

UPPSALA
UNIVERSITET

UPTEC IT 23031
Examensarbete 30 hp
September 2023

Comparisons of Server-side Rendering and Client-side Rendering for Web Pages

Carl Nordström, August Dixelius



Civilingenjörsprogrammet i informationsteknologi



UPPSALA
UNIVERSITET

Comparisons of Server-side Rendering and Client-side Rendering for Web Pages

Carl Nordström, August Dixelius

Abstract

Page load speed on the web is one of the most important metrics for retaining users and the way a web page is rendered plays a big part in page load speed. There are many different ways to render a web page and the design of websites differs a lot, so in this paper a website was developed with differently designed layouts and then tested with client-side rendering and server-side rendering . The performance for each rendering method was tested with Google Lighthouse and Google DevTools Performance Insights. The tests showed that for web pages with fewer elements the choice of rendering method does not have a significant impact on the page loading speed of the web page. When the web page contains more content, which can be in the form of additional media or API calls, server side rendering is the better the choice when looking at rendering performance.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Erik Karlsson Ämnesgranskare: Maya Neytcheva

Examinator: Lars-Åke Nordén

Sammanfattning

Tiden det tar för en hemsida att ladda in är ett av de viktigaste måtten för att behålla användare och sättet som en hemsida är renderad har en stor påverkan på hur lång tid det tar för en sida att ladda in. Det finns många olika sätt att rendera en hemsida och designen av hemsidor skiljer sig. Så i det här projektet så utvecklades en hemsida med vyer som skiljde sig åt och dessa testades med client-side-rendering och server-side-rendering. Prestandan för varje renderingsmetod var testad med Google Lighthouse och Google DevTools. Testerna visade att för hemsidor med färre element så hade valet av renderingsmetod ingen signifikant påverkan på laddningshastigheten för hemsidan. När hemsidan innehåller mer innehåll, såsom mer media eller API-anrop, är server-side-rendering det bättre valet om man kollar på renderingsprestanda.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Purpose and Goals 2

2 Background **3**

2.1 What Happens When a User Navigates to a website? 3

2.2 Hypertext Transfer Protocol and Transmission Control Protocol 3

2.3 Browser Rendering 4

2.4 The Browser Engine Render Tree 5

2.5 Client-Side Rendering 6

2.6 Server-Side Rendering 7

2.7 Other Rendering Techniques 8

2.7.1 Static Site Generation 8

2.7.2 Incremental Static Rendering 9

2.7.3 Deferred Static Rendering 9

2.8 JavaScript Engine 9

2.9 Different Browser Engines 10

2.10 Rendering Performance Metrics 10

2.10.1 First Contentful Paint (FCP) 10

2.10.2 Largest Contentful Paint (LCP) 10

2.10.3 Total Blocking Time (TBT) 11

2.10.4 Cumulative Layout Shift (CLS) 11

2.10.5 Speed Index (SI) 11

2.10.6 Time to Interactive (TTI) 11

2.11 The History of Web Rendering 12
2.12 JavaScript Frameworks 12

3 Related Work **13**

3.1 Comparison Between Client-Side and Server-Side Rendering in the Web Development 13
3.2 A Hybrid Web Rendering Framework on Cloud 13
3.3 Measuring Web Latency and Rendering Performance: Method, Tools, and Longitudinal Dataset 14
3.4 A Performance Comparative on Most Popular Internet Web Browsers . 14

4 Method **15**

4.1 Tools for Testing Rendering Performance 15
4.1.1 Lighthouse 15
4.1.2 Chrome DevTools 16
4.1.3 GTmetrix 16
4.1.4 Testing Tools That Were Used 16
4.2 Web Frameworks 17
4.3 The different Test Design Layouts 17
4.3.1 Javascript 17
4.3.2 Images 17
4.3.3 API 18
4.4 Next.js 18
4.5 Compared Metrics 19
4.6 Testing Variables 19
4.7 Conducting the Tests 20

5 Architecture **20**

5.1 Web Page Architecture 20

5.2 Technical Testing Environment 20

6 Results **21**

6.1 JavaScript Test Results 21

6.2 Image Test Results 22

6.3 API Test Results 26

7 Discussion **31**

8 Conclusions **33**

9 Future Work **34**

10 Contributions of the Authors **38**

11 Appendix **39**

11.1 Test results 39

1 Introduction

The way in which websites are built is changing rapidly. The usage of websites and the time spent online has increased in the last years [2]. With more devices being connected to the internet and new emerging technologies the importance of fast and responsive websites is rising. Our ability to concentrate has become worse and the demand for everything to be fast is growing, especially when it comes to websites [3, 4] .

Load time is one of the factors that helps websites retain their users [5] [3] [6]. One way to make sure that the load time is faster is by using different rendering techniques. To render the content of a website the browsers need to know the design and layout of the elements in the website and what elements can be interacted with and what to do with this interaction. To do this the browser needs to download data from the server where the website is being hosted. Then, before anything can be shown, the browser needs to interpret the code as pixels on the screen. Poor rendering performance can result in sluggish page load times, choppy or inconsistent animations, and a frustrating overall user experience. This thesis aims to explore what a web developer can do to improve the rendering performance of their website when choosing which way a website should be rendered.

Websites can differ substantially from each other, and the optimal rendering technique for each website may not be the same. One website may have a lot of user interactivity while another has close to none, the types of elements and the amount of elements is also a factor that distinguishes many websites. Though, independent of the websites, a general goal is to reduce the load time as it is helpful when it comes to user retention.

The bandwidth of the users also varies, some users may have internet speeds that are up to 20-30 times faster than other users. When users that have a bandwidth which requires them to wait at least seconds, the way a website is rendered starts to impact the load time significantly. But if the user has a bandwidth which almost instantly loads all web pages the rendering method may not impact the rendering performance. There are also many different types of devices and web browsers that are used when accessing a website, these may all have an affect of the perceived rendering performance of a website.

1.1 Motivation

As the internet is growing and new websites are being developed every day, the need to create a website that stands out or holds up to the best websites is increasing in importance. Therefore knowing which rendering technique that is best in a performance aspect for a specific website and how the choice of rendering technique impacts the user experience is of significance.

This study gives test results based on different rendering techniques. This is something that would lead to better understanding of when different rendering techniques should be used and make people take a more informed choice when choosing how to render their website.

The difference between various rendering methods is good to know when developing web sites. In this thesis we test different rendering methods by the usage of a modern JavaScript framework, to aid developers in the decision of what rendering technique to use depending on the site they are developing.

1.2 Purpose and Goals

The purpose of this thesis is to understand what rendering technique fits a certain type of website, or if a mix of rendering techniques could enable performance improvements. The advantages of both these solutions will be compared. The criteria that will be compared are:

- performance of different rendering techniques.
- performance for users with varying hardware specifications and network speeds.
- development time and complexity of different rendering techniques.

The goal of the project is to establish how different rendering techniques perform for web pages with various types of layouts, and from that draw conclusions to what rendering techniques would be the most beneficial for certain types of websites. The rendering techniques will be tested using the new version of the Next.js framework, Next.js 13, which is a React framework with rendering functionalities. Different bandwidth speeds will be taken into consideration when testing the rendering techniques.

2 Background

This section describes the process of what happens when a user tries to access a website and the details of the rendering process of different web rendering techniques. Different metrics of measuring the web rendering performance is described in the later parts of the section.

2.1 What Happens When a User Navigates to a website?

The first thing that happens when a user navigates to a new website is that the browser of the user needs to find out where the server is located, it does this by doing a domain name system (DNS) lookup. The browser remembers the location of a server for a set amount of time that can be decided in the DNS settings of the server. The browser of the user makes a request for a certain content that it wants and the server sends the content in the form of an HTML file to the browser and the browser then tries to render the content. The way this HTML file is formatted is a significant factor in web performance and this is one of the main ways where rendering techniques differ from each other.

2.2 Hypertext Transfer Protocol and Transmission Control Protocol

Hypertext Transfer Protocol (HTTP) is a protocol which is based on a client-server model. It specifies how data is sent between client and server and is used for websites. To send data HTTP generally uses the Transmission Control Protocol (TCP) to send data and establish a connection between the client and the server [7].

When TCP is establishing a connection it uses a technique called *slow start*. With slow start packets are sent at a slower rate in the beginning to hinder congestion. As packets are successfully sent the amount of data that can be sent at once increases as long as no packet of data is dropped. When a packet is dropped the amount of data that can be sent at once is decreased to the start level, before starting over with the same process of increasing the amount of data that can be sent between the client and server [8].

2.3 Browser Rendering

The task of the browsers is to load an HTML file that is provided from a remote server and then display it in the correct way and also adding user interactability [9]. The part of the browser that renders a web page and displays it to the user is the so-called browser engine, also known as a layout engine or rendering engine. It is the browser engine that receives the HTML file and reads the raw bytes of data from the HTML that is generated from the HTML, CSS and JavaScript.

However, the browser engine can not read the raw data, it needs to work with a document object model (DOM). To convert the raw data into the DOM the browsers engine first converts the raw data to characters. The character encoding of the HTML is deciding the raw data to character conversion. These characters are then in turn converted to tokens. These tokens are created from every start and end HTML tag in the file. So these tokens have different properties depending on the HTML tag, a $\langle a \rangle$ tag token has different traits compared to the token of an $\langle html \rangle$ tag.

After the tokens are created they are then in turn converted into nodes. The nodes can be seen as a separate entity in the document object model tree. The time that the creation of the DOM tree takes depends on the size of the HTML file. This DOM tree is accessible via the DOM API. In the DOM API you can use an object, property or method that is in production, for instance HTML tags and CSS styles have their handles in the DOM API. This tree is built from the top down incrementally.

As soon as the first link tag to a CSS style sheet in the HTML the browser engine sends a request to fetch the style sheet. The browser engine now also receives raw bytes of CSS data. The same conversion of raw bytes into characters and then tokens as in the construction of the DOM tree occurs, which in turn is converted into nodes in a tree which is built concurrently with the DOM tree. This is called the CSS object model (CSSOM). The CSSOM does not have any way to access it like the DOM API has, it is kept hidden from the user. As CSS has inheritance where the styles may come from a parent element, the structure of the tree is important due to the browser engine having to recursively go through the tree to determine what styles affects a certain element. This means that the CSSOM tree can not be built incrementally as a CSS rule in the bottom can overrule one that occur higher up in the tree structure.

Now there are two independent trees, the CSSOM and the DOM tree, which respectively contain the information of how the different elements should be styled and the relationships of the HTML elements. By combining these two trees the render tree is created. But as already mentioned, CSSOM does not have an API to access it so the browser exposes the CSSOM node of a DOM element by providing a high level API

that works on the DOM element itself. This means that the developer can access and alter the CSS properties of a CSSOM node.

On top of CSS and HTML the browser engine also handles JavaScript. JavaScript can change the way elements of both DOM and CSSOM are rendered. What this means for DOM is that when the HTML parser encounters JavaScript it executes it before continuing with the parsing. Though, for CSSOM the JavaSript is not executed before the CSS renderer is done with CSSOM [10].

2.4 The Browser Engine Render Tree

When the browser engine has the render tree which is either constructed by the browser itself in the case of client-side rendering or by the server in the case of server-side rendering, the browser engine can start with the layout of the page [11]. The browser engine starts by calculating the exact location and size of all the objects on the page. When all positions and sizes of the objects are decided the browser engine can start "painting" the screen. It does this by first creating layers for the elements as these can change position, look or geometry in the website. This helps the browser engine when it is performing paint operations during scrolling, rezising the window and it also helps it to paint draw elements correctly in stacking order(along the z-axis). Each layer is then painted separately by the browser engine. Each pixel inside the layer gets the correct border, background color, text etc, stated by the property of each element. Now the browser has different layers in the form of bitmap images that are sent to the GPU which finally draw it on the screen.

The way JavaScript interacts with this rendering process with client-side rendering is that when a $\langle script \rangle$ tag is encountered the construction of the DOM is paused until the entire script has finished executing, this is called script-blocking. This is due to JavaScript's ability to alter both CSSOM and DOM. This in turn also affects the position of your $\langle script \rangle$ tags as you have to make sure that the objects that you are operating on may not have been created if the tag is put in the wrong place. Objects, such as images and videos are not script-blocking so these can be rendered in the background.

This entire path of rendering is called the critical rendering path (CRP) and is one of the most important part to optimise for websites. It starts with the construction of the DOM tree, then the CSSOM tree followed by the render tree, then the layout operation, after that the painting is done and finally the composition which is when the GPU actually shows everything.

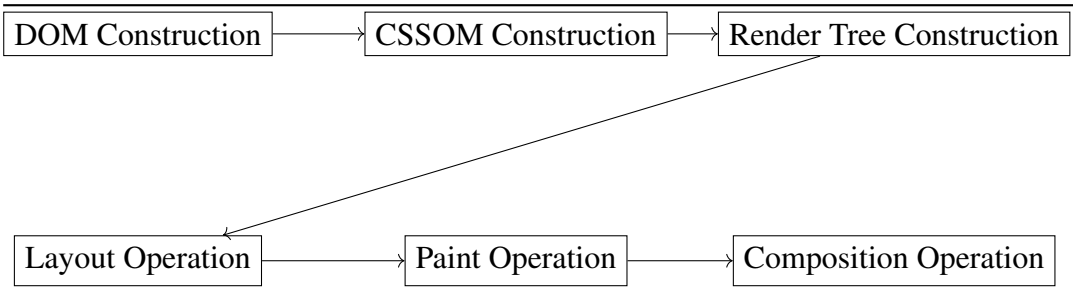


Figure 1 The critical rendering path of CSR, the top part of the figure is not present with SSR.

2.5 Client-Side Rendering

Client-Side rendering is one way to render content in the browser. When using CSR there is no complete HTML telling the browser how to render the content of the page. Instead the server sends an empty HTML document together with JavaScript when the client requests data from the server. The browser engine then starts to download the JavaScript. When the JavaScript is downloaded the browser executes it and from the instructions in the JavaScript data the HTML is populated and displayed in clients browser window [12].

Figure 2 shows the flow of how CSR is executed in the browser. In the example in the image the framework React is used, but other frameworks of JavaScript can be used to achieve the same result.

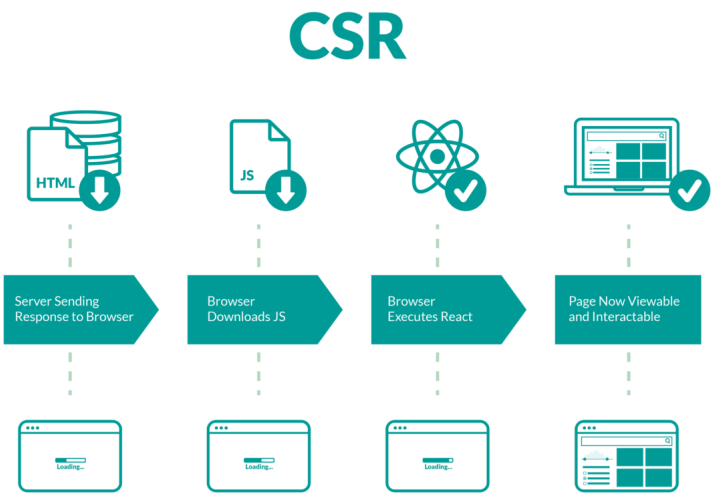


Figure 2 Client-Side Rendering

2.6 Server-Side Rendering

One of the most common ways to render a websites is by using server-side rendering (SSR). When you visit a website that uses SSR, your browser sends a request to the server and then in response it gets back a fully rendered HTML file and display it on the screen. This means that all the JavaScript and all the CSS files are rendered and that you do not have to wait for the browser to load these files before the content is visible. If you try to access a different part of the website the browser sends another request to the server and obtain a new HTML file that it has to display.

When an HTML file is being rendered and it contains JavaScript that it has to fetch, it fetches the JavaScript files to add functionality such as event handlers to buttons, this is called hydration. Hydration is a process that takes a lot of time and resources which can make SSR seem slower when there is a lot of interactivity. SSR helps with search engine optimisation(SEO) due to all the links and texts being loaded by the server which means that it is easily crawled. It is also beneficial for sites such as social media sites where social media crawlers have issues with crawling JavaScript. As there is a higher demand on the server when using SSR, it can be more costly to have a website with SSR compared to CSR and it is not optimal for websites where there are frequent server requests.

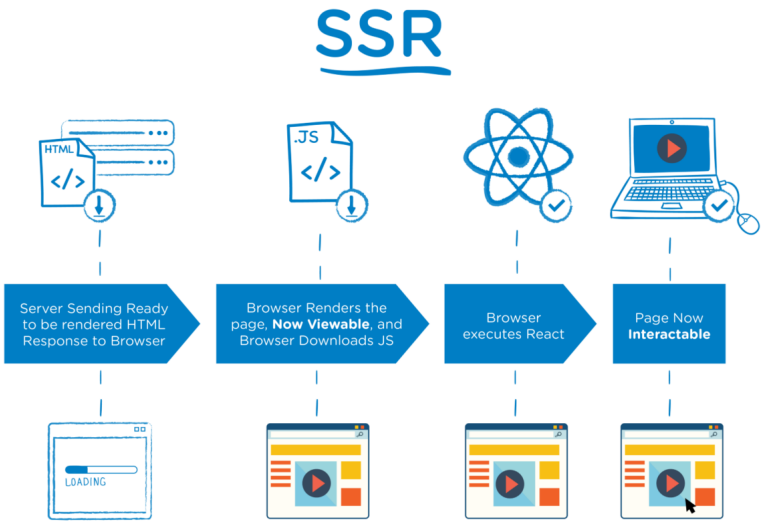


Figure 3 Server-Side Rendering

2.7 Other Rendering Techniques

In this section we describe more rendering techniques beside SSR and CSR.

2.7.1 Static Site Generation

Another method of rendering a website is static site generation (SSG). When using SSG is used the entire website is retrieved at build time and only a single request is sent at the beginning of the visit to the website. This means that when new data is being added to a server, rebuilding the entire application is required. Distributed persistent rendering (DPR) is a form of rendering which allows the developer to defer the rendering of a single page of the website to the first time a user is visiting, and then caches it to the content delivery network (CDN). This means that at build time the most visited pages can be prerendered and less popular pages is built on-demand.

2.7.2 Incremental Static Rendering

There is also a method called incremental static rendering (ISR). With this method the page is rendered at build time and then cached. The cached page is shown when a user makes a request while a new build is started in the background. The invalidation of the cached page is not triggered until the new page is generated. Edge rendering is also a way to render a website. Here ”edge” refers to to the CDN level, and by using this method an already rendered page is getting atomic updates if locale data or personalization is needed.

2.7.3 Deferred Static Rendering

Deferred static rendering is another way to render a website. With this method the developer can choose to defer building certain parts of the the website that is not often accessed by users. This means that when the developer wants to build their site again the parts of the website that is deferred does not have to be built again.

2.8 JavaScript Engine

Every browser has a JavaScript engine that runs the JavaScript of a website [13]. Javascript engines typically use a call stack and a heap where the call stack is where the code is executed and unstructured memory is stored in the heap. JavaScript is used to only be a interpreted language meaning that an interpreter runs the JavaScript and executes it line by line. Compilation is the alternative where a compiler takes the source code and turn it into machine code and then it is executed by the CPU.

Nowadays JavaScript is run with a mix of these two called Just-In-Time compilation. The JavaScript engine takes the JavaScript and parses it and creates a data structure called the abstract data tree. The abstract data tree is used to to convert the JavaScript into machine code. The machine code is run in the JavaScript engines call stack stack right away. The code is first run with an optimised version of the machine code and then while it is running it is being optimised and recompiled in the background.

Browsers have their own JavaScript engines which may have small differences but generally they work in the same way. Google Chrome uses their V8 engine, Microsoft Edge uses Chakra, Mozilla Firefox uses Spider Monkey and Safari uses JavaScript Core Webkit.

2.9 Different Browser Engines

Browsers have their own browser engine, Chrome uses Blink, Firefox uses Quantum, safari uses WebKit and so on. The differences between these browser engines lies in more of a high-level context such as how multiple processes in multiple tabs should be handled and which of the older web standards should be supported. The way a browser renders text is not standardised so there is no guarantee that all browsers works in the same way but there are some common principles which all browser engines adhere to. The way to parse the HTML code to the DOM tree which is often called DOM parsing is usually done by the browser engines providing the DOMParser web API to construct the DOM tree from the HTML code. This code is incrementally built into the DOM tree one node at a time from the top of the tree to bottom. This means that the initial data can be shown as soon as possible.

2.10 Rendering Performance Metrics

Here we describe different rendering performance metrics are described and what information they provide the developer of regarding the rendering of a website.

2.10.1 First Contentful Paint (FCP)

In FCP, the time that it takes from the moment the user navigates until the first DOM content has been rendered by the browser is measured (in seconds) [14]. Images, texts, non-white $\langle canvas \rangle$ are all considered as DOM content.

2.10.2 Largest Contentful Paint (LCP)

As the name suggests, LCP measures the time needed to render the largest element of the website [15]. The time considered to be a good score is 2.5 seconds by google. The size that is taken into account is the size that the user can see, so if the element is clipped or has non-visible overflow these parts of the element is ignored. As the largest element of the page might change during the loading time due to the loading usually being done in stages. The first largest element has its loading time measured with a performance entry, if a larger element is loaded a new performance entry is added and it measures the loading time of the new largest element. Some websites add new elements to the DOM when more content become available in later parts of the rendering, when this occurs and the element is the largest of the website the element is counted as the largest and

its loading time is measured. When the user starts to interact with the page the browser stops reporting the new elements due to interactions with the website can alter what is visible to the user. LCP can be used as an approximation of when the main content of a website is rendered.

2.10.3 Total Blocking Time (TBT)

What TBT measures is the total time that it takes from the moment when a user navigates to a page to when the user can interact with the website [16]. Mouse clicking, screen tapping or keyboard presses are all counted as interacting with the website. The way this is measured is by adding the blocking portion (the time when the browser is waiting for other task to finish) of all long tasks (tasks that take more than 50 ms to execute) between FCP and time to interactive. This means that all time exceeding the 50 ms cutoff is added to the TBT.

2.10.4 Cumulative Layout Shift (CLS)

When content on a website moves during the loading time of a website it is known as a layout shift [17]. This may be due to content being loaded asynchronously or that DOM elements being added on top of already existing components. What CLS measures is the largest burst of layout shifts (is when one or more individual layout shifts occur with less than 1-second in between each shift) that occurs in a session window (maximum of 5 seconds). This is a metric where you want the result to be as low as possible, 0.1 is seen as a good score.

2.10.5 Speed Index (SI)

What SI measures is the time that it takes to visually display the content of the website during a page load [18]. It is measured by first capturing a video of the loading of the website and computing the progress of the visual content being added each frame.

2.10.6 Time to Interactive (TTI)

TTI measures is the time that it takes for a web page to become fully interactive, thus, the site displays useful content which FCP is the used for, the the page responds to the interactions of user within 50 milliseconds and lastly when for most event handlers on

the page are registered [19]. A time less the 3.8 seconds is seen as a good time. A slow TTI is usually due to poorly optimised JavaScript work.

2.11 The History of Web Rendering

The first web page was created in 1991. It was a simple page with very little styling. It consisted of only text and hyperlinks. For the first years of web rendering the web pages only contained static content, i.e. content that did not change or were interactive. This meant that the page was downloaded once and after that no communication was needed with the server [20].

2.12 JavaScript Frameworks

The first years of the web was built using only HTML, shortly after CSS and JavaScript was introduced. There was some problems regarding this, since everything was new there existed no standardisation on how data should be delivered to the browsers. This resulted in years of trying to find a standard for HTML, but foremost CSS and JavaScript. In the early 2000s different JavaScript frameworks were built and one of the most common JavaScript frameworks, JQuery was introduced [21]. It helped modify the the HTML and CSS as well as the DOM, and made it easier to use for example animations [22].

In the 2010s more advanced JavaScript frameworks started to emerge. In 2010 the first version Angular was released which was the first largely used framework for web projects [23]. It helped developers develop more complex websites in less time. But as web projects increased in size this version of Angular was not filling the needs of the developers and the complexity of building websites grew.

Soon after the release of Angular other JavaScript frameworks were released. Two of them were React and Vue [24, 25]. These are two of the most used JavaScript frameworks today. React helped developers build web interfaces by dividing parts of the interface in to smaller components, which made the development quicker. With the usage of React it also became easier to write components that changed frequently. Vue on the other hand is a framework that was built by Evan You who was inspired by concepts in Angular when he created it [26].

3 Related Work

In this section scientific papers that relate to this paper are discussed.

3.1 Comparison Between Client-Side and Server-Side Rendering in the Web Development

In the project Iskandar et al. [27] benchmark testing between CSR and SSR is provided. The test parameters are first content paint, speed index, time to interactive, first meaningful paint, first idle CPU and estimated input latency. For all these parameters except the estimated input latency the speedup using SSR was between 60% and 133%. The input latency was the same. They came to the conclusion that SEO was improved when using SSR compared to CSR but that the accessibility was better with CSR.

The differences between our project and the study by Iskandar et al. is the way that the websites that are implemented, and the types and the different layouts of the pages that are tested. In this project React and the web framework Next.js are used to make the website client-side rendered or server-side rendered, whereas it is not specified how Iskandar et al. achieved it. The layout of the page where Iskandar et al. tested their rendering performance on was on a simple login page with the test case of wrong password being input. In contrast our testing is done on multiple different pages where there are more different types of input and a higher amount of elements to be rendered, resembling what is expected from a normal website.

3.2 A Hybrid Web Rendering Framework on Cloud

In this paper by Kumar et al. [28] two methods for a hybrid web rendering engine is presented where the rendering is split between the server and the client. With the idea to improve the rendering of web browsers that have cloud assisted rendering such as Opera Mini and UC Mini. The rendering is assisted by a cloud server which provides a more lightweight version of the HTML file to the client which helps the process by making it faster and saves on the bandwidth for the client. This way of rendering usually makes the rendering a lower quality than if it was only to be done by the client. In their solution the pixel by pixel rendering is identical to that of a more traditional way of rendering. The main purpose of this architecture is to offload the rendering burden of clients with lacking hardware or low bandwidth.

The first of the two methods generates a layer tree file at the server which contains information about the different HTML layers. In the second method the authors generate a render file that contains the drawing commands instead of the layer tree file. Their conclusion regarding the different methods is that the first method generates a larger file than the second method but that it has a higher rendering accuracy, leading to a trade off between bandwidth and rendering accuracy.

Similarly to our project they also did performance comparisons of their different rendering methods but their focus was more towards the size of the file and power usage of the user.

3.3 Measuring Web Latency and Rendering Performance: Method, Tools, and Longitudinal Dataset

Asrese et al [29] presents a tool for measuring Web Quality of Service metrics such as, the download time, the DNS lookup time and the time to first byte called WePR. It can also capture Web complexity metrics, which includes the size and number of objects that make up a website. Their testing shows that the websites of Google, YouTube and Facebook all improved their DNS lookup time from January 2014 to July 2017. But only Google improved their download time and time to first byte. In their tests the rendering time was faster in 80% of the time compared to the page load time. In regards to performance their testing show that the size and amount of objects and the number of scripts are big factors when it comes to performance. The physical location of the server also affects the rendering time, where a server with a physical location that is farther away has a longer rendering time.

They also developed a tool that measured the same things but also tested metrics such as rendering times, which they described as Web Quality of Experience.

3.4 A Performance Comparative on Most Popular Internet Web Browsers

The performance of different web browsers are tested in this paper by Mohammed et al [30]. The authors compare five common web browsers, Google Chrome, Mozilla Firefox, Microsoft Edge, Opera, and Brave. They compare the GPU, CPU and RAM usage using tools such as Speedometer and MotionMark. Their testing conclude that Chrome utilised memory and CPU more and that their GPU usage was lower. Microsoft Edge had the reverse relationship with using the GPU more but using the CPU

and memory less. When the authors tested the speed of the different browsers Microsoft Edge had the best results but as Chrome utilises the high hardware of the user better, the speed depends on the specifications of the computer that is used when testing.

Just like in our project performance testing on the web is conducted but the work by Mohammed et al. looks at different browsers instead of how the websites are implemented. The metrics used in Mohammed et al. measures the usage of GPU, CPU and RAM and therefore different testing tools are used compared to this project.

4 Method

In this section we describe different tools for browser rendering performance and the different test design layouts that were used. The last part of the section describes how the tests were conducted.

4.1 Tools for Testing Rendering Performance

We describe different tools for testing the rendering performance of a web page, discuss, their advantages and disadvantages and explain the reason for our choice of testing tools.

4.1.1 Lighthouse

Lighthouse is an automated tool that evaluates a website [31]. The tool is owned by Google and is open source. What lighthouse does is that it audits a web page on a website and gives you a number ranging from 1 to 100 on five different metrics which are performance, SEO, accessibility, best practises and progressive web apps. It not only rates a website but it also gives suggestions on how to improve the website. The performance is measured by five different performance metrics for web pages, first contentful paint (FCP), largest contentful paint (LCP), total blocking time (TBT), cumulative layout shift (CLS) and speed index(SI). Lighthouse is available for free in the Google Chrome web browser.

4.1.2 Chrome DevTools

Chrome DevTools is a set of tools that is built into the Chrome web browser which can provide the developer with information regarding the performance of a website [32]. The DevTools are comprised of a set of seven different panels, the elements panel, the console panel, the sources panel, the network panel, the performance panel, the memory panel and the application panel. The one that is of interest to us is the performance panel, where you can see how quickly the page is loading and rendering, and the network panel which shows what resources are loaded and the time that it takes for the resources to load.

Performance Insights which is of the most importance in this project is a part of Google DevTools [33]. Performance Insights can record the screen of the user that is operating the browser and measures values such as FCP and LCP. The recording may be played back again to look at it again, to see what parts of the page that is loading at what time.

Chrome DevTools also has more features that are useful when testing our website, such as the device mode which lets you test the the performance and the way the website is presented on a simulation of the device of the developers choice. There is also a feature called Network Throttling, which lets the developer change the speed of the network and to test how the website works for users with differing network speeds. These features can be used together with Performance Insights to record the screen when throttling is used.

4.1.3 GTmetrix

GTmetrix is another tool for testing the performance of a web page. Similarly to Lighthouse it measures FCP, LCP, TBT, SI and CLS [34]. GTmetrix further allows you to see how long time each resource takes to load and also see a breakdown of why it takes that long time to load, due slow server response times, large images, or poorly optimised code. There is also a feature which allows you to see how your website performs over time.

4.1.4 Testing Tools That Were Used

Lighthouse and Chrome DevTools Performance Insights were used to assess our system. The reason for choosing these two is that they give the ability to measure metrics that are good indications for the rendering performance of a website such as FCP, LCP and SI. They also give the ability to throttle the network speed of the testing environment.

Another reason behind the choice of these tools is that they are integrated in the Chrome DevTools which comes with the Google Chrome browser, and they are commonly used by developers to analyse page speed.

4.2 Web Frameworks

There are many different frameworks used to develop websites. When deciding which to use for this system, the performance was taken into consideration, and secondly the support and documentation as well as and number of users. Frameworks that suited the needs of the system were Gatsby, Next and Flutter. The latter being less common and released two years ago, has less support for SSR and does not currently support SSG. Gatsby on the other hand is a web framework for creating statically generated websites (SSG) and not server side generated (SSR). We decided to use Next.Js to be able to compare rendering speeds using SSR and CSR.

4.3 The different Test Design Layouts

In this section we describe the way the tests used to compare rendering techniques are built and conducted. The creation of the JavaScript tests, the image tests and the API tests is also described.

4.3.1 Javascript

To simulate execution of JavaScript a for-loop has been used as shown in Figure 4. The loop is iterated a billion times and for each iteration in the loop the number of the current iteration is multiplied by itself three times before being added to the return value.

The reason for choosing to use this test for JavaScript is because it is a heavy task for the CPU and, therefore, we can see the differences in rendering speed when using different rendering techniques, networks speeds and hardware.

4.3.2 Images

To test how the rendering performance were affected by different amounts of data being sent and rendered different sizes of image data were used. Image files of JPEG format and images of a combined size ranging from 500 kB to 5000 kB were used.

```
const javaScriptTest = () => {
  let returnValue = 0;
  for (let i = 0; i < 1000000000; i++) {
    returnValue = i * i * i;
  }
  return returnValue;
};
```

Figure 4 The JavaScript function used in out tests

4.3.3 API

When performing the API tests, wanted to see how different amounts of API calls to the API affected the performance of CSR and SSR. To test this the API, APIS.GURU was used, which is a Wikipedia for public WEB APIs. [35].

4.4 Next.js

As mentioned in Section 4.2 this project is built and tested using Next.js framework. Next.js is a framework built upon React written using JavaScript or Typescript. Next.js contains tools that help the developer to create a website with tools that can help to increase the rendering speed of the web page. The feature that is specifically looked at in this project is Next.js ability to use different rendering techniques. Small changes can be done to the code to tell Next.js in what way the page should be rendered. In this project version 13 of Next.js has been used. [36].

In Next.js the components, which can be pages or small parts of pages are by default server side rendered. A server component can use either SSG or SSR. Changing between the two is made in the function where data is being fetched. The default is to use SSR while calling the method which fetches data from an API. To make a server component static the following can be added to the fetch function call: { cache: 'force-cache' }. To make a component rendered on the client side with CSR, the following line of code can be added at the top of the file, ("use client";)

With Next.js you can divide a page into smaller or larger components. One component can for example consist of of only a single row of text up to a whole page consisting of

e.g. buttons, texts and images [37].

Different components can be rendered in different ways. Interactive components need to be client side rendered to work, so when testing our system using SSR, interactive components, e.g. buttons and inputs, have been made into leaf components such that only those parts of the page is rendered using CSR.

4.5 Compared Metrics

The metrics that were chosen for assessing the rendering performance of our website are FCP, LCP and SI. The reason for choosing FCP is that with using this metric is that when a website starts loading it is the first indicator for the user that a website is responsive. The difference in FCP between SSR and CSR is the difference in time that it takes for the server having to render the entire page and sending the larger HTML file to the client and the time it takes for the the server to send a smaller HTML file and the client having to render the file.

LCP was chosen due to it being the metric that tells when the largest component is done rendering, which is usually the main component on a web page. The time measured by LCP is also usually the same as the time it take for the entire page to load due to the largest content often taking the longest time to load.

When using Lighthouse the SI metric was used as well. This metric gives the time to display the content of the web page

4.6 Testing Variables

The tests are conducted on web pages containing varying amounts of image elements, API-calls and also on a JavaScript heavy function. This is done to have tests which can be compared with a wider variety of website designs. The tests were also run with different internet speeds, first with a normal level of bandwidth and request latency then with a reduced bandwidth of 1.44 megabits per second and an added 560 milliseconds of request delay and then further ran with a bandwidth of 400 kilobits per second and an added request latency of two seconds. The two latter options are what Google DevTools define as *fast 3G* and *slow 3G* respectively. This is to test for user with a varying degree of internet speeds. The tests were also ran with a 6x CPU slowdown which makes the CPU run 6 times slower than usual.

4.7 Conducting the Tests

Both Lighthouse and the Performance Insights tab from Google Dev Tools were used when conducting the tests, as they are among the most commonly used tools. As mentioned in Section 4.1.1, Lighthouse measures many different metrics, and the ones that were looked at was FCP, LCP and Speed Index. Performance Insights gives us similar metrics, and the ones that were chosen to examine were FCP and LCP, which could be compared with the Lighthouse results of the same metric.

Next js was used to select which rendering method was used for the different tests. As mentioned in Section 4.4 Next.js has built-in support for switching between different rendering methods.

A typical test setup would look like this: SSR would be used with no throttle on a web page with 500 kB of image data using Lighthouse to gather the performance metrics. The test would be ran five times to ensure that the result were consistent and the average of these test was saved as our test result.

5 Architecture

The architecture of the website that was used when doing the performance test is described in the first part of this section. Then the technical conditions of the testing is described.

5.1 Web Page Architecture

The web page that is used for the testing is built with Next.Js. The base design of the web page that is used in the testing consists of some small text fields and a button. This page is then altered between different tests to display for example images, JavaScript heavy calculations or API responses as mention in Section 4.3.

5.2 Technical Testing Environment

The tests were conducted on a MacBook Air M1, 2020 with an Apple M1 processor, using 8 GB of RAM and running on the operative system macOS Ventura 13.3.1. It was tested on Google Chrome Version 114.0.5735.198 (Official version) (arm64)

The NEXT.js server and the client browser were running on the local MacBook machine when executing the tests.

6 Results

In this section we present the results from the conducted tests. The tests are run five times and the average of these is the final result.

6.1 JavaScript Test Results

These are the results from the JavaScript tests that are described in Section 4.3.1

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.8 (0.09)	2.1 (0.06)	2.5 (0.1)	1.8 (0.09)
LCP	4.8 (0.06)	4.8 (0.06)	5.3 (0.09)	5.6 (0.09)
SI	4.1 (0)	4.3 (0)	4.8 (0.04)	4.9 (0.04)

Figure 5 JavaScript test results using CSR (lighthouse). The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.9 (0.05)	2.1 (0)	2.6 (0.04)	1.9 (0)
LCP	1.9 (0.05)	2.1 (0.04)	2.6 (0)	1.9 (0)
SI	1.9 (0.04)	2.2 (0.05)	2.7 (0.07)	1.9 (0)

Figure 6 JavaScript test results using SSR (lighthouse). The results are in seconds. The standard deviation is in parenthesis.

The test results when running the tests with Google Lighthouse can be seen in Figures 5 and 6. The difference between CSR and SSR when comparing the FCP is very small, only 0.1 seconds. The difference in LCP and SI is larger, ranging between 2.1 and 2.9 seconds when comparing different internet speed. When comparing with the CPU throttling the result were similar with the FCP being within 0.1 seconds of each other and the LCP and SI having a a difference of 3 to 3.6 seconds.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.08 (0.006)	1.8 (0.1)	6.2 (0.07)	0.12 (0.006)
LCP	0.6 (0.06)	3.1(0.1)	9.4 (0.1)	3.3 (0.1)

Figure 7 JavaScript test results using CSR (Performance Insights). The results are in seconds.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.08 (0.004)	1.78 (0.004)	6.2 (0.06)	0.36 (0.004)
LCP	0.08 (0.004)	1.78 (0.004)	6.2 (0.06)	0.36 (0.004)

Figure 8 JavaScript test results using SSR (Performance Insights). The results are in seconds.

The test results when testing with Google DevTools Performance Insights can be seen in Figures 7 and 8. The results for the LCP show that when using no throttling the difference in LCP and FCP is 0.5 seconds and the difference increases as slower internet speeds are used. At *slow 3G* the difference between LCP and FCP is 3.2 seconds. When looking at the CPU throttle our tests show that there is a difference of 0.24 seconds looking at the FCP, where CSR is faster. Though when looking at LCP the SSR is 3 seconds faster than when using CSR.

6.2 Image Test Results

Here we present the results from the JavaScript tests that are described in Section 4.3.2. The bar charts show how much faster the LCP is when using SSR compared to CSR. When using Lighthouse the difference compared to Google DevTools Performance Insights is that SI is also shown. The FCP was very similar between different tests as well as the standard deviation being less than 10 percent as can be seen in the Appendix 11, so the FCP and the standard deviation are omitted in these graphs.

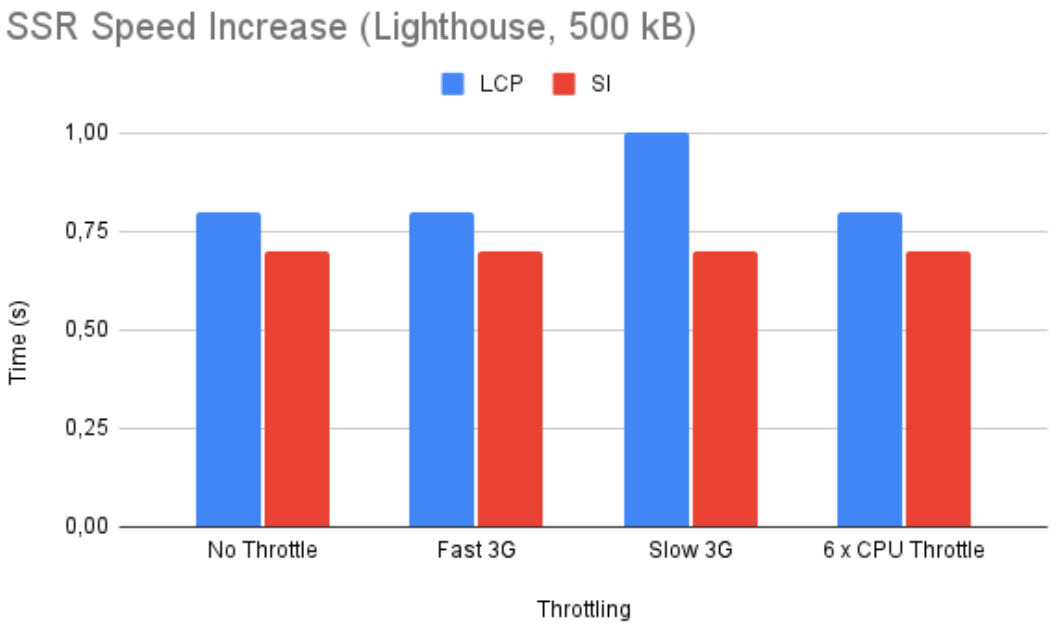


Figure 9 Image test results using 500 kB of image data (lighthouse).

The test results for the image test when using Lighthouse and 500 kB of image data can be seen in Figure 9. When looking at the results, the speed increase using SSR instead of CSR is quite consistent. The speed increase in seconds when using SSR observing LCP is ranging between 0.75 and 1 seconds. The speed increase is largest when using a *slow 3G* internet speed. For the Speed index value from Lighthouse the increase is consistent across the the different internet speeds as well as for the CPU slowdown test.

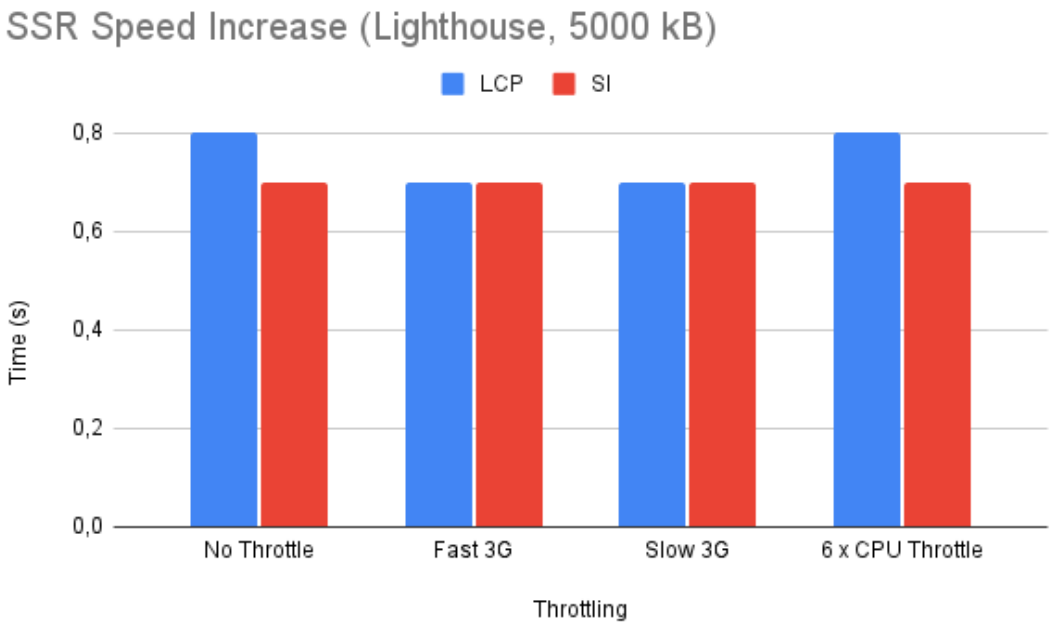


Figure 10 Image test results using 5000 kB of image data (lighthouse).

In Figure 10 a speed increase for using SSR compared to CSR using Ligthouse measurements can be observed. The results show that the increase in LCP is fairly consistent for all the internet speeds as well as for the CPU throttle. The speed increase for LCP using SSR is ranging between 0.7 to 0.8 seconds. For the speed index value given by Lighthouse we see that its value is increased by 0.7 not depending on the internet speed or CPU slowdown.

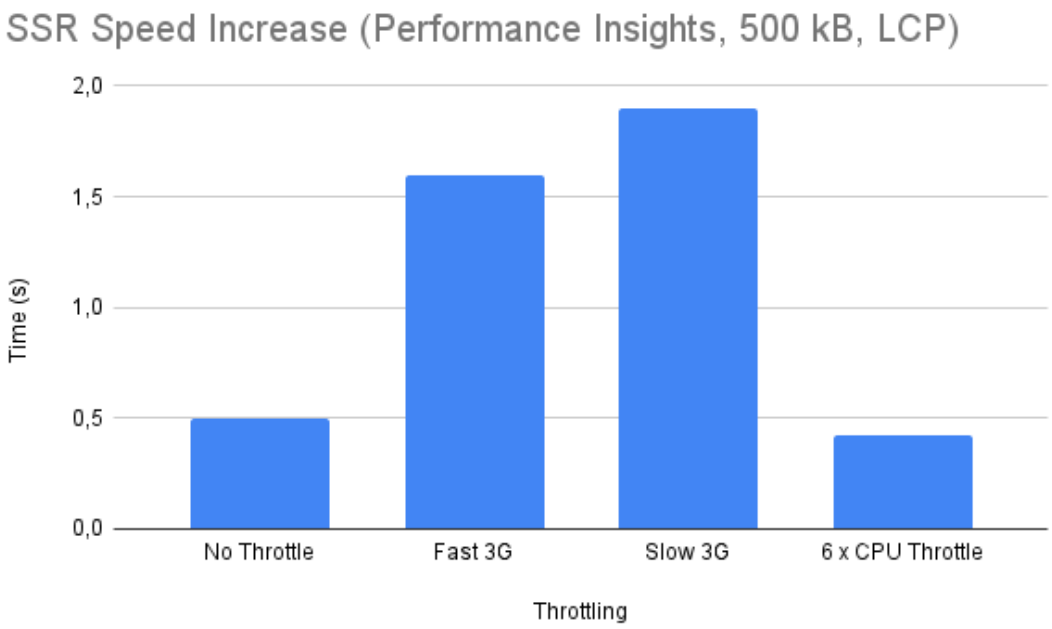


Figure 11 Image Test Results Using 500 kB of image data (Performance Insights).

In Figure 11 the LCP speed increase when using SSR instead of CSR and using image data of 500 kB can be observed. There is a speed increase when using SSR no matter the internet speed or CPU throttle. Though when using the internet speeds *slow 3G* or *fast 3G* there is a larger increase compared to no throttle in internet speed. The speed increase when using internet throttle is about 1.5 seconds when using *fast 3G* and 1.9 seconds when using *slow 3G*, and for no internet throttle and the CPU slowdown measurements there is between 0.4 and 0.5 seconds speed increase when using SSR instead of CSR.

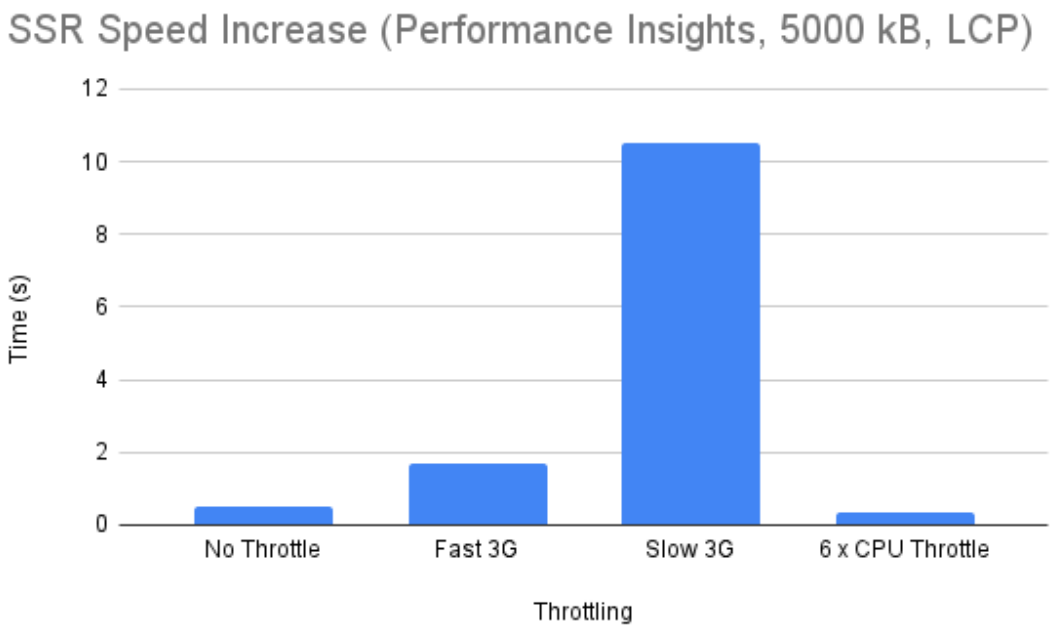


Figure 12 Image Test Results Using 5000 kB of image data (Performance Insights).

From the results in Figure 12 we observe the speed increases when using SSR compared to CSR when looking at LCP and using 5000 kB of image data. For all the tests SSR has better performance. When testing with a slower internet speed the difference between SSR and CSR increases. When using *fast 3G* the increase in speed for SSR is just shy of 2 seconds and when using *slow 3G* the increase in speed for SSR is almost 11 seconds. When using internet speed that is not throttled and with or without CPU throttle the difference is between 0.36 and 0.49 seconds.

6.3 API Test Results

We shoe next the results from the JavaScript tests, described in Section 4.3.3.

For the API tests the figures show the results of the Performance Insights when comparing LCP between CSR and SSR. There is also a diagram for no throttle with Lighthouse measurements. Each figure shows the comparison between SSR and CSR when performing different amounts of API calls for different internet speeds and a 6 x CPU slowdown. The number of API calls that were tested are 10, 20, 50 and 100, though the results are displayed in line diagrams created from the test results that were collected

which can be seen in the Appendix 11. As the difference in FCP was small and the standard deviation was very low they are omitted from these graphs, the exact numbers can be found in the Appendix 11.

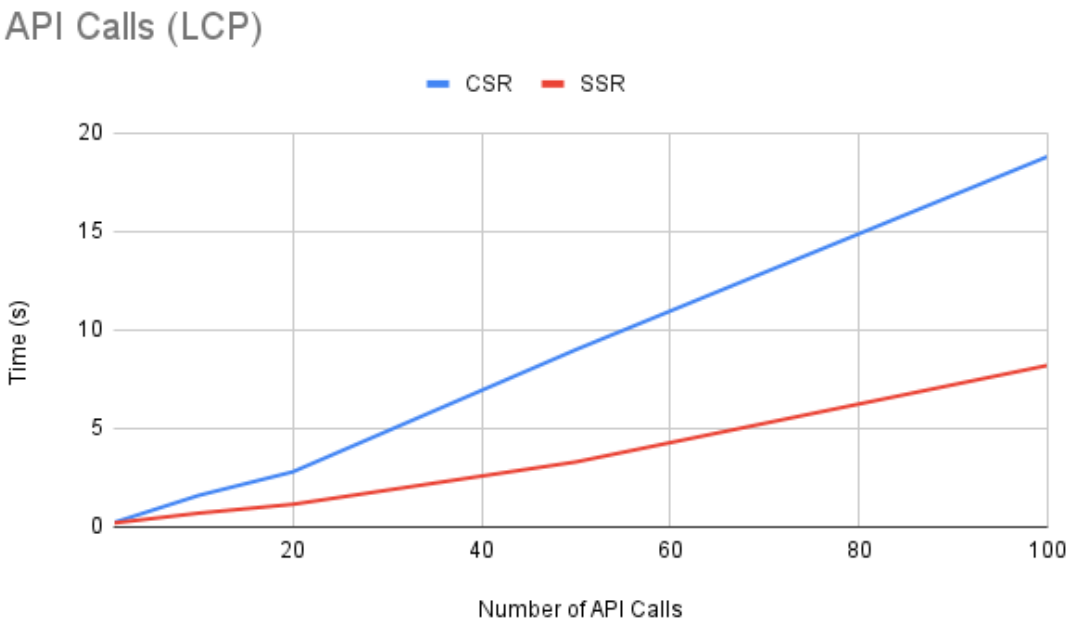


Figure 13 LCP for different amount of API calls for SSR and CSR (no throttle).

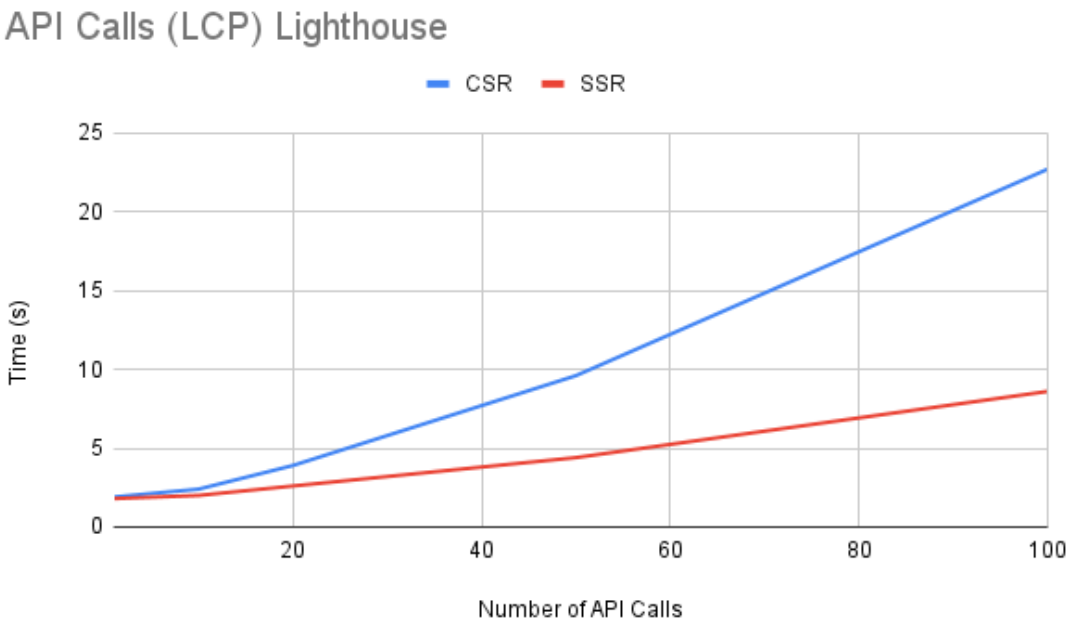


Figure 14 LCP for different amount of API calls for SSR and CSR in Lighthouse

Figure 13 and 14 shows that it is faster to do the API calls on the server side using SSR. Though for a single API call there is near to no difference, but as the amount of API calls increases the difference when looking at LCP is increasing. Figure 13 and 14 show how long time it takes for SSR and CSR for different number of API calls. The same trends can be seen when comparing Performance Insights and Lighthouse. Though the difference between the two performance testing tools is that the results from Lighthouse are for all the values a couple of seconds slower.

When looking at the slower internet speeds Figure 15 and 16 shows that the for few API calls there is just a small difference in LCP, but as we reach 50 API calls the difference is around 3 to 4 seconds. As the number of calls increases the rendering techniques seem to stay close in terms of time, with a slight increase in LCP when using CSR.

Figure 17 shows the results of LCP difference between CSR and SSR when using the 6 x CPU slowdown. The trend of the LCP difference as the number of API calls increases are similar to the ones without CPU slowdown. Though, when comparing the results without throttling a speed up when using CPU throttle is seen.

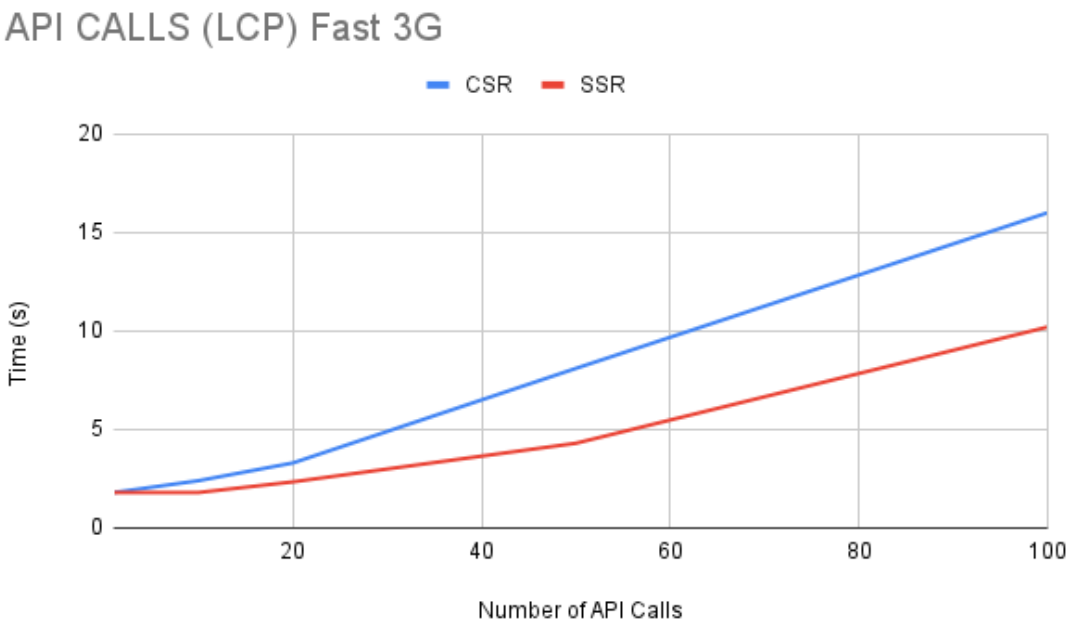


Figure 15 LCP for different amount of API calls for SSR and CSR (Fast 3G)

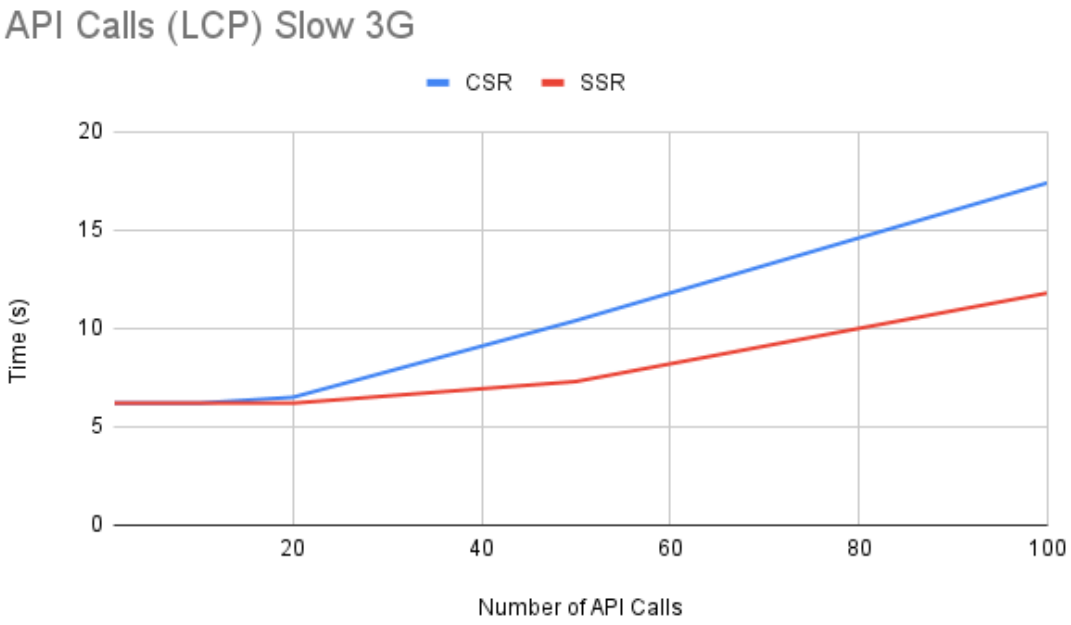


Figure 16 LCP for different amount of API calls for SSR and CSR (slow 3g)

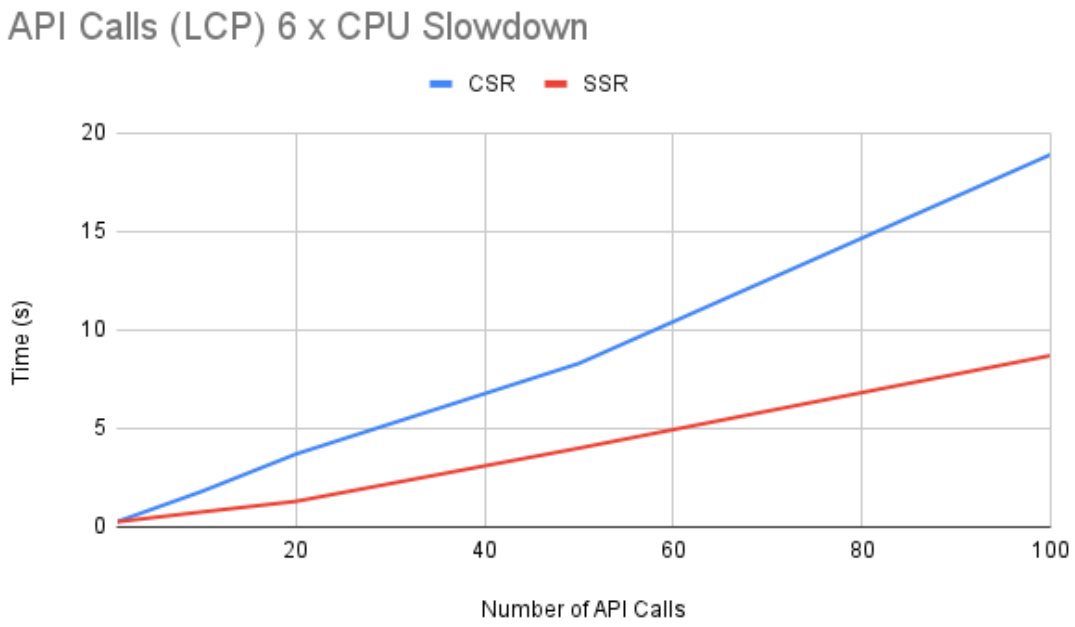


Figure 17 LCP for different amount of API calls for SSR and CSR (6 x CPU Slowdown)

7 Discussion

When considering the results from Google Lighthouse and comparing those with what was observed when loading the website there was a big discrepancy, but a consistent trend of worse results when using slower internet speed or a layout with a higher complexity could still be seen. When using Google DevTools Performance Insights the results lined up more closely to what the human eye observed. In this discussion the focus is mainly on the results from performance insights, since they are more close to what the user experience would be like.

In this project performance testing has been done comparing SSR and CSR on three different types of layouts consisting of API calls, CPU heavy tasks or differing images data sizes. The tests that were done by adding images to the web page show that when the total size of the images is small, the choice of rendering technique does not have a large impact on when the page is done rendering, only affecting the LCP by 0.4 seconds slower using CSR without throttling the internet speed. The reason that CSR is slower might be due to CSR having to execute the Next.js JavaScript before it is viewable to

the user. Though, while using SSR the HTML that is being sent to client is already viewable once it is downloaded and processed by the client browser.

Testing with a worse bandwidth and with a slowdown on the CPU also did not have a significant impact with a lower amount of images. The results gathered using *slow 3G* show that the time increase for LCP is 1.9 seconds when using CSR instead of SSR. Though the LCP is 15.7 seconds for the SSR, thus the extra 1.9 seconds is not a major time saver. When the size of the images are larger the performance difference of the rendering techniques got bigger. When lowering the bandwidth to *slow 3G* the LCP decreases by over 10 seconds when SSR is used compared to CSR. This may be due to when CSR has to fetch the images from the server it takes a longer time when it has to send more requests to the server. The server on the other hand is in our case hosted at same location as the image files which makes the request for images when using SSR faster.

When doing the image tests with a six time reduction of the CPU, the performance stayed quite similar to the no throttling tests. Though when using SSR there is a slight time increase in the results. A reason for why the CPU slowdown did not impact the rendering that much can be because the CPU we did the tests on still had enough processing power after the slowdown to still efficiently render the web page.

When looking at the JavaScript tests it shows that SSR is faster than CSR for all the network speeds as well as when using CPU throttle. The reason may be that when executing the JavaScript task when using CSR the calculation is done after the data is sent from the server, and when SSR is used the calculation is done at build time and then stored just to access statically when rendering. When using CPU throttle the CSR is a bit over two seconds slower, which is what is expected since there is many computations that require processing power.

The API testing show that there is a linear increase of LCP when the number of of API calls rise when using no throttle. The difference in LCP is almost non-existent when looking at a single API call. With an added internet slowdown the linear increase of LCP is also apparent but starts with a higher amount of API calls, starting at 20 API calls. The difference between CSR and SSR is very small for up to 20 API calls when using slow 3G.

From the results it can be seen that when adding another API call a constant time is being added to the LCP, though the time is less for SSR than CSR it is still constant for both techniques. This means that the Next.js server handles each new API call faster than the client browser.

SSR is usually used when a website does not have a lot of user interaction where the page is not changing frequently, since it requires additional communication with the

server. When there is a low amount of users on the website SSR is preferred due to the load on the server being easily handled. CSR is usually used in websites where there is a lot of user interactions such as social media where many updates are sent. As these two cases are true for our tests it means that SSR performing better is the expected result. Even though the results are expected they tell how much the rendering techniques differ and that the difference is small with modern computers and high quality internet.

To make the tests less skewed, more tests could have been conducted. For example, tests with more interactive elements or the usage of other metrics like *time to interactive* could have been used.

In our tests we are not taking in to consideration that when using SSR the load on the server might affect the rendering performance. If there are more users than the server can handle in an effective way there can be a bottleneck that negatively affects the performance. In contrast the server load is smaller when using CSR as the server does not have to render the page.

8 Conclusions

The tests show that SSR has a faster LCP compared to CSR in all cases regardless of the type of tests that are executed and using different internet speeds or CPU slowdown was not an exception to this result. Without throttling the differences are small between CSR and SSR. But when adding a network slowdown with a website layout that has a lot of complexity such as image data, the differences between SSR and CSR grows larger where SSR is the better performer. This means that when choosing between the two different rendering techniques the choice comes down to the network speed of the users of the website and the complexity of the website.

The tests show that when using many API calls the Next.js server handle the calls quicker than the browser does but when there is only a single call or just a few the performance difference is small. This means that SSR is to be preferred when multiple API calls has to be executed on a website.

When implementing a website in Next.js 13 the complexity and time that it takes to develop the site may increase when using SSR. The reason behind this is because you need to divide the interactive components with the rest to make it work. This means that you need to have a more complex implementation in mind when developing a website using SSR compared to if you were to build everything in CSR. While using CSR everything can be built in the same component or view.

In general when the internet speed is getting slower the usage of SSR can clearly be seen to be the better option. In the tests this is when we use a bandwidth of 1.44 megabits per seconds and an added 560 milliseconds of request delay (fast 3G), the LCP is reduced by 0.75 up to 1.4 seconds when using SSR.

9 Future Work

For future work, testing on different types of devices such as smart phones and tablets would be a something of interest to study. Comparing the ease of implementation between SSR and CSR with different front-end frameworks is also a variable that could be explored. This could perhaps be done by a user survey or analysing the amount of code needed to write for the different implementations.

The energy efficiency of CSR and SSR could be compared with the different loads of the servers and the clients devices.

There are also more metrics that we did not include in this project that could be interesting to look at as well, for example the time it takes for a web page to be interactive.

References

[1]

[2] "Average time spent per day with digital media in the United States from 2011 to 2024." https://www.statista.com/statistics/262340/daily-time-spent-with-digital-media-according-to-us-consumsers/, 2023. Accessed: 2023-02-13.

[3] D. An and P. Meenan, "Why marketers should care about mobile page speed," 2016.

[4] J. Wagner, *Web Performance in Action: Building Fast Web Pages*. Simon and Schuster, 2016.

[5] D. F. Galletta, R. Henry, S. McCoy, and P. Polak, "Hello world," *Journal of Silly Statements*, 2016.

[6] "How loading time affects your bottom line." https://blog.kissmetrics.com/wp-content/uploads/2011/04/loading-time.pdf. Accessed: 2023-02-13.

[7] J. C. Mogul, "The case for persistent-connection http," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, (New York, NY, USA), p. 299–313, Association for Computing Machinery, 1995.

[8] W. Stevens, "Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," tech. rep., 1997.

[9] S. Konger, "How browsers work — the rendering engine," 2023. Accessed 2023-07-15.

[10] "Populating the page: how browsers works," 2023. Accessed 2023-07-21.

[11] "How browser rendering works — behind the scenes," 2021. Accessed 2023-07-21.

[12] "Client-side rendering vs. server-side rendering: which one is better," 2020. Accessed 2023-07-21.

[13] A. Yaduvanshi, "Inside the javascript engine," 2022. Accessed 2023-07-20.

[14] Google, "First contentful paint," 2016. Accessed 2023-04-24.

[15] Google, "Largest contentful paint," 2020. Accessed 2023-04-24.

[16] Google, "Total blocking time," 2019. Accessed 2023-04-24.

[17] P. Walton and M. Mihajlija, "Cumulative layout shift," 2019. Accessed 2023-04-24.

[18] Google, "Speed index," 2019. Accessed 2023-04-24.

[19] Google, "Time to interactive," 2019. Accessed 2023-05-11.

[20] D. Shivalingaiah and U. Naik, "Comparative study of web 1.0, web 2.0 and web 3.0," 2008.

[21] "jquuery," 2023. Accessed 2023-09-07.

[22] M. Wanyoike, "History of front-end frameworks," 2018. Accessed 2023-04-29.

[23] "Angular," 2023. Accessed 2023-09-07.

[24] "React," 2023. Accessed 2023-09-07.

[25] "Vue," 2023. Accessed 2023-09-07.

[26] M. Levlin, "Dom benchmark comparison of the front-end javascript frameworks react, angular, vue, and svelte," 2020.

[27] T. F. Iskandar, M. Lubis, T. F. Kusumasari, and A. R. Lubis, "Comparison between client-side and server-side rendering in the web development," *IOP Conference Series: Materials Science and Engineering*, vol. 801, p. 012136, may 2020.

[28] N. K. SG, P. K. Madugundu, J. Bose, and S. C. S. Mogali, "A hybrid web rendering framework on cloud," in *2016 IEEE International Conference on Web Services (ICWS)*, pp. 602–608, 2016.

[29] A. S. Asrese, S. J. Eravuchira, V. Bajpai, P. Sarolahti, and J. Ott, "Measuring web latency and rendering performance: Method, tools, and longitudinal dataset," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 535–549, 2019.

[30] A. Mohamed and I. Ismail, "A performance comparative on most popular internet web browsers," *Procedia Computer Science*, vol. 215, pp. 589–597, 2022. 4th International Conference on Innovative Data Communication Technology and Application.

[31] Google, "Lighthouse overview," 2016. Accessed 2023-04-24.

[32] Google, "Chrome devtools," 2015. Accessed 2023-04-25.

[33] "Performance insights: Get actionable insights on your website's performance," 2022. Accessed 2023-08-01.

[34] GTmetrix, "everything-you-need-to-know-about-the-new-gtmetrix-report-powered-by-lighthouse," 2020. Accessed 2023-05-12.

[35] "Apis.guru," 2023. Accessed 2023-07-20.

[36] NEXT.js, "React essentials," 2023. Accessed 2023-05-14.

[37] NEXT.js, "From javascript to react," 2023. Accessed 2023-05-14.

10 Contributions of the Authors

This master thesis has been written by August Dixelius and Carl Nordström. The work has been divided equally between the two authors. Though parts of the the thesis has been divided between the two, there has been a good cooperation between the authors and both have been invested in what the other person is doing. To be updated on the other persons work we have had close contact each day during the writing of this thesis.

11 Appendix

11.1 Test results

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.8 (0.04)	2.1 (0)	2.6 (0)	1.8 (0.04)
LCP	2.6 (0)	2.9 (0)	3.4 (0)	2.7 (0.04)
SI	2.5 (0.04)	2.8 (0)	3.3 (0)	2.6 (0)

Figure 18 Regular CSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.8 (0.04)	2.1 (0.04)	2.6 (0)	1.9 (0)
LCP	1.8 (0.04)	2.1 (0.04)	2.6 (0.06)	1.9 (0.06)
SI	1.9 (0.04)	2.2 (0.04)	2.7 (0.06)	1.9 (0.06)

Figure 19 Regular SSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.1 (0.006)	1.8 (0.06)	6.2 (0.06)	0.36 (0.09)
LCP	0.5 (0)	2.6 (0.12)	8.9 (0.12)	0.52 (0.006)

Figure 20 Regular CSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.06 (0)	1.78 (0)	6.2 (0.04)	0.36(0.004)
LCP	0.06 (0)	1.78 (0.004)	6.2 (0.04)	0.36 (0.004)

Figure 21 Regular SSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.8 (0.09)	2.1 (0.06)	2.5 (0.1)	1.8 (0.09)
LCP	4.8 (0.06)	4.8 (0.06)	5.3 (0.09)	5.6 (0.09)
SI	4.1 (0)	4.3 (0)	4.8 (0.04)	4.9 (0.04)

Figure 22 JavaScript CSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.9 (0.05)	2.1 (0)	2.6 (0.04)	1.9 (0)
LCP	1.9 (0.05)	2.1 (0.04)	2.6 (0)	1.9 (0)
SI	1.9 (0.04)	2.2 (0.05)	2.7 (0.07)	1.9 (0)

Figure 23 JavaScript SSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.08 (0.006)	1.8 (0.1)	6.2 (0.07)	0.12 (0.006)
LCP	0.6 (0.06)	3.1(0.1)	9.4 (0.1)	3.3 (0.1)

Figure 24 JavaScript CSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.08 (0.004)	1.78 (0.004)	6.2 (0.06)	0.36 (0.004)
LCP	0.08 (0.004)	1.78 (0.004)	6.2 (0.06)	0.36 (0.004)

Figure 25 JavaScript SSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.8 (0.04)	2.1 (0)	2.6 (0)	1.8 (0.04)
LCP	3.8 (0.07)	4.1 (0.06)	4.6 (0.10)	3.9 (0.07
SI	3.1 (0.10)	3.4 (0.11)	3.83.4 (0.11)	3.2 (0.1)

Figure 26 500x1 CSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.8 (0.07)	2.1 (0.07)	2.6 (0.1)	1.9 (0.07)
LCP	3.0 (0.08)	3.3 (0.13)	3.6 (0.12)	3.1 (0.10)
SI	2.4 (0.04)	2.7 (0.04)	3.1 (0.06)	2.5 (0.04)

Figure 27 500x1 SSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.1 (0.007)	1.8 (0.07)	6.2 (0.16)	0.2 (0.06)
LCP	0.6 (0.06)	5.9 (0.28)	17.6 (0.40)	0.60 (0.06)

Figure 28 500x1 CSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.07 (0.004)	1.8 (0.06)	6.2 (0.06)	0.14 (0.006)
LCP	0.1 (0.006)	4.3 (0.25)	15.7 (0.43)	0.18 (0.007)

Figure 29 500x1 SSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.8 (0.06)	2.1 (0.07)	2.6 (0.1)	1.8 (0.06)
LCP	5.6 (0.1)	5.8 (0.15)	6.3 (0.19)	5.7 (0.19)
SI	4.3 (0)	4.6 (0.04)	5.0 (0.04)	4.4 (0)

Figure 30 500x10 CSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.8 (0.07)	2.1 (0)	2.6 (0)	1.9 (0)
LCP	4.8 (0.07)	5.1 (0)	5.6 (0)	4.9 (0)
SI	3.6 (0.04)	3.9 (0.07)	4.3 (0.04)	3.7 (0.04)

Figure 31 500x10 SSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.1 (0.006)	1.8 (0.08)	6.3 (0.08)	0.2 (0.006)
LCP	0.6 (0.006)	13.7 (0.14)	48.9 (1.3)	0.6 (0.04)

Figure 32 500x10 CSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.08 (0.006)	1.8 (0.006)	6.3 (0.1)	0.14 (0.009)
LCP	0.11 (0.006)	12 (0.19)	38.4 (1.3)	0.26 (0.025)

Figure 33 500x10 SSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.9 (0)	7.1 (0)	N/A	1.9 (0)
LCP	1.9 (0)	7.1 (0)	N/A	1.9 (0)
SI	19 (0.04)	7.1 (0.04)	N/A	1.9 (0)

Figure 34 API x1 CSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.8 (0.04)	7.0 (0.04)	N/A	1.9 (0)
LCP	1.8 (0.04)	7.0 (0.04)	N/A	1.9 (0)
SI	1.9 (0.04)	7.1 (0.04)	N/A	1.9 (0.04)

Figure 35 API x1 SSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.22 (0.01)	1.8 (0.09)	6.2 (0.04)	0.26 (0.004)
LCP	0.22 (0.004)	1.8 (0.09)	6.2 (0.04)	0.26 (0.004)

Figure 36 API X1 CSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.2 (0.007)	1.8 (0.014)	6.2 (0.06)	0.26 (0.07)
LCP	0.2 (0.007)	1.8 (0.014)	6.2 (0.06)	0.26 (0.07)

Figure 37 APIX x1 SSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	2.4 (0.13)	N/A	N/A	2.5 (0.10)
LCP	2.4 (0.13)	N/A	N/A	2.5 (0.10)
SI	2.5 (0.04)	N/A	N/A	2.6 (0.04)

Figure 38 API x10 CSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	2 (0.07)	7.4 (0.05)	N/A	2.3 (0.08)
LCP	2 (0.07)	7.4 (0.05)	N/A	2.3 (0.08)
SI	2 (0.04)	7.4 (0.06)	N/A	2.3 (0.04)

Figure 39 API x10 SSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.6 (0.06)	2.4 (0.05)	6.2 (0.07)	1.8 (0.04)
LCP	1.6 (0.06)	2.4 (0.05)	6.2 (0.07)	1.8 (0.04)

Figure 40 API x10 CSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	0.7 (0.03)	1.8 (0.05)	6.2 (0.06)	0.76 (0.006)
LCP	0.7 (0.03)	1.8 (0.05)	6.2 (0.06)	0.76 (0.006)

Figure 41 API x10 SSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	3.9 (0.06)	N/A	N/A	4.4 (0.05)
LCP	3.9 (0.06)	N/A	N/A	4.4 (0.05)
SI	3.9 (0.06)	N/A	N/A	4.4 (0.05)

Figure 42 API x20 CSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	2.6 (0.07)	N/A	N/A	2.7 (0.09)
LCP	2.6 (0.07)	N/A	N/A	2.7 (0.09)
SI	2.6 (0.07)	N/A	N/A	2.7 (0.09)

Figure 43 API x20 SSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	2.8 (0.04)	3.3 (0.05)	6.5 (0.15)	3.7 (0.06)
LCP	2.8 (0.04)	3.3 (0.05)	6.5 (0.15)	3.7 (0.06)

Figure 44 API x20 CSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	1.15 (0.006)	2.34 (0.01)	6.2 (0.08)	1.3 (0.04)
LCP	1.15 (0.006)	2.34 (0.01)	6.2 (0.08)	1.3 (0.04)

Figure 45 API x20 SSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	9.6 (0.06)	N/A	N/A	10.2 (0.12)
LCP	9.6 (0.06)	N/A	N/A	10.2 (0.12)
SI	9.6 (0.06)	N/A	N/A	10.2 (0.12)

Figure 46 API x50 CSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	4.4 (0.05)	N/A	N/A	4.6 (0.05)
LCP	4.4 (0.05)	N/A	N/A	4.6 (0.05)
SI	4.4 (0.05)	N/A	N/A	4.6 (0.05)

Figure 47 API x50 SSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	9 (0.13)	8.1 (0.03)	10.4 (0.04)	8.3 (0.03)
LCP	9 (0.13)	8.1 (0.03)	10.4 (0.04)	8.3 (0.03)

Figure 48 API x50 CSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	3.3 (0.02)	4.3 (0.02)	7.3 (0.07)	4.0 (0.04)
LCP	3.3 (0.02)	4.3 (0.02)	7.3 (0.07)	4.0 (0.04)

Figure 49 API x50 SSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	22.7 (0)	N/A	N/A	23.0 (0)
LCP	22.7 (0)	N/A	N/A	23.0 (0)
SI	22.7 (0.04)	N/A	N/A	23.0 (0)

Figure 50 API x100 CSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	8.6 (0.01)	N/A	N/A	8.9 (0.02)
LCP	8.6 (0.01)	N/A	N/A	8.9 (0.02)
SI	8.6 (0.04)	N/A	N/A	8.9 (0.05)

Figure 51 API x100 SSR lighthouse. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	18.8 (0.04)	16.0 (0.16)	17.4 (0.26)	18.9 (0.14)
LCP	18.8 (0.04)	16.0 (0.16)	17.4 (0.26)	18.9 (0.14)

Figure 52 API x100 CSR performance insights. The results are in seconds. The standard deviation is in parenthesis.

Metric	No Throttling	Fast 3g	slow 3g	6x CPU Throttling
FCP	8.2 (0.04)	10.2 (0.05)	11.8 (0.06)	8.7 (0.04)
LCP	8.2 (0.04)	10.2 (0.05)	11.8 (0.06)	8.7 (0.04)

Figure 53 API x100 SSR performance insights. The results are in seconds. The standard deviation is in parenthesis.