full-stack applications with React components for UI. Next.js provides for example tools to handle application structure, routing, and configuration. Next.js was initially released in 2016, latest major version 14 was released in October 2023 [Next.js releases 2024]. Prior to version 13, Next.js had only one router version called Pages Router, which is still supported for the older versions of Next.js.

The new router called App Router in the app directory for doing routing and layout for web applications was added in Next.js version 13, which was beta released in October 2022. React version 18 features like React Server Components are supported with this newer App Router. [Next.js 2022]. A production-ready stable version of the App Router was released in May 2023 [Next.js 2023]. React has stated that the most comprehensive implementation of React Server Components architecture is in Next.js App Router [React 2024a].

5.2 Server Components rendering in Next.js

React Server Components are rendered only on the server. Each route in a Next.js application can have different server rendering strategies, the route can be rendered either statically or dynamically. For Server Components Next.js has three server rendering options: Static Rendering, Dynamic Rendering, or Streaming. [Next.js 2024b]. When compared to the rendering patterns in chapter three, Nex.js Static Rendering is similar to Static Site Generation (SSG) or when revalidation is added then it’s similar to Incremental Static Regeneration (ISR), Dynamic Rendering is similar to Server-Side Rendering (SSR), Streaming is similar to Streaming Server-Side Rendering (Streaming SSR). One difference is that in the rendering phase in Next.js, there are happening two different renderings, the rendering of the React Server Component Payload and the rendering of the HTML.

In Next.js Static Rendering option is the default, with it rendering is done at build time or when data is revalidated. Rendering switches automatically to dynamic rendering if data is not cached or code contains specific dynamic functions that need information that can

be obtained only when a user sends a request to the application. Dynamic rendering means in this case that rendering is done at request time for every user. With Streaming rendering can be built gradually by streaming finished chunks, without waiting for everything to finish before sending rendered UI to the client. [Next.js 2024b]

React Server Component Payload (RSC Payload) is a data format that holds the information about rendered Server Components, and props that need to be passed later to the Client Components, and placeholders for Client Components with references to the JavaScript files. [Next.js 2024b]

Figure 14 displays a diagram of Next.js default caching procedure for statically rendered route, the diagram also shows how the static rendering works in Next.js. On a server at build time when a route is started rendering, first data is fetched from a data source and then data is cached separately in Data Cache. Request memorisation is optimisation that remembers if there has been the same fetch request already, then data is not fetched multiple times unnecessarily. With the data React Server Component Payload is rendered, then HTML is rendered according to the RSC Payload and Client Component JavaScript. Full Route Cache stores the created RSC Payload and HTML, which can then be sent to a client when the route is requested by a user. On the client, Next.js uses an in-memory Router Cache to store the RSC Payload. The Router Cache is only stored for the time of user session. Full Route Cache and Data Cache on server are persistent. [Next.js 2024b]

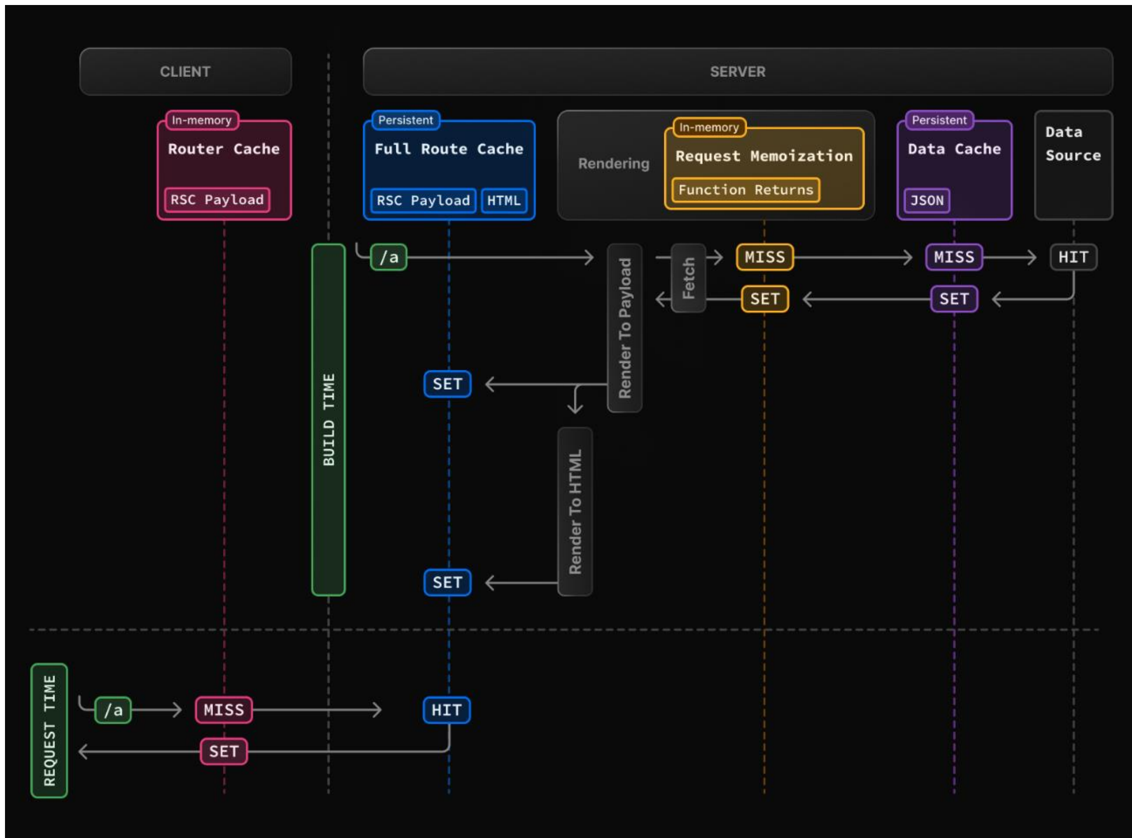


Figure 14. Next.js default caching procedure for statically rendered route [Next.js 2024b].

Differences between static and dynamic route visits are visualised in Figure 15. When a user makes an initial visit to a static route, RSC Payload and rendered HTML are fetched from a server's Full Route Cache. On the client, RSC Payload is set on Router Cache, and the page for the route is shown to the user. Dynamic routes don't use a Full Route Cache. When a user makes the initial visit to a dynamic route, the requested route is then rendered and created RSC Payload and HTML is sent to the client, where RSC Payload is set on the Router Cache, and the page for the route is shown to the user. On subsequent navigation to the same routes, both static and dynamic routes behave similarly by checking the Router Cache and using the cached RSC Payload. If the Router Cache contains the requested route RSC Payload, there is no need to send the request to the server. [Next.js 2024b]

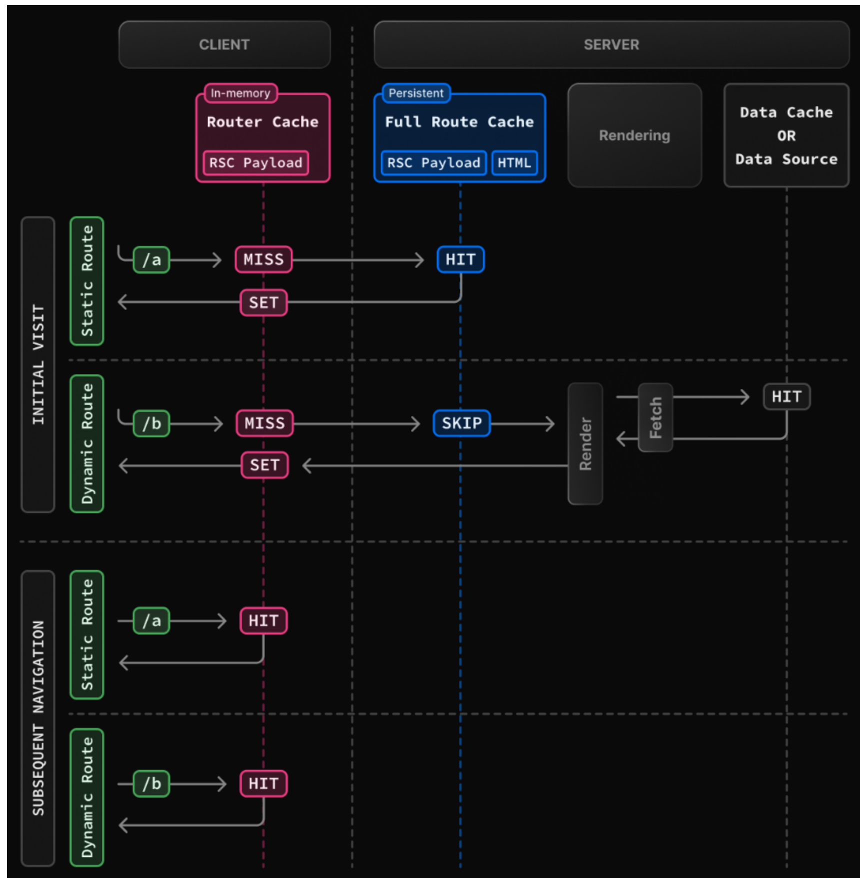


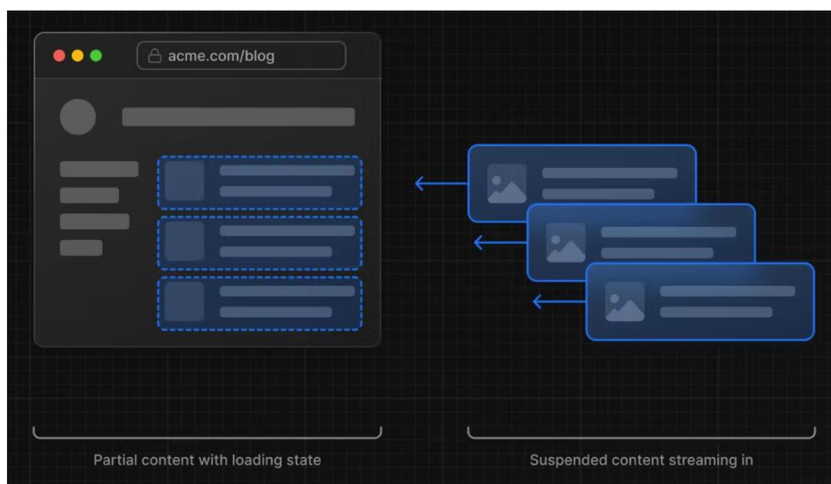
Figure 15. Differences of static and dynamic routes [Next.js 2024b].

On an initial visit to a route Client Components are prerendered on a server to get a fast initial HTML page shown for a user. The first page shown is not yet interactable. On the client JavaScript is used to hydrate the Client Components, to make them interactable. When the user later navigates to the same route, the Client Components are rendered fully on the client. This is done by using the loaded JavaScript and RSC Payload to reconcile the Client Component tree and Server Component tree into a single tree and update the DOM. [Next.js 2024b]

Server-side rendering with data fetching can take time, and to fully render a page on a server all data must be fetched first. These process parts block other parts starting before the first finishes. On the client React also needs everything to be downloaded before React can start a hydration process. From requesting a page to finishing the hydration process, all tasks are sequential, so the time that a user has to wait for the page to have any inter-activity can be long. Streaming can help with this problem. When adding Streaming with React Suspense components, the page can have suspended components, that are later streamed to the client. This allows faster initial page rendering and loading states to be shown to a user. [Next.js 2024b]. When a page contains components that fetch data, it is

beneficial to suspend those components and stream them later when data fetching and rendering finishes. This way process is split into parts that can be done simultaneously and not everything has to wait for previous tasks to finish. This should also reduce the time that a user has to wait for the page to have interactivity.

Streaming suspended components to a client is visualised in Figure 16. In Figure 16a there is a page view that contains a loading state for the components that are suspended. Figure 16b shows how suspended components are streamed in chunks from a server to a client after components are fully rendered. On the client, the suspended components are swapped to their own places, and the loading state is no longer visible. [Next.js 2024b]



(a) (b)

Figure 16. Streaming server-rendered components to a client [Next.js 2024b].

6 Case study

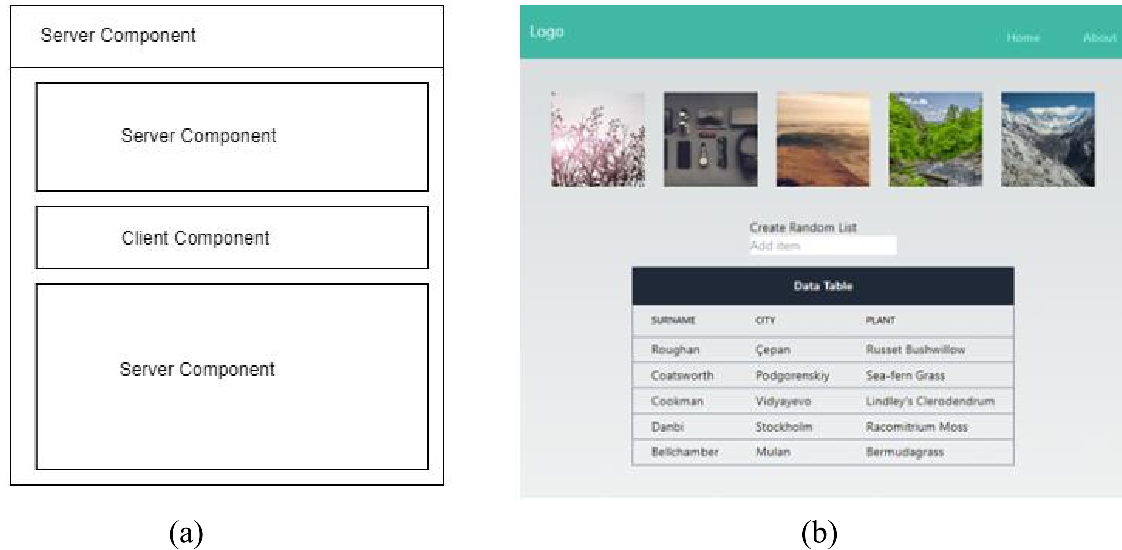
This chapter introduces a demo web application, that was built with Next.js and React Server Components, the details are in Section 6.1. Streaming SSR implementation part is talked about in Section 6.2. Section 6.3. goes through the experiment and how it was conducted. Experiment results and analysis are presented in Section 6.4.

6.1 Web application prototype

A simple demo web application, that uses Next.js and React Server Components, was developed during the research. This demo web application is named as List-app, and the source code is stored in a GitHub repository [Kettunen 2024]. The application is used in an experiment, where static and dynamic rendering options are compared by evaluating performance metric results.

In this research, the web application landing page (index page) is referred to as a home page. The home page component layout is visualised in Figure 17a, and the rendered home page displayed in a browser is shown in Figure 17b. Home page contains four components, three of which are Server Components, and one is a Client Component. The list below summarises home page components:

- Nav component provides links to application routes, for a user to navigate on the application between routes.
    - Server Component
- ImageList component fetches image data from an external data source and displays images.
    - Server Component
- RandomList component lets a user create a list with items user inputs in the field.
    - Client Component
- PersonList component displays a table with data, the used mock data is stored with the code.
    - Server Component



(a) (b)

Figure 17. (a) Home page component layout. (b) Rendered home page displayed in browser.

6.2 Streaming SSR example

React's Suspense component is used to enable streaming server-side rendered components [React 2024c]. With Suspense individual components can be suspended, while other components on a page can be server-rendered. In the built production version of the demo web application that uses dynamic rendering, a combination of SSR and Streaming SSR is used.

Listing 4 contains Home page code in a file called page.js, which is special Next.js file, which will define a page UI for a route. The UI for the home route is defined in this code. From Listing 4, it is visible that ImageList and PersonList components are wrapped with Suspense components and a Spinner component is used as a fallback until child components are finished rendering. ImageList component fetches data from an external source, and sometimes data fetching can take longer time than anticipated. If components that fetch data would not use Suspense, the whole page would need to wait until even the slower data-fetching components do finish rendering, before sending the rendered page to a browser. With Suspense, the page can be rendered with a fallback component, in this case with a Spinner component, for the initial render and later switch the finished component to its place.

```
import Nav from '@/components/nav';
import ImageList from '@/components/imageList';
import PersonList from '@/components/personList';
import RandomList from '@/components/randomList';
import Spinner from '@/components/spinner';
import { Suspense } from 'react';

export default function Home() {
  return (
    <>
      <Nav />
      <div className="flex … ">
        <Suspense fallback={<Spinner />}>
          <ImageList />
        </Suspense>
        <RandomList />
        <Suspense fallback={<Spinner />}>
          <PersonList />
        </Suspense>
      </div>
    </>
  );
}
```

Listing 4. Home page code.

PersonList component uses static asset data, which is inserted in a table in the component. Later in the experiment test phase, it was found out that this kind of static asset data does not activate the use of the Suspense component. PersonList is never suspended nor streamed. The source code was not altered due to the find in the test phase, even though Suspense does not work with PersonList component. From the home page only ImageList component can be suspended, because ImageList fetches data from an external data source.

6.3 Experiment

Next.js provides two rendering options for a route, to be static or dynamic. Static is the default option. For the tests, two versions of the demo application were added in different version control branches. One version is for the static home route tests and the other is for dynamic home route tests. In the static home route version, when a production version is being built Next.js fetches all needed data, creates a static build, and caches it. In the dynamic home route version, Next.js doesn’t render dynamic routes at build time instead dynamic route is rendered when a route is requested by a user.

The experiment was conducted with the two production versions, one with static home route and other with dynamic home route. The initial load of the home page was analysed with Lighthouse tests. Lighthouse performance tests were chosen to gather information about how static route with SSG rendering pattern and dynamic route with combination of SSR and Streaming SRR rendering pattern options differ.

First the Lighthouse tests were done with emulating desktop device with custom throttling. Lighthouse uses network throttling to alter current networks speed, so that the simulation is more consistent with regular network speeds. Later Lighthouse tests were done with emulating mobile device with slow 4G throttling.

The Lighthouse tool developed by Google was used to carry out performance audit. First the page that needs to be audited is opened in the browser. Lighthouse tests were run in Google Chrome browser with Chrome DevTools. Incognito window was used to avoid the influence of browser extensions on the tests results. In the Lighthouse tab, after selecting mode, device and categories, the page analysing can start. In this experiment Lighthouse was run in navigation mode, which analyses single page load without any user interactions. Device selection had mobile or desktop options. The tests were carried out with categories: performance, accessibility, best practices, and SEO. Only the results from performance tests were later analysed. After each audit run, a report was generated.

The used Lighthouse [2024] version tracks the following five performance metrics:

- First Contentful Paint (FCP)
- Speed Index (SI)
- Largest Contentful Paint (LCP)
- Total Blocking Time (TBT)
- Cumulative Layout Shift (CLS)

First Contentful Paint is the time measurement it takes to screen have first element to be rendered after a user has requested a page. Speed Index is the time measurement that tells how fast content is painted to a screen when a page is loaded. Speed Index is calculated after a full page load from the progression between frames. Largest Contentful Paint is the time measurement when the largest component on screen is rendered. Total Blocking Time is the time measurement of when a page can’t respond to any user interaction. Cumulative Layout Shift is the measurement of visible components shifting place while a page is being rendered. [Lighthouse 2024]

Table 1 contains the Lighthouse performance metric values categorised into three categories: fast, moderate, and slow. If test result values are in the fast category, then tested application has good performance. If test result values are in the moderate category, then tested application performance needs some improvement. If test result values are in the slow category, then tested application performance is poor and improvement is needed to serve better user experience. [Lighthouse 2024]

Performance metric	Fast	Moderate	Slow
First Contentful Paint (s)	0–1.8	1.8–3	over 3
Total Blocking Time (ms)	0–200	200–600	over 600
Speed Index (s)	0–3.4	3.4–5.8	over 5.8
Largest Contentful Paint (s)	0–2.5	2.5–4	over 4
Cumulative Layout Shift	0–0.1	0.1–0.25	over 0.25

Table 1. Lighthouse performance metric values categorised as fast, moderate, or slow. Information in the table is gathered from Lighthouse [2024] documentation.

6.4 Results and analysis

In the experiment two production versions are built, one with a **dynamic home route** and the other with a **static home route**, and then versions are audited with Lighthouse performance tests. One test round was emulated with a desktop device and the other with a mobile device. All tests were done ten times, performance tests were run multiple times to provide more accurate values with reduced variable interference. The averages of the performance values are shown in the Tables 2 and 3.

The static routes can use Next.js Static Rendering which is like a combination of SSG or ISR rendering pattern and rendering of the React Server Component Payload. If data revalidation is not added in the implementation, then SSG is used instead of ISR. The dynamic routes can use either Next.js Dynamic Rendering which is like a combination of SSR and rendering of the RSC Payload, or it uses Next.js Streaming which is like a combination of Streaming SSR and rendering of the RSC Payload. Dynamic routes can also use the combination of the two. When talking about Next.js route rendering, the patterns also contain RSC Payload rendering, even though there might be a mention only of SSG, ISR, SSR, or Streaming SSR.

In the web application demo implementation static home route uses SSG, because data revalidation was not added. The other version's dynamic home route implemented Streaming SSR with the React Suspense for the ImageList component, other parts of the page use SSR. Suspense component is used in both code versions, but static production

build does not use streaming, because static routes are already fully rendered with fetched data at build time. Source code differs only in the data fetching part whether the data can be cached by Next.js or not. Static route uses cached data, dynamic route does not cache data on a server.

Both tested versions fetch 30 pictures in the ImageList component, instead of the 5 pictures in the original demo application. Picture quantity was increased because 5 pictures produced minimal differences in preliminary tests.

Table 2 contains performance test result averages, from the tests that were done with desktop device emulation. First Contentful Paint values are both great, very slight 0.1 seconds difference, static version's first content being rendered slightly quicker. Total Blocking Time was zero milliseconds in both versions, meaning that there were no long tasks blocking the main thread, that would keep the page unresponsive to the user. Speed Index value in the dynamic route is 62,5% faster, being significantly quicker, which tells that the visible parts of the page are fully shown quicker in the dynamic version compared to the static version. Largest Contentful Paint values are both good, with only 0.1 seconds difference, the dynamic version is 20% faster than the static version, painting the largest component on screen. Cumulative Layout Shift is perfect 0 with the static route, the dynamic route contains minor layout shifting, which is due to a fallback part implementation in Suspense component.

	Dynamic route	**Static route**
First Contentful Paint (s)	0.3	0.2
Total Blocking Time (ms)	0	0
Speed Index (s)	0.3	0.8
Largest Contentful Paint (s)	0.4	0.5
Cumulative Layout Shift	0.002	0

Table 2. Performance test result averages, Lighthouse tests with desktop device.

Table 3 contains performance test result averages, from the tests that were done with mobile device emulation. First Contentful Paint values are both still quite low with the same 0.1 seconds difference, static version's first content being rendered in 0.8 seconds and dynamic version rendering first content in 0.9 seconds. Total Blocking Time metric is showing differences between versions, the dynamic version being 72% faster than the static version, meaning that in the static version long tasks are not blocking the main thread. Even though Total Blocking Time values are still quite low only 7 milliseconds in dynamic version and 25 milliseconds in static version, the 72% difference in this scale

is quite significant. Speed Index value in dynamic route is 56 % faster than in the static route. Largest Contentful Paint values have 0.2 second difference, the dynamic version is 13% faster than the static version painting the largest component on screen. Cumulative Layout Shift metric gives zero in the static route version, the dynamic route version has some layout shifting.

	Dynamic route	**Static route**
First Contentful Paint (s)	0.9	0.8
Total Blocking Time (ms)	7	25
Speed Index (s)	1.1	2.5
Largest Contentful Paint (s)	1.3	1.5
Cumulative Layout Shift	0.004	0

Table 3. Performance test result averages, Lighthouse tests with mobile device.

Based on the values of the five performance metrics (FCP, SI, LCP, TBT, CLS), Lighthouse calculates the overall Performance score, which can be from 0 to 100 [Lighthouse 2024]. For the dynamic route, all Performance score values were 100 when tested with desktop and mobile emulations. For the static route, Performance score values were 100 when tested with desktop emulation, in the test with mobile emulation the test's Performance score value averaged to 100. These maximum performance scores indicate that the tested web application is quite minimalistic, as it performed so well in the tests without adding performance improvements. In the Lighthouse [2024] documentation is said that the maximum value of 100 is hard to get, for a good user experience sites should be in the range of 90 to 100.

When both test results from Tables 2 and 3 are compared to the Lighthouse metric value categories in Table 1, all test result values fit in the fast category. Even though there are slight differences, all the test result values are good, and both compared rendering approaches, SSG and combination of SSR and Streaming SSR, will produce fast rendering of the web application.

After analysing test results in both desktop and mobile emulations, the differences between the dynamic and static versions results are then discussed here. The static route option seems to be rendering slightly faster the first content, which can be seen in the values of the First Contentful Paint metric. But especially in the mobile emulated test with a slower network, the dynamic route option has better values in Total Blocking Time, Speed Index, and Largest Contentful Paint metrics. Those values tell that the dynamic

route version executes tasks faster, populates the page faster, and paints the largest content to screen faster. The value differences of the Cumulative Layout Shift metric are due to the implementation faults, which are easily fixable, so the CLS metric values are left out of this comparison. Overall, the dynamic route version performed slightly better in the Lighthouse performance tests. The combination of SSR and Streaming SSR in the dynamic route version seems to produce slightly better performance, than the used SSG in the static route version. The performance differences are quite minimal in the test result.

These tests were conducted by running production versions on my own computer. The production versions could yield similar or different test results if run on other computers or by using hosting services. The experiment's test results give an indicator of how the static and dynamic versions perform, but the results don't give the absolute truth. The benefits of SSG could be more prominent if static content were cached by using a Content Delivery Network (CDN). By using CDNs the static content can be served easily and makes a SSG solution more scalable than SSR, which needs to be rendered on each request. When choosing which rendering pattern to use depends also on the requirements. If content does not update frequently, like blog posts or marketing materials, then SSG is a better choice. SSG is also more cost-efficient than SSR because the application needs to be rendered only at build time. If ISR is used instead of SSG, then data revalidation is also possible. If the content is user-specific or otherwise content needs to be updated frequently, then SSR or Streaming SSR is the rendering pattern to choose.

7 Conclusion

This study collected information about rendering patterns, React Server Components, and how React Server Components are rendered in Next.js. There are many rendering patterns that can be used with React and React Server Components. The main concepts of rendering patterns of Static Site Generation (SSG), Client-Side Rendering (CSR), Server-Side Rendering (SSR), Incremental Static Regeneration (ISR), and Streaming Server-Side Rendering (Streaming SSR) were introduced in Chapter 3. Later in the Next.js Framework chapter, it was explained how rendering works in Next.js when React Server Components are used. The main ideas of React Server Components rendering in Next.js are covered in the following paragraphs.

When an application's page that contains React Server Components and Client Components is rendered, React Server Component Payload (RSC Payload) is created to hold the needed information of the combined component tree. RSC Payload has the information about rendered Server Components and props that need to be passed later to the Client Components, and placeholders for Client Components with references to the JavaScript files [Next.js 2024b].

In Next.js when using React Server Components there are three server rendering options: Static Rendering, Dynamic Rendering, or Streaming [Next.js 2024b]. When compared to the rendering patterns in chapter three, where rendering of the HTML is explained in different patterns, Next.js Static Rendering is like Static Site Generation or when revalidation is added then it's like Incremental Static Regeneration, Dynamic Rendering is like Server-Side Rendering, Streaming is like Streaming Server-Side Rendering. The main difference is that in the rendering phase on a server, Next.js does also the rendering of the React Server Component Payload. The rendered HTML is only used to show quickly a noninteractive page when a user makes the initial request for a route. React Server Component Payload is used on the client to update DOM and with JavaScript the Client Components can be hydrated, which adds the interactive parts to the page. On subsequent navigations, React Server Component Payload is used to render a page on the client.

In web applications built by Next.js, rendering patterns can differ by each route, but a route can only be fully static or dynamic. The static routes use Next.js Static Rendering. The dynamic routes use either Next.js Dynamic Rendering, or it uses Next.js Streaming. Dynamic routes can also use the combination of the two. When talking about rendering

in Next.js (when using App Router), the patterns mentioned here also contain RSC Payload rendering, even though the pattern might be only referred to as SSG, ISR, SSR, or Streaming SSR.

To be able to compare cases in the case study part of the research, a demo web application was developed using Next.js and React Server Components. Two application versions were implemented, one with a static home route that uses SSG, and the other with a dynamic home route that uses a combination of SSR and Streaming SSR. The two production builds were tested with Lighthouse performance tests, where the initial load of the home page (index page) was analysed.

Lighthouse performance tests results were analysed to get information about how the two rendering options differ from each other. Based on the Lighthouse performance test results, the static route version is rendering slightly faster the first content. But overall, the used dynamic route version is performing slightly better in the tests, this becomes evident in the mobile emulated test with a slower network, where the dynamic route option has better values in Total Blocking Time, Speed Index, and Largest Contentful Paint metrics. In tests, the dynamic route version executes tasks faster, populates the page faster, and paints the largest content to a screen faster. The combination of SSR and Streaming SSR in the dynamic route version seems to produce slightly better performance, than the used SSG in the static route version. But the performance differences are quite minimal in the test results, so there is no clearly better performer found in this test.

Both versions, static route with SSG and dynamic route with a combination of SSR and Streaming SSR, produced test results that fit in the fast category in the Lighthouse metric value categories. This indicates that the tested web application was quite simple. These test values give an indication of how these two versions perform in this one test environment, but test results can vary when done in another environment. When choosing a rendering pattern also requirements are a big factor that influence which rendering pattern to choose. The default rendering option in Next.js is SSG, which is more cost-efficient than SSR because an application needs to be rendered only at build time, and if ISR is used then data revalidation is also possible. Dynamic rendering with SSR or Streaming SSR are the right patterns to choose when content is user-specific or otherwise content needs to be updated frequently.

Further research could be done on areas that were not studied during this research. The initial load of the home page was analysed with the Lighthouse tests, further research could be done on the subsequent navigation where RSC Payload is used to render a page

on the client. The built web application demo is small, so the differences between the rendering patterns, when evaluated with performance metrics, are quite small. Further studies with using a more complex application, could yield more comprehensive differences in performance tests. Also adding a version that uses only SSR could be an option that could have different results than the combined SSR and Streaming SSR. Further research could also be done by comparing web applications built with this new style Next.js with App Router and React Server Components to the older Next.js Pages Router without React Server Components. That comparison could better bring out the Server Component's zero JavaScript bundle size benefits.

References

Abramov, Dan. 2021. New Suspense SSR Architecture in React 18. Available: https://github.com/reactwg/react-18/discussions/37 (Checked 12.02.2024)

Angular. 2024. Angular documentation. Available: https://angular.io/docs (Checked 23.02.2024)

Astro. 2024. Astro documentation. Available: https://docs.astro.build/en/getting-started/ (Checked 24.02.2024)

Coccia, M., & Benati, I. 2022. Comparative Studies. In *Global Encyclopedia of Public Administration, Public Policy, and Governance* (pp. 2207–2213). Springer International Publishing. https://doi.org/10.1007/978-3-030-66252-3_1197

DOM. 2024. DOM - Living Standard. Available: https://dom.spec.whatwg.org/ (Checked 6.03.2024)

Dumez, H. 2015. What Is a Case, and What Is a Case Study? *Bulletin de Méthodologie Sociologique, 127*(1), 43–57. https://doi.org/10.1177/0759106315582200

ECMA-262. 2024. ECMA-262, ECMAScript language specifications. Available: https://ecma-international.org/publications-and-standards/standards/ecma-262/ (Checked 16.02.2024)

Ecma International. 2024. History. Available: https://ecma-international.org/about-ecma/history/ (Checked 16.02.2024)

Engineering at Meta. 2017. Relicensing React, Jest, Flow, and Immutable.js. Available: https://engineering.fb.com/2017/09/22/web/relicensing-react-jest-flow-and-immutable-js/ (Checked 17.02.2024)

Gatsby. 2024. Gatsby documentation. Available: https://www.gatsbyjs.com/docs (Checked 24.02.2024)

Google Developers. 2024a. Inside look at modern web browsers (part 2). Available: https://developer.chrome.com/blog/inside-browser-part2 (Checked 07.02.2024)

Google Developers. 2024b. Inside look at modern web browsers (part 3). Available: https://developer.chrome.com/blog/inside-browser-part3 (Checked 06.03.2024)

HTML. 2024. HTML - Living Standard. Available: https://html.spec.whatwg.org/ (Checked 29.01.2024)

Javatpoint. 2024. What is Dom in React? Available: https://www.javatpoint.com/what-is-dom-in-react (Checked 19.02.2024)

Kettunen, Tellervo. 2024. List-app - web application demo. Available: https://github.com/TelluK/list-app (Checked 15.04.2024)

Lighthouse. 2024. Lighthouse performance scoring. Available: https://developer.chrome.com/docs/lighthouse/performance/performance-scoring (Checked 26.03.2024)

MDN Web Docs. 2024a. CSS: Cascading Style Sheets. Available: https://developer.mozilla.org/en-US/docs/Web/CSS (Checked 30.01.2024)

MDN Web Docs. 2024b. JavaScript. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript (Checked 30.01.2024)

MDN Web Docs. 2024c. Populating the page: how browsers work. Available: https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work (Checked 07.02.2024)

Meta. 2021. Introducing Meta: A Social Technology Company. Available: https://about.fb.com/news/2021/10/facebook-company-is-now-meta/ (Checked 17.02.2024)

Next.js. 2022. Next.js 13. Available: https://nextjs.org/blog/next-13 (Checked 13.02.2024)

Next.js. 2023. Next.js 13.4. Available: https://nextjs.org/blog/next-13-4 (Checked 15.02.2024)

Next.js. 2024a. Incremental Static Regeneration (ISR). Available: https://nextjs.org/docs/pages/building-your-application/data-fetching/incremental-static-regeneration (Checked 12.02.2024)

Next.js. 2024b. Next.js App Router documentation. Available: https://nextjs.org/docs/app (Checked 29.02.2024)

Next.js releases. 2024. Next.js releases. Available: https://github.com/vercel/next.js/re-leases (Checked 4.03.2024)

Nuxt. 2024. Nuxt documentation. Available: https://nuxt.com/ (Checked 24.02.2024)

Ollila, R., Mäkitalo, N., & Mikkonen, T. 2022. Modern Web Frameworks: A Comparison of Rendering Performance. *Journal of Web Engineering*, 21(3), 789-814. https://doi.org/10.13052/jwe1540-9589.21311

Osmani, Addy. 2023. Learning JavaScript Design Patterns, 2nd Edition. O'Reilly Media, Inc.

Osmani, Addy & Jason Miller. 2024. Rendering on the Web. Available: https://web.dev/articles/rendering-on-the-web (Checked 08.02.2024)

Patterns.dev. 2024. React - Rendering Patterns. Available: https://www.patterns.dev/re-act (Checked 08.02.2024)

Qwik. 2024. Qwik documentation. Available: https://qwik.dev/ (Checked 24.02.2024)

React. 2022. React v18.0. Available: https://react.dev/blog/2022/03/29/react-v18 (Checked 14.02.2024)

React. 2023. React Labs: What We've Been Working On – March 2023. Available: https://react.dev/blog/2023/03/22/react-labs-what-we-have-been-working-on-march-2023 (Checked 15.02.2024)

React. 2024a. React documentation. Available: https://react.dev/ (Checked 15.02.2024)

React. 2024b. React documentation (legacy). Available: https://legacy.re-actjs.org/docs/getting-started.html (Checked 19.02.2024)

React. 2024c. Suspense component. Available: https://react.dev/reference/react/Suspense (Checked 15.04.2024)

React release. 2022. Release v18.0.0. Available: https://github.com/facebook/react/releases/tag/v18.0.0 (Checked 14.02.2024)

React releases. 2024. Release releases. Available: https://github.com/facebook/react/releases (Checked 6.03.2024)

React RFCs. 2024a. RFC: React Server Components. Available: https://github.com/reactjs/rfcs/blob/main/text/0188-server-components.md (Checked 15.02.2024)

React RFCs. 2024b. RFC: Server module conventions. Available: https://github.com/reactjs/rfcs/blob/main/text/0227-server-module-conventions.md (Checked 15.02.2024)

Remix. 2024. Remix documentation. Available: https://remix.run/ (Checked 24.02.2024)

Solid. 2024. Solid documentation. Available: https://www.solidjs.com/ (Checked 24.02.2024)

Sommerville, Ian. 2007. Software engineering. 8th ed. Harlow: Addison-Wesley. pp. 270-275.

State of JS. 2024. State of JS 2022. Available: https://2022.stateofjs.com/en-US (Checked 23.02.2024)

Svelte. 2024. Svelte documentation. Available: https://svelte.dev/ (Checked 24.02.2024)

SvelteKit. 2024. SvelteKit documentation. Available: https://kit.svelte.dev/docs/introduction (Checked 24.02.2024)

Taivalsaari, A., Mikkonen, T., Pautasso, C., Systä, K. 2019. Client-Side Cornucopia: Comparing the Built-In Application Architecture Models in the Web Browser. In: Escalona, M., Domínguez Mayo, F., Majchrzak, T., Monfort, V. (eds) *Web Infor-*

mation Systems and Technologies. WEBIST 2018. Lecture Notes in Business Information Processing, vol 372. Springer, Cham. https://doi.org/10.1007/978-3-030-35330-8_1

Thakkar, Mohit. 2020. Building React Apps with Server-Side Rendering Use React, Redux, and Next to Build Full Server-Side Rendering Applications. 1st ed. Berkeley, CA: Apress.

The Open Source Initiative. 2024. The MIT Licence. Available: https://opensource.org/license/mit/ (Checked 17.02.2024)

TypeScript. 2024. TypeScript for the New Programmer. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html (Checked 29.01.2024)

Vepsäläinen, J., Hellas, A., & Vuorimaa, P. 2023a. Implications of Edge Computing for Static Site Generation. *Proceedings of the 19th International Conference on Web Information Systems and Technologies - WEBIST*; SciTePress, pages 223-231. https://doi.org/10.5220/0012173900003584

Vepsäläinen, J., Hellas, A., & Vuorimaa, P. 2023b. The State of Disappearing Frameworks in 2023. *International Conference on Web Information Systems and Technologies, WEBIST - Proceedings*, 232–241. https://doi.org/10.5220/0012174000003584

Vue. 2024. Vue documentation. Available: https://vuejs.org/guide/introduction.html (Checked 23.02.2024)

Web Reference. 2024. A Brief History of ECMAScript Versions in JavaScript. Available: https://webreference.com/javascript/basics/versions/ (Checked 16.02.2024)

Wu, Chung. 2022. How React server components work: an in-depth guide. Available: https://www.plasmic.app/blog/how-react-server-components-work (Checked 28.02.2024)