Mathias Beke

# On the Comparison of Software Quality Attributes for Client-side and Server-side Rendering

**Principal Adviser**
Prof. dr. Serge Demeyer

**Assistant Adviser**
dr. Gulsher Laghari

Dissertation Submitted in June 2018 to the Department of Mathematics and Computer Science of the Faculty of Sciences, University of Antwerp,
in Fulfillment of the Requirements for the Degree of Master of Science in Computer Science: Software Engineering.

Universiteit Antwerpen

# Contents

# Nederlandstalige Samenvatting

Het doel van dit onderzoek is om de verschillen tussen client-side rendering en server-side rendering te bestuderen. Daarnaast is het ook de bedoeling om ontwikkelaars te helpen in de keuze tussen deze twee rendering paradigma's.

Allereerst worden — naast het aangeven van de noodzaak voor deze vergelijkende studie — de verschillen tussen client-side en server-side rendering uitgelegd. Hierna volgt een overzicht van de evolutie van het world wide web, en een verduidelijking van welke technologieën er effectief gebruikt worden om webapplicaties met beide rendering paradigma's te implementeren.
Daarna worden de verschillende software kwaliteitsattributen geïntroduceerd en besproken waarop deze vergelijkende studie gebaseerd is. Verder is er een enquête gedaan a.h.v. een uitgebreide lijst van kwaliteitsattributen. Deze enquête biedt eerste inzichten in wat ontwikkelaars denken over de verschillen tussen de twee rendering paradigma's, en geeft nogmaals aan dat er nood is aan een vergelijkende studie

Het eerste onderzoek wordt gedaan in de pilot study a.h.v. zelfgeschreven webapplicaties. Op deze projecten worden *performance* en *development time* gemeten.
Vervolgens wordt een case study gedaan a.h.v. bestaande opensourceprojecten. Op deze *Github* projecten wordt de *performance*, *scalability*, *development effort* en *availability* onderzocht.

Er wordt vastgesteld dat client-side rendering de voorkeur geniet wanneer er belang gehecht wordt aan de volgende kwaliteitsattributen: *server bandwidth*, *server throughput*, *next page load times*, *usability* en *scalability*. Server-side rendering daarentegen geniet de voorkeur voor volgende kwaliteitsattributen: *initial page loads*, *development effort*, *search engine optimisation* en *browser compatibility*. Gebaseerd op deze bevindingen wordt een beslissingsboom opgesteld om developers te helpen de keuze te maken tussen de twee rendering paradigma's.

# Abstract

The main goal of this study is to investigate the differences between client-side rendering and server-side rendering and to advise developers on making the choice between those two rendering paradigms.

First, the need for such a comparative study is explained, as well as the two rendering paradigms. This is followed by an overview of how the world wide web changed over the years and an explanation of the current technologies that are actually used for implementing web applications with both client-side rendering and server-side rendering.
Then, the different software quality attributes on which the comparison is based are introduced and explained. Furthermore, a survey is performed on an extensive list of quality attributes. This survey gives first insights in what developers think about the differences between the two paradigms, and shows again the need for a comparative study.

The first investigation is done by means of a pilot study on self-written web applications. On this projects the *performance* and *development effort* are measured.
Then, a case study is done on existing open source projects. On these existing *Github* repositories, the *performance*, *scalability*, *development effort* and *availability* are investigated.

It is found that client-side rendering is to be preferred when the following attributes are important: *server bandwidth*, *server throughput*, *next page load times*, *usability* and *scalability*. Server-side rendering, on the other hand, is the be preferred for the following attributes: *initial page loads*, *development effort*, *search engine optimisation* and *browser compatibility*. Based on this findings a decision tree is composed to guide developers in making the choice between the two paradigms.

# Acknowledgments

I would like to thank Prof. dr. S. Demeyer and dr. G. Laghari for all the advice and feedback on this thesis. I am also grateful to them that I could introduce and develop my own research topic.

I am profoundly grateful for all the input that T. Truyts gave me on the subject and for his patience to proofread this thesis twice and continuously listen to my thoughts about the research.

I would also like to thank everyone who participated in the survey, especially those who helped me distribute the survey on social media.

Finally, my heartfelt thanks go to my friends and family, who always support and tolerate me. Special thanks go to my parents, who patiently encouraged me and are always there to help me.

# List of Figures

# List of Tables

# CHAPTER 1
# Introduction

With the arrival of Web 2.0, the Internet changed drastically from a whole of static pages, to more dynamic content with enriched user interaction [52]. This paradigm shift from server-side rendering to client-side rendering, which delegates more processing work to the client, resulted into *single page applications* – websites using client-side rendering instead of fetching HTML pages rendered at server [55] [23].
This thesis discusses the differences between these two rendering paradigms and compares them on software quality attributes.

## 1.1 Why Compare Client-side with Server-side Rendering

There are numerous consequences of choosing one paradigm over the other. The preference of one paradigm over the other has for instance some ramifications for the page load times, development, and maintenance cost [95] [77]. The increased page load times may translate to customer abandonment [62]. Similarly, search engine optimisations and browser compatibility are also impacted by the choice [11] [6].

**Lack of Literature**
Literature around these consequences is found mainly in blogposts, in which the personal experience and opinion of one developer or one team is proposed. Although there exists literature about modern technologies, including single page applications and client-side rendering [57] [37] [54], searching for academic literature on this specific topic yields no results.

**Difficulty for Developers**
On the other hand, organisations struggle to decide which rendering strategy to choose for their specific applications. For example, Twitter originally used server-side rendering and then switched to client-side rendering in 2010 [76]. Then, they switched back to server-side rendering again in 2012 [90].
This implies that a comparison to help objectively decide which paradigm to choose from the two would be helpful.

**Conclusion 1.1.1**
There is a clear need for a comparative study between client-side rendering and server-side rendering.

## 1.2 How Compare Client-side with Server-side Rendering?

Choosing one rendering paradigm over the other has its consequences. This implies that to decide which rendering paradigm to choose from the two, the two paradigms need to be compared in terms of software engineering quality attributes.

The goal of this research is to get insights on which paradigm yields the best results for which quality attribute, such that developers can make informed decisions when choosing one rendering paradigm over another.

First the criteria of the comparison, i.e. quality attributes, were selected. A survey was conducted on developers to get their insights on the quality attributes.
A pilot study was done by comparing equivalent self-written test projects on the selected quality attributes. Then, a more elaborate study was done by comparing real-world projects found on *Github*.

## 1.3 Thesis Outline

To begin, the research is introduced in this chapter (**Chapter 1**): In **Section 1.1** the need for this comparative study is explained.

*The content of this thesis is further structured like this:*

In **Chapter 2** the difference between client-side rendering and server-side rendering is explained. The chapter also gives an overview of how web development has evolved until now.

In **Chapter 3** the selected quality attributes are introduced, and the impact of client-side rendering and server-side rendering is explained (Section 3.1). Based on this selection first insights were gathered using a developer oriented survey (Section 3.2).

The first actual measurements are done in **Chapter 4**: equivalent test projects were programmed with two development stacks: one equivalent pair was created in *Go*, the other was created in *Laravel*.

In **Chapter 5**, more thorough measurements were done on real-world repositories — which were found on *Github*.

The validity of those results is discussed in **Chapter 6**.

The overal conclusions are drawn in **Chapter 7**, and a decision three is proposed to help developers choose the most suited paradigm.

Finally, generalisations are proposed in **Chapter 8**.

The performance related parts from **Chapter 4** and **Chapter 5**, as well as the generalisations from **Chapter 8** were submitted in a paper to the *ProWeb 2018*[1] conference. Although the paper got rejected, a lot of useful feedback was gather from the reviews and incorporated in this document.

[1] `https://2018.programming-conference.org/track/proweb-2018-papers`

# CHAPTER 2
# Background

## 2.1 Client-side rendering and server-side rendering

Before the research about client-side rendering and server-side rendering can be discussed, the differences between these paradigms should be explained. Those differences are thus explained in this section.
We talk about server-side rendering if for each change the user makes, the complete page must be sent back again by the server. If only the changed content is updated by the client, we talk about client-side rendering. [95] [45]

### 2.1.1 Server-side rendering

Server-side rendering is mostly used in older web apps and static web pages. PHP, APS.NET, Java server pages, ... they were all conceived to render HTML code on the server. This means that the server fetches the data from the database, renders this data in HTML, and sends it back to the client (Figure 2.1).

An example of server-side rendering is the website of the University of Antwerp[1]: looking into the source code, returned by the server, shows that the complete HTML page is rendered by the server.

[1] `https://www.uantwerpen.be/nl/`

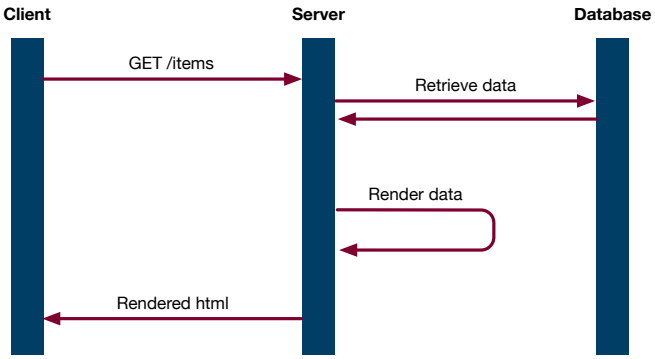



**Figure 2.1:** Sequence of server-side rendering

### 2.1.2 Client-side rendering

In modern web development (and software development in general), the data and the front-end are both treated separately [37]. The backend is developed in one programming language, and exposes its data through an API. The client fetches the data and renders it. Whether it is a web app or a mobile app, they fetch the data from the same API.
Putting this in the context of a web app, this means that the server first sends the code of the web app (mostly being a simple HTML page with minimal elements, and Javascript code containing all the logic) to the client. The client then fetches the data from the servers' API. This data is then rendered by the client (Figure 2.2).

Rendering the data is mostly done by the use of a *Javascript front-end framework* in modern web development. The world of these frameworks is dominated by Angular[2] and React[3]. There are of course a lot of growing frameworks available (like Vue[4]), but the former are the most used [28] [30].

An example of this is the website of the city of Antwerp[5]: only the most necessary HTML code is passed to the client, after which the client renders all the pages itself.

[2] `https://angular.io`
[3] `https://reactjs.org`
[4] `https://vuejs.org`
[5] `https://www.antwerpen.be`

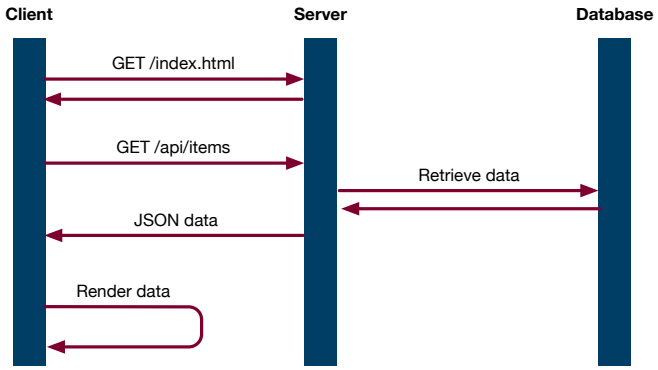



**Figure 2.2:** Sequence of client-side rendering

## 2.2 Evolution of Web development

The evolution of web development and web technologies is explained in this section. This explains how the shift from server-side rendering to client-side rendering came about.

### 2.2.1 The World Wide Web

The world wide web was initially invented in 1989 by Tim Berners-Lee [9]. The very first version of the world wide web at CERN was built upon the first versions of HTTP (HyperText Transfer Protocol) [46] and HTML (HyperText Markup Language) [8]. These web pages were served by the first web server [51]. Bernes-Lee also wrote the first web browser to access this newly created web [7].

The world wide web has drastically changed since its original conception in 1989. To get a better view on this evolution, the web is divided into different phases: Web 1.0, Web 2.0, Web 3.0, and Web 4.0 [42] [3] [27].

#### Web 1.0

The original world wide web — as proposed by Tim Berners-Lee — consisted of static pages, through which users could share information [3]. It was monodirectional, meaning that organisations shared information (like brochures, catalogues, ...) in a read-only way. This data was presented on static HTML pages which changed infrequently over time. Users could not contribute to existing web pages.

#### Web 2.0

The term Web 2.0 was orignally invented by Dale Dougherty [3] [67]. Tim O'Reilly later defined Web 2.0 as follows [66]:

> *Web 2.0 is the business revolution in the computer industry caused by the move to the internet as platform, and an attempt to under-*

> *stand the rules for success on that new platform. Chief among those rules is this: Build applications that harness network effects to get better the more people use them. (This is what I've elsewhere called "harnessing collective intelligence.")*

Web 2.0 can be denoted by the following synonyms to describe its meaning [3]:

- Wisdom Web
- People-centric Web
- Participative Web
- Read-write Web

Allowing users to both read from web pages and write to web pages made the web become bi-directional. This allowed webpages to be updated more frequently, but also allowed user contributions, make it possible for crowd-sourced platforms, wikis, and social networks to be created.

**Web 3.0**
Web 3.0, also called *semantic web*, extends the web pages from Web 2.0 with machine readable meta-data. This makes discovery, integration, automation, and reuse amongst multiple applications of data possible [3]. This machine-consumable data is shared by services/API's. Internet of Things also comes into play here to crowd-source this data [42].

**Web 4.0**
Web 4.0, also called WebOS, is nothing more than an *idea* right now [3]. The idea is that the web becomes an intelligent system which enables symbioses between humans and machines.

### 2.2.2 Evolution of Technologies

The evolution of websites also means the evolution of the technologies and techniques used to build those websites.

**Static HTML**

In the days of Web 1.0, where web pages did not change dynamically, browsers were very rudimental and websites were small. So it was adequate to create a static HTML file containing all the content and design which would never be automatically updated. Such an HTML file looks like this:

```
<!DOCTYPE html>
<html>
    <head>
        <title>This is a title</title>
    </head>
    <body>
        <p>Hello world!</p>
    </body>
</html>
```

Later, when browsers developed visual abilities, style sheets were needed. Håkon W Lie proposed a first version of CSS (Cascading Style Sheet), which is still used today [49]. A simple stylesheet looks like this:

```
body {
    background-color: silver;
}

p {
    font-size: bold;
}
```

Example: A good example of a static website is the Prof. Demeyer's website[6]. Very simple layout, built only with HTML and CSS.

**Dynamic Web Pages**

Together with the first dynamic websites came the need for the first server-side programming technologies. In early days programming on the server-side was done directly in the webserver. Later the CGI (Common Gateway Interface) standard was developed, which made it possible for the webserver to interact with any local process.
Later, languages like Perl, Java, PHP, and ASP (and others) became popular for server-side programming [69].

[6] http://win.ua.ac.be/~sdemey/

Example: Any website built with a content management system (like *Drupal, Wordpress, Joomla, ...*) is a dynamic website. The website of the student circle[7] is a good example of such a dynamic website.

**Web Applications**

Websites becoming richer in features requires more flexibility and better usability. This is where client-side programming comes into play. Without the need of completely reloading a page, content can be changed by end-users. The mostly used technology for this purpose is Javascript [42].
When client-side features became even more important, frameworks were conceived for client-side rendering (and controlling) of pages. The most known frameworks here are: Ember[8], Angular[9] and React[10].

Example: Online web applications like *Google Drive*, *Slack*, ...

**Progressive Web Applications**

The next step in web development is making standalone web applications which can also be used offline. They are called PWA (Progressive Web Applications) [10] [72]. Making such applications is not possible without having full control at the client-side. Server-side rendering is thus impossible and fully fledged front-end frameworks are used to build these kind of applications. Communication is handled through API's.

Example: *Twitter Mobile*[11]

[7] `https://www.winak.be`
[8] `https://emberjs.com`
[9] `https://angular.io`
[10] `https://reactjs.org`
[11] `https://mobile.twitter.com/home`

### 2.2.3 Evolution of Web and Technologies

The evolution of the world wide web combined with the evolution of the technologies is shown in Figure 2.3.

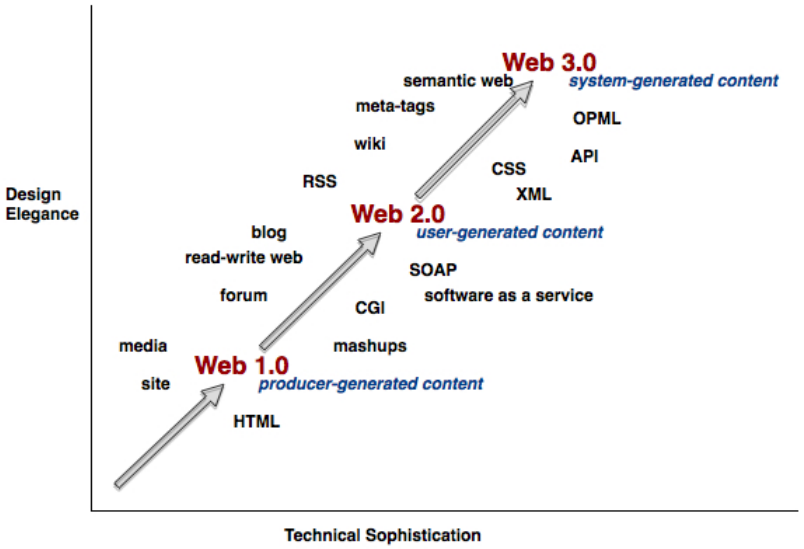



**Figure 2.3:** Evolution of web and technologies
©*Clark Quinn* [71]

## 2.3 Overview of Technology Stack

This section gives an overview of the respective technologie stacks for client-side rendering and server-side rendering and how representative technologies to use were chosen for this study.
To talk about the different technologies and their popularity, first the conceptual differences between the two paradigms must be pointed out. Figure 2.4 shows that the styling, markup language, server-side language (and framework) and database system are equivalent for both of the paradigms.
The client-side rendering specific parts in the technology stack are: a client-side language (and framework) and an API, implemented in the server-side language.

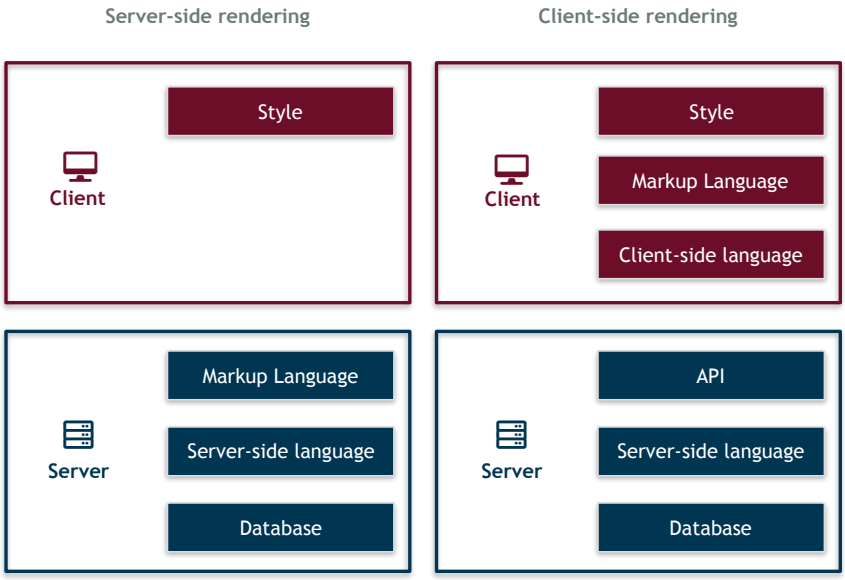



**Figure 2.4:** Technology stacks for client-side rendering and server-side rendering.

To have representative results in this research, the popularity of each technology in the stack is taken into account, and the most popular technologies were used:

1. *Style & markup Language*
   Styling and markup is not affected by the choice of paradigm. There is also no choice possible, since *HTML & CSS* are the de facto standards.
2. *Database*
   Database system used is not affected by the choice of paradigm. Since

the majority of website use a relational database, *MySQL*[12] and *SQLite*[13] are used in this research [36].

3. *Client-side language & framework*
   The front-end language on the web is de facto *Javascript*, there is no choice. There are however numerous frameworks that do client-side rendering. By investigating *React, Angular* and *Vue* in this study, the majority ($> 60\%$) of front-end frameworks (doing actual rendering) are analysed [81] [88] [14].
4. *Server-side language & framework*
   For the server-side language, *Go* and *PHP* were considered, but most of the research was conducted using *PHP*. Both *Wordpress* and *Laravel* were considered as back-end frameworks, to make the study as representative as possible: *PHP* is used on more than 80% of server-side scripted websites, and more than 80% of those *PHP* websites use *Wordpress* or *Laravel* [89] [13] [15].
5. *API*
   The choice of server-side framework has it consequences for the used API: *Wordpress* comes with a REST API by default, so in this research GraphQL and SOAP are not considered. REST is currently also the most popular API technology [83].

**Conclusion 2.3.1**
The used technologies in this research are representative for the real world.

[12] `https://www.mysql.com`
[13] `https://www.sqlite.org/index.html`

CHAPTER 3

# Software Quality Attributes

Before comparing client-side rendering with server-side rendering, the domain must be established. A good choice of attributes/criteria to do the comparison must be made.

General literature gives a good view on the existing software quality attributes [25] [74] [56] [70]. But this doesn't specifically gives a list of attributes which are important and relevant for web development, a subset of web development related attributes is thus needed.
That's why literature was consulted to know which quality attributes are to be considered in the context of web development, i.e. the criteria for this research.
Based on this literature, the attributes to consider were selected [64] [60] [59] [63] [65] [32] [97] [50] [68]. Various academic literature, like Nabil, Mosad and Hefny, derived the quality attributes for web development from the *ISO9126 standard for software engineering product quality* [59] [65] [32] [97] [50]. Together with this literature based on *ISO9126*, other literature was used to narrow down the relevant software quality attributes and make an actual selection for this research [64] [60] [63] [68].

## 3.1 Quality Attributes

### 3.1.1 Usability & User Experience

Although usability and user experience are tightly coupled together, they are not the same: Usability is about the task effectiveness, i.e. doing things intuitively and easily. It defines how well the requirements are met [17] [5]. User experience, on the other hand, is what a user feels when interacting with the product, i.e. the emotional connection to the task. [17] [5]

The most obvious consequence when choosing between server-side and client-side rendering is the user experience. Client-side rendering allows rich interactions by the user [34]. It allows not only entry and exit animations for static elements, but also makes it possible for users to fetch new dynamic content on a webpage without having to wait for a complete new page load. It allows also for a more usable way to submit forms or other data. Additionally, client-side rendering enables lazy-loading of content and images. [34] [45]. Client-side rendering furthermore makes it possible to show loading animations to the user [45].

Server-side rendering, on the other hand, also has its advantages when talking about user experience [31] [45] [94]. When using client-side rendering, the client needs to wait on the download of the Javascript front-end app before downloading and rendering the content. When using server-side rendering, on the other hand, a fully rendered page is sent to the client, which yields a faster initial load time. So this yields a better user experience on the first load [91].
This slow initial load time for client-side rendering can be overcome by rendering the initial page loads on the server [20] [43].

In general there is a clear consensus that client-side rendering yields a better usability and user experience [34] [45] [92] [86] (Also see the conducted survey Section 3.2). That's why this quality attributes is not further investigated in this research.

### 3.1.2 Performance

One of the most quantifiable metrics is the performance. Performance denotes how responsive and stable a system is under a certain load, i.e. to execute any request within a given time limit [56]. It can be measured in terms of response time, resource utilisation, throughput, and efficiency [74]. Performance can be measured on test cases, or estimated by profiling the actual project.
There are two point of views that need to be taken into account when talking about performance: performance at the client and performance at the server. More importantly, for the client *initial* and *subsequent* load times and for the server *throughput* and *bandwidth*.

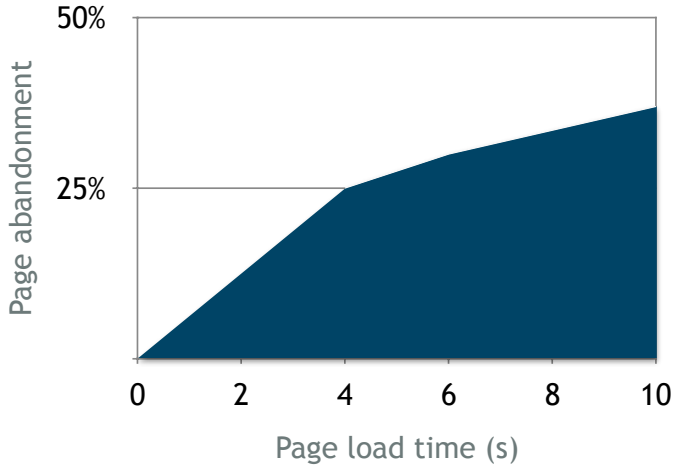



**Figure 3.1:** Page abandonment rate in relation to initial load time [22].

**Client**

Based on Figure 2.2 and Figure 2.1, one can compare the requests made in both paradigms. This comparison can be based on the *network waterfall*. This waterfall shows when a request starts loading, and how long it takes to load. So it shows which requests depend on which other requests. An example of what such a *network waterfall* looks like in the development tools of the web browser can be found in Appendix Figure 8.20.

**Client: initial page load time.**

The first client metric to look into is the *initial page load time*.
In case of server-side rendering everything is obviously handled on the server, from the database interactions to rendering the HTML content. Client-side rendering, on the other hand, requires that first the front-end Javascript code is downloaded and executed by the client, after which a request is made to the server. Finally the returned data by the server must then be rendered by the client. This means that more roundtrips have to be made, which of course increases the initial page load [91] [45] [95].
The initial load times are important because people tend to leave sites that load slowly: this is called the page abandonment rate [62]. If a webpage does not load within 2 seconds, 10% of the visitors will abandon the page (Figure 3.1), resulting in missed sales [75] [29] [21] [85].

**Client: subsequent page load time.**

Not only the initial page load is important, but also the load times of all the consecutive user interactions. Subsequent page loads can be faster on client-side rendering, since less data has to be transmitted [95].

Websites with shorter average session load times have a higher conversion rate [18] [80]. The conversion rate denotes the number of visitors who are actually paying customers.
It can be generalized as: Conversion rate $= \frac{\text{Number of goal achievements}}{\text{Number of visitors}}$.
Stadnik et. al. state that the conversion rate quickly drops if the page load times increase [80]. So it is clearly relevant to look a bit deeper in those load times.
When using server-side rendering, the complete new HTML page is fetched from the server and rendered at the client. While with client-side rendering, only the part that changes is requested, and encoded in say JSON. Thus, client-side rendering has the following consequences for a next page load: less work to be done by the server, less bytes to be transferred, and only small part of rendered page needs to be refreshed, yet the browser must actually render it.

**Server**
**Server: throughput**
Similar to the performance measures from the perspective of the client, the performance measures from the perspective of the server are also equally relevant. A first important performance measure is the throughput — number of requests handled per second by the server. The goal is to serve as many clients as possible with as few resources as possible.
In case of server-side rendering, for each request the page must be rendered contrary to client-side rendering where only the data is serialised. If the server must only return a serialised data object (client-side rendering) and not a fully rendered HTML page (server-side rendering), the throughput (requests per second handled by the server) can increase [77] So one can assume that client-side rendering will yield a higher throughput.

**Server: bandwidth**
The differences between client-side rendering and server-side rendering in terms of used bandwidth can be found in the size of the content based requests. Of course there is always the overhead of assets like images and movies, but encoding text by rendering the data in a complete HTML page is completely different than using a more compact representation like JSON or GraphQL serialised object.
Client-side rendering requests to APIs are smaller in terms of bandwidth compared to server-side rendered HTML files [77].

Consider for instance the following HTML snippet:

```
<div class="post">
    <h1>Hello world!</h1>
    <div class="meta">
        <span class="author">John Doe</span>
        <span class="date">21/02/2018</span>
    </div>
    <div class="content">
        Donec ullamcorper nulla non metus auctor.
    </div>
</div>
```

The very same data, encoded in JSON, is comparatively smaller:

```
{
    "post": {
        "title":   "Hello world!",
        "author":  "John Doe",
        "date":    1519223545,
        "content": "Donec ullamcorper nulla non metus auctor."
    }
}
```

The performance metrics are investigated in both the pilot study (Chapter 4) and the case study (Chapter 5).

### 3.1.3 Development Effort

The complexity of client-side rendered apps is argued to be higher than server-side rendered [26] [4]. In single page applications the client needs to fetch data from the server. This means that effort should be put into creating a complete and secure API, and the communication with that API: serialising at the server, unserialising at the client, or taking care of network delay or connection loss, ...
There is also the problem of duplicates: models not only exist at the server side, but when more complex operations with those models are needed, they also need to be implemented at the client side. Often the language used at the client differs from the one at the server. This yields duplicate code [84].

Building a front-end web app also means that not only the build environment at the server must be configured and maintained, but that effort must now also be put into maintaining a build environment for the client-side code.

Development effort is investigated in both the pilot study (Chapter 4) and the case study (Chapter 5).

### 3.1.4 Maintainability

Maintainability denotes how easy it is to make changes to the existing system [56]. Also taking into account the ease of updating the tests to these changes [74].
An example in this comparison: client-side rendering involves a more differentiated setup and more interactions, it is is obviously more difficult to maintain all these [12].

Maintainability is not something that can be easily studied by looking at source code [44]. It is to be studied a posteriori, i.e. based on historical data that is gathered [33]. This makes it useless to look at maintainability in this research, since it cannot be covered in the pilot study and case study that was performed.

### 3.1.5 Scalability

Scalability is the ability of the system to handle an increased load [56]. A system is scalable if it can gracefully handle an increased usage/load [70].
When using client-side rendering, only the data has to come from the server running the API. The actual Javascript code and assets can be hosted on CDNs (content delivery networks), which can scale horizontally very well [34] [20].

Scalability is investigated in the case study (Chapter 5).

### 3.1.6 Compatibility

#### Browser Support

Server-side rendering is argued to yield higher compatibility [91]. Since the browser has less work to do, and more responsibilities are delegated to the server, server-side rendering offers higher compatibility with different clients and browsers [45]. Although most people use modern browsers, there is still a portion of users on older or less standardised browsers that do not fully support modern techniques used in the most popular front-end frameworks [1] [2].

Browser compatibility all depends on the used Javascript frameworks and the actual browsers of the website visitors. So this is something that is not relevant for this study, but that each organisation can easily check for themself by looking at website statistics and at the requirements of the framework they want to use.

#### Search Engine Optimisation

Search Engine Optimisation is important for most websites. Depending on the complexity of the crawler visiting your website, client-side rendering

[1] `https://angular.io/guide/browser-support`
[2] `https://reactjs.org/docs/react-dom.html#browser-support`

could have a bad impact on your Search Engine Optimisation. For instance, some crawler do not execute the Javascript code on your website, thus it does not get the content and is not able to crawl any interesting data. Recently, the crawlers have evolved and most of them execute Javascript code now, meaning that they support client-side rendering [35]. Google, for instance, was the first to fully support client-side rendering in their crawlers [61] [58] [95]:

> *Times have changed. Today, as long as you're not blocking Googlebot from crawling your JavaScript or CSS files, we are generally able to render and understand your web pages like modern browsers.*

Unfortunately, this is not the case for all crawlers. Baidu for instance, the largest search engine in China, does not support client-side rendering yet [47].
So if Search Engine Optimisation really matters, one should also render the pages on the server if the website is accessed by a crawler [91].

Because compatibility is hard to measure without real content and real users, it is left out from this research.

### 3.1.7 Security

Measuring security on source code is impossible. Security can only be measured on an actual organisation and actual security breaches [48] [1]. So this is again a quality attribute that's not relevant in this research.

### 3.1.8 Reliability & Availability

Reliability and availability describe whether the web site is available for users, and whether it behaves correctly [63]. Investigating this involves the user and an actual running system. Thus, once more, this quality attribute cannot be researched in this study.
The availability of the two systems is however shortly discussed in the case study (Chapter 5). When abstracting the user away and calculating it with an uptime monitoring tool, it can be measured.

## 3.2 Opinion of Developers on Quality Attributes

To get the first insights on the software quality attributes a survey was conducted. The goal of this survey is to reach out to professional developers and ask them for which quality factors they prefer over a rendering paradigm and to ask them which factors are most important in the decision of choosing between the two rendering paradigms.
Thus, it allows to learn about developers' perceptions about the client-side rendering and server-side rendering paradigms.

The survey was distributed on various social media (Facebook, Twitter, Reddit), mostly in web development related communities or via influencers in web development. In total 63 respondents filled in the complete survey.

### 3.2.1 Survey Questions

#### Demographic Related Questions

To get some insight in the profile of the people filling the survey, demographic related questions were added to the survey. To do so, existing developer oriented surveys were studied. One of the most important surveys about software engineering — performed each year — is the *Stack Overflow Developer Survey*[3]. This survey contains a lot of interesting demographic questions. The demographic questions in this survey (Table 3.1) are thus extracted from the *Stack Overflow Developer Survey*.

| # | Question |
|---|---|
| A | Age |
| B | Occupation |
| C | Gender |
| D | Highest Education |
| E | Experience in Software Development |
| F | Years Since Learning to Code |
| G | Years of Professional Coding Experience |
| H | Company Size |

**Table 3.1:** List of demographic related questions in the survey.

#### Software Engineering Related Questions

To better understand which quality attributes depend on which metrics, dependencies were looked up in literature and used to compose the survey questions. The following dependencies were extracted from literature:

1. **Testability** depends on Modifiability and Flexibility [2].
2. **Code Quality** depends on Number of Defects, Maintainability, Portability, Efficiency, Usability, Functionality, Testability [41] [25].

[3] `https://insights.stackoverflow.com/survey/2018/`

3. **Extensibility** depends on Modifiability, Maintainability, Scalability [39].

These dependencies and other interesting quality attributes combined yield the following list of individual metrics to ask questions about in the survey:

1. Modifiability
2. Flexibility
3. Number of Defects
4. Efficiency
5. Usability
6. Testability
7. Extensibility
8. Modifiability
9. Maintainability
10. Scalability
11. Development effort
12. Duplicates
13. SEO
14. Performance

The list of composed questions based on these metrics can be found in Table 3.2. For most of the questions the respondents could chose between:

1. *"Client-side rendering"*
2. *"Server-side rendering"*
3. *"There is no noticeable difference"*
4. *"I don't know"*

A complete list of questions with all the answers can be found in the appendix (page 77).

| # | Question |
|---|---|
| 1 | Do you have experience with client-side rendering frameworks / single page application frameworks (like Angular, React, Vue, ...)? |
| 2 | What's your preference? Server-side rendering or client-side rendering? |
| 3 | Which of the following metrics influence your choice between server-side rendering and client-side rendering the most? |
| 4 | Which paradigm is most testable? |
| 5 | Which paradigm is more modifiable? |
| 6 | Which paradigm is more flexible? |
| 7 | Which paradigm requires most development effort? |
| 8 | Which paradigm yields most duplicate code? |
| 9 | Which one of the paradigms yields more code defects/bugs? |
| 10 | Which paradigm yields more maintainable code? |
| 11 | Which paradigm yields more efficient code? |
| 12 | Which paradigm yields better usability? |
| 13 | Which paradigm results in best extendability? |
| 14 | Which paradigm scales best? |
| 15 | Do you think the impact of client-side rendering on SEO is still noticeable anno 2018? |
| 16 | Did you notice impact on server load using one of the paradigms? |

**Table 3.2:** List of software engineering related questions in the survey.

### 3.2.2 Results from Survey

In this section, the results are presented, first starting with the demographics of the respondents. Then, the answers on the atomic questions are analysed. Lastly, the dependencies between metrics are examined.

#### Demographics

For the results of the survey to make sense, the survey should be filled in by people who have expertise. This is why a first look into results was to confirm whether enough web development experts had filled in the survey. In total, 63 respondents filled in the complete survey. The following conclusions were drawn about these respondents: The vast majority (83%) of the respondents are professional developers (Appendix Figure 8.1), which have finished higher education (Appendix Figure 8.2) and have at least a couple years of experience in software development (Appendix Figure 8.3). 81% of respondents state that they have experience with modern front-end frameworks (Appendix Figure 8.4).

In summary, the respondents in the survey are representative and have the required expertise to provide useful insights about the topic.

#### Quality Attributes

Looking at the software quality attributes that were in the survey, there is a first pattern that catches the eye: there is always a part of respondents that feel there is no noticeable difference (grey in the pie charts). But then again, in most survey questions there is also a clear preference in the developers choice.

Next step consists of looking into the individual software quality attributes asked in the survey. From interpreting these response, the following major observations become apparent:

1. Most developers prefer client-side rendering. Appendix Figure 8.5 shows that 44% prefer client-side rendering, compared to 27% preferring server-side rendering.
2. There seems no single software quality attributes that influences the developers in the sample to prefer client-side rendering or server-side rendering. Yet, the development effort and performance do matter more to 18% and 21% developers respectively in their choice to prefer certain rendering technology, compared to around 10% to 14% for the other factors. (Appendix Figure 8.6).
3. However, around two times as much developers think that client-side rendering is more modifiable (Appendix Figure 8.8), flexibele (Appendix Figure 8.9), extendable (Appendix Figure 8.16) and scalable (Appendix Figure 8.17) compared to server-side rendering.

4. The above advantage of client-side rendering are apparently opposed — again more than two times as much developers prefer one over another — to client-side rendering requiring more development effort (Appendix Figure 8.10) and, client-side rendering yielding more duplicate code (Appendix Figure 8.11) and more code defects (Appendix Figure 8.12). Client-side rendering also still has its drawbacks regarding SEO (search engine optimisation) (Appendix Figure 8.18).
5. The maintainability (Appendix Figure 8.13) and efficiency (Appendix Figure 8.14) of the code do not seem to be clearly favoured. Answers from the developers are almost evenly divided between the two paradigms.
6. Usability on the other hand is clearly better for client-side rendering: 50% vs 22% of developers prefer client-side rendering over server-side rendering for usability (Appendix Figure 8.15).
7. Most developers (44% vs 16% ) think server-side rendering is most testable (Appendix Figure 8.7)
8. Server-side rendering can yield a higher server load (Appendix Figure 8.19).

## 3.3 Conclusion

The first part of this chapter discussed the various software quality attributes and made a selection of attributes that need to be investigated in this research.

The survey shows that developers perceive differences. But these differences are not always well pronounced. So there is a need to do research the selected quality attributes in more detail. This is done in the pilot study (Chapter 4) and the case study (Chapter 5).

**Conclusion 3.3.1**
The following software quality attributes will be investigated in this research:

1. Performance (page loads, throughput and bandwidth)
2. Development Effort
3. Scalability
4. Availability

# CHAPTER 4
# Pilot Study

In this chapter a pilot study is performed on quality attributes.
The goal of this experiment is to get a first quantifiable comparison between client-side rendering and server-side rendering. This is done by creating two versions of the same simple web application (one with client-side rendering and one with server-side rendering) and perform analysis and benchmarks on them. This web application consists of a rudimental CMS (content management system) front-end: i.e. browsing pages and pagination.
Such empirical pilot study is the most controlled way to analyse the quantifiable metrics [53]. By doing the empirical pilot study on two completely identical web applications, other factors — besides the used rendering paradigm — are excluded.

## 4.1 Methodology

### 4.1.1 Web application

To do a meaningful comparison, a web application had to be implemented. This web application must not only represent the functionalities of a real world website but must also represent the technologies used by real developers.
The web application written for this research is a simple implementation of content management system. It contains simple pages which can be read-/browsed like on a real-world website.
Instead of writing it twice (client-side rendered version and a server-side rendered version), it was written four times: two equivalent projects. One client-side and server-side rendered version written in Go, and another client-side

and server-side rendered written in PHP/Laravel. *Vue*[1] was used as front-end framework for the two versions. Although *Vue* only respresents one of the front-end frameworks, the specific relevant differences between frameworks for this study are minimal [82]. The other most relevant frameworks, *Angular* and *React* are used in the case study (described in Chapter 5).
The server-side rendered versions use the built-in templating libraries of the languages. The client-side rendered versions consume the REST API exposed by the back-end, again written in the respective languages.
In this pilot study the Go implementation represents the more modern approach, were large frameworks are replaced with smaller packages [87].
The PHP/Laravel implementation represents mainstream web development. PHP is still the most used programming language on the web, and Laravel the most popular framework for PHP [16] [73].

**Source Code**

The source code of the CMS can be found on Bitbucket:

*Go implementation:*
`https://bitbucket.org/MathiasB/thesis-cms-go`

*Laravel implementation:*
`https://bitbucket.org/MathiasB/thesis-cms-laravel`

### 4.1.2 Measurement tools

The measurements done in the browser were performed using the browser's network inspection tools[2]. They were performed on the test CMS that was written.

The benchmarking of the server throughput and bandwidth is done with *ApacheBench*[3].

**Apache Bench**

*Apache Bench* is part of the *apache2-utils* packages (or *Apache* itself). These can be installed on Ubuntu as follows:
`$ apt-get install apache2-utils`

On Mac, *Apache* and *Apache Bench* are installed by default.

*Apache Bench* can then be invoked as follows:
`ab -n 1000 -c 100 http://hostname:port/`

[1] `https://vuejs.org`
[2] `https://developers.google.com/web/tools/chrome-devtools/`
[3] `https://httpd.apache.org/docs/2.4/programs/ab.html`

Where `n` is the number of requests (1000 was used in this study) and `c` is the number of concurrent connections.

## 4.2 Results

### 4.2.1 Client: initial page load time

If the *network waterfall* for both client-side rendering and server-side rendering for the same page are compared, it quickly becomes clear that client-side rendering incurs slower first page load time. Analysing the waterfall shows that the initial page load times of a website are dependent on a couple of factors including latency, number of requests, size of the requests, etc. Using a Javascript framework for client-side rendering can for example yield more blocking requests to be made before the content can be rendered, which takes more initial load time. However, not all of these depending factors can easily be improved. For instance, the latency to the client is fixed and the rendering paradigm cannot influence this. Yet, the number of requests and the size of these requests can be controlled.
Comparing Figure 4.1 and Figure 4.2, it is apparent that when using client-side rendering, the initial page load is slower. This is because after loading the index file, the complete front-end framework must be loaded followed by a request to the API to fetch the actual data. Once this data is fetched, the actual page is rendered. Contrarily, using server-side rendering, only some static assets need to be loaded after the index file before the page is rendered.

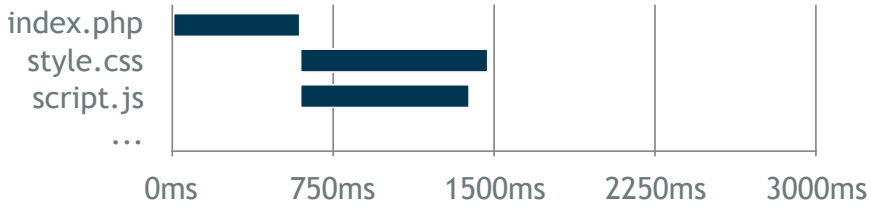



**Figure 4.1:** Network waterfall of server-side rendering for initial page load.

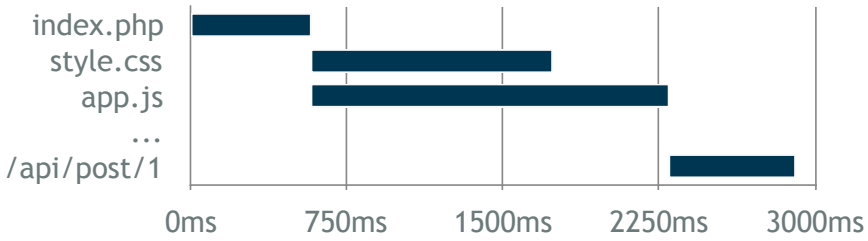



**Figure 4.2:** Network waterfall of client-side rendering for initial page load.

Looking into the network waterfalls it becomes clear that some requests depend on each other, blocking the actual rendering of the content until that chain of blocking requests is resolved.

### 4.2.2 Client: subsequent page load time

Figure 4.3 shows that, in case of server-side rendering, the complete index page is requested again, and all the assets are reloaded (from the browser cache, hence the very short times). To the contrary, for the client-side rendering (Figure 4.4) only the data from the API is fetched and nothing else.

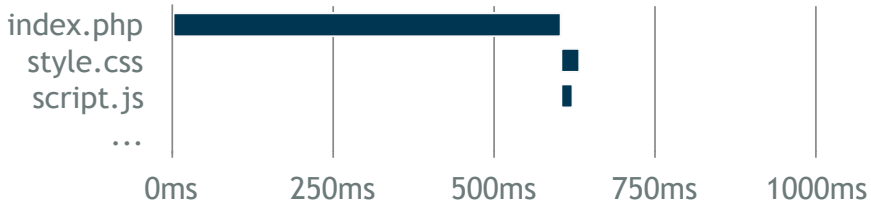



**Figure 4.3:** Network waterfall of server-side rendering for next page load.

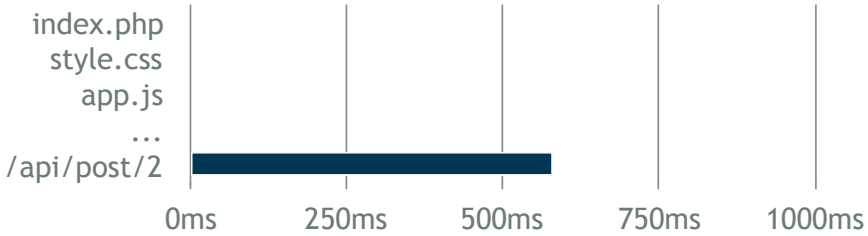



**Figure 4.4:** Network waterfall of client-side rendering for next page load.

In this test case for a small application the difference in time is insignificant and may not likely affect the conversion rate though.

### 4.2.3 Server: throughput

Comparing the throughput of calls to a templated code (representing server-side rendering) with calls to the a REST API (representing client-side rendering) on the Go project supports this assumption. Figure 4.5 shows that using client-side rendering (i.e. the API calls) yields an increase in throughput of almost 100%.

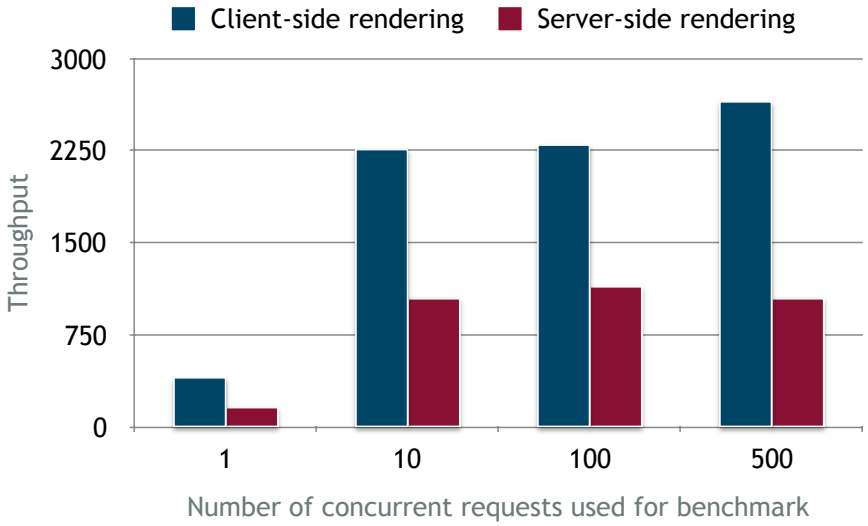



**Figure 4.5:** Throughput in Go project: API call compared to rendered page.

Performing the same experiment on the Laravel project, however, doesn't give the same enormous differences as the Go project (Figure 4.6). Client-side rendering shows in the Laravel project only a very small increase in throughput in most cases. For a very high amount of concurrent connections server-side rendering even scored better.

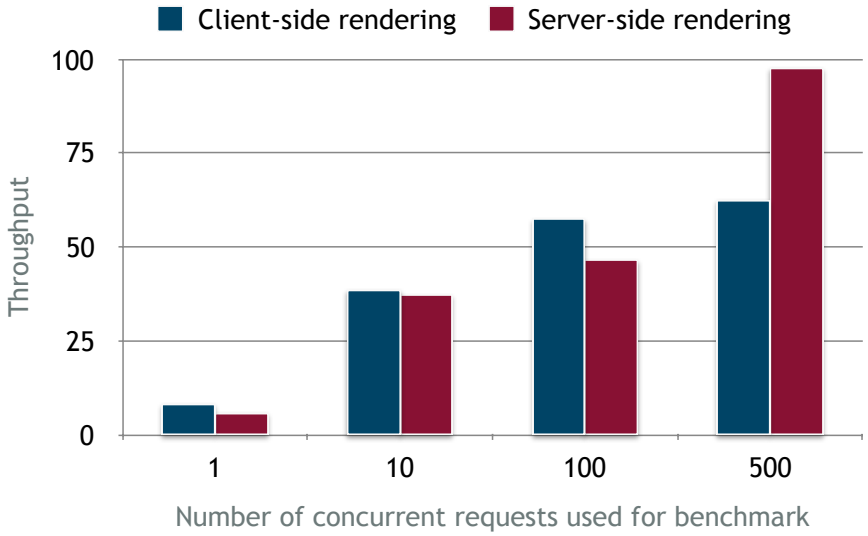



**Figure 4.6:** Throughput in PHP/Laravel project: API call compared to rendered page.

So there is a clear need for more benchmarks on existing projects. This is doen in the case study chapter (Chapter 5).

### 4.2.4 Server: bandwidth

After the throughput, the consumed bandwidth was measured. For the Go web app, Figure 4.7 shows that 90% less bandwidth is used when using client-side rendering instead of server-side rendering.

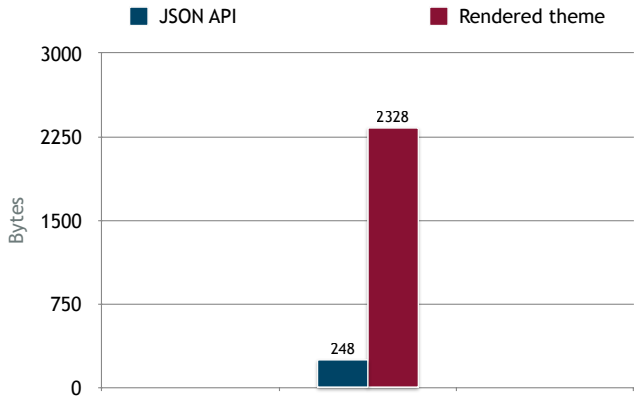



**Figure 4.7:** Bandwidth used by Go project: API call compared to rendered page.

### 4.2.5 Development Effort

When looking at the lines of code (LOC) of the two equivalent test web apps (Figure 4.8), it is clear that for this experiment more development effort was needed. Looking at the actual time it took to write these two test web apps (Figure 4.9), yields the same pattern.
So for this experiment, the needed development effort for client-side rendering is more than double the needed effort for server-side rendering.

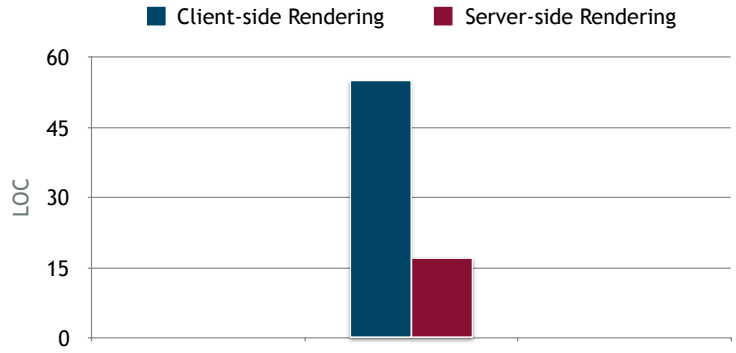



**Figure 4.8:** LOC (lines of code) for the same test web app.

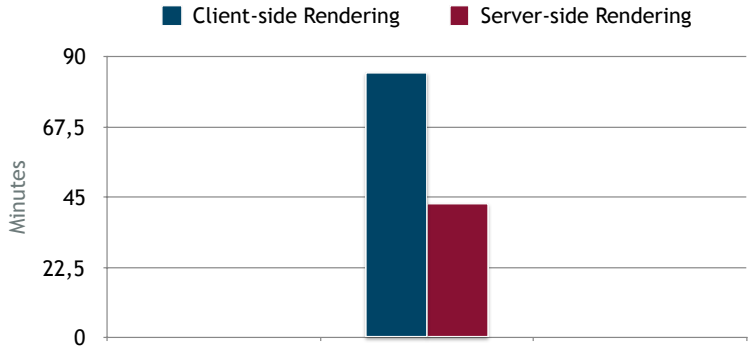



**Figure 4.9:** Time spent on writing the two test web apps.

LOC is argued to be not the most interesting metric to use for estimation [19] and thus generalisation doesn't make much sense.
Yet, the LOC provides a first quantifiable and tangible metric for development effort.

## 4.3 Conclusion

In this pilot study, the first tangible results were found. There results not only showed the first differences, but also showed that more comparison is needed.

**Conclusion 4.3.1**
The following preliminary results were found:

1. Client-side rendering yields slower initial page load.
2. Client-side rendering yields faster next page load.
3. Client-side rendering yields an increased server throughput.
4. Client-side rendering yields less server bandwidth consumption.
5. Client-side rendering yields higher development effort.

**Conclusion 4.3.2**
Research on real web applications is needed.

That is why in the next chapter open source projects were investigated, like *Wordpress*, since it is a complete content management system and the most popular on the entire internet [13]

CHAPTER 5

# Case Study on Existing Projects

## 5.1 Motivation

To get better insights in metrics in real-world situations, sticking with a single small empirical experiment is not sufficient. More data is needed, and therefore a case study was performed on existing real-world *Github*[1] repositories.
A case study is the ideal investigation to look into the impact of a given paradigm where it is isolated from other factors [53] in existing projects. There is of course a need for existing projects for the investigation to make sense, this is where *Github* comes in to play. *Github* is a platform where people host their git repositories. Those repositories not only contain the versioned source code, but also all reported bug reports (called *Issues* on *Github*). *Github* also allows users to *Star* (i.e. liking a repository) and *Fork* (i.e. creating a new user copy of a repository in which they can make their own changes) repositories. So *Github* is a very rich data source on software development [40].

*Github* repositories are existing projects, implementing solutions for real-world problems by real developers and are used by real people. This makes them perfect to research software quality as perceived by the multiple stakeholders.

[1] `https://github.com`

## 5.2 Methodology

### 5.2.1 Selecting Repositories

Analysing the impact of both client-side rendering and server-side rendering can only be successful when identical projects can be found, where the only difference resides in whether they are rendered on the client or the server. So pairs of projects with exactly the same features and popularity are needed. Collecting such pairs of repositories was done by searching on client-side rendering technologies using *Github* search. These technologies include among others: *Vue*, *Angular*, *React*, *Ember*.
As result, each of these terms (and more) yields an enormous list of repositories. So for each repository in that list the project size, features, popularity, and technologies were taken into account. This is done by looking into the documentation, websites, and source code of each candidate.

Once a good — actively maintained with reasonable features and popularity — client-side rendered repository was found, an alternative was searched with the same features/size, but implemented with server-side rendering technology. This is done by doing a *Github* search on the features/topic of the found client-side rendered results. Again, for each of the resulting repositories of that search, they were manually checked if they truly matched with the client-side rendered counterpart.

Then, for each repository the metadata was extracted by means of accessing *Github's* API and by cloning the git repositories to be able to compare them.

### 5.2.2 Analysis

#### Performance

The performance benchmarks were ran using the same tools used in the pilot study. But this time they were ran on the found projects, more specifically on *Wordpress*. Server-side rendering is tested by simply invoking the Wordpress index, which uses the theme. Client-side rendering is tested by accessing the Wordpress REST API.

#### Scalability

Besides performance, horizontal scalability is also important to measure. It goes one step further by not looking at the throughput, but looking at how the throughput increases when it is distributed over more servers. To look at scalability, the waterfalls for client-side rendering and server-side rendering were implemented so they could be replayed to mimic meaningful load. This was implemented using Go (since it offers useful mechanisms for concurrency). Then the waterfalls were first replayed on a single server as host for the application. Next, another server was added — on which the assets were hosted — and the waterfalls replayed again to see if this increased the throughput on the server.

**Source Code**

The source code of the written benchmarking tool can be found on Bitbucket:
`https://bitbucket.org/MathiasB/thesis-benchmark-tool`

**Development Effort**

Development time cannot be calculated post factum, so only LOC is used to compare the projects. Since LOC is used as a measure, the programming language expressiveness should be taken into account while analysing [24].

**Availability**

Lastly, a monitoring was ran for one week on two equivalent Wordpress setups. To do so, a scenario of a visitor browsing the website was implemented in a front-end testing framework — *Cypress*[2] was used here — and executed each 10 minutes.

**Source Code**

The source code of the written front-end tests can be found on Bitbucket:
`https://bitbucket.org/MathiasB/thesis-frontend-tests`

## 5.3 Analysed Repositories and Their Characteristics

Finding the matching pairs of repositories to analyse proved to be very hard task (and close to impossible): it is a time-consuming task, and there seemed only few equivalents on all (or even the majority) of factors.
A thorough search through *Github* yielded three interesting pairs of repositories. See Table 5.1 and Table 5.2 showing a list of client-side rendered repositories and server-side rendered repositories respectively, with their name and type, as well as the metadata extracted from them. Each n-th item in the first table forms a pair with the n-th item in the second table.

| Client-side Rendered Repositories | | Stars | Forks | Issues | LOC |
|---|---|---|---|---|---|
| Angular-Wordpress-Theme | Wordpress Theme | 389 | 124 | 25 | 852 |
| Foxhound | Wordpress Theme | 283 | 32 | 72 | 14072 |
| Buka | Book management software | 383 | 53 | 25 | 101772 |

**Table 5.1:** Selected client-side rendered repositories

[2] `https://www.cypress.io`

| Server-side Rendered Repositories | | Stars | Forks | Issues | LOC |
|---|---|---|---|---|---|
| theme | Wordpress Theme | 749 | 77 | 54 | 1095 |
| casper | Wordpress Theme | 473 | 105 | 111 | 13329 |
| cops | Book management software | 686 | 148 | 300 | 28837 |

**Table 5.2:** Selected server-side rendered repositories

Although Table 5.1 and Table 5.2 show the best matched pairs found, however the comparison of the repositories shows that there are minor differences in the projects. Those should be taken into account when comparing them:

The main features of the projects are quite similar (especially for the Wordpress themes). But still some projects are bigger than others. This is another problem, because larger project have of course more chance of having bugs, lines of code, more code to be executed — and thus possibly lower throughput, ... [96].

Client-side rendered projects also tend to have more Javascript code, whereas Server-side rendered projects tend to have more PHP code. If the lines of code needed to express the same feature is larger for a given programming language, it should be taken into account when comparing the LOC. Therefore, the difference in expressiveness for Javascript and PHP were compared by counting the the LOC (using CLOC[3]) for numerous code snippets performing exactly the same task. Comparing these LOC counts clearly show that in this case there is no significant difference in expressiveness that should be taken into account. See Figure 5.1.

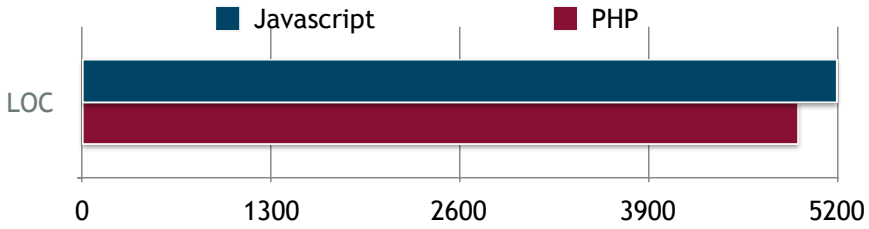



**Figure 5.1:** LOC compared for exactly the same tasks, written in both Javascript and PHP.

The client-side rendered Wordpress themes use the built-in Wordpress REST API. So the LOC for the API part is not taken into account when just counting the lines in the repositories. That's why — when researching them

[3] `https://github.com/AlDanial/cloc`

on the LOC — the LOC of the Wordpress REST API should be added to them. The LOC of the logic of the Wordpress REST API equals 7140[4].

## 5.4 Results

### 5.4.1 Client: initial page load time & subsequent page load time

The load waterfalls observed in this case study (Appendix Figure 8.20, Figure 8.21, Figure 8.22, Figure 8.23) completely matched the waterfalls from the test projects in the pilot study. See Figure 4.1 and Figure 4.2. So the results from the pilot study also hold in this case study:

1. The initial page load is faster when using server-side rendering.
2. Subsequent page loads are a little bit faster when using client-side rendering, when using browser caching. However, if no caching is used, there is a noticeable difference: 160ms for client-side rendering compared to 600ms for server-side rendering.

### 5.4.2 Server: throughput

As already done on the test projects in the pilot study, the server throughput was measured. Figure 5.2 shows that for Wordpress projects, client-side rendering yields a higher throughput than server-side rendering.

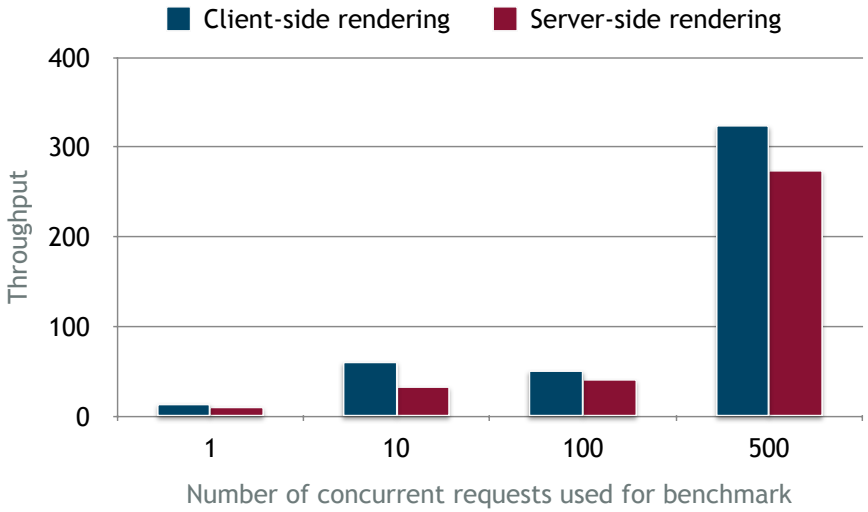



**Figure 5.2:** Throughput in Wordpress: API call compared to rendered theme.

### 5.4.3 Server: bandwidth

Comparing the server bandwidth for the Wordpress themes yields again the same results as for the Go and Laravel test projects: client-side rendering

[4] https://github.com/WordPress/WordPress/tree/master/wp-includes/rest-api

transmits a lot less data for the same request. For this test setup 50% less data was used when using client-side rendering. See Figure 5.3.

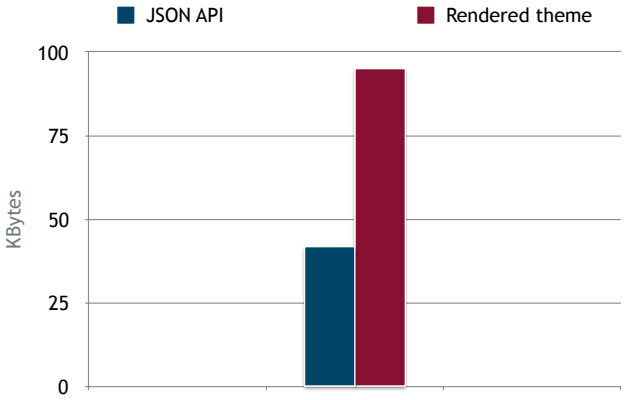



**Figure 5.3:** Bandwidth used by Wordpress: API call compared to rendered page.

However, looking at the raw requests is not enough. For real-world websites it is considered best practice to compress the server responses with Gzip [79]. So the bandwidth was also compared after Gzipping. Figure 5.4 shows that indeed, when using Gzip, 30% less data is transmitted. But even after Gzipping client-side rendering still uses 33% less bandwidth.

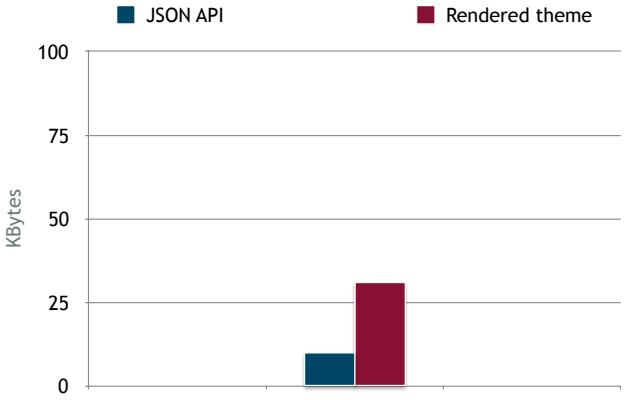



**Figure 5.4:** Bandwidth used by Wordpress when using Gzip compression: API call compared to rendered page.

### 5.4.4 Scalability

To look how the throughput increases when the load is distributed over an additional server, the scalability is studied. This was firstly done by solely looking at the rendering, not considering static assets. Figure 5.5 shows that

— when only taking rendering itself into account — server-side rendering cannot scale over an extra server, whereas client-side rendering can. So, even if the difference is not large, client-side rendering scales better than server server-side rendering.

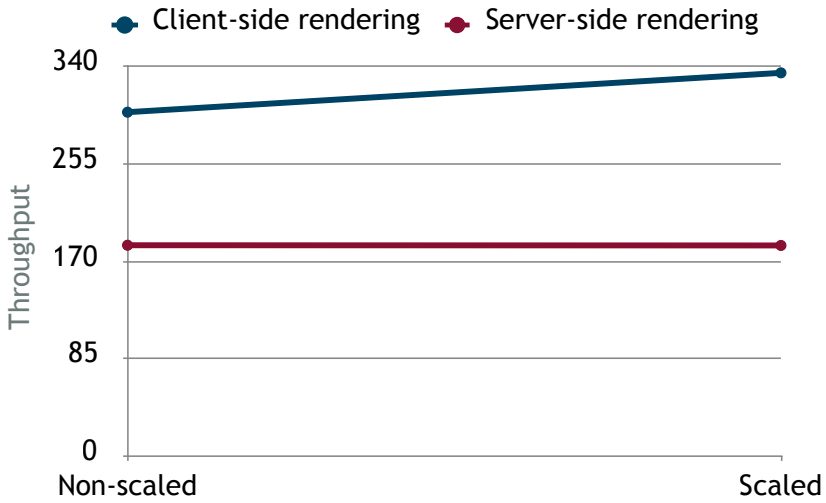



**Figure 5.5:** Scalability of Wordpress without loading other assets.

But of course, fetching some assets — corresponding to the waterfalls described earlier in the pilot study (Figure 4.1 and Figure 4.2) — represents the load of real websites more. When also putting the assets on the second server, it is observed that server-side rendering has a small improvement in throughput and client-side rendering — although small — a larger increase in throughput. See Figure 5.5.

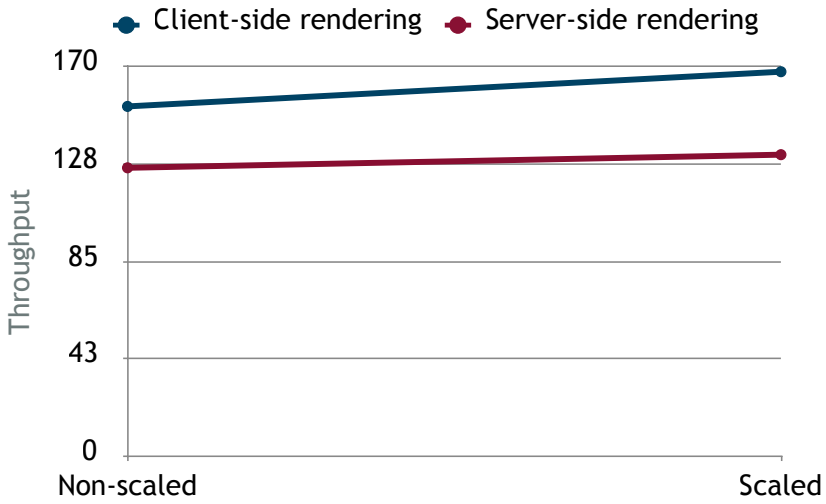



**Figure 5.6:** Scalability of Wordpress when loading other assets.

These scalability results are what can be expected. Because previously Figure 5.2 showed that only 16% of time is spent on rendering, the upper-bound on speedup factor when scaling on more servers cannot be higher than 1.18, following Amdahl's Law. Amdahl defined the upper-bound of the speedup, caused by parallelism as follows: $S \leq \frac{1}{1-p}$, where $S$ is the speedup and $p$ is the portion of code that can be parallelised [93].

### 5.4.5 Development Effort

The development effort in this case study is only considered in terms of LOC. As pointed out before, the LOC of the Wordpress REST API were added to the measured LOC of the client-side rendered Wordpress themes to make the comparison meaningful. This yields the LOC in Figure 5.7, conforming to the results of the pilot study were implementing the client-side rendered version required more development effort.

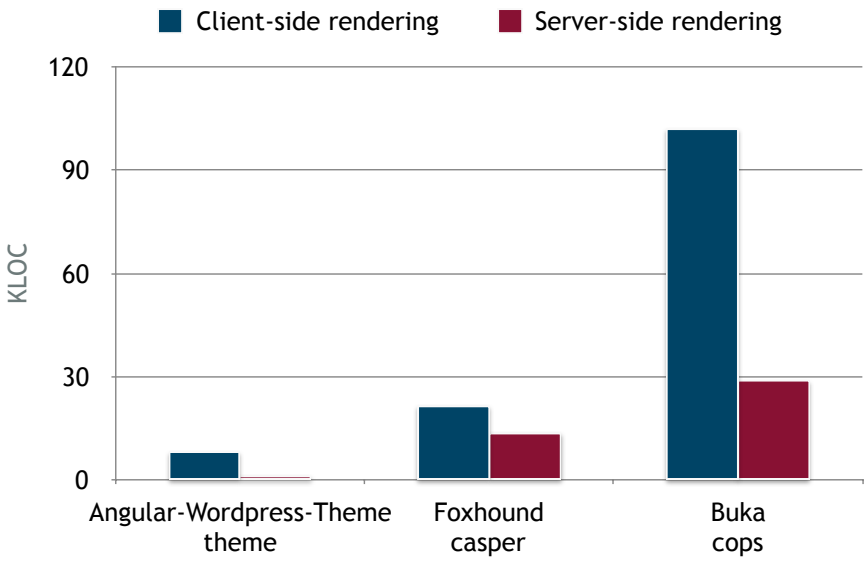



**Figure 5.7:** KLOC (thousands of lines of code) for each of the Github projects.

### 5.4.6 Availability

Running the front-end test continuously on the two Wordpress instances (one for each paradigm), yielded for both instances an uptime of 100%. This can be expected since the same server software is used and the same type of servers are used.

## 5.5 Conclusion

This case study proved the findings of the pilot study (Chapter 4) and added some new insights.

**Conclusion 5.5.1**
The following results were found on the open source projects:

1. Client-side rendering yields slower initial page load.
2. Client-side rendering yields faster next page load.
3. Client-side rendering yields an increased server throughput.
4. Client-side rendering scales better than server-side rendering.
5. Client-side rendering yields less server bandwidth consumption.
6. Client-side rendering needs more LOC for the same functionality.
7. There is no difference in availability for this particular test setup.

# CHAPTER 6
# Threats To Validity

In this research 5 out of the 8 different software quality attributes were compared in both a pilot study with newly written projects, and a case study with existing projects. Those other 3 attributes were left out because they couldn't be measured in the scope of this research.
This makes that the comparative study doesn't takes into account all relevant quality attributes. However, they were covered in the survey (Section 3.2), which gives a small indication of what the outcome would be if those were to be considered.

Although interesting projects for the case study were found on *Github*, it were only 3 interesting pairs of equivalent repositories, covering only a small part of web application functionalities.

The study being done on the test projects written in *Go* and *Laravel* combined with the *Wordpress* test cases yields a representative study. They cover the most significant portion of used web software [16] [13]. There are however other major platforms that were not considered in this study, like *APS.NET*, Java, *Ruby on Rails*, *Django*, and possible others, but Section 2.3 showed that *PHP* is the most used [16] [89].

The tests on the client were ran with three major front-end frameworks *Vue*, *Angular* and *React* [38] [82]. Together they give a good representation of used front-end frameworks in real world, as seen in Section 2.3.

How the used technologies represent the real world was throughly discussed in Section 2.3.

# CHAPTER 7
# Conclusions

This research started by introducing the need for a comparative study regarding client-side rendering and client-side rendering (Section 1.1). Then the fast evolving of the web and it's technologies was considered in Chapter 2. Chapter 3 first introduces relevant software quality attributes, selected from literature (Section 3.1). A developer oriented survey was then conducted on these attributes (Section 3.2).

The pilot study on self-written test projects in Chapter 4 combined with the case study on existing projects in Chapter 5 made it possible to measure the quality attributes and to point out the differences between client-side rendering and server-side rendering.

**Conclusion 7.0.1**
The proposed software quality attributes could be quantified.

Those quantifications made it possible to analyse the differences between the quality attributes and propose which of the paradigms is to be used to optimise a certain quality attribute.
This yields the following conclusions:

**Conclusion 7.0.2**
Client-side rendering is to be preferred when the following quality attributes are important:

1. Performance: server bandwidth
2. Performance: server throughput
3. Performance: next page loads
4. Usability
5. Scalability

**Conclusion 7.0.3**
Server-side rendering is to be preferred when the following quality attributes are important:

1. Initial page loads
2. Development Effort
3. Search Engine Optimisation
4. Browser Compatibility

Based on the gathered results, a decision tree was constructed which helps developers in making the choice between client-side rendering and server-side rendering. See Figure 7.1. This decision tree also proposes the hybrid approach, which consists of serving the first page with server-side rendering — requiring of course more development effort.

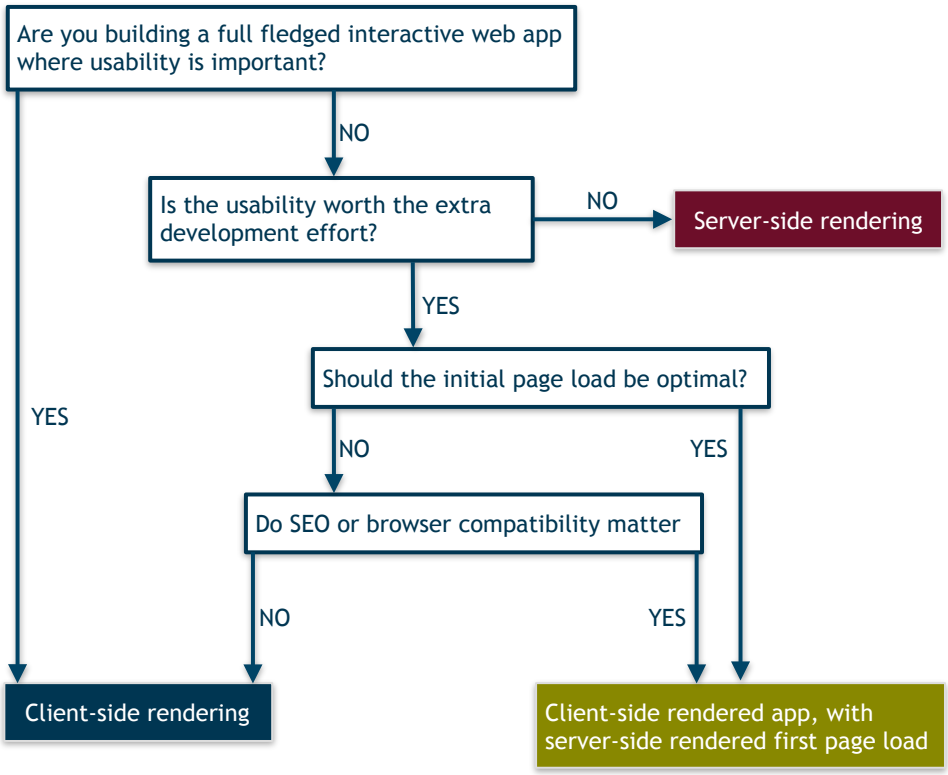



**Figure 7.1:** Decision tree which aids developers in making the choice between client-side rendering and server-side rendering.

# CHAPTER 8
# Future Work

Future work consists of generalising the found results.
The performed case study could be enhanced to a larger scale: doing more studies with different project types would allow for more generalisable results. It could also allow the theoretical models for the development effort and usability to be worked out.
To make attributes comparable to other attributes — in order to estimate the impact of attributes —, the results for the attributes should be denoted as general as possible. This future work section proposes a way to generalise quality attributes in terms of money. Money is always something that is considered and it can — once quantified — be compared between attributes. That way the impact of the rendering paradigms can be expressed as a sum of increases and decreases in revenue, yielding a total which aids in making the choice between the paradigms.

The remainder of this chapter sketches how quality attributes could be generalised.

**Client page load time**
Based on the network waterfalls from the pilot study (Figure 4.1 and Figure 4.2) it becomes clear that some requests depend on each other, blocking the actual rendering of the content until that chain of blocking requests is resolved.
The load time $T_r$ of a single blocking request $BR$ depends on the fixed average client bandwidth $B_c$, client average latency $L_c$, server processing delay $D_s$, and the size of the request $S(req)_{BR}$ and response $S(res)_{BR}$. Thus, the load time $T_{BR}$ can be computed as in Equation 8.0.1.

**Equation 8.0.1**

$$T_{BR} = L_c + \frac{S(req)_{BR}}{B_c} + D_s + \frac{S(res)_{BR}}{B_c}$$

Therefore, to generalise based on the waterfall, the complete load time $T_{load}$ is the sum of all consecutive blocking requests in the waterfall (Equation 8.0.2).

**Equation 8.0.2**

$$T_{load} = \sum_{i=1}^{n} T_{BR_i}$$

Where $n$ is the number of blocking requests in a rendering session. Using the above generalisation, the increase or decrease in load times for each individual project can be estimated.

Since the page abandonment rate is proportional to the load times, and the generated revenue $R$ is proportional to the page abandonment rate, using the abandonment rate $AR(T_{load})$ for the calculated load time the gain or loss in revenue can estimated as in Equation 8.0.3.

**Equation 8.0.3**

$$R = (1 - AR(T_{load})) * Average(R_{client}) * \#visitors$$

Where $Average(R_{client})$ is the average revenue generated per visitor. And $\#visitors$ is the number of visitors of the website. The difference in revenue for both paradigms can then be denoted as in Equation 8.0.4.

**Equation 8.0.4**

$$\Delta R = R_C - R_S$$

Where $R_C$ is the revenue estimated for the client-side rendering paradigm and $R_S$ is revenue estimated for the server-side rendering paradigm.

**Client: subsequent page load time**

Again since the difference in conversion rate $CR(T_{load})$ is proportional to the consecutive load times and the revenue is proportional to the conversion rate [78], the change in revenue (reusing the $T_{load}$ from Equation 8.0.2) is as in Equation 8.0.5.

**Equation 8.0.5**

$$R = CR(T_{load}) * AR(checkout) * \#visitors$$

Where $CR(T_{load})$ is the conversion rate for a given load time. This means that a $\Delta R$ can again be calculated as seen in Equation 8.0.4.

**Server: throughput**
The server throughput could also be generalised. For others projects one could profile the application (assuming there is already a server-side implementation) and calculate the amount of time spent on rendering. The upper bound of the gain in throughput $\delta_{throughput}$ is then given in Equation 8.0.6.

**Equation 8.0.6**

$$\delta_{throughput} = \frac{total\ time}{total\ time - rendering\ time}$$

Where *total time* is the total time needed to process a request and *rendering time* is time spent on rendering the HTML content for the particular request. A higher throughput means that less server resources are needed to handle the same amount of clients. This means that after calculating the upper bound of the gain in throughput, also the lower bound of optimised server cost can be calculated (assuming the current server cost $ServerCost_C$ needed for the throughput is known).

**Equation 8.0.7**

$$ServerCost_C = \frac{ServerCost_S}{\delta_{throughput}}$$

Where $ServerCost_C$ is the server costs for client-side rendering, while $ServerCost_S$ is the server costs for server-side rendering.
This can then again be written as a difference in value to compare those values with other values in the framework:

**Equation 8.0.8**

$$\Delta cost = ServerCost_C - ServerCost_S$$

Where $\Delta cost$ is the difference in cost and is useful to compare the increase or decrease in cost by switching between the paradigms.

**Server: bandwidth**
The server bandwidth could also be generalised:

**Equation 8.0.9**

$$\delta_B = \frac{B_{serialised}}{B_{HTML}}$$

Where $\delta_B$ is the gain in bandwidth, because serialising the data (denoted by $B_{serialised}$) generates smaller data size than HTML encoded data (denoted by $B_{HTML}$). This can lead to a decrease in bandwidth cost.

Given the number of requests *#requests* handled by the server, the size of those requests *size(request)* and server data cost *cost(byte)*, the cost of the bandwidth *cost(B)* can be calculated as in Equation 8.0.10.

**Equation 8.0.10**

$$cost(B) = \#requests * size(request) * cost(byte)$$

The new cost as a result of the gain in bandwidth for client-side rendering $C$ is then as in Equation 8.0.11.

**Equation 8.0.11**

$$cost_C(B) = \delta_B * cost_S(B)$$

This can then again be written as a difference in value to compare those values with other values in the framework (Equation 8.0.12).

**Equation 8.0.12**

$$\Delta cost = \quad cost_C(B) - cost_S(B)$$

# References


[1] Zed Abbadi. "Security metrics what can we measure?" In: *Open Web Application Security Project (OWASP), Nova Chapter meeting presentation on security metrics, viewed*. Vol. 2. 2011.

[2] MH Abdullah and Reena Srivastava. "Testability Measurement Model for Object Oriented Design (TMMOOD)". In: *arXiv preprint arXiv:1503.05493* (2015).

[3] Sareh Aghaei, Mohammad Ali Nematbakhsh, and Hadi Khosravi Farsani. "Evolution of the world wide web: From WEB 1.0 TO WEB 4.0". In: *International Journal of Web & Semantic Technology* 3.1 (2012), p. 1.

[4] Jose Aguinaga. *How it feels to learn JavaScript in 2016*. [Online; accessed 21-February-2018]. 2016. URL: `https://hackernoon.com/how-it-feels-to-learn-javascript-in-2016-d3a717dd577f`.

[5] W. Albert and T. Tullis. *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics*. Interactive Technologies. Elsevier Science, 2013. ISBN: 9780124157927.

[6] Kristofer Baxter. *Netflix Technology Blog - Making Netflix.com Faster*. [Online; accessed 10-January-2018]. 2015. URL: `http://techblog.netflix.com/2015/08/making-netflixcom-faster.html`.

[7] Tim Berners-Lee. "The WorldWideWeb browser, 1990". In: *URL http://www.w3. org/People/Berners-Lee/WorldWideWeb. html. Pristupljeno* 5 (2013).

[8] Tim Berners-Lee and Daniel Connolly. "Hypertext markup language (html)". In: *CERN, Geneva, Switzerland* 13 (1993).

[9] Timothy J Berners-Lee. *Information management: A proposal*. Tech. rep. 1989.

[10] Andreas Biørn-Hansen, Tim A Majchrzak, and Tor-Morten Grønli. "Progressive web apps: The possible web-native unifier for mobile development". In: *Proceedings of the 13th International Conference on Web Information Systems and Technologies*. 2017, pp. 344–351.

[11] Spike Brehm. *Airbnb Engineering & Data Science - Isomorphic JavaScript: The Future of Web Apps*. [Online; accessed 10-January-2018]. 2013. URL: http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/.

[12] Spike Brehm. *The future of web apps is - ready? - isomorphic JavaScript*. [Online; accessed 24-May-2018]. 2013. URL: https://venturebeat.com/2013/11/08/the-future-of-web-apps-is-ready-isomorphic-javascript/.

[13] BuiltWith. *CMS Usage Statistics*. [Online; accessed 20-May-2018]. 2018. URL: https://trends.builtwith.com/cms.

[14] BuiltWith. *JavaScript Library Usage*. [Online; accessed 11-June-2018]. 2018. URL: https://trends.builtwith.com/javascript/javascript-library.

[15] BuiltWith. *Laravel Usage Statistics*. [Online; accessed 11-June-2018]. 2018. URL: https://trends.builtwith.com/framework/Laravel.

[16] BuiltWith. *Programming Language Usage*. [Online; accessed 2-June-2018]. 2018. URL: https://trends.builtwith.com/framework/programming-language.

[17] Domain7 contributors. *Usability vs. User Experience: What's the difference?* [Online; accessed 4-January-2018]. 2014. URL: https://www.slideshare.net/domain7/ux-vs-usability.

[18] Cliff Crocker, Aaron Kulick, and Balaji Ram. *Real User Monitoring @ Walmart.com: A Story in 3 Parts*. [Online; accessed 28-December-2017]. 2012. URL: https://www.slideshare.net/devonauerswald/walmart-pagespeedslide.

[19] Serge Demeyer. *Software Metrics*. [Online; accessed 30-December-2017]. 2017. URL: http://ansymore.uantwerpen.be/system/files/uploads/courses/SE3BAC/10Metrics17.pdf.

[20] Sebastian De Deyne. *Considering Single Page Applications*. [Online; accessed 1-June-2018]. 2018. URL: https://sebastiandedeyne.com/slides/considering-single-page-applications.pdf.

[21] Roger Dooley. *Don't Let a Slow Website Kill Your Bottom Line*. [Online; accessed 28-December-2017]. 2012. URL: https://www.forbes.com/sites/rogerdooley/2012/12/04/fast-sites/#4f42d2eb53cf.

[22] Kit Eaton. *How One Second Could Cost Amazon $1.6 Billion In Sales.* [Online; accessed 12-June-2018]. 2012. URL: https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales.

[23] Jason Farrell and George S Nezlek. "Rich internet applications the next stage of application development". In: *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on.* IEEE. 2007, pp. 413–418.

[24] Matthias Felleisen. "On the expressive power of programming languages". In: *Science of computer programming* 17.1-3 (1991), pp. 35–75.

[25] Rudolf Ferenc, Péter Hegedűs, and Tibor Gyimóthy. "Software product quality models". In: *Evolving software systems.* Springer, 2014, pp. 65–100.

[26] Dan Gebhardt. *THE 'DEVELOPMENT DRAWBACKS' OF JAVASCRIPT WEB APPLICATIONS.* [Online; accessed 21-February-2018]. 2013. URL: http://www.cerebris.com/blog/2013/08/08/the-development-drawbacks-of-javascript-web-applications/.

[27] Brian Getting. "Basic Definitions: Web 1.0, Web. 2.0, Web 3.0". In: *Practical eCommerce: Insights for Online Merchants.* (2007). URL: http://www.practicalecommerce.com/articles/464-Basic-Definitions-Web-1-0-Web-2-0-Web-3-0.

[28] Github. *Collection: Front-end JavaScript frameworks.* [Online; accessed 2-November-2017]. 2017. URL: https://github.com/collections/front-end-javascript-frameworks.

[29] Gomez. *Why Web Performance Matters: Is Your Site Driving Customers Away?* [Online; accessed 10-January-2018]. 2017. URL: http://www.mcrinc.com/Documents/Newsletters/201110_why_web_performance_matters.pdf.

[30] Sacha Greif. *Front-end Frameworks.* [Online; accessed 2-November-2017]. 2016. URL: http://stateofjs.com/2016/frontend/.

[31] Alex Grigoryan. *The Benefits of Server Side Rendering Over Client Side Rendering.* [Online; accessed 20-February-2018]. 2017. URL: https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8.

[32] Samer Hanna and Ali Alawneh. "An Approach of Web Service Quality Attributes Specification". In: *Communications of the IBIMA Journal (ISSN: 1943-7765)* (2010).

[33] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A practical model for measuring maintainability". In: *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE. 2007, pp. 30–39.

[34] Cory House. *Here's Why Client-side Rendering Won – freeCodeCamp*. [Online; accessed 20-February-2018]. Dec. 2016. URL: https://medium.freecodecamp.org/heres-why-client-side-rendering-won-46a349fadb52.

[35] Patrick Hund. *SEO vs. React: Web Crawlers are Smarter Than You Think*. [Online; accessed 21-February-2018]. 2016. URL: https://medium.freecodecamp.org/seo-vs-react-is-it-neccessary-to-render-react-pages-in-the-backend-74ce5015c0c9.

[36] solid IT. *DB-Engines Ranking*. [Online; accessed 11-June-2018]. 2018. URL: https://db-engines.com/en/ranking.

[37] Madhuri A Jadhav, Balkrishna R Sawant, and Anushree Deshmukh. "Single page application using angularjs". In: *International Journal of Computer Science and Information Technologies* 6.3 (2015), pp. 2876–2879.

[38] Benjamin Jakobus. *VueJS vs Angular vs ReactJS with Demos*. [Online; accessed 5-June-2018]. 2017. URL: http://www.dotnetcurry.com/vuejs/1372/vuejs-vs-angular-reactjs-compare.

[39] Niklas Johansson and Anton Löfgren. "Designing for Extensibility: An action research study of maximizing extensibility by means of design principles". B.S. thesis. 2009.

[40] Eirini Kalliamvakou et al. "The promises and perils of mining github". In: *Proceedings of the 11th working conference on mining software repositories*. ACM. 2014, pp. 92–101.

[41] Yiannis Kanellopoulos et al. "Code quality evaluation methodology using the ISO/IEC 9126 standard". In: *arXiv preprint arXiv:1007.5117* (2010).

[42] David Kreps and Kai Kimppa. "Theorising Web 3.0: ICTs in a changing society". In: *Information Technology & People* 28.4 (2015), pp. 726–741.

[43] Gajus Kuizinas. *Pre-rendering SPA for SEO and improved perceived page loading speed*. [Online; accessed 1-June-2018]. 2017. URL: https://medium.com/@gajus/pre-rendering-spa-for-seo-and-improved-perceived-page-loading-speed-47075aa16d24.

[44] Nupul Kukreja. *Measuring Software Maintainability*. [Online; accessed 1-June-2018]. 2015. URL: https://quandarypeak.com/2015/02/measuring-software-maintainability/.

[45] Sergey Laptick. *Client Side vs Server Side UI Rendering. Advantages and Disadvantages.* [Online; accessed 20-February-2018]. 2017. URL: https://blog.webix.com/client-side-vs-server-side-ui-rendering/.

[46] Tim Berners Lee. "HTTP 0.9". In: (1991).

[47] Simon Lesser. *Technical and On-Page SEO Guide for Baidu.* [Online; accessed 21-February-2018]. 2017. URL: http://www.dragonmetrics.com/technical-on-page-seo-guide-baidu/.

[48] Michael Lester. *Measuring security.* [Online; accessed 1-June-2018]. 2016. URL: https://www.csoonline.com/article/3112029/security-awareness/measuring-security.html.

[49] Hakon Wium Lie. "Cascading HTML style sheets-a proposal". In: *World Wide Web Consortium (W3C)* (1994).

[50] Ben Lilburne et al. "Measuring quality metrics for web applications". In: *Proceedings of the 15th Information Resources Management Association International Conference, held in New Orleans, USA, 23-26 May, 2004.* 2004.

[51] Ari Luotonen, Henrik Frystyk, and Tim Berners-Lee. *CERN HTTPD public domain fullfeatured hypertext/proxy server with caching, 1994.*

[52] Michael Mazzei. "Web 2.0." In: *Salem Press Encyclopedia of Science* (2017).

[53] Emilia Mendes, Nile Mosley, and Steve Counsell. "The need for web engineering: An introduction". In: *Web Engineering.* Springer, 2006, pp. 1–27.

[54] Ali Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Delft University of Technology, Netherlands, 2009.

[55] Ali Mesbah and Arie Van Deursen. "Migrating multi-page web applications to single-page Ajax interfaces". In: *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on.* IEEE. 2007, pp. 181–190.

[56] Microsoft. *Design Fundamentals: Quality Attributes.* [Online; accessed 7-May-2018]. URL: https://msdn.microsoft.com/en-us/library/ee658094.aspx.

[57] Michael S Mikowski and Josh C Powell. "Single page web applications". In: *B and W* (2013).

[58] Luca Mugnaini. *SPA and SEO: Google (Googlebot) properly renders Single Page Application and execute Ajax calls.* [Online; accessed 21-February-2018]. 2017. URL: https://medium.com/@l.mugnaini/spa-and-seo-is-googlebot-able-to-render-a-single-page-application-1f74e706ab11.

[59] Doaa Nabil, Abeer Mosad, and Hesham A Hefny. "Web-Based Applications quality factors: A survey and a proposed conceptual model". In: *Egyptian Informatics Journal* 12.3 (2011), pp. 211–217.

[60] Muhammad Nadeem et al. "Analysis and Comparison of Web Development Platforms Based on Software Quality Attributes in Network Management System". In: *Journal of Advances in Computer Networks*. Vol. 5. 1. Springer. 2017, pp. 18–21.

[61] Kazushi Nagayama. *Google Webmaster Central Blog: Deprecating our AJAX crawling scheme*. [Online; accessed 21-February-2018]. 2015. URL: https://webmasters.googleblog.com/2015/10/deprecating-our-ajax-crawling-scheme.html.

[62] Fiona Fui-Hoon Nah. "A study on tolerable waiting time: how long are Web users willing to wait?" In: *Behaviour & Information Technology* 23.3 (2004), pp. 153–163. DOI: 10.1080/01449290410001669914. eprint: https://doi.org/10.1080/01449290410001669914. URL: https://doi.org/10.1080/01449290410001669914.

[63] Jeff Offutt. "Web software applications quality attributes". In: *Quality Engineering in Software Technology (CONQUEST 2002)* (2002), pp. 187–198.

[64] Vivek Ojha. *12 Attributes of a good web application architecture*. [Online; accessed 1-June-2018]. 2015. URL: https://www.linkedin.com/pulse/12-attributes-good-web-application-architecture-vivek-ojha/.

[65] Luis Olsina, Guillermo Lafuente, and Gustavo Rossi. "Specifying quality characteristics and attributes for websites". In: *Web Engineering*. Springer, 2001, pp. 266–278.

[66] Tim O'Reilly. "Web 2.0 Compact definition: trying again. 2006". In: *http://radar.oreilly.com/2006/12/web-20-compact-definition-tryi. html. Acesso em* 18.01 (2008), p. 215.

[67] Tim O'reilly. *What is web 2.0*. 2005.

[68] Meysam Ahmadi Oskooei, Salwani Binti Mohd Daud, and Fang-Fang Chua. "Modeling quality attributes and metrics for web service selection". In: *AIP Conference Proceedings*. Vol. 1602. 1. AIP. 2014, pp. 945–952.

[69] Pingdom. *A history of the dynamic web*. [Online; accessed 6-May-2018]. 2007. URL: https://royal.pingdom.com/2007/12/07/a-history-of-the-dynamic-web/.

[70] Max Pool. *The 7 Software "-ilities" You Need To Know*. [Online; accessed 7-May-2018]. 2007. URL: http://codesqueeze.com/the-7-software-ilities-you-need-to-know/.

[71] Clark Quinn. *Beyond Web 2.0*. [Online; accessed 7-May-2018]. 2009. URL: https://blog.learnlets.com/2009/07/beyond-web-2-0/.

[72] Alex Russell. *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. [Online; accessed 7-May-2018]. 2015. URL: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/.

[73] Fintan Ryan. *Language Framework Popularity: A Look at PHP*. [Online; accessed 2-June-2018]. 2016. URL: http://redmonk.com/fryan/2016/11/01/language-framework-popularity-a-look-at-php/.

[74] Durgesh Samadhiya, Su-Hua Wang, and Dengjie Chen. "Quality models: Role and value in software engineering". In: *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*. Vol. 1. IEEE. 2010, pp. V1–320.

[75] Alberto Savoia. "The science and art of web site load testing". In: *International Conference on Software Testing Analysis & Review, Orlando, FL*. 2000.

[76] Britt Selvitelle. *The Tech Behind the New Twitter.com*. [Online; accessed 10-January-2018]. 2010. URL: https://blog.twitter.com/engineering/en_us/a/2010/the-tech-behind-the-new-twittercom.html.

[77] Brett Slatkin. *Experimentally verified: 'Why client-side templating is wrong'*. [Online; accessed 21-February-2018]. 2015. URL: https://www.onebigfluke.com/2015/01/experimentally-verified-why-client-side.html.

[78] D. Soman and S. N-Marandi. *Managing Customer Value: One Stage at a Time*. World Scientific, 2010. ISBN: 9789812838278.

[79] Steve Souders. "High-performance web sites". In: *Communications of the ACM* 51.12 (2008), pp. 36–41.

[80] Wiktor Stadnik and Ziemowit Nowak. "The Impact of Web Pages' Load Time on the Conversion Rate of an E-Commerce Platform". In: (2017), pp. 336–345.

[81] A. Sviatoslav. *The Best JS Frameworks for Front End*. [Online; accessed 11-June-2018]. URL: https://rubygarage.org/blog/best-javascript-frameworks-for-front-end.

[82] TechMagic. *ReactJS vs Angular5 vs Vue.js – What to choose in 2018?* [Online; accessed 6-June-2018]. 2018. URL: https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d.

[83] Google Trends. *REST API vs SOAP API vs GraphQL API*. [Online; accessed 11-June-2018]. 2018. URL: https://g.co/trends/PTjYq.

[84] Malte Ubl. *Tradeoffs in server side and client side rendering.* [Online; accessed 21-February-2018]. 2015. URL: https://medium.com/google-developers/tradeoffs-in-server-side-and-client-side-rendering-14dad8d4ff8b.

[85] Edward Upton. *How does page load speed affect bounce rate?* [Online; accessed 28-December-2017]. 2017. URL: https://blog.littledata.io/2017/04/07/how-does-page-load-speed-affect-bounce-rate/.

[86] Chad Van Lier. *How Single Page Applications (SPAs) Can Dramatically Improve Your Customer Experience.* [Online; accessed 1-June-2018]. 2017. URL: https://www.exclamationlabs.com/blog/how-single-page-applications-spas-can-dramatically-improve-your-customer-experience/.

[87] Shiju Varghese. *Introduction to Web Development in Go.* [Online; accessed 2-June-2018]. 2016. URL: https://blog.rubylearning.com/introduction-to-web-development-in-go-3a126626eab.

[88] W3Techs. *Usage of JavaScript libraries for websites.* [Online; accessed 11-June-2018]. 2018. URL: https://w3techs.com/technologies/overview/javascript_library/all.

[89] W3Techs. *Usage of server-side programming languages for websites.* [Online; accessed 11-June-2018]. 2018. URL: https://w3techs.com/technologies/overview/programming_language/all.

[90] Dan Webb. *Improving performance on twitter.com.* [Online; accessed 10-January-2018]. 2012. URL: https://blog.twitter.com/engineering/en_us/a/2012/improving-performance-on-twittercom.html.

[91] Jeff Whelpley. *Use Cases for Server Side Rendering.* [Online; accessed 1-June-2018]. 2015. URL: https://medium.com/@jeffwhelpley/use-cases-for-server-side-rendering-2fc6389b3f7d.

[92] Joost Willemsen. *Improving User Workflows with Single-Page User Interfaces.* [Online; accessed 1-June-2018]. 2006. URL: https://www.uxmatters.com/mt/archives/2006/11/improving-user-workflows-with-single-page-user-interfaces.php.

[93] Lloyd G Williams and Connie U Smith. "Web Application Scalability: A Model-Based Approach." In: *Int. CMG Conference.* 2004, pp. 215–226.

[94] Alexander Zarges. *Client vs serverside rendering – the big battle?* [Online; accessed 1-June-2018]. 2013. URL: https://blog.mwaysolutions.com/2013/11/08/client-vs-serverside-rendering-the-big-battle-2/.

[95] Adam Zerner. *Client-side rendering vs. server-side rendering.* [Online; accessed 21-February-2018]. 2017. URL: https://medium.com/@adamzerner/client-side-rendering-vs-server-side-rendering-a32d2cf3bfcc.

[96] Luyin Zhao and Sebastian Elbaum. "Quality assurance under the open source development model". In: *Journal of Systems and Software* 66.1 (2003), pp. 65–75.

[97] Hazura Zulzalil et al. "A case study to identify quality attributes relationships for web-based applications". In: *IJCSNS* 8.11 (2008), p. 215.

# Appendix

## Survey

### Pie-charts

#### Demographic Related Questions

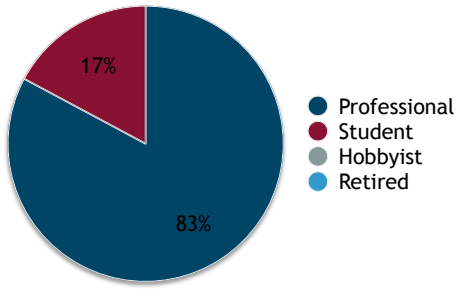



**Figure 8.1:** Occupation of the respondents

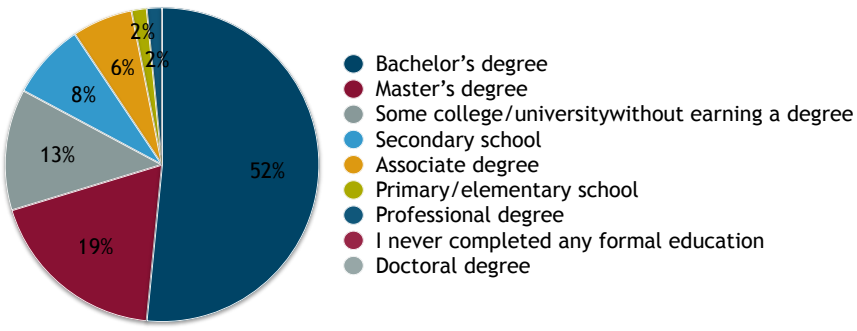



**Figure 8.2:** Highest Education of the respondents

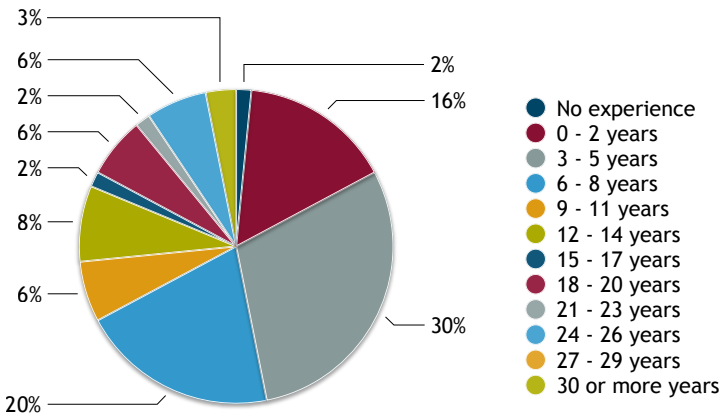



**Figure 8.3:** Experience in Software Development

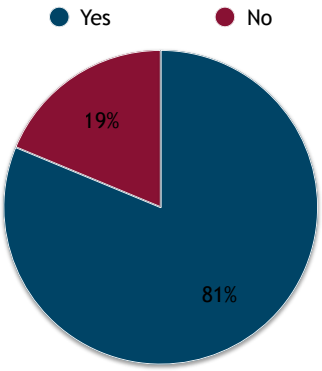



**Figure 8.4:** Experience with client-side rendering frameworks (like Angular, Vue, React).

### Quality Attributes Related Questions

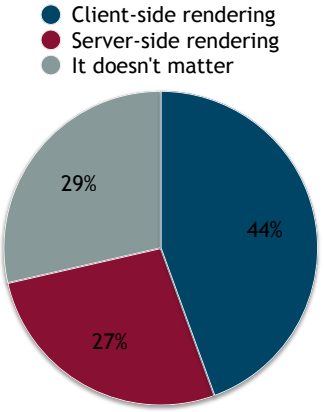



**Figure 8.5:** Preference of paradigm.

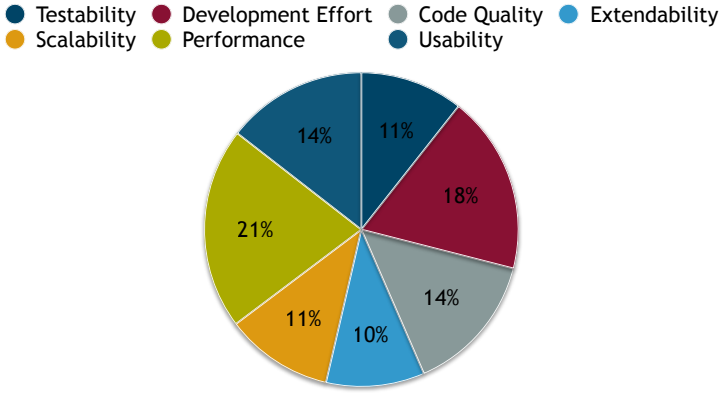



**Figure 8.6:** Which metrics have the most influence on the choice of paradigm.

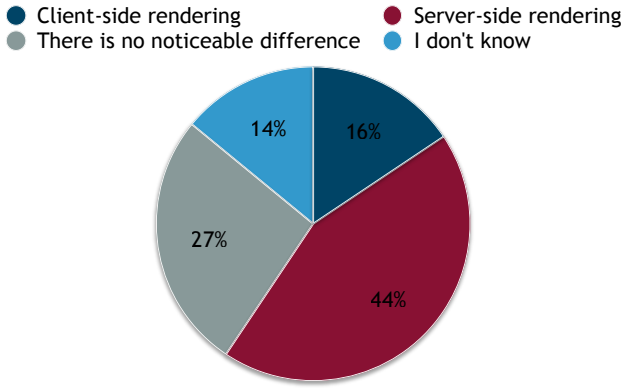



**Figure 8.7:** Which paradigm is most testable.

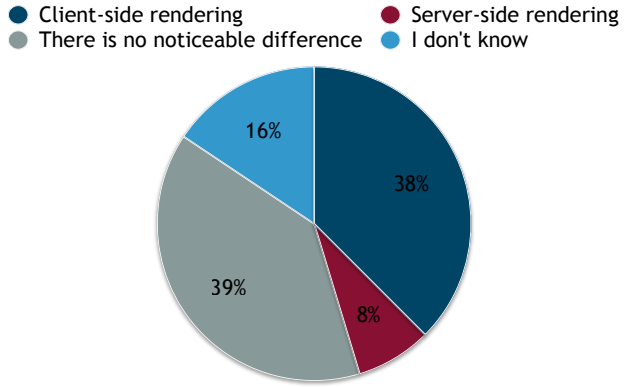



**Figure 8.8:** Which paradigm is most modifiable.

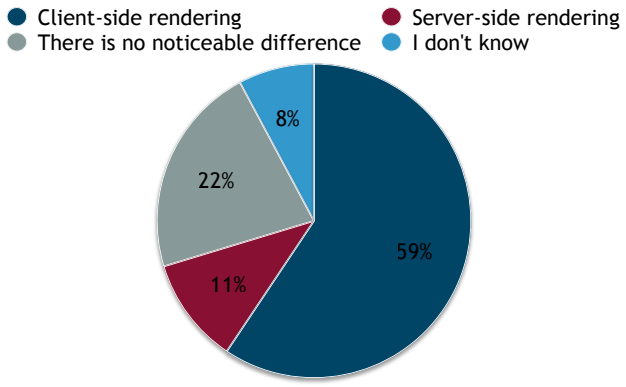



**Figure 8.9:** Which paradigm is most flexible.

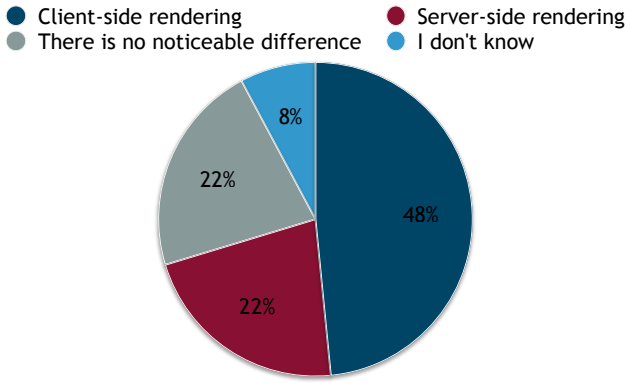



**Figure 8.10:** Which paradigm requires most development effort.

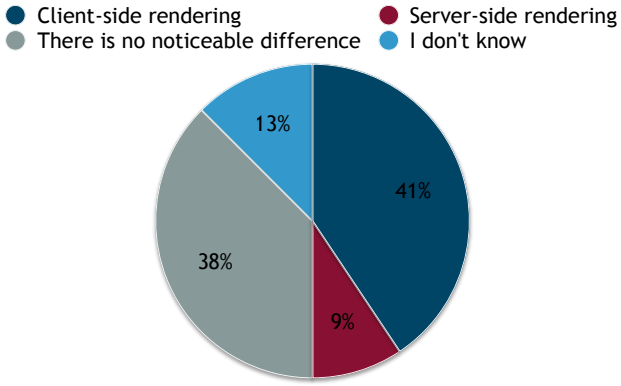



**Figure 8.11:** Which paradigm yields most duplicate code.

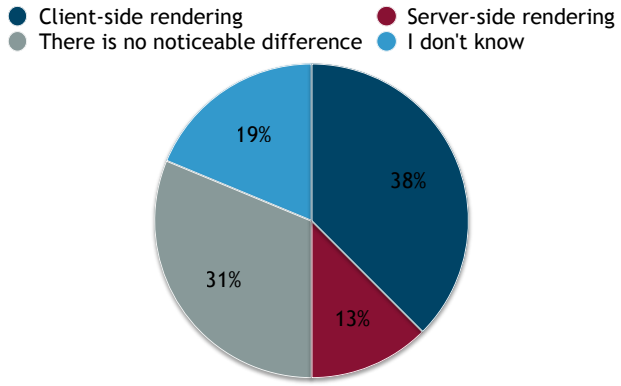



**Figure 8.12:** Which paradigm yields code defects/bugs.

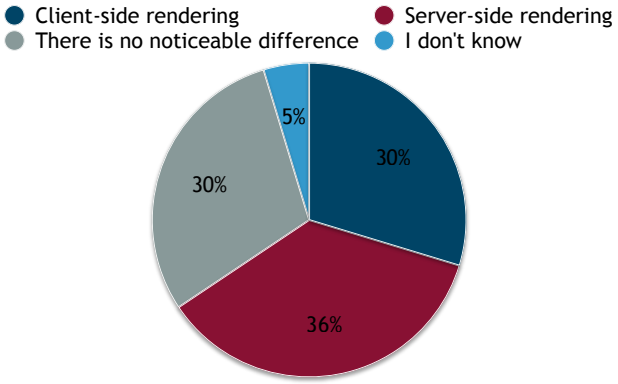



**Figure 8.13:** Which paradigm yields more maintainable code?

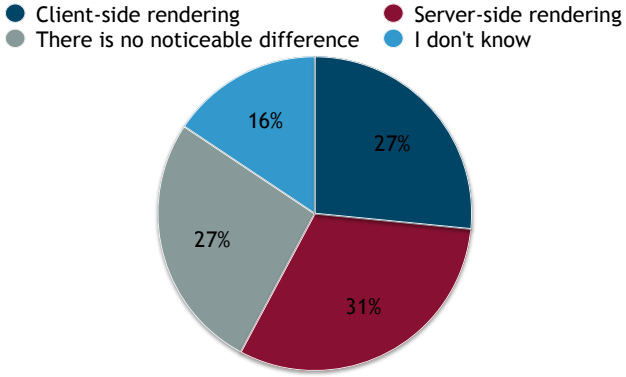



**Figure 8.14:** Which paradigm yields more efficient code?

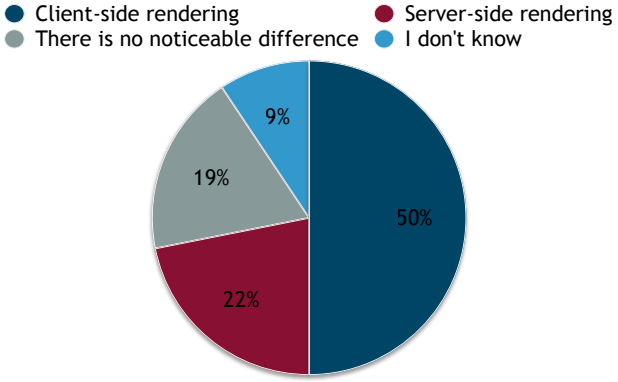



**Figure 8.15:** Which paradigm yields better usability?

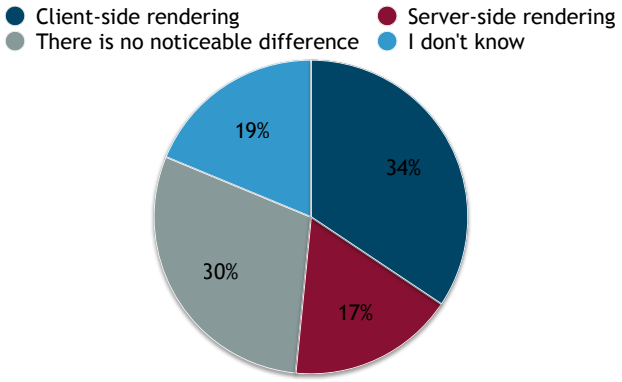



**Figure 8.16:** Which paradigm results in best extendability?

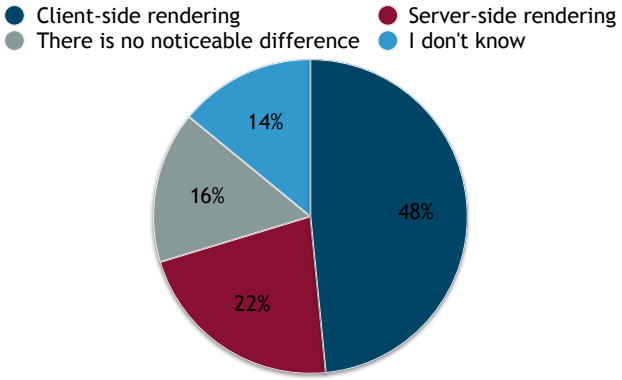



**Figure 8.17:** Which paradigm scales best?

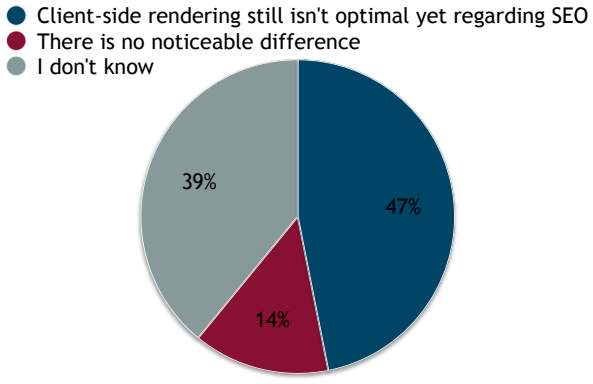



**Figure 8.18:** Do you think the impact of client-side rendering on SEO is still noticeable anno 2018?

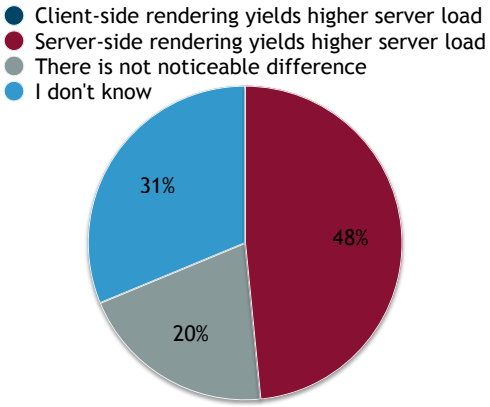



**Figure 8.19:** Did you notice impact on server load using one of the paradigms?

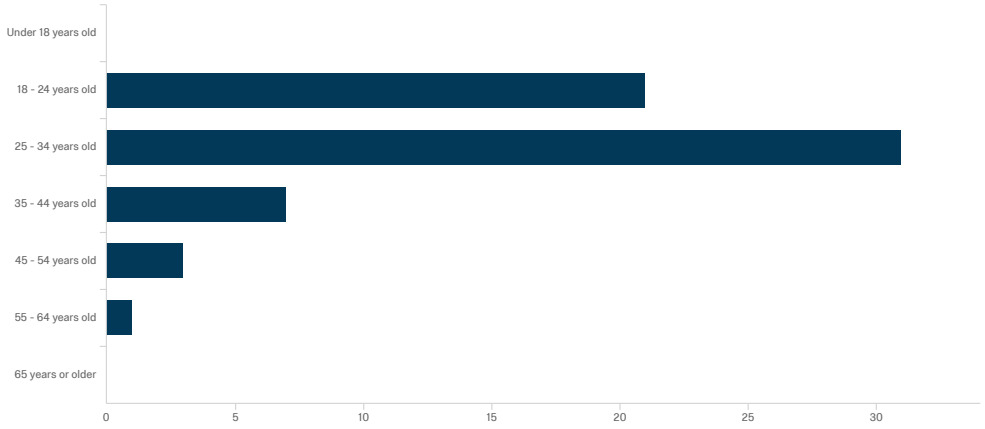
### Complete Survey Questions with Answers

# Thesis Survey

*Comparing Client-side rendering with Server-side rendering*

June 6, 2018 9:52 AM MDT

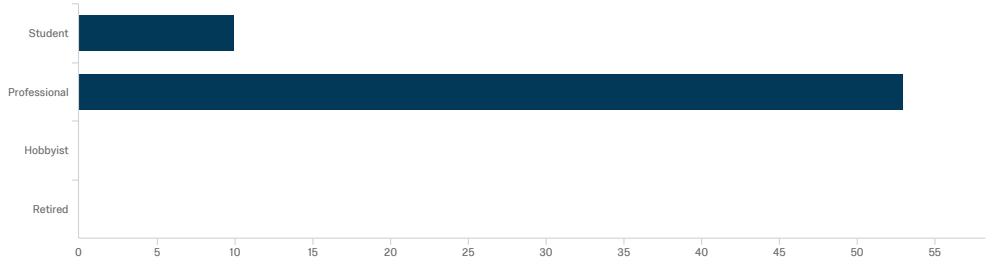
## 1 - Age




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Under 18 years old | 0.00% | 0 |
| 2 | 18 - 24 years old | 33.33% | 21 |
| 3 | 25 - 34 years old | 49.21% | 31 |
| 4 | 35 - 44 years old | 11.11% | 7 |
| 5 | 45 - 54 years old | 4.76% | 3 |
| 6 | 55 - 64 years old | 1.59% | 1 |
| 7 | 65 years or older | 0.00% | 0 |
| | | | 63 |

Showing Rows: 1 - 8 Of 8

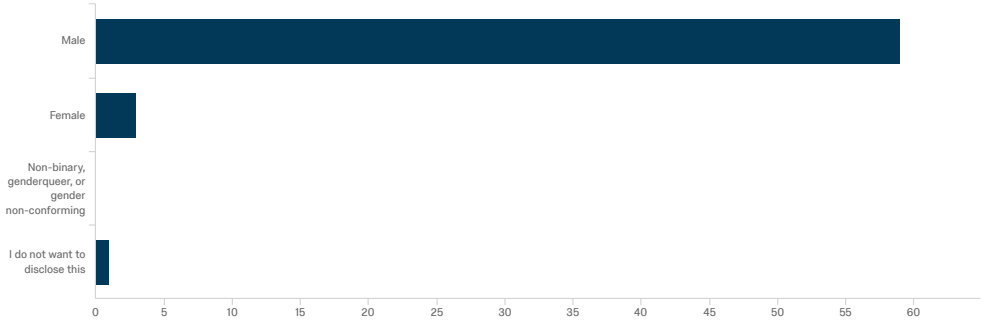
## 1-1 - Occupation




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Student | 15.87% | 10 |
| 2 | Professional | 84.13% | 53 |
| 3 | Hobbyist | 0.00% | 0 |
| 4 | Retired | 0.00% | 0 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

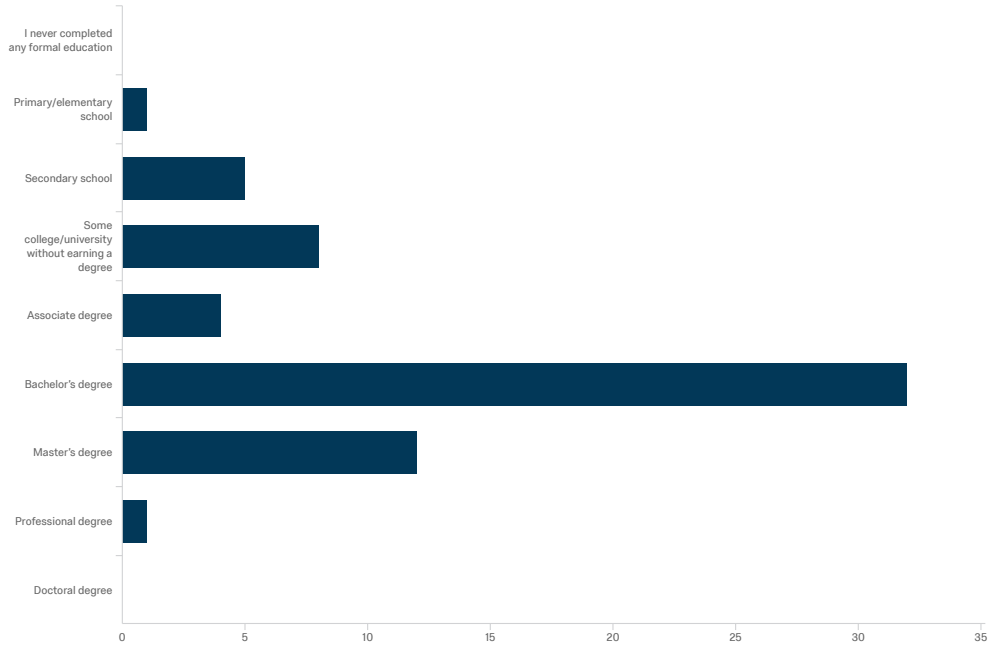
## 1-2 - Gender




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Male | 93.65% | 59 |
| 2 | Female | 4.76% | 3 |
| 3 | Non-binary, genderqueer, or gender non-conforming | 0.00% | 0 |
| 4 | I do not want to disclose this | 1.59% | 1 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

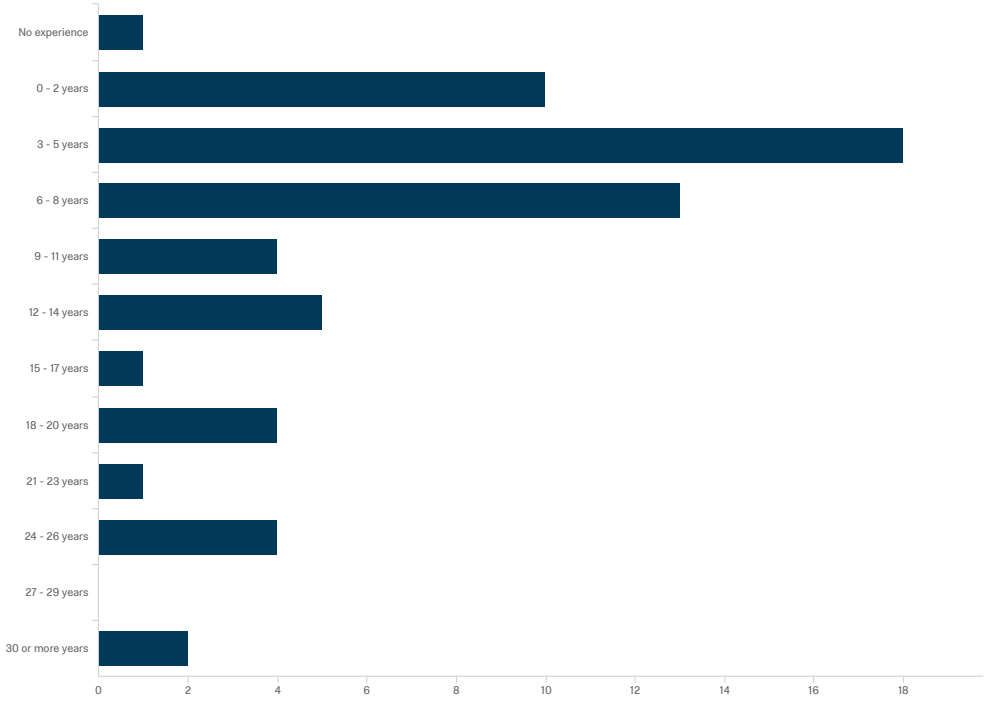
## 1-3 - Highest Education




| # | Field | Choice Count | |
|---|---|---|---|
| 1 | I never completed any formal education | 0.00% | 0 |
| 2 | Primary/elementary school | 1.59% | 1 |
| 3 | Secondary school | 7.94% | 5 |
| 4 | Some college/university without earning a degree | 12.70% | 8 |
| 5 | Associate degree | 6.35% | 4 |
| 6 | Bachelor's degree | 50.79% | 32 |
| 7 | Master's degree | 19.05% | 12 |
| 8 | Professional degree | 1.59% | 1 |
| 9 | Doctoral degree | 0.00% | 0 |
| | | | 63 |

Showing Rows: 1 - 10 Of 10

## 1-4 - Experience in Software Development

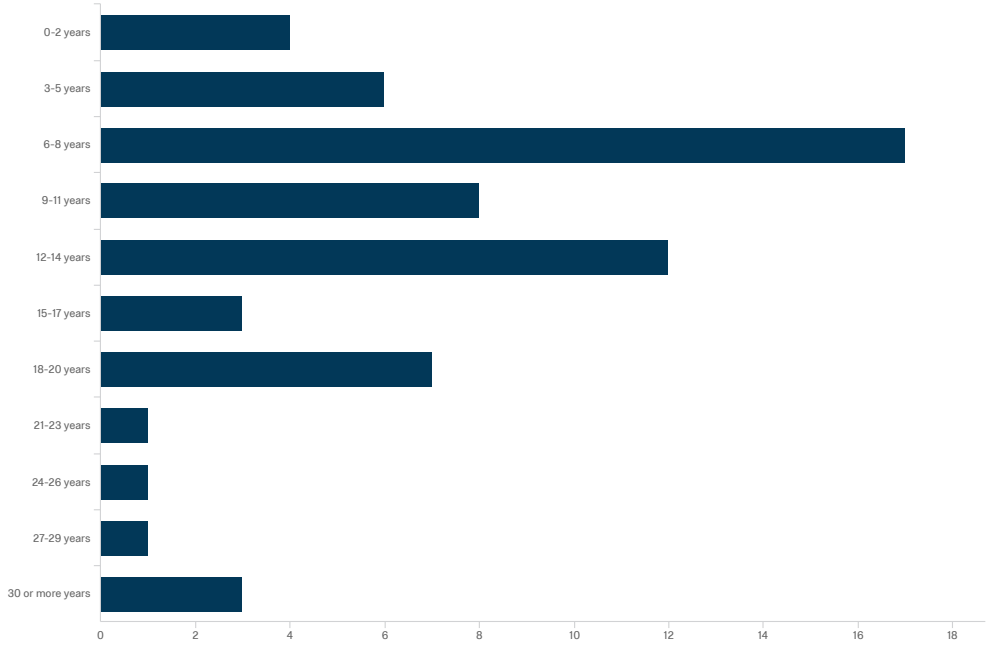



| # | Field | | Choice Count |
|---|---|---|---|
| 1 | No experience | 1.59% | 1 |
| 2 | 0 - 2 years | 15.87% | 10 |
| 3 | 3 - 5 years | 28.57% | 18 |
| 4 | 6 - 8 years | 20.63% | 13 |
| 5 | 9 - 11 years | 6.35% | 4 |
| 6 | 12 - 14 years | 7.94% | 5 |
| 7 | 15 - 17 years | 1.59% | 1 |
| 8 | 18 - 20 years | 6.35% | 4 |
| 9 | 21 - 23 years | 1.59% | 1 |
| 10 | 24 - 26 years | 6.35% | 4 |

| | | | |
|---|---|---|---|
| 11 | 27 - 29 years | 0.00% | 0 |
| 12 | 30 or more years | 3.17% | 2 |
| | | | 63 |

Showing Rows: 1 - 13 Of 13

## 1-5 - Years Since Learning to Code

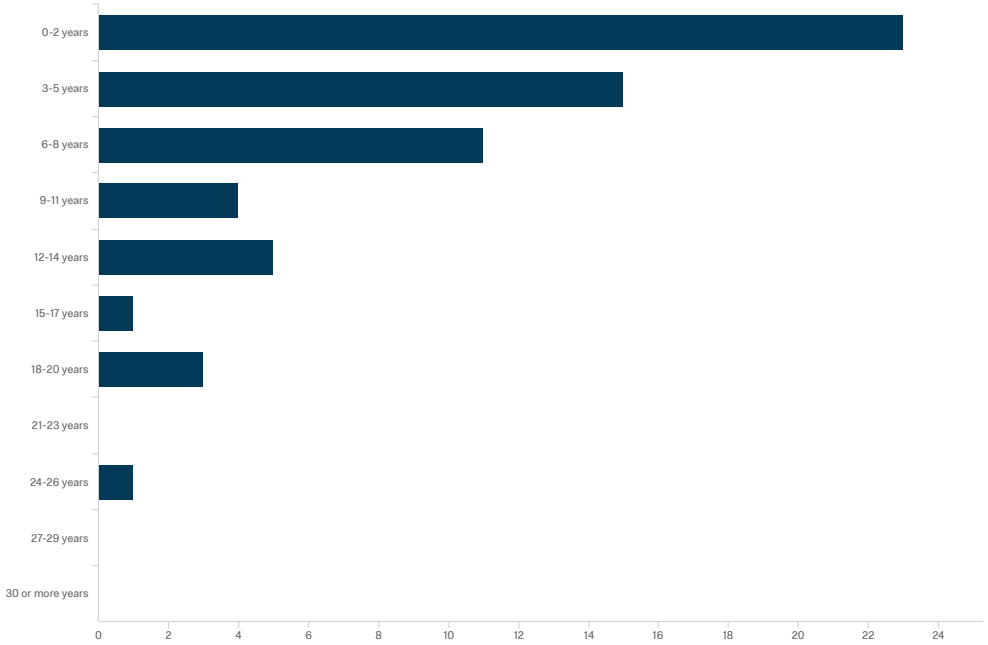



| # | Field | Choice Count | |
|---|---|---|---|
| 1 | 0-2 years | 6.35% | 4 |
| 2 | 3-5 years | 9.52% | 6 |
| 3 | 6-8 years | 26.98% | 17 |
| 4 | 9-11 years | 12.70% | 8 |
| 5 | 12-14 years | 19.05% | 12 |
| 6 | 15-17 years | 4.76% | 3 |
| 7 | 18-20 years | 11.11% | 7 |
| 8 | 21-23 years | 1.59% | 1 |
| 9 | 24-26 years | 1.59% | 1 |
| 10 | 27-29 years | 1.59% | 1 |
| 11 | 30 or more years | 4.76% | 3 |

63

Showing Rows: 1 - 12 Of 12

## 1-6 - Years of Professional Coding Experience

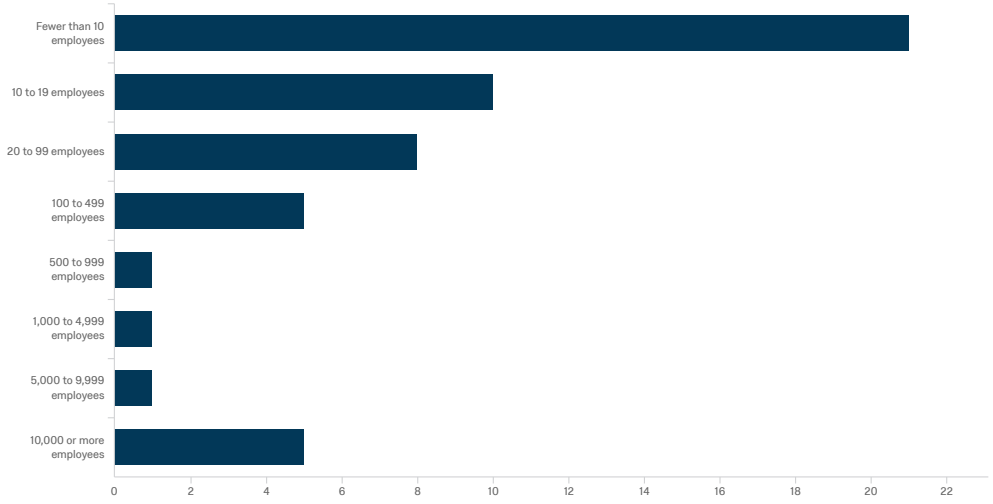



| # | Field | Choice Count | |
|---|---|---|---|
| 1 | 0-2 years | 36.51% | 23 |
| 2 | 3-5 years | 23.81% | 15 |
| 3 | 6-8 years | 17.46% | 11 |
| 4 | 9-11 years | 6.35% | 4 |
| 5 | 12-14 years | 7.94% | 5 |
| 6 | 15-17 years | 1.59% | 1 |
| 7 | 18-20 years | 4.76% | 3 |
| 8 | 21-23 years | 0.00% | 0 |
| 9 | 24-26 years | 1.59% | 1 |
| 10 | 27-29 years | 0.00% | 0 |
| 11 | 30 or more years | 0.00% | 0 |

63

Showing Rows: 1 - 12 Of 12

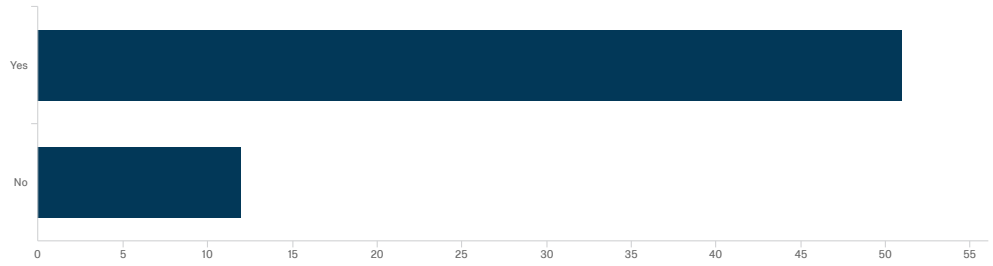
## 1-7 - Company Size




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Fewer than 10 employees | 40.38% | 21 |
| 2 | 10 to 19 employees | 19.23% | 10 |
| 3 | 20 to 99 employees | 15.38% | 8 |
| 4 | 100 to 499 employees | 9.62% | 5 |
| 5 | 500 to 999 employees | 1.92% | 1 |
| 6 | 1,000 to 4,999 employees | 1.92% | 1 |
| 7 | 5,000 to 9,999 employees | 1.92% | 1 |
| 8 | 10,000 or more employees | 9.62% | 5 |
| | | | 52 |

Showing Rows: 1 - 9 Of 9

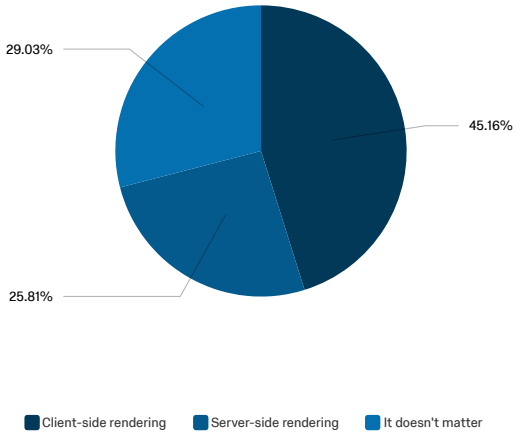
1-8 - Do you have experience with client-side rendering frameworks / single page application frameworks (like Angular, React, Vue, ...)?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Yes | 80.95% | 51 |
| 2 | No | 19.05% | 12 |
| | | | 63 |

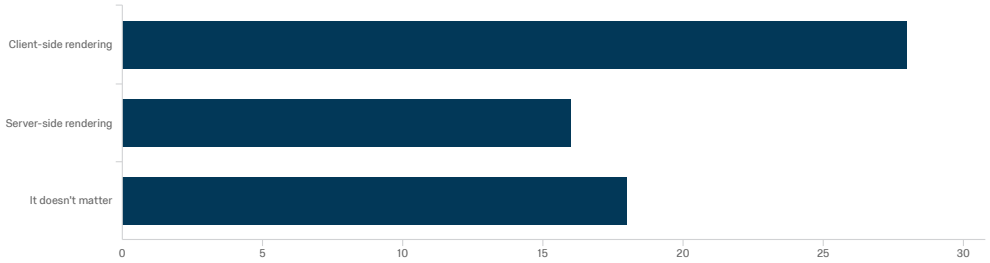
Showing Rows: 1 - 3 Of 3

1-9 - What's your preference? Server-side rendering or client-side rendering?

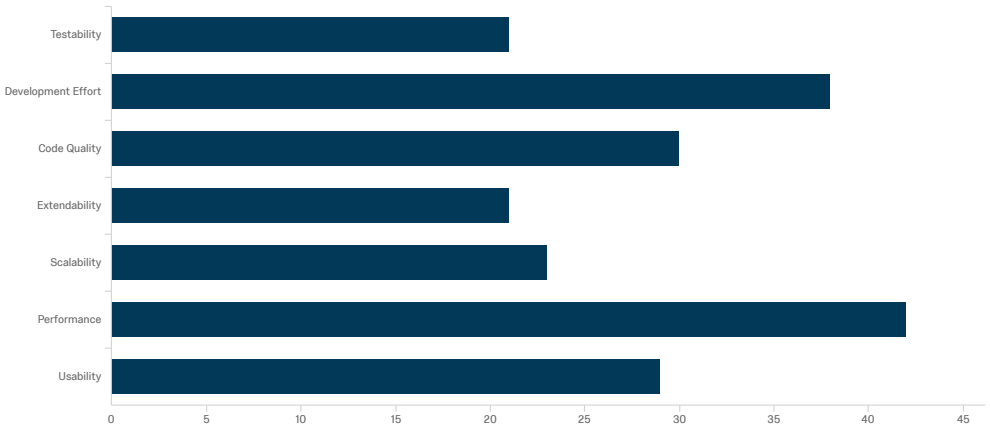



| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 45.16% | 28 |
| 2 | Server-side rendering | 25.81% | 16 |
| 3 | It doesn't matter | 29.03% | 18 |
| | | | 62 |

Showing Rows: 1 - 4 Of 4



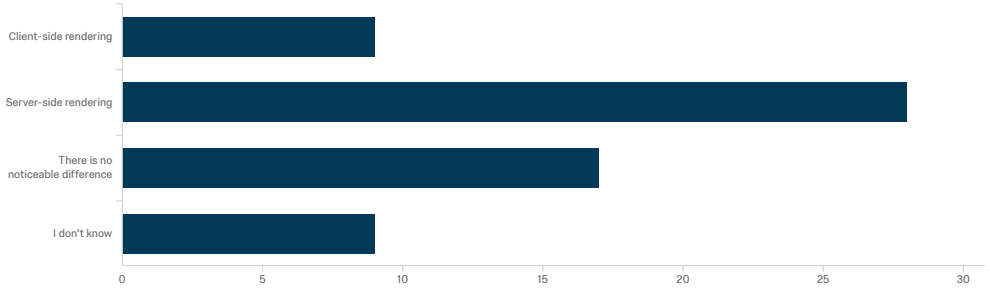
1-10 - Which of the following metrics influence your choice between server-side rendering and client-side rendering the most?




| # | Field | Choice Count | |
|---|---|---|---|
| 1 | Testability | 10.29% | 21 |
| 2 | Development Effort | 18.63% | 38 |
| 3 | Code Quality | 14.71% | 30 |
| 4 | Extendability | 10.29% | 21 |
| 5 | Scalability | 11.27% | 23 |
| 6 | Performance | 20.59% | 42 |
| 7 | Usability | 14.22% | 29 |
| | | | 204 |

Showing Rows: 1 - 8 Of 8

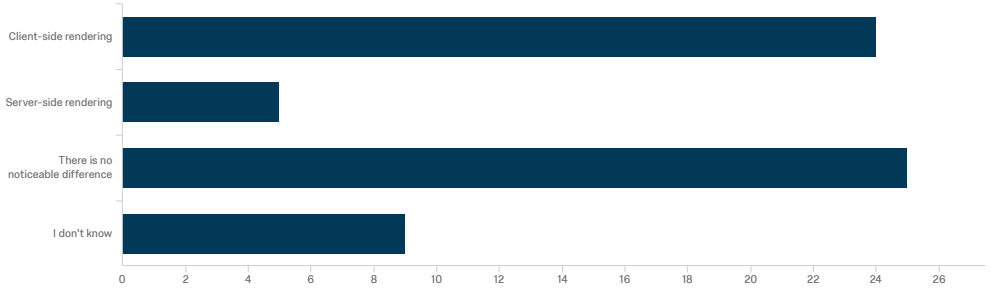
1-11 - Which paradigm is most testable?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 14.29% | 9 |
| 2 | Server-side rendering | 44.44% | 28 |
| 3 | There is no noticeable difference | 26.98% | 17 |
| 4 | I don't know | 14.29% | 9 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

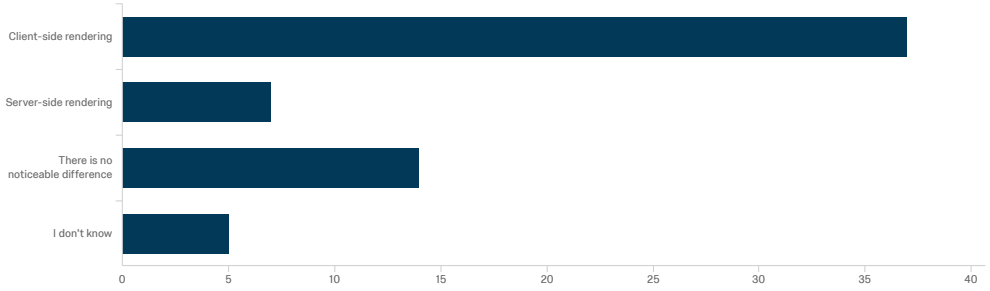
1-12 - Which paradigm is more modifiable?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 38.10% | 24 |
| 2 | Server-side rendering | 7.94% | 5 |
| 3 | There is no noticeable difference | 39.68% | 25 |
| 4 | I don't know | 14.29% | 9 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

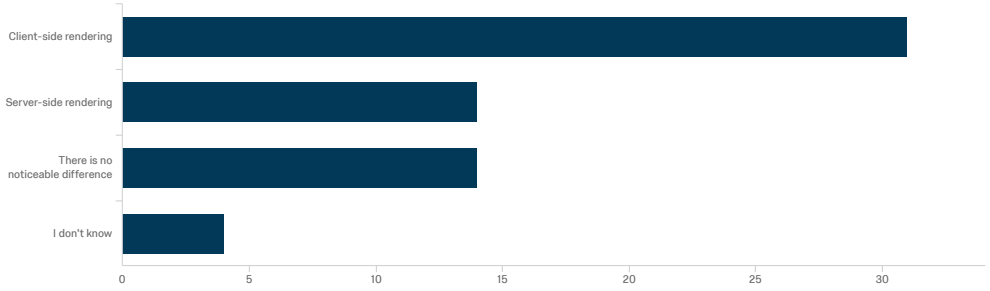
1-13 - Which paradigm is more flexible?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 58.73% | 37 |
| 2 | Server-side rendering | 11.11% | 7 |
| 3 | There is no noticeable difference | 22.22% | 14 |
| 4 | I don't know | 7.94% | 5 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

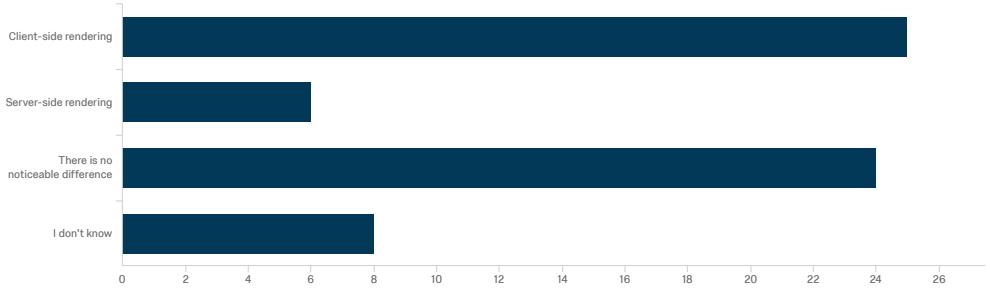
1-14 - Which paradigm requires most development effort?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 49.21% | 31 |
| 2 | Server-side rendering | 22.22% | 14 |
| 3 | There is no noticeable difference | 22.22% | 14 |
| 4 | I don't know | 6.35% | 4 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

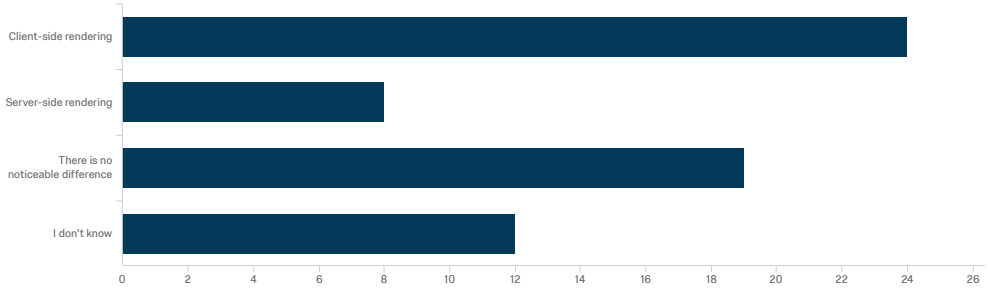
1-15 - Which paradigm yields most duplicate code?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 39.68% | 25 |
| 2 | Server-side rendering | 9.52% | 6 |
| 3 | There is no noticeable difference | 38.10% | 24 |
| 4 | I don't know | 12.70% | 8 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

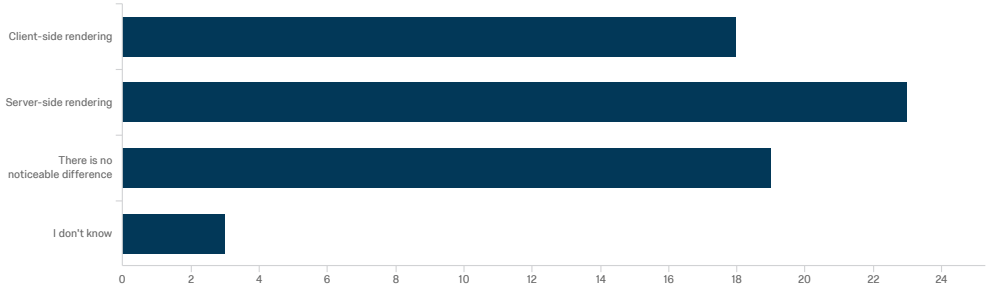
1-16 - Which one of the paradigms yields more code defects/bugs?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 38.10% | 24 |
| 2 | Server-side rendering | 12.70% | 8 |
| 3 | There is no noticeable difference | 30.16% | 19 |
| 4 | I don't know | 19.05% | 12 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

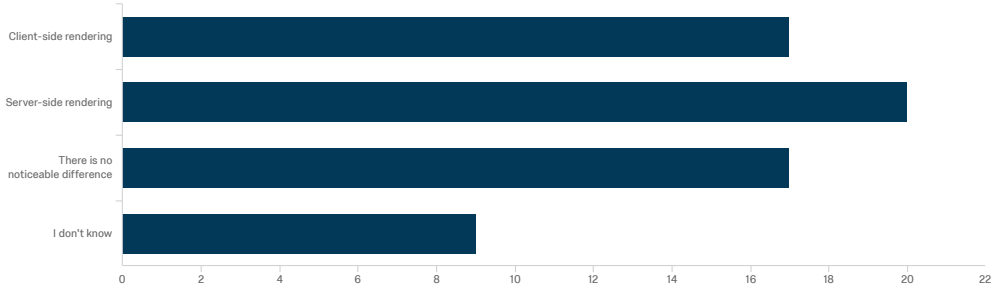
1-17 - Which paradigm yields more maintainable code?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 28.57% | 18 |
| 2 | Server-side rendering | 36.51% | 23 |
| 3 | There is no noticeable difference | 30.16% | 19 |
| 4 | I don't know | 4.76% | 3 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

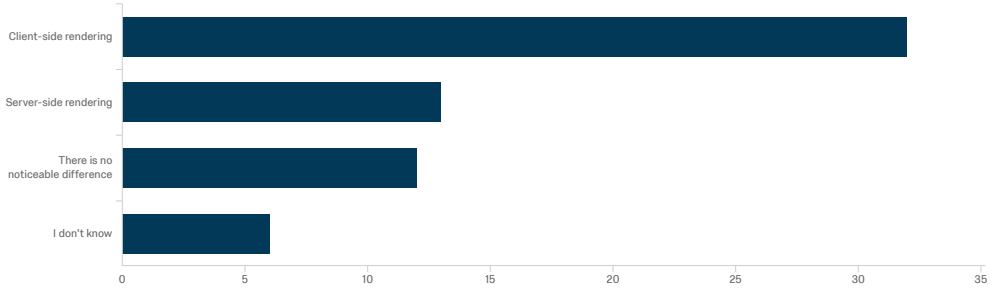
1-18 - Which paradigm yields more efficient code?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 26.98% | 17 |
| 2 | Server-side rendering | 31.75% | 20 |
| 3 | There is no noticeable difference | 26.98% | 17 |
| 4 | I don't know | 14.29% | 9 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

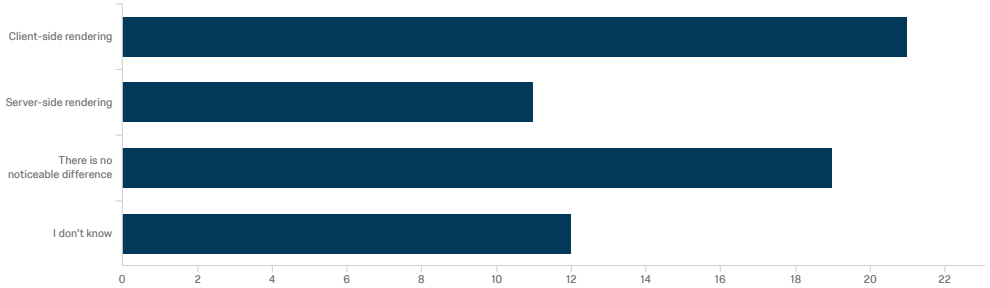
1-19 - Which paradigm yields better usability?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 50.79% | 32 |
| 2 | Server-side rendering | 20.63% | 13 |
| 3 | There is no noticeable difference | 19.05% | 12 |
| 4 | I don't know | 9.52% | 6 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

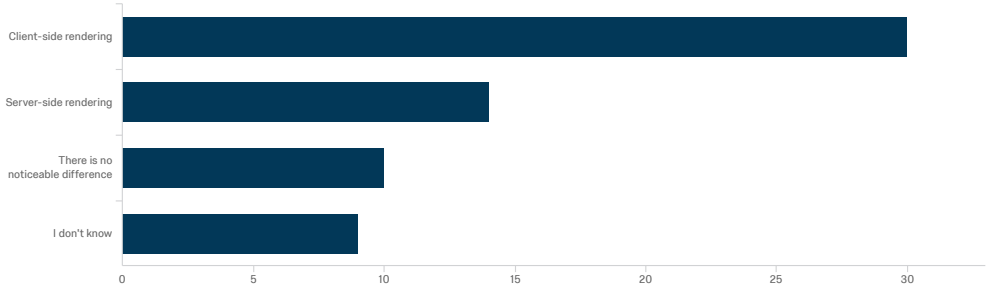
1-20 - Which paradigm results in best extendability?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 33.33% | 21 |
| 2 | Server-side rendering | 17.46% | 11 |
| 3 | There is no noticeable difference | 30.16% | 19 |
| 4 | I don't know | 19.05% | 12 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

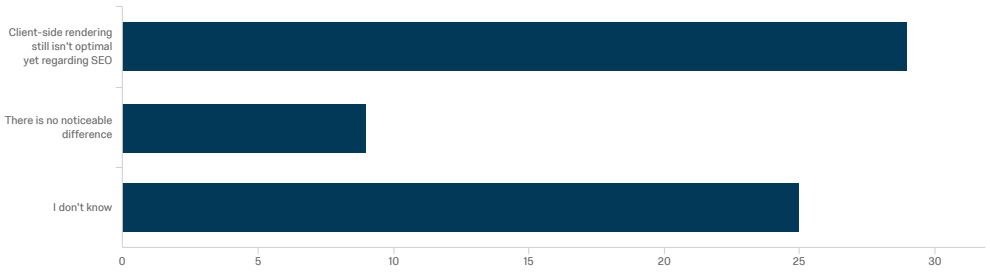
1-21 - Which paradigm scales best?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering | 47.62% | 30 |
| 2 | Server-side rendering | 22.22% | 14 |
| 3 | There is no noticeable difference | 15.87% | 10 |
| 4 | I don't know | 14.29% | 9 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

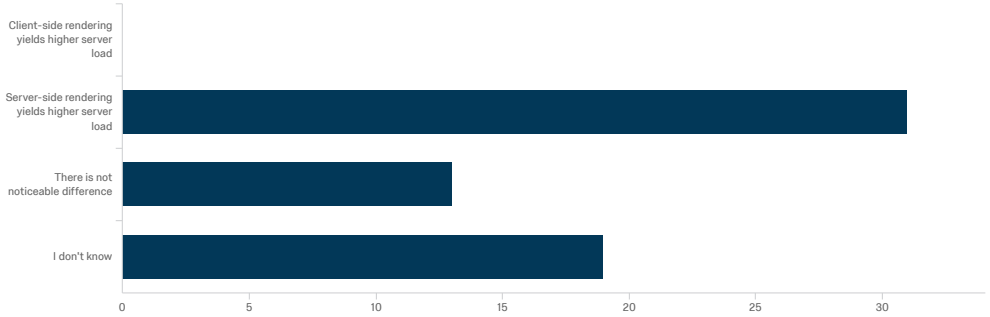
1-22 - Do you think the impact of client-side rendering on SEO is still noticeable anno 2018?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering still isn't optimal yet regarding SEO | 46.03% | 29 |
| 2 | There is no noticeable difference | 14.29% | 9 |
| 3 | I don't know | 39.68% | 25 |
| | | | 63 |

Showing Rows: 1 - 4 Of 4

1-23 - Did you notice impact on server load using one of the paradigms?




| # | Field | | Choice Count |
|---|---|---|---|
| 1 | Client-side rendering yields higher server load | 0.00% | 0 |
| 2 | Server-side rendering yields higher server load | 49.21% | 31 |
| 3 | There is not noticeable difference | 20.63% | 13 |
| 4 | I don't know | 30.16% | 19 |
| | | | 63 |

Showing Rows: 1 - 5 Of 5

**End of Report**

## Case Study Waterfalls

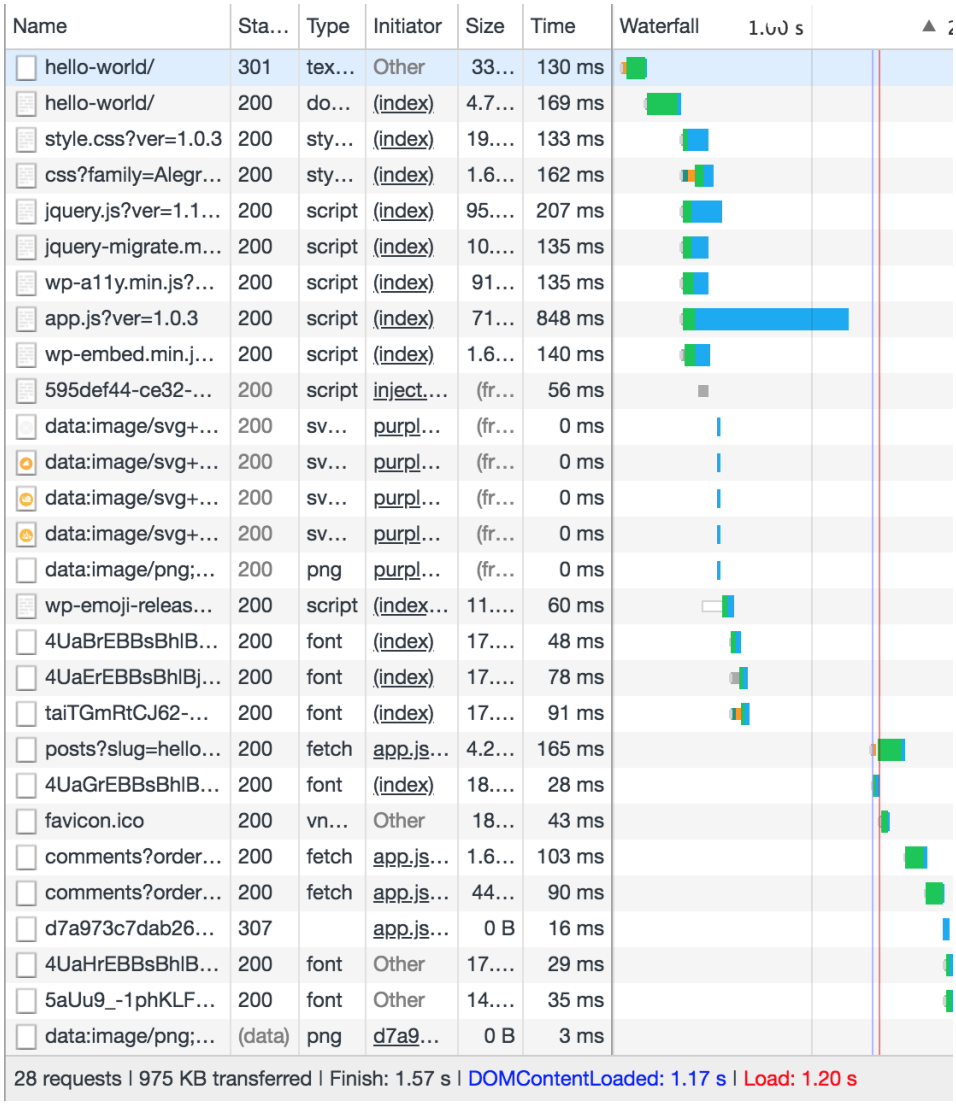



**Figure 8.20:** Network waterfall of client-side rendering for initial page load for Wordpress theme.

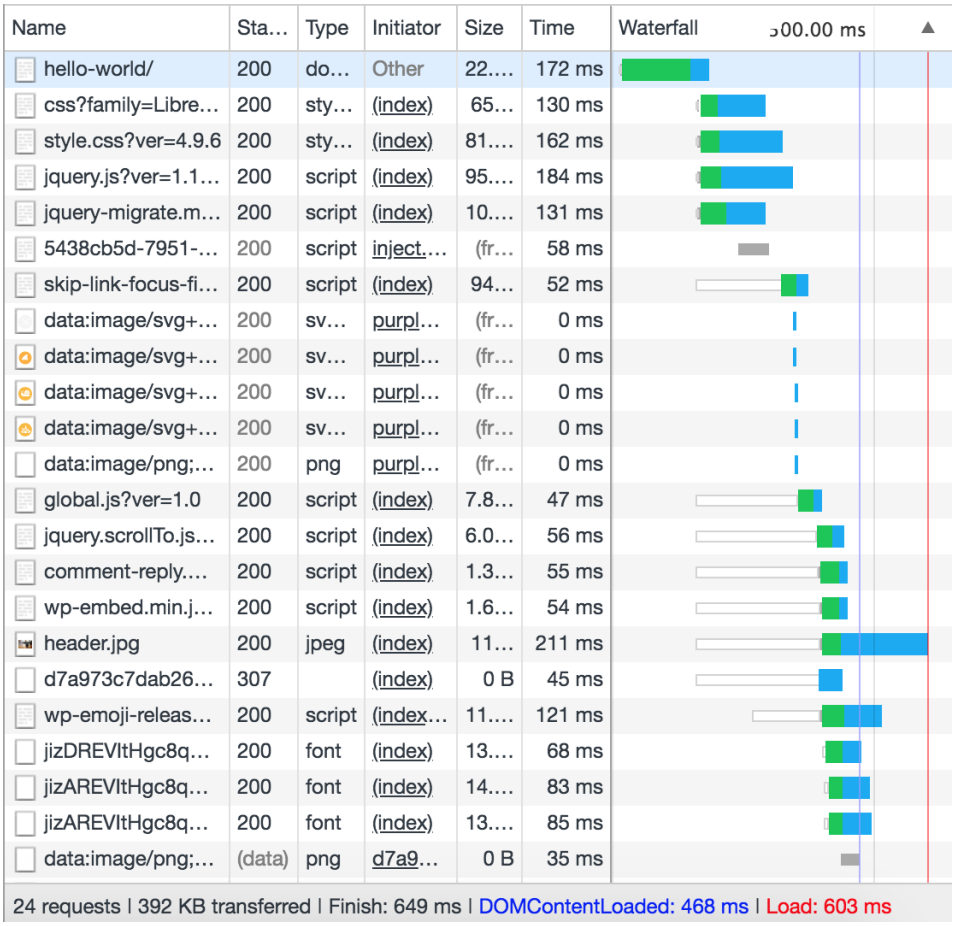



**Figure 8.21:** Network waterfall of server-side rendering for initial page load for Wordpress theme.

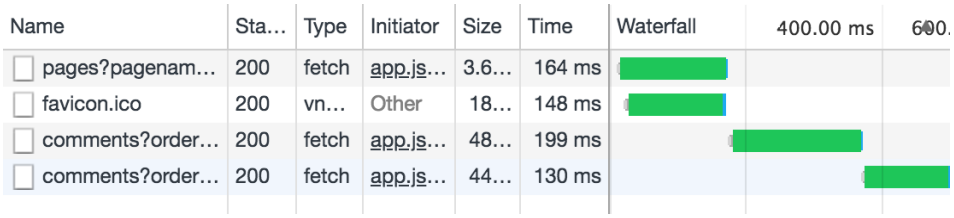



**Figure 8.22:** Network waterfall of client-side rendering for next page load for Wordpress theme.

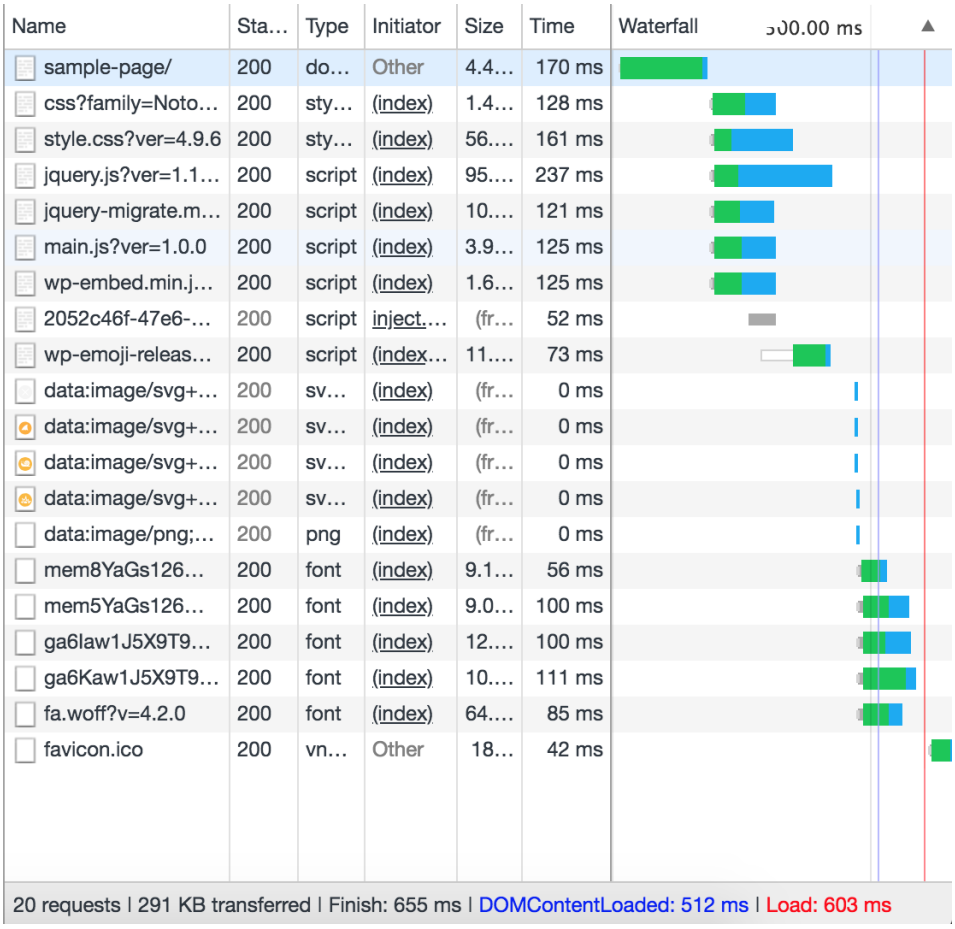



**Figure 8.23:** Network waterfall of server-side rendering for next page load for Wordpress theme.