ÅBO AKADEMI

Optimizing the Critical Rendering Path for Decreased Website Loading Time

---

Gabriel Kivilohkare

Master of Science Thesis

Supervisors: Annamari Soini, Jan Westerholm
Department of Information Technologies
Faculty of Science and Engineering
Åbo Akademi University

May 5, 2020

Abstract

Users expect websites to load faster while websites are becoming larger and more complex. Simultaneously, users prefer mobile devices that use networks with high latencies. This thesis aims to evaluate different optimization strategies for decreasing the website loading time. General optimization strategies and optimization strategies for the above-the-fold content are presented and tested in this thesis. Finally, tests are run with all the optimization strategies for a compound effect. The conclusion is that even large websites can load the above-the-fold content in a short time for good user experience.

Keywords: Critical rendering path, Website loading time, JavaScript, Cascading style sheets, Critical CSS, Critical JavaScript

List of Figures

Contents

Chapter 1

Introduction

Low loading time is crucial for websites in several ways. Low loading time has a major role in providing the end-user with a good user experience. Moreover, website loading time has an immense impact on how the user behaves on a website. Studies show that 53% of mobile users will abandon a website that loads for more than three seconds [3]. Therefore website loading time is critical for *conversions*, which is the activity that a business wants the user to do, e.g. a purchase in e-commerce. Finally, the long loading time will have a negative influence on a website's search engine ranking.

Most businesses have an online presence and a large number of businesses rely on the Internet. The loading time of a business's website can have a vast economic impact on the business. Slow-loading websites make users abandon websites, which makes businesses directly lose potential customers.

The number of mobile users is continuously increasing, which poses challenges for the loading speed of websites due to mobile networks with high latency and mobile devices with less processing and memory capabilities than traditional computers. Another challenge for website loading time is that websites tend to become more complex with more functionalities. Simultaneously, users expect websites to load faster.

Recent research shows that there are several ways to decrease the loading time of websites, including back-end and front-end optimizations. This thesis focuses on

optimizing the *critical rendering path* for a decreased loading time of the above-the-fold content. The critical rendering path (CRP) can be defined as the steps the browser needs to take to render a web site [4]. Above-the-fold content is the content that is initially visible in the browser when the website first loads. Content that appears after scrolling is called below-the-fold content. By optimizing the critical rendering path for the above-the-fold content, meaningful content can be shown rapidly to the user, while the website is not fully loaded. This gives the user a sense of a high performing website.

1.1 Purpose

Traditionally website loading time has been considered the time from when the user opens a website until the website is fully loaded. The goal of this thesis is to find solutions for optimizing the critical rendering to decrease the loading time of the above-the-fold content. A website is not necessarily completely loaded once it is rendered, and certain resources can be loaded in the background while prioritizing only the critical resources needed for the rendering of the website. This thesis attempts to outline the following for each optimization strategy:

- How much does implementing this strategy impact website loading time?
- How much work is required to implement this strategy?
- Does this strategy cause any side effects?

1.2 Thesis Structure

The second chapter of this thesis is an introduction to what the browser does when a website is loaded, how website loading time is measured, and which factors have a main impact on the website loading time. The third chapter is an overview of the different strategies for decreasing website loading time. These strategies are divided into two categories: general optimization strategies and above-the-fold content optimization strategies. In the fourth chapter, the optimization strategies

are implemented, tested, and compared. The fifth chapter discusses the results of the experiments and answers the research questions. In the last chapter, a conclusion is reached and future work is discussed.

Chapter 2

Background

This chapter introduces how websites are rendered and what key metrics can be used for measuring website loading performance. The information in this chapter is essential for understanding the research and discussions in the following chapters.

2.1 Loading a Website

When a user types in a domain in the address bar of a browser and navigates to a website, the browser needs to find the Domain Name System (DNS) record of the domain. The DNS is a system for giving resources that are connected to the internet a human friendly name, as in the case of websites, translating domains to IP addresses [5]. The browser examines different caches in order to find the DNS record. First, it examines its own cache and if the DNS record is not found, it examines the operating system cache followed by the router cache and the ISP (Internet Service Provider) cache. If the information is not found in any of the caches, a recursive DNS query is initiated from the ISP's DNS servers. Caching significantly improves the performance of translating the domain to an IP address [6].

Once the browser has found the IP address of the domain, either from cache or through DNS query, the browser sends an HTTP, HTTPS, or HTTP2 request to

the website host server. The host responds and sends back the requested HTML as a response, as shown in the picture below.

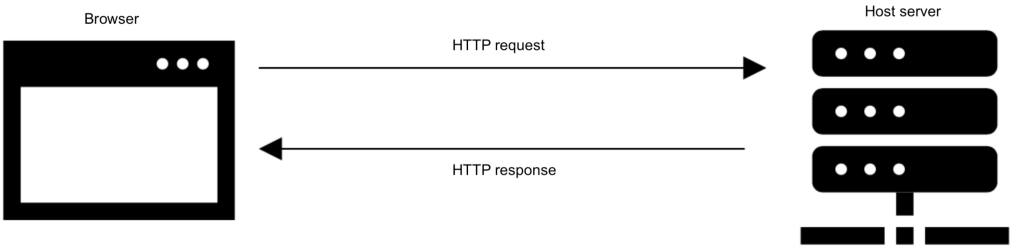

FIGURE 2.1: HTTP request and response.

The browser processes the HTML markup and builds the Domain Object Model (DOM) tree. A Cascading Style Sheets Object Model (CSSOM) is built from the styles associated with the DOM. The DOM and the CSSOM are both individual data structures, where the DOM represents the content of the processed HTML and the CSSOM represents only the styling. The DOM and CSSOM are combined into a render tree. Hereafter, a layout is generated and finally, the web page is painted. We will cover all these steps in detail in the following sub-chapters, as they are key elements in the loading process of a website. The critical rendering path is the steps explained above, in other words, the steps the browser has to take in order to show a website on a user's screen.

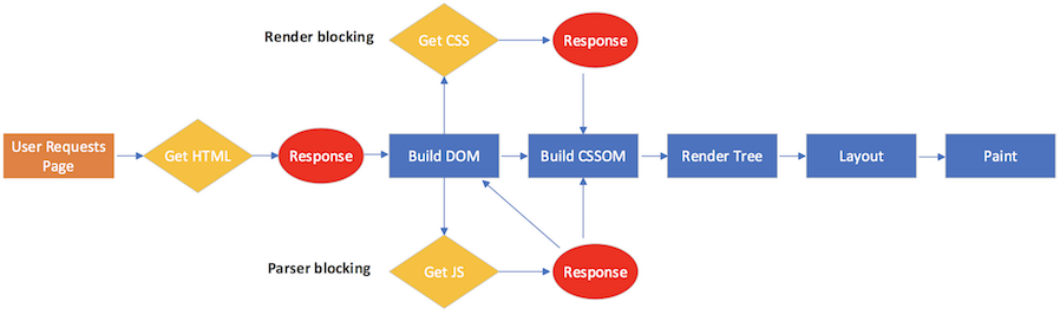

FIGURE 2.2: The construction of the Domain Object Model.

2.1.1 Construction of the DOM

For the browser to construct the DOM, the browser has to convert all the raw HTML bytes to characters, followed by converting the character strings to tokens, followed by converting tokens to objects, and finally building the DOM tree [7]. Figure 2.2 displays the relationships between each object. Each object has a certain set of rules and a specific purpose.

Consider this very simple piece of HTML code, which represents a website with a small text and a picture:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    <p>Hello <span>world</span>!</p>
    <div><img src="sample-photo.jpg"></div>
  </body>
</html>
```

The DOM shows how different objects are linked in a tree-like structure. Every tag in the HTML is represented as an object in the DOM. A graphical representation of the DOM in the above example would look like this:

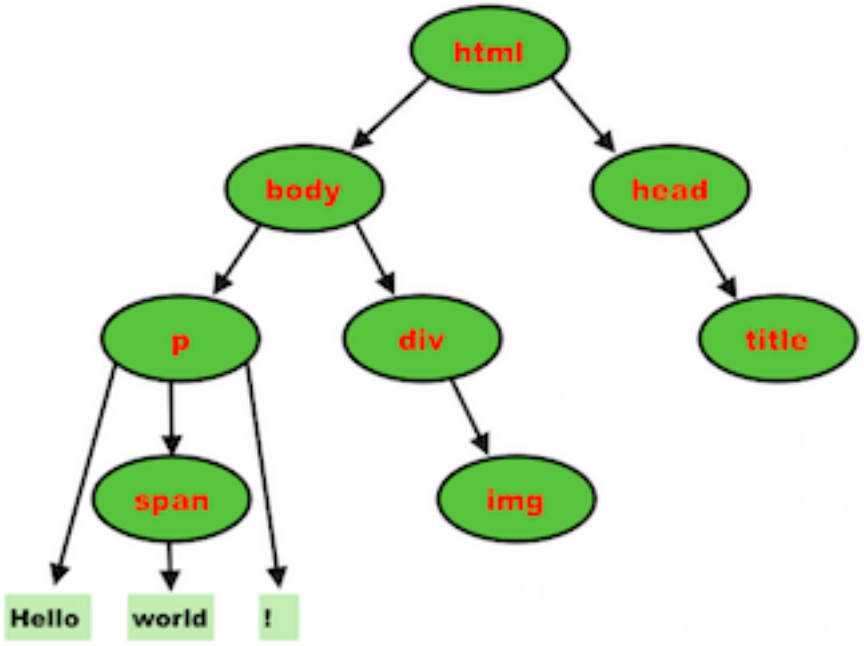

FIGURE 2.3: A graphical representation of the Domain Object Model.

The construction of the DOM can be time-consuming if there is a large amount of HTML.

2.1.2 Construction of the CSSOM

During the DOM construction, the browser might encounter a reference to an external Cascading Style Sheet (CSS).

```
<link href="example.css" rel="stylesheet">
```

Immediately after the browser encounters the CSS reference, it requests the resource and receives the content of the CSS file as a response. Similarly to the DOM construction, the CSSOM is constructed by first converting the CSS bytes to characters, hereafter characters are converted to tokens and lastly, tokens are converted to nodes. Ultimately the nodes are linked to a tree-like structure containing styling attributes, which is called CSSOM. The styles in the CSSOM automatically override browser-specific default styles, so-called user-agent-styles.

As an example, the following CSS would give the word ”world” the color red in our previous example.

```
p span {
    color: red;
}
```

Browsers read CSS from right to left. In our example, the browser would first check for ”span” -objects, then check if those have ”p” -objects as parents. If the criteria match, the styling will be applied to the object.

2.1.3 Render Tree, Layout, and Paint

As seen in Figure 2.2, the render tree is constructed by combining the DOM and the CSSOM [8]. The final render tree contains only visible nodes and their corresponding CSSOM rules. All of the visible content that is shown on the web page is included in the render tree. Nodes that are hidden by a CSS display property are not included in the render tree. Likewise, nodes in the DOM that have no content are omitted from the render tree.

During the layout stage, the browser calculates the exact size and position for each node in the render tree. This is a recursive process, beginning from the root node and traversing the render tree. The browser has to run layout each time the render tree is updated or the size of the viewport changes.

The final stage of website rendering is the paint stage. In this stage, the browser paints the result from the layout stage on the screen.

2.2 Above-the-fold Content

The above-the-fold is the visible content that is seen on the web browser when the website loads. Fast rendering of this content gives the user an impression of an instant load, even though most of the resources might not have loaded.

Considering the importance of loading time to the user experience, optimizing the loading of the above-the-fold content could significantly improve the user experience on heavy websites. This content can vary on different devices such as computers and mobile devices.

Optimizing the above-the-fold content is a solution for two scenarios, loading large web pages and loading any web page with a poor connection. The average website is about 2 MB in size [2], and large web pages may be tens of megabytes in size. In mobile networks, poor connections are often related to high latency. By finding a solution for the two scenarios above, it is possible to determine the optimal size of the critical rendering path.

2.3 Stages of Loaded Sites

Traditionally, website loading time has been considered as a single metric, that is how long it takes for the website to load completely. Even though this is a real and relevant metric, it does not give an accurate overview of the loading time corresponding to the user experience. The user experience is different considering the loading time of a fully-loaded site versus a partly loaded site. Different stages serve different purposes, as explained in the following sections.

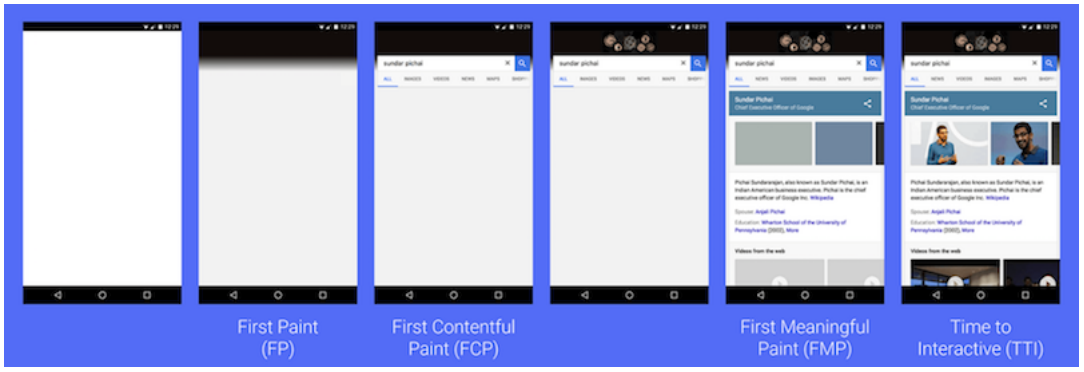

FIGURE 2.4: Different website loading stages demonstrated [1].

2.3.1 Time to First Paint

The time to first paint is the time it takes for the browser to show the very first pixels on the user's screen after navigation. The first paint indicates that the server has responded with an HTTP status code 200 and that the web page is reachable and loading. The content could be e.g. a background color, and the web page is not yet useful nor does it provide interesting content to the user at this stage.

2.3.2 Time to First Contentful Paint

The time to first contentful paint is the time it takes for the browser to paint the first elements from the domain object model. This is the first consumable content that can bring value to the user. The content can be e.g. text, a canvas element, or an image.

2.3.3 Time to First Meaningful Paint

The time to first meaningful paint is the time it takes for the browser to show meaningful content that is useful for the user. This is a very critical metric as this content is the most important part of the page. For example, in an e-commerce product page, the first meaningful paint would include a picture and description of the product, which is what the user came to look for. The definition of meaningful content varies from site to site and there is no general specification that would apply to all cases. At this stage, the browser cannot necessarily respond to user interactions.

2.3.4 Time to Interactive

The time to interactive is the time it takes for the browser to render the web page and become ready for user interaction. Asynchronous JavaScript may not have loaded at this stage, but the JavaScript main thread is idle. All synchronous

JavaScript has to be loaded, and often JavaScript is needed for user interaction in modern websites. Therefore in some cases, not all JavaScript can be run asynchronously.

2.4 Website Loading Time

Fiona Fui-Hoon Nah finds in her empirical study that a tolerable waiting time for retrieving information is approximately two seconds [9]. She noted that after two seconds, shifts in focus or interference with short-term memory occurred. This is in line with research from the limits of short-term memory after two seconds of waiting [10]. Also, as previously stated, most mobile users abandon a website after three seconds of waiting. Therefore, a loading time of a maximum of two seconds is a reasonable goal for any website.

The loading time of a website is affected by several factors. The main factors include the website itself, the webserver that hosts the website, the network, and the connection. These factors are presented in detail in the following subchapters.

2.4.1 Network Latency

Network latency has an impact on the loading time. The network latency is the time it takes for the browser to receive a response to an HTTP request. In order to show a website on a browser, at least one HTTP request is made by the browser to receive the HTML from the host server. Each request is subject to latency and, on average, a website that is loaded on mobile client causes 71 HTTP requests [11].

Network latency is largely impacted by the physical distance between a user's device and the responding webservers. A content delivery network (CDN) reduces network latency by storing content in multiple locations and serving users from the closest locations. Additionally, the host server performance, including its hardware, affects network latency.

2.4.2 Connection

The internet connection of the device that loads a website plays a crucial role in the loading time of a website. The global average mobile download speed is 30.46 Mbps and the global average fixed broadband download speed is 74.64 Mbps [12], as demonstrated in the table below.

TABLE 2.1: Global averages per connection type [12]

Connection type	Download speed	Latency
Mobile	30.46 Mbps	42 ms
Fixed broadband	74.64 Mbps	24 ms

However, a problem that will likely persist is that the downloading speed and latency will vary [13]. This is caused by, among other factors, network congestion and traffic shaping.

As the average size of a website is 2 megabytes, which translates to 16 megabits, the download speed is not a bottleneck in loading the average website. For larger websites, however, optimizations need to be in place in order to reach a short loading time on mobile devices. Mobile connections have higher latencies than fixed broadband connections that cause further delays in the loading of a website.

2.4.3 The Data Size of the Website

The median data size of websites has increased steadily in the past years. The larger a website is, the more bytes are downloaded, causing an increased loading time. The average size of a website is 2 MB and the largest part is the images. While the amount of JavaScript is constantly increasing, in an average website it takes up 21.7% of the total size.

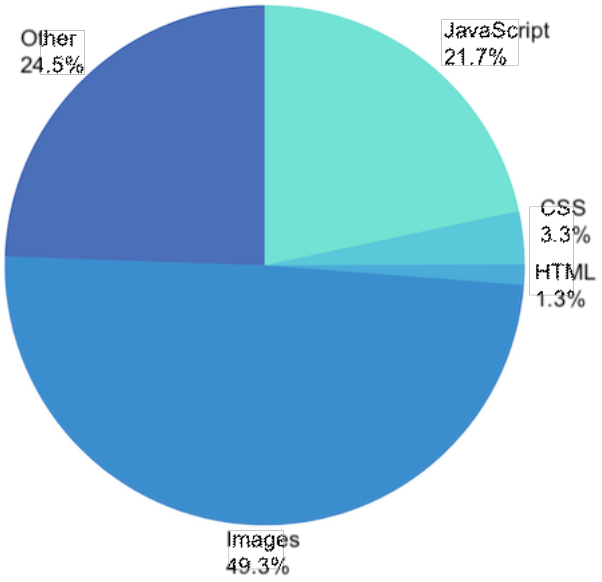

FIGURE 2.5: Average size per content type [2].

In the picture above the distribution of the bytes per content type is visualized. Other content types are mostly represented by videos and fonts.

Many optimization strategies for decreasing website loading time impact the size of the website. As images account for 49.3% of the size of an average website, optimizing the delivery of images has the largest potential in decreasing the loading time of a website.

2.4.4 Website Loading Order

The loading order of resources in a website has an impact on the loading time. Resources that are fetched before the first pixels are painted on the screen increase the loading time. Upon displaying a website, the webserver gives the HTML as a response to the browser. The HTML typically contains links to scripts, style sheets and images.

Optimization strategies that relate to the loading order of a website are presented and tested in this thesis. These strategies separate the content of the resources

into critical and non-critical content and download the respective contents in different ways. The order of the website's resources and way of downloading them is explained further below.

2.4.4.1 Render and Parser Blocking Resources

As seen in Figure 2.2, CSS is render-blocking and JavaScript is parser-blocking. When the browser encounters a style sheet, it requests the resource from the server and the parser continues. Therefore, CSS is render-blocking but as the parser continues, it is not parser-blocking. Once all CSS resources have been loaded the browser can paint the result. The fetching of each resource causes a round trip and therefore increases the loading time. CSS rules that are not part of the above-the-fold content can be loaded at a later stage.

When the browser encounters a script tag, the parser has to wait until the resource is fetched and executed before it can continue, making the JavaScript parser-blocking. In addition to the round trip that the fetching of the resource causes, the execution increases the loading time. The loading of JavaScript resources is more challenging than loading CSS resources from a performance point of view.

2.4.4.2 Synchronous and Asynchronous Resources

Synchronous loading of resources for a web page means loading and executing resources in the order they appear in the HTML. This might slow down the loading of the web page, as the browser does not proceed with rendering the web page before the resource is loaded and executed.

Asynchronous loading does not stop the rendering while loading the resource. With asynchronous loading, multiple resources can be downloaded simultaneously. Loading resources asynchronously typically decreases the time to first paint, but the resources are not available before they have been loaded. Therefore, it is important to identify what resources can be loaded asynchronously. Thus, in order to find out which resources shall be loaded asynchronously for decreased loading time and which resources shall be loaded synchronously for good

user experience, these ways of loading will be examined in further detail in this thesis.

2.4.5 Webserver

A webserver is, simply put, a software that can accept requests and returns replies [14]. A webserver can contain one or several websites depending on its setup. A webserver can be located on different types of hardware and can serve different purposes. For example, a webserver can be located on a server computer in a data center, on a laptop, or on a printer. Typically, websites are hosted on webservers that are run from data centers.

A webserver can store and deliver websites and it uses HTTP communication in order to show a website in a browser. It hosts typically static HTML, CSS, and JavaScript files along with images. These resources are usually handled on the client side in the browser. Depending on how the browser handles these resources, they have an impact on the loading time of the website.

Most web servers support server-side scripting that enables code to be executed on the webserver instead of the browser. Typically, advanced business logic and retrieval of dynamic data are done on the server side. As an example, information on whether a user is logged in to a web portal is ordinarily fetched through a server-side script. Server-side scripts are processed by the processor and memory of the webserver, while client-side scripts consume the resources av the user's device. Some server-side scripts, such as poorly written WordPress plugins, can cause severe increases on the loading time of a website.

The most widely used webserver is the Apache HTTP server that serves 38.9% of all websites [15]. Other popular webservers include Nginx and Cloudflare. The Apache HTTP server is a free open-source webserver software that was launched in 1995, while the second most popular webserver, Nginx, was launched in 2004. Nginx and Apache HTTP server have different architectures and serve different needs. When choosing a webserver it is important to understand the requirements of the website. Nginx is suitable for high-traffic websites due to its event driven

architecture, while Apache HTTP server gives the user more flexibility in terms of modules.

The settings of a webserver can have an impact on the loading time. For example, in the Apache HTTP server, it is possible to enable compression and caching directly from the webserver settings. One of the most common optimization strategies for decreased loading time of a website is gzip that is a method for compressing resources. Gzip can be enabled directly in the Apache webserver configurations and it is discussed in further detail in chapter 3.1.3.

2.4.6 Client

The client is a software that sends a request to the web server, commonly a web browser. The browser uses the resources, such as the processor and memory, of the computer, mobile phone, or other device it is used on. Client-side scripts are run on the user's device [16] and can, therefore, increase the loading time of a website.

The loading time of a website can be impacted by the web browser that is used for displaying the website. There are major differences in how older and newer browsers handle scripts, style sheets, and images. Older browser, such as Firefox 3.0 and Internet Explorer 6, are not capable of downloading resources in parallel [17]. Therefore, it is not possible to download resources asynchronously with these browsers. Most modern browsers, on the other hand, are capable of downloading resources in parallel. Performance related issues are thus often caused by old browsers. As modern browsers are not a bottleneck in loading websites, they are not further examined in this thesis. Modern browsers that can download resources in parallel, include Google Chrome 74, Safari 10, Firefox 70, Internet Explorer 11, and newer versions of these browsers.

Different browsers support different CSS properties. Consequently, websites might appear different when using different browsers. For example, the CSS property *text-orientation*, which sets the orientation of text in a line, is not supported by Internet Explorer 11 while is is supported by Google Chrome 74 and Firefox 70. Hence, cross-browser testing is an important part of designing a website.

Chapter 3

Optimization Methods

The optimization methods discussed in this thesis are divided into two sections. In the first section, general methods are introduced. The second section focuses on methods for optimizing the critical rendering path. However, the methods introduced in the first section will have an impact on the critical rendering path as well.

3.1 General Website Performance

There are several ways to decrease the loading time of a website and improve its performance. The following principles are general and apply to any website. These principles also decrease the loading time of the above-the-fold content.

3.1.1 Reduction of Requests

As Michael Butkiewicz found in his study, the number of requests has more impact on the loading time than the number of bytes transferred [18]. The browser sends an HTTP request each time the parser encounters a request for a new element. Each request requires a round trip to a server, causing increased loading time.

The number of requests can be minimized by combining resources. Style sheets and scripts can be combined into one CSS file and one script file respectively.

However, this does not guarantee reduced loading time. Depending on the website structure, asynchronous loading of specific resources will decrease loading time while increasing HTTP requests.

Another technique for decreasing the number of HTTP requests is using CSS sprites [19]. One CSS sprite is a collection of multiple images in one file. The images within the sprite are separated by using coordinates in the desired CSS properties.

3.1.2 Minify Resources

By minifying resources, we make them smaller in size and therefore the page request becomes smaller. In practice, minifying is the removal of unnecessary characters in a file. For example, when minifying a CSS file, all line breaks, comments and unnecessary blank characters are removed. An example of an unminified CSS snippet is presented below.

```
.example {
    padding: 5px 10px 10px 5px;
}

.second {
    border-radius: 5px;
}
```

The same CSS snippet is significantly shorter when minimized, as shown in the code snippet below.

```
.example{padding:5px 10px 10px 5px;}.second{border-radius:5px;}
```

The readability of the files suffers when minifying, and the minified versions are not intended to be used in development. Similarly to minifying CSS, JavaScript minification reduces script size without modifying any essential processes [20].

3.1.3 Compressing Resources

When a user navigates to a website, a request is made to a server and the browser receives resources. The smaller these resources are, the faster the website will load. The resources can be made significantly smaller by compressing them. For plain text files, such as script, HTML, and CSS -files, a method called gzip compression is effective. Gzip replaces repetitive strings in files with pointers that use less space, which reduces file size [21]. This procedure is fairly similar to minification, however, the two methods complement each other. Hereafter, gzip compresses the file to a zip file, which the browser unzips. The compression takes place directly on the server and it can be activated with a few lines of code on most common web servers.

3.1.4 Delivering Images

Optimally, images shall be no larger than they appear on the end user's screen. Scaling down large images results in downloading unnecessary bytes, which in turn results in longer loading time. For optimal performance, images shall be optimized for different resolutions. A background image for a mobile device might be several times smaller than one for a large screen. High-quality images are larger in size than lower quality images. Removing meta-data and compressing images will decrease the size of the images.

3.1.5 Caching

The browser can store resources in its cache memory, providing a significant decrease in loading time. This will not impact the first page visit, but future visits and visits to other pages that share the same resources will have a decreased loading time. Different types of resources can be cached, commonly pictures, CSS, and HTML. The two common caching methods are called *expires header* and cache-control. The *expires header* is an HTTP date after which the specified resource is stale. The cache-control uses a maximum age for the specified resource,

which can be set in seconds. Cache-control gives more control over a website's caching compared to *expires header*.

3.1.6 CSS Delivery

CSS resources can be delivered in two ways: as external resources or in-lined in the HTML. The external resources can be loaded either asynchronously or synchronously. As discussed in 2.1.4, CSS is render blocking as when the browser encounters an external style sheet it requests the resource while the parser continues.

While the asynchronous loading of these resources decreases the loading time, asynchronicity must be applied with caution. Consider loading all style sheets asynchronously, where no style sheets would be loaded on the initial page render. Therefore, in this scenario, the initial page render would be unstyled. Once the asynchronous parallel task of loading the style sheets is finished, the style sheet would be applied to the website. This event is called flash of unstyled content (FOUC), which causes the illusion of longer loading time and a poorly designed website.

When loading CSS, using @import calls, which imports style sheets into other style sheets, will increase loading time [19]. The preferred method is using link tags, as using @import will add one more roundtrip to the page load. Moreover, inlining CSS to HTML elements is considered a bad practice, as it will cause repetition, a larger file size, and it will be hard to maintain.

3.1.7 JavaScript Delivery

Similarly to CSS, JavaScript can be delivered as external resources or in-lined in the HTML. The external JavaScript can be loaded either asynchronously or synchronously. Synchronous loading of JavaScript, which is the default way of loading resources, is parser blocking. When loading these resources asynchronously, however, they will not block DOM construction and will continue downloading in parallel.

Similarly to asynchronous loading of style sheets, asynchronous loading of scripts might cause issues. In modern web pages, the amount of JavaScript has been rapidly growing during the last years. Optimizing JavaScript delivery has become increasingly important. While loading all JavaScript asynchronously would make the web page load faster, the functionality that the JavaScript provides would not be usable before it has loaded completely.

Asynchronous loading of all resources is not the optimal solution for user experience, but neither is waiting for all resources to load. Therefore, loading only the critical resources during page render and asynchronously loading the non-critical resources would optimize both loading time and user experience.

JavaScript resources can be loaded synchronously, asynchronously with the async tag, and asynchronously with the defer tag. Synchronous loading in the head section of the HTML is the least effective way of loading, as the browser pauses the parsing of the HTML while fetching and executing the script, as illustrated in the picture below. Synchronous loading of scripts should only be applied to critical resources. JavaScript can be loaded later in the HTML as well, often right before the closing body tag, allowing the browser to render the web page before the resource is loaded.

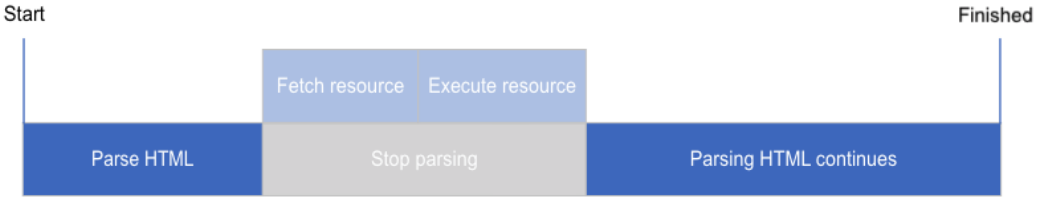

FIGURE 3.1: Synchronous loading of JavaScript.

Asynchronous loading of scripts using the async tag allows the browser to continue parsing the HTML while fetching the resource. However, when the resource is fetched, the parsing is paused while the script is executed.

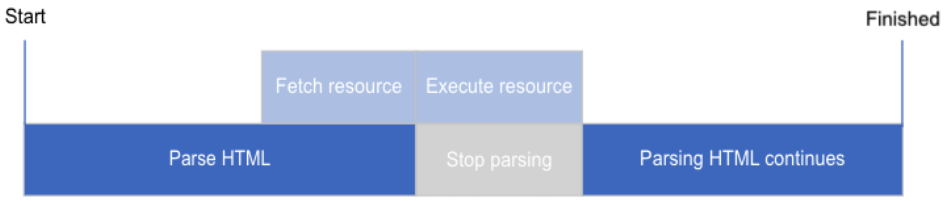

FIGURE 3.2: Asynchronous loading of JavaScript.

Using the defer tag, the browser continues parsing the HTML while loading the resource and executes the script once the browser has finished parsing. This method is optimal for performance. However, the same performance can be achieved by placing the script with an async tag at the end of the HTML.

FIGURE 3.3: Loading JavaScript with the defer attribute.

3.2 Above-the-fold Content Performance

By prioritizing the loading of the above-the-fold content, any website can show the user critical content in minimal time. This is an important factor from a business perspective, as it is a best practice to show the unique selling point in the above-the-fold content. The principles discussed in section 3.1 apply also to the above-the-fold loading time. However, to optimize the loading time of the above-the-fold content, the following methods make a significant difference:

1. Minimize the number of critical resources
2. Minimize the length of the critical path
3. Minimize the number of critical bytes

3.2.1 Minimize the Number of Critical Resources

Critical resources can be defined as those scripts and style sheets that are required for interactivity and styling of the above-the-fold content. For fast delivery of the above-the-fold content, the number of critical resources has to be minimized. The fewer resources the browser needs to fetch, the less work is needed to show the content.

3.2.2 Minimize the Critical Path Length

The critical path length is the number of round trips between the browser and the host server. The number of round trips between servers for the above-the-fold content should optimally be zero when all the critical scripts and style sheets are in-lined in the head section of the HTML. However, it is not feasible to inline all critical resources in every case. In these situations, it is recommended that critical resources are downloaded as early as possible.

3.2.3 Minimize the Number of Critical Bytes

Critical bytes is the number of bytes in the critical rendering path. For a decreased loading time, each critical resource must be compressed and minified, as discussed in the previous chapter. The number of critical bytes can be optimized by including only the relevant style sheets and scripts for the above-the-fold content.

3.2.3.1 Critical CSS

The critical CSS is formed from the classes and id's in the HTML elements of the above-the-fold content. Parsing the style definitions based on these classes and id's from all style sheets ultimately creates the critical CSS. The critical CSS can be removed from the style sheets as it is in-lined in the HTML. Once the critical CSS is parsed from the original style sheets, the style sheets can be loaded

asynchronously. The above-the-fold content is different on different devices, which must be taken into account when parsing the critical CSS from the style sheets.

3.2.3.2 Critical JavaScript

The critical JavaScript is the JavaScript that the browser requires for initializing a web page. This JavaScript handles basic user interactions and adjusts the layout. Typically the critical portions of JavaScript in a website includes functionality relating to navigation or a sign-up form. For more complex user interactions and functionality, the rest of the JavaScript is loaded asynchronously or deferred. Similarly to the critical CSS, the critical portions of the JavaScript need to be identified based on what portions are most important for the user experience.

Chapter 4

Experiments

4.1 Introduction

The experiments in this chapter will use an unoptimized test website for testing the optimization strategies introduced in chapter 3. First, goals and the setup of the experiments are presented followed by implementation of the optimization strategies. Finally, results from the experiments are presented.

4.1.1 Optimization Strategies Tested

The following general optimization strategies will be tested in the experiments: minimizing CSS, minimizing JavaScript, reducing the number of requests, compressing resources and delivering images. Caching will not be tested as it does not affect the loading time of the first page visit. All general optimization strategies will be applied to the test website.

For decreasing above-the-fold content loading time, two strategies will be tested. First, optimization of critical CSS delivery will be tested followed by testing of critical JavaScript optimizations. Finally, all optimization strategies presented above will be tested together for evaluating the compound effect of the strategies.

4.2 Goal

The goal of the experiments is to test the above presented optimization strategies separately and to evaluate how much these impact the website loading time. The second goal of the experiments is to evaluate the amount of work needed to implement each optimization strategy. This will be evaluated based on the effort required to implement each optimization strategy on the test website. Based on this it can be evaluated whether an optimization strategy is worth implementing.

The third goal is to discover how much impact all the optimization strategies presented in this thesis have combined. As the test website is larger than the average website, combining the optimization strategies will give valuable insights into whether large websites can be loaded in less than two seconds, which is, as discussed in chapter 2.4, the maximum tolerable waiting time.

4.3 Experiment Design

For accuracy, each optimization is tested five times. The results are recorded and the average loading time is calculated based on these results. Minimum and maximum loading times are recorded for reference.

Each general optimization method is tested separately on the unoptimized test website. Hereafter, all general optimizations are applied together and tested. As the general optimization methods support the above-the-fold content performance, the above-the-fold optimization methods are applied on top of the general optimization methods.

4.3.1 Experiment Environment

The test environment is designed to have minimal interference from factors that might affect the loading time of the test website. Therefore, the test website is hosted locally on an Apache webserver on a Macbook Pro, and the client device

is an emulated mobile device. The connection is limited with simulated network throttling, as explained below.

4.3.1.1 Webserver

The experiment webserver is located on a 15-inch Macbook Pro 2018, which has a 2.5-GHz Intel Core i7 processor and 16 GB 2400 MHz DDR4 RAM-memory. The webserver that hosts the test website is an Apache/2.4.39 (Unix) HTTP server. The webserver has a 64-bit architecture. As each DNS lookup increases the loading time, in order to minimize the number of DNS lookups, all external resources are loaded from the same webserver.

4.3.1.2 Client

The client device is an emulated Nexus 5X mobile device which simulates the typical hardware that a user might have. The emulated device has an Android 6.0.1 operating system and uses Chrome Lighthouse as browser. As discussed in chapter 2.4.6, modern browsers are capable of handling multiple resources in parallel, which is the case for Chrome Lighthouse.

4.3.1.3 Connection

To limit the downloading speed, network throttling is simulated to provide realistic loading times. Google's Lighthouse v3 is used for testing the loading times. To increase the accuracy of the experiments and to provide realistic loading times, the downloading bandwidth was limited by simulating network throttling. The network was limited to a 1,638.4 Kbps throughput which equals a poor 4G connection. The main benefit of simulating the network throttling is low variance between test runs.

A simulated fixed latency of 150 ms is used, which means that each HTTP request has the same fixed latency. As presented in chapter 2.4.2, the network latency and connection might vary. Therefore, in order to minimize variance, the experiment downloading speed is constant and the latency is fixed.

4.3.2 Experiment Website

For these experiments, an test website was created. This test website has no optimizations in place and it was designed to have a high loading time. The website is significantly larger than an average website.

4.3.2.1 Design

The test website is a simple website with 714 words of text and five images. It starts with a banner section with a large image, text, and a button. This is followed by several rows and columns of text, images, buttons, and links. At the bottom of the page there is a footer section with links.

The design of the website is responsive and it scales to every device. The above-the-fold content is different for different devices, as demonstrated in the picture below.

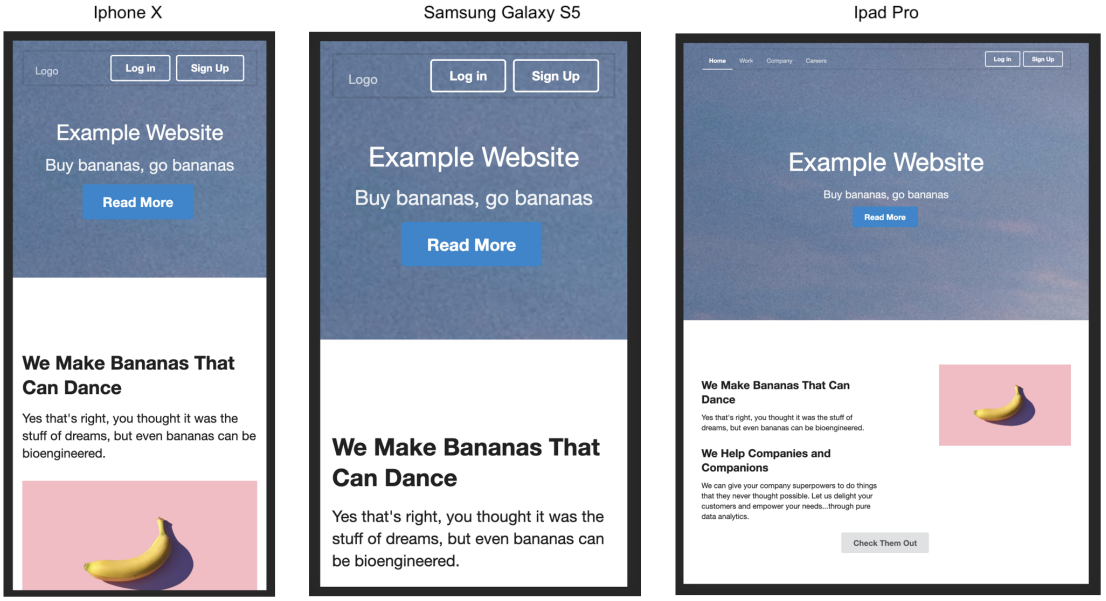

FIGURE 4.1: Test website on different devices.

The unoptimized test website consists of the following resources:

- One HTML file (size: 14 KB)
- Seven JavaScript files (total size: 8 MB)
- Six CSS files (total size: 7,8 MB)
- Five images (total size: 5,6 MB)

4.3.2.2 Loading of Resources

On the test website, all external resources are loaded synchronously in the head section. All of the style sheets are loaded and all of the JavaScript files are fetched and executed before the browser parses the body of the HTML. The banner image is loaded in-lined in the HTML body, making it unavailable in the first contentful paint, as illustrated in the picture above. As all the external resources are loaded before the first contentful paint, the time from first contentful paint to interactive is short.

4.3.2.3 HTML

The HTML file is unminified and uncompressed and it contains 347 lines of code. The head section contains all the scripts and style sheets and the body section contains all the visible content of the website. All external resources are located on the same server and within the same folder as the HTML file. This makes the retrieval of these files consistent with minimal dependencies.

4.3.2.4 JavaScript

The JavaScript files are loaded synchronously as external files in the head section of the HTML. Each JavaScript file creates one HTTP request. These files are loaded without any HTML attributes. The test website does not contain any JavaScript in-lined in the HTML. The only user interaction that requires JavaScript is the mobile navigation bar that opens up when clicking the logo, as demonstrated in the picture below.

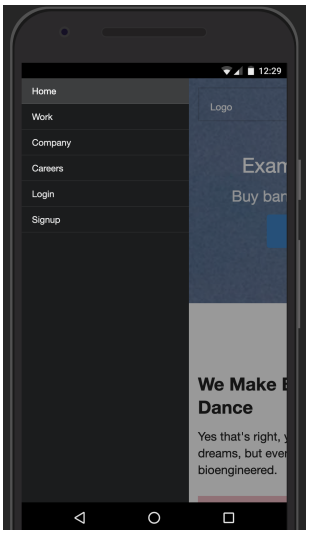

FIGURE 4.2: Mobile navigation bar.

The rest of the JavaScript is not used on the website and is created for demonstrating the effects of optimizing JavaScript delivery. Unused JavaScript causes a longer loading time if fetched and executed without optimizations. Therefore, unused JavaScript is a real problem in modern websites.

4.3.2.5 CSS

All style sheets are loaded as external resources in the head area of the HTML file after the JavaScript files, as demonstrated in the code snippet below.

```
<head>
 <title>Homepage</title>

 <!-- JavaScript -->
 <script src="jquery.min.js"></script>
 <script src="javascript.js"></script>
 <script src="javascript2.js"></script>
 <script src="javascript3.js"></script>
 <script src="javascript4.js"></script>
 <script src="javascript5.js"></script>
```

```
  <script src="script.js"></script>

  <!-- CSS -->
  <link rel="stylesheet" type="text/css" href="stylesheet.css">
  <link rel="stylesheet" type="text/css" href="stylesheet2.css">
  <link rel="stylesheet" type="text/css" href="stylesheet3.css">
  <link rel="stylesheet" type="text/css" href="stylesheet4.css">
  <link rel="stylesheet" type="text/css" href="stylesheet5.css">
  <link rel="stylesheet" type="text/css" href="styles.css">

</head>
```

The HTML of the test website contains one inlined CSS element, which is the banner image. The rest of the CSS is loaded as external files.

4.3.2.6 Images

The banner image is loaded as part of the above-the-fold content. It is 6000 pixels in width and 400 pixels in height and is loaded as such on every device without scaling to a device-specific size. Additionally, the test website contains four identical pictures that are 363 KB each. These pictures include meta-data and are downsized in the CSS. This means that the pictures are downloaded in their original sizes and made smaller in the code, causing unnecessary bytes to be downloaded. Depending on the device, one of these pictures is part of the above-the-fold content.

4.3.3 Loading Stages

The test website has a notably high loading time with 80.1 seconds until the first pixel is painted on the screen, as illustrated in the picture below.

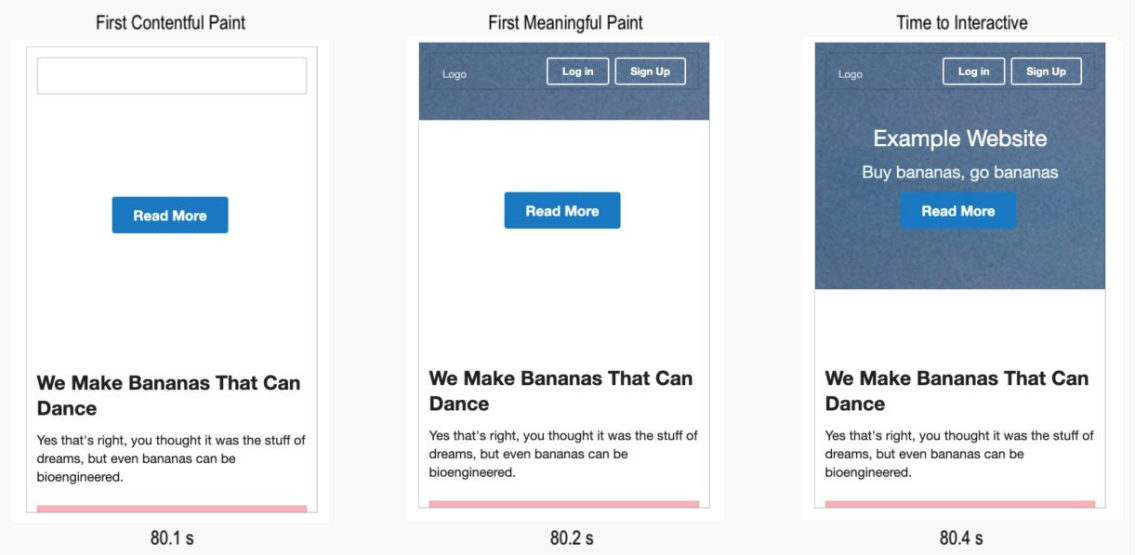

FIGURE 4.3: Loading stages of the unoptimized sample website.

The high loading time makes the website unusable in practice and most users would likely abondon the site before it would have loaded. The time from the first contentful paint to interactive is short due to the fact that most resources are loaded synchronously before the first paint.

4.3.4 Limitations

The test website contains only static client-side content. It does not contain any server-side code such as dynamic PHP content. No strategies for optimizing code are implemented. Therefore, the execution time of the JavaScript code is not examined. All the tests are run on a mobile device simulating a relatively poor connection with a 1,638.4 Kbps throughput.

4.4 Implementation

4.4.1 General Methods

The general optimization methods are implemented separately to the unoptimized test website. The practical implementation for each general optimization strategy is presented below. Finally, all general optimization strategies are applied together on the unoptimized test website.

4.4.1.1 Minimizing CSS

All CSS resources were minified using an online tool called CSS Minifier [22]. The tool removes all unnecessary characters, such as comments and unnecessary white spaces, from the CSS files. Each CSS file was minified separately. The total size of the CSS files was reduced from 7.8 MB to 6.4 MB.

4.4.1.2 Minimizing JavaScript

All JavaScript resources were minified using a open-source tool called Minify [23]. The tool minified the JavaScript files by removing all unecessary characters and shortening patterns. Each JavaScript file was minified separately. The total size of the JavaScript files was reduced from 8 MB to 3.1 MB resulting in a significant decrease in bytes to be downloaded.

4.4.1.3 Reducing the Number of Requests

All external CSS files were merged into one CSS file. Similarly, all external JavaScript files were merged into one JavaScript file. When merging the resources, the order of the content was kept unchanged. The number of requests was reduced from 13 requests to two requests. Each request causes one round-trip, which increases the loading time. One CSS file and one JavaScript file is optimal in this case.

4.4.1.4 Delivering Images

As discussed in chapter 2.4.3, the median website consists to 49.3% of images. Therefore, there is potential to decrease the amount of bytes the browser needs to download. First, in order to apply this strategy, the images were scaled down to their optimal size. The optimal size is based on what device and resolution the page is loaded on. In this experiment, we optimize the images for a mobile device, more specifically an emulated Nexus 5X. Hereafter, the image is compressed using TinyPNG. TinyPNG reduces the file size by decreasing the number of colors in the images and removing meta-data. The difference between the original image and the compressed image is barely visible.

The sample website has five images. The main background image is loaded as in-lined CSS in the HTML, while the other smaller images are loaded directly in the HTML. All images cause one HTTP request.

4.4.1.5 Compressing Resources

In these experiments, gzip is used to compress resources. The following resources are compressed: CSS files, HTML files, and JavaScript files. Gzip allows output from the local server to be compressed before being sent to the client over the network. The implementation of gzip on an Apache webserver is fast and easy. It requires enabling the *mod deflate* module in the Apache configuration. Also, compressible file types need to be added to the Apache configuration file. Once the changes are made, the Apache webserver requires a restart.

4.4.1.6 All General Optimizations Combined

After applying all the above-mentioned general optimization methods to the test website, the size of the website has decreased from 21.4 MB to 9.0 MB. The number of requests decreased from 19 to 8. The optimizations work well together and there were no issues in combining them. The CSS and JavaScript resources were minified before they were merged.

4.4.2 Critical Rendering Path Specific Optimizations

To optimize the loading time of the above-the-fold content, the number of critical resources shall be minimized, the length of the critical path shall be minimized, and the number of critical bytes shall be minimized. To do this, we need to deliver CSS and JavaScript resources optimally.

4.4.2.1 Critical CSS

As presented in chapter 2.4.4.1, loading CSS blocks the rendering of the website. Therefore it is important to identify what parts of the CSS are needed to style the above-the-fold content and loading the rest of the CSS after the page render.

The critical CSS is extracted from all the CSS files and the HTML file using Critical Path CSS Generator [24]. The external CSS was merged into one file and minified in the general optimization strategies. The critical CSS is extracted and inserted to the head section of the HTML. The rest of the CSS is placed in an external CSS file which is loaded at the end of the HTML file, right before the closing body tag. This allows the browser to render the page before loading the rest of the CSS. Before the optimization, the stylesheet was loaded in the head, as shown in the code snippet below.

```
<html>
  <head>
    <title>Title</title>
    <script src="script.js"></script>
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
    <p>Hello <span>world</span>!</p>
  </body>
</html>
```

The optimized delivery of the CSS is shown in the code snippet below. The critical portions of the CSS can be removed from the external CSS file.

```
<html>
  <head>
    <title>Title</title>
    <script src="script.js"></script>
    <style>/*CRITICAL CSS CONTENT*/</style>
  </head>
  <body>
    <p>Hello <span>world</span>!</p>
    <link rel="stylesheet" type="text/css" href="styles.css">
  </body>
</html>
```

4.4.2.2 Critical JavaScript

Finding the critical JavaScript is more challenging than finding the critical CSS. For optimal performance, the critical parts of the JavaScript shall be identified and in-lined in the head section of the HTML. Usually, the critical JavaScript includes modules for interactivity in the above-the-fold content, such as functionality for navigation.

In this case, the critical JavaScript is identified manually and inserted in the head section of the HTML. The rest of the JavaScript is loaded using the *defer* attribute, which, as explained in chapter 3.1.7, means that the external JavaScript is fetched asynchronously and executed once HTML parsing is done. The code snippet below demonstrates the HTML after in-lining the critical JavaScript and deferring the non-critical JavaScript.

```
<html>
  <head>
    <title>Title</title>
    <script>/*CRITICAL JAVASCRIPT CONTENT*/</script>
    <link rel="stylesheet" type="text/css" href="styles.css">
    <script src="/script.js" defer></script>
  </head>
```

```
  <body>
    <p>Hello <span>world</span>!</p>
  </body>
</html>
```

4.5 Experiments

4.5.1 Unoptimized Website

The unoptimized test website creates 19 requests and has a total size of 21.4 MB. None of the optimizations presented in this thesis is applied to the website. The table below displays the loading times of the unoptimized website.

Table 4.1: Loading times for the unoptimized test website

Loading Times - Unoptimized Website			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	80.1 s	80.0 s	80.2 s
First Meaningful Paint	80.2 s	80.2 s	80.1 s
Time to Interactive	80.4 s	80.4 s	80.4 s

The time to contentful paint is 80.1 seconds which is very high. This is mostly because all external resources are fetched and executed before the browser starts the rendering of the visible content.

4.5.2 General Optimization Strategies

4.5.2.1 Minimizing CSS

After minimizing the CSS, the website size of the website decreased from 21.4 MB to 20.1 MB. The amount of requests remains the same as for the unoptimized website. The table below displays the loading times after minimizing the CSS.

Table 4.2: Loading times after minifying the CSS of the test website

Loading Times - CSS Minimized			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	73.3 s	73.3 s	73.3 s
First Meaningful Paint	73.4 s	73.4 s	73.4 s
Time to Interactive	73.7 s	73.6 s	73.8 s

As the browser has fewer bytes to download, the loading time of the website is decreased. The differences between loading stages are insignificant as the external JavaScript and CSS resources are render blocking and these are loaded and executed before the first pixel is painted on the screen.

4.5.2.2 Minimizing JavaScript

After minimizing the JavaScript, the size of the website is decreased by 4.9 MB. The size of the external JavaScript was reduced by 61%. The table below displays the loading times after minimizing the JavaScript.

Table 4.3: Loading times after minifying the JavaScript of the test website

Loading Times - JavaScript Minimized			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	54.3 s	54.3 s	54.3 s
First Meaningful Paint	54.4 s	54.4 s	54.4 s
Time to Interactive	54.7 s	54.5 s	54.8 s

Unsurprisingly, the loading time of the website decreased significantly due to fewer bytes to be downloaded.

4.5.2.3 Reducing the Number of Requests

After combining the external files into one CSS file and one JavaScript file, the number of requests was decreased from 19 to 8. The size of the website remained

the same. The table below displays the loading times after reducing the number of requests.

Table 4.4: Loading times after reducing the number of requests on the test website

Loading Times - Reduced Number of Requests			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	78.1 s	78.1 s	78.2 s
First Meaningful Paint	78.1 s	78.1 s	78.2 s
Time to Interactive	78.5 s	78.4 s	78.6 s

Reducing the number of requests resulted in minor decreases in loading time for all the loading stages.

4.5.2.4 Delivering Images

After optimizing the images, the size of the website decreased from 21.4 MB to 15.3 MB. The total size of the images decreased from 5.6 MB to 199 KB. The table below displays the loading times after optimizing the images.

Table 4.5: Loading times after optimizing images on the test website

Loading Times - Image Optimization			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	80.0 s	80.0 s	80.1 s
First Meaningful Paint	80.0 s	80.0 s	80.1 s
Time to Interactive	80.2 s	80.1 s	80.3 s

The decrease in the loading time is very insignificant despite the decrease in the size of the website. This is due to the fact that the images are downloaded in the body of the HTML. The browser still needs to fetch and execute the external resources in the head section of the HTML before it can render the body.

4.5.2.5 Compressing Resources

After compressing HTML, JavaScript, and CSS the number of requests is unchanged. The browser fetches the compressed resources and unzips them. The size of the compressed website is 7.6 MB while the size of the unzipped website is 21.4 MB. The table below displays the loading times after compressing resources.

Table 4.6: Loading times after compressing resources on the test website

Loading Times - Compressed Resources			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	14.6 s	14.5 s	14.7 s
First Meaningful Paint	14.7 s	14.6 s	14.8 s
Time to Interactive	14.7 s	14.7 s	14.7 s

The decreases in loading times for all the loading stages are very significant. The differences between the loading stages are small as the browser performs most actions before the first pixel is painted on the screen.

4.5.2.6 General Strategies Combined

After applying all the general optimization methods, the website creates eight requests and has a size of 9.0 MB. The table below displays the loading times after applying all of the optimization methods.

Table 4.7: Loading times after applying all the general optimization methods on the test website

Loading Times - All General Optimizations			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	8.7 s	8.6 s	8.7 s
First Meaningful Paint	8.7 s	8.6 s	8.7 s
Time to Interactive	9.4 s	9.4 s	9.4 s

Applying all the general optimization methods results in a remarkable decrease in loading time for all the loading stages. Despite the decrease in loading time,

8.7 seconds time to first paint is not a good result, and most users would abandon the website before it would be loaded.

4.5.3 Critical Rendering Path Optimization Strategies

The above-the-fold optimizations are applied to a version of the test website where all the general optimizations are in place. The general optimization strategies decrease the loading time of the above-the-fold content and partly overlap with the critical rendering path optimization strategies.

4.5.3.1 Critical CSS

After inlining the critical CSS to the head section and moving the external CSS to the closing body tag the size of the website grew from 9.0 MB to 9.1 MB since the critical CSS was moved to the head section, while the external CSS file was unmodified. This translates to 0.1 MB of critical CSS. The table below displays the loading times after optimizing critical CSS delivery.

Table 4.8: Loading times after applying critical CSS optimizations on the test website

Loading Times - Critical CSS			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	4.8 s	4.8 s	4.8 s
First Meaningful Paint	6.8 s	6.8 s	6.8 s
Time to Interactive	9.4 s	9.3 s	9.5 s

The time to first contentful paint is significantly shorter after applying this method. The rendering time of the head section of the HTML is shorter because there are no external CSS resources. The JavaScript resources block the parsing, resulting in a relatively high loading time.

4.5.3.2 Critical JavaScript

After in-lining the critical JavaScript to the head section and loading the external JavaScript with the *defer* attribute, the size of the website remains at 9.0 MB and the number of requests at eight. The table below displays the loading times after optimizing critical JavaScript delivery.

Table 4.9: Loading times after applying critical JavaScript optimizations on the test website

Loading Times - Critical JavaScript			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	5.0 s	5.0 s	5.0 s
First Meaningful Paint	5.0 s	5.0 s	5.0 s
Time to Interactive	9.6 s	9.5 s	9.6 s

Applying the critical JavaScript optimization strategy decreases the website loading time. The time to interactive is slightly longer after applying this strategy, as the browser has more JavaScript to fetch and execute. The website is interactive after 5.0 seconds as all critical JavaScript interactions are loaded before this. The time to interactive measurement indicates when the asynchronous JavaScript is fetched and executed, which is not needed for critical interactions on the test website.

4.5.3.3 All Optimization Methods Combined

After applying all the optimization methods discussed in this chapter, the website is 9.1 MB in size and creates eight requests. The table below displays the loading times for the optimized website.

Table 4.10: Loading times after applying all optimizations on the test website

Loading Times - All Optimization Methods Combined			
Loading Stage	Average	Minimum	Maximum
First Contentful Paint	0.6 s	0.6 s	0.6 s
First Meaningful Paint	0.6 s	0.6 s	0.6 s
Time to Interactive	9.7 s	9.6 s	9.8 s

The time to first contentful paint and the time to first meaningful paint are both drastically shorter than on the unoptimized website. The time to interactive is comparably high due to the fact that the amount of asynchronous JavaScript is high.

4.6 Results

4.6.1 General Optimization Methods

By applying all the general optimization strategies introduced in this thesis, the test website's loading time decreased significantly. The first contentful paint was reduced from 80.1 seconds to 8.7 seconds. The table below compares the loading times for all the general optimization methods tested and discussed above.

Table 4.11: Comparisons of loading times for the general optimization methods

General Optimization Comparison			
Strategy	First Contentful Paint	First Meaningful Paint	Time to Interactive
Before Optimizations	80.1 s	80.2 s	80.4 s
Minimizing CSS	73.3 s	73.4 s	73.7 s
Minimizing JS	54.3 s	54.4 s	54.7 s
Delivering Images	80.0 s	80.0 s	80.2 s
Compressing Resources	14.6 s	14.7 s	14.7 s
Reducing Requests	78.1 s	78.1 s	78.5 s
Combined	8.7 s	8.7 s	9.4 s

Optimizing image delivery caused an insignificant reduction in loading time, while all the other strategies caused a significant decrease in loading time. Compressing resources was the most effective general optimization strategy for the test website. Minimizing JavaScript decreased the loading time more than minimizing CSS because it reduced the size of the website by 4.9 MB while minimizing CSS reduced the size only by 1.3 MB.

Reducing requests caused a two-second decrease in time to first paint. This is fairly low because the resources are on the same server. Fetching resources from other external servers would have likely increased the loading time.

4.6.2 Above-the-fold Optimization Methods

When optimizing the above-the-fold content performance, we see a large difference in loading time between the different loading stages, as demonstrated in the table below.

Table 4.12: Comparisons of loading times for the above-the-fold optimization methods

Above-the-fold Optimizations			
Strategy	First Contentful Paint	First Meaningful Paint	Time to Interactive
Before Optimizations	8.7 s	8.7 s	9.4 s
Critical JS	5.0 s	5.0 s	9.6 s
Critical CSS	4.8 s	6.8 s	9.4 s
Combined	0.6 s	0.6 s	9.7 s

Optimizing critical CSS delivery decreased the loading time slightly more than optimizing JavaScript delivery. The amount of critical JavaScript is significantly lower than the amount of critical CSS. Combined, however, the compound effect is impressive.

4.6.3 Optimized Website

By applying all the optimization methods described above to the sample website, the first contentful paint loading time of the sample website is decreased from 80.2 seconds to 0.6 seconds. However, the time to interactive measurement had a worse result after applying the above-the-fold content optimization strategies. This is due to the increased size of the website. The picture below illustrates the different loading stages after applying all the optimization strategies presented above on the sample website.

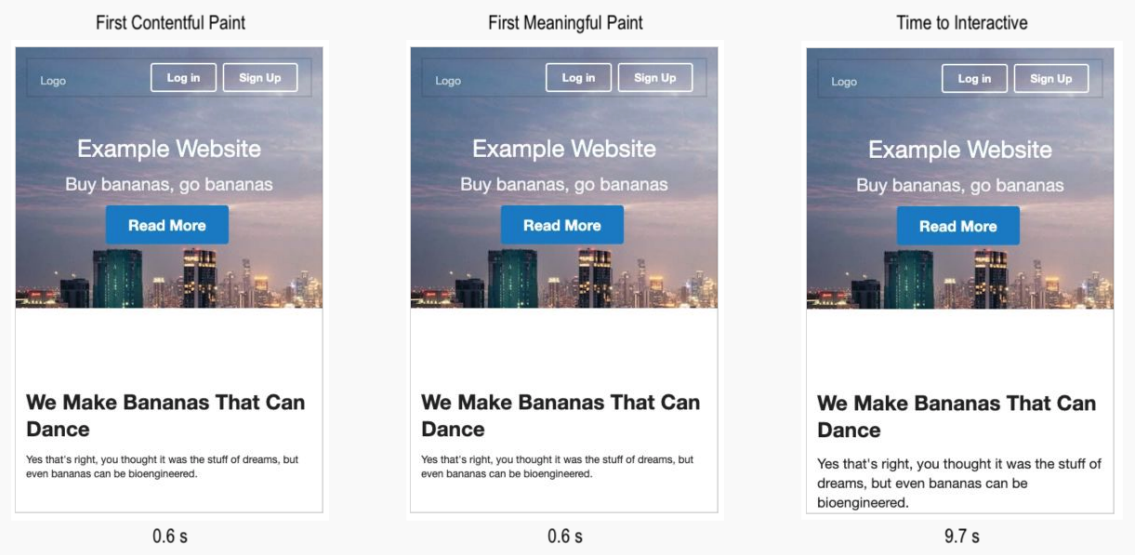

FIGURE 4.4: Loading stages of the optimized sample website.

As we can see from the above picture, the website appears ready to use in 0.6 seconds despite the asynchronous resources being handled in the background. All the interactions of the website, including the mobile navigation bar, are ready after 0.6 seconds as these were part of the critical JavaScript.

Chapter 5

Discussion

5.1 General Optimization Methods

5.1.1 Compressing Resources

Compressing resources was the most effective strategy for the experiment website, causing a 65.5 second decrease in loading time. Because this strategy was very fast and easy to implement on an Apache web server, it is highly recommended to use this strategy. Compressing resources has no negative side effects and is used by more than 80% of all websites [25].

5.1.2 Minimizing JavaScript

Minimizing JavaScript caused a 25.8 second decrease in first contentful paint loading time on the experiment website. Applying this strategy is easy and fast using widely available tools. As this strategy caused a 61% reduction in the size of the JavaScript code, applying it is recommended for any website.

Minimizing JavaScript makes the scripts hard to read and edit. It is advisable to combine and minify these resources automatically when creating a production

build. Therefore, the potential side effect of decreased readability can be eliminated. Also, when combining the JavaScript files it is necessary to keep the code in correct order. Otherwise, there are no side effects to this optimization strategy.

5.1.3 Minimizing CSS

Minimizing CSS caused a 6.8 second decrease in first contentful paint loading time on the experiment website. Applying this strategy is easy and fast using widely available tools. Even though applying this strategy did not yield significant time savings, it is advisable to apply this to any website as it reduces the size of the website and is easy to apply.

Similarly to JavaScript, minimized CSS is hard to read and edit, and it is recommended that also CSS files are minified and combined automatically as part of creating a production build of the website. Otherwise, this strategy has no negative side effects.

5.1.4 Reducing the Number of Requests

Reducing the number of requests caused a two second decrease in time to first contentful paint on the experiment website. This is rather insignificant, but the reduction in loading time could be significant if the external files would have been on different servers and would have had higher latencies. On the experiment website, the external resources were all on the same server. If they would have been on a different server, the loading time would have been affected by the location and connection to the server. Therefore, having a minimal number of resources and hosting them on the same server is recommended.

This strategy was easy to apply. When combining resources it is important to keep the execution order of the code similar to the original loading order. No negative side effects were identified.

5.1.5 Delivering Images

Optimizing the delivery of the images caused a 0.1 second decrease in loading time on the experiment website. This is a surprisingly poor result considering that the total size of the website was reduced from 21.4 MB to 15.3 MB.

However, the background image loads faster and gives the impression of a fast render, despite the insignificant decrease in loading time. Also, after the optimization, the image fits the mobile screen well. These matters are demonstrated in the pictures below.

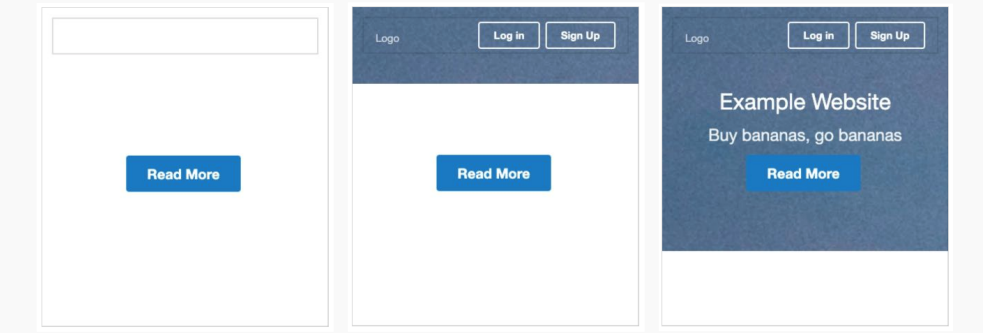

FIGURE 5.1: Unoptimized image delivery.

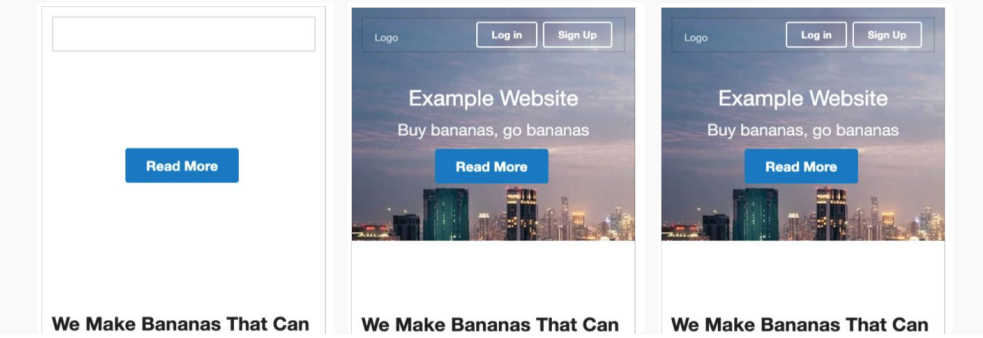

FIGURE 5.2: Optimized image delivery.

Applying this strategy requires insignificant effort and has a positive effect on the user experience. Even though applying this strategy yielded an insignificant decrease in loading time on the experiment website, it is recommended to apply this to any website due to improved user experience.

The delivery of images can be further optimized. After applying this strategy, the images caused six HTTP requests. This can be decreased by combining the

images into one image and using CSS sprites for displaying the images. There are no negative side effects of optimizing image delivery.

5.1.6 Combining General Optimization Strategies

The combination of all the general optimization strategies caused a 71.4 second decrease in the time to first paint on the experiment website, which resulted in 8.7 seconds in time to first paint. The size of the website decreased to 9.0 MB, which is significantly larger than the average website of 2 MB. The decrease in loading time is significant and improves the usability of the website radically. However, a loading time of 8.7 seconds is not sufficient for good user experience.

In order to provide a good user experience on the experiment website with the experiment setup, above-the-fold content optimization strategies need to be applied. This applies also to websites that have large external style sheets and scripts.

As presented in chapter 2.4, a two-second loading time of a website is a reasonable goal. Considering that the bandwidth throughput was 1,638.4 Kbps and the latency was 150 ms on the experiment website, in order to achieve a time to first paint of less than two seconds, the total size of resources that load before the first pixel is painted on the screen can be at maximum 3,031.04 kilobits, which translates to 378,88 kilobytes. This calculation does not take into account any other factors such as DNS lookups, rendering, painting, or the execution time of the critical JavaScript. This means that the actual number of critical bytes should be less than 378,88 kilobytes on the experiment website in order to reach a time to first paint in less than two seconds.

Combining the strategies required that each strategy had to be implemented separately. All of these strategies work well together and no negative side effects were identified.

5.2 Critical Rendering Path

5.2.1 Critical CSS

Optimizing the delivery of CSS decreased the time to first paint by 3.9 seconds. The time to first meaningful paint decreased by 1.9 seconds and the time to interactive was unchanged.

As the critical CSS was extracted from the CSS file and in-lined in the HTML, the critical portion of the CSS was loaded twice. Hence, the size of the website was larger and the time to interactive measurement gives a worse result.

Applying this strategy manually requires significant effort for unoptimized websites. However, there are tools available for extracting the critical CSS that makes this strategy easy to apply. Applying this strategy is highly recommended for any website.

Nevertheless, applying this strategy could cause some CSS to be loaded twice, as in the experiments in this thesis. However, this can be avoided by identifying and erasing the critical CSS from the CSS file.

5.2.2 Critical JavaScript

Optimizing the delivery of JavaScript resulted in a significant decrease in loading time. This was expected, as the size of the critical JavaScript was 20 lines of code, which is a very small amount. The external JavaScript was loaded with the *defer* attribute, meaning it was fetched in parallel with the parsing of the HTML and executed after the HTML was parsed. Therefore, only the critical JavaScript was loaded when the first pixel was painted on the screen.

Applying this strategy requires a significant amount of work when done manually. Extracting the JavaScript is a more complex task than extracting the CSS, but there are tools available for this purpose.

This strategy is worth implementing especially for websites with large amounts of JavaScript. A negative side effect of this strategy is that some user interactions may not be available before the JavaScript has loaded.

5.3 Combining All Optimization Strategies

Once all the optimization strategies were implemented on the experiment website, the first meaningful paint loading time decreased from 80.2 seconds to 0.6 seconds. Even though the time to interactive loading did not decrease after applying the above-the-fold content optimization strategies, the result is phenomenal.

The time to first contentful paint and the time to first meaningful paint are important metrics for user experience. Both metrics are 0.6 seconds for the optimized experiment website. From a user's perspective, the experiment website looks ready to use after 0.6 seconds of loading although the browser is still working in the background.

The time to interactive is 9.7 seconds for the optimized experiment website, which is quite high. However, the critical JavaScript, which is loaded after 0.6 seconds, contains the functionalities that allow the user to perform the critical interactions. In this case, the only critical interaction for a user is the navigation bar that uses JavaScript. Therefore, the experiment website is usable after 0.6 seconds of loading.

The experiments were simulated with a poor 4G connection, an average webserver, and an average client device. However, the loading time of the optimized experiment would have been much faster with a better connection. The connection played a more crucial role in the experiments, as the webserver and user device did not cause bottlenecks for the loading of the website.

The average size of a mobile website is 1.9 MB [11], while the size of the experiment website was 9.1 MB after applying all the optimization strategies. Therefore, an average website may have an even lower loading time after applying these strategies, depending on the number of critical bytes the average website has.

5.4 Further Optimizations

The loading time of the experiment website was reduced to a satisfactory level. However, some optimizations could be added for further reduction in loading time. Identifying and removing all unused JavaScript and CSS would have made the experiment website smaller in size, causing fewer bytes to be downloaded. This would have decreased the time to interactive loading time.

Implementing a cache solution would decrease the loading time for returning visitors. Nevertheless, this would not affect the loading time for users visiting the website for the first time.

This thesis did not take into account code optimizations and how they affect the loading time of a website. Optimizing client-side and server-side code could potentially decrease the loading time, depending on how the website is built. The experiment website did not use any server-side code.

Chapter 6

Conclusion

In this master's thesis, seven strategies for decreasing the website loading time were tested. These strategies were divided into two groups, general optimization strategies and critical rendering path specific optimization strategies. The general optimization strategies focus on the overall decrease in loading time, while the critical rendering path specific strategies focus on delivering the above-the-fold content as fast as possible.

The different strategies were tested using the metrics first contentful paint, first meaningful paint, and time to interactive. These metrics provide an accurate understanding of the loading process of websites.

The following general optimization strategies for decreasing website loading time were tested: minimizing CSS, minimizing JavaScript, reducing the number of requests, compressing resources, and delivering images.

All of the general optimization strategies tested in this thesis reduced the loading time of the experiment website. Compressing resources had the greatest impact on the loading time of the strategies tested. Finally, all these strategies were implemented to the experiment website together. This resulted in a significant reduction in loading time.

The following critical rendering path specific optimization strategies for decreasing the loading time of the above-the-fold content were tested: critical CSS and critical JavaScript

The combination of the critical rendering path specific optimization strategies resulted in a very significant reduction in loading time. The amount of work required to implement these strategies was moderate on the test website but the implementation can potentially be automated. The conclusion is that even a large website can load the above-the-fold content remarkably fast and provide an excellent user experience with the optimization strategies introduced in this thesis.

Bibliography

[1] Google LLC. Online. Accessed May 2018. No longer available. URL https://developers.google.com/web/updates/2017/06/user-centric-performance-metrics/.

[2] HTTP Archive. Page weight. Online. Accessed April 2020. URL https://httparchive.org/reports/page-weight?view=grid.

[3] Google LLC. Find out how you stack up to new industry benchmarks for mobile page speed. Online. Accessed April 2020 . URL https://think.storage.googleapis.com/docs/mobile-page-speed-new-industry-benchmarks.pdf.

[4] Pratiksha H. Shroff and Seema R. Chaudhary. Critical rendering path optimizations to reduce the web page loading time. *2017 2nd International Conference for Convergence in Technology (I2CT)*, 2017.

[5] M. Jalalzai, W. Shahid, M. Iqbal. Dns security challenges and best practices to deploy secure dns with digital signatures. *Applied Sciences and Technology (IBCAST) 2015 12th International Bhurban Conference on, pp. 280-285*, 2015.

[6] Martin L. Abbott and Michael T. Fisher. *Scalability Rules: 50 Principles for Scaling Web Sites, 2nd Edition*, Addison-Wesley Professional, August 2011.

[7] Ilya Grigorik. Online. Accessed April 2020. URL https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model.

[8] Ilya Grigorik. Online. Accessed April 2020. URL https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-constructionl.

[9] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *9th Americas Conference on Information Systems, AMCIS 2003, Tampa, FL, USA, on*, August 4-6, 2003.

[10] Robert B. Miller. Response time in man-computer conversational transactions. *AFIPS Conference Proceedings, 1968,33 (Pt. 1), 267–277*, 1968.

[11] HTTP Archive. State of the web. Online. Accessed April 2020. URL https://httparchive.org/reports/state-of-the-web#pctHttps.

[12] Ookla LLC. Speedtest global index. Online. Accessed April 2020. URL https://www.speedtest.net/global-index.

[13] Marshini Chetty, David Haslem, Andrew Baird, Ugochi Ofoha, Bethany Sumner and Rebecca E. Grinter. Why is my internet slow?: Making network speeds visible. *International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, on*, May 7-12, 2011.

[14] Patrick Killelea. *Web Performance Tuning, 2nd Edition*, O'Reilly Media Inc., March 2002.

[15] W3 Techs. Usage statistics of web servers. Online. Accessed April 2020. URL https://w3techs.com/technologies/overview/web_server.

[16] Rick Lehtinen, G. T. Gangemi. *Computer Security Basics*, O'Reilly Media, June 2006.

[17] Steve Souders. Browser script loading roundup. Online. Accessed April 2020. URL https://www.stevesouders.com/blog/2010/02/07/browser-script-loading-roundup/.

[18] Vyas Sekar Butkiewicz Michael, V. Madhyastha Harsha. Characterizing web page complexity and its impact. *IEEE/Acm Transactions On Networking, vol. 22, no. 3*, June 2014.

[19] Steve Souders. *High Performance Web Sites - Essential Knowledge for Frontend Engineers*, O'Reilly Media, December 2008.

[20] Yasutaka Sakamoto, Shinsuke Matsumoto, Seiki Tokunaga, Sachio Saiki, Masahide Nakamura. Empirical study on effects of script minification and http compression for traffic reduction. *2015 Third International Conference on Digital Information, Networking, and Wireless Communications (DINWC)*, 2015.

[21] Chris Coyier. Online. Accessed April 2020. URL https://css-tricks.com/the-difference-between-minification-and-gzipping/.

[22] Andrew Chilton. Online. Accessed April 2020. URL https://cssminifier.com/.

[23] Matthias Mullie. Online. Accessed April 2020. URL https://github.com/matthiasmullie/minify.

[24] Jonas Ohlsson Aden. Online. Accessed April 2020. URL https://jonassebastianohlsson.com/criticalpathcssgenerator/.

[25] W3 Techs. Usage statistics of gzip compression for websites. Online. Accessed April 2020. URL https://w3techs.com/technologies/details/ce-gzipcompression/all/all.

Swedish Summary

Introduktion

Snabb nedladdning av webbsidor är viktigt av flera orsaker. Nedladdningstiden påverkar hur slutanvändaren upplever webbsidan samt hur hen beter sig på webbsidan. Eftersom webbsidors roll är betydlig för allt flera företag, har webbsidors nedladdningstid direkt påverkan på affärsverksamheten. Dessutom blir dagens webbsidor mer komplexa samtidigt som användarna kräver kortare nedladdningstider. I denna avhandling fokuseras på hur nedladdningstiden av webbsidors kritiska innehåll kan minimeras. Kritiska innehållet är det innehåll som användaren ser när hen öppnar webbsidan innan skrollning.

Bakgrund

När en användare navigerar till en webbsida skickar webbläsaren en HTTP-, HTTPS- eller HTTP2-begäran till servern där webbsidan ligger. Webbläsaren får ett svar i HTML varifrån den bygger ett träd för domänobjektsmodellen (DOM) samt ett träd för en stilmallsobjektmodell (CSSOM). DOM representerar webbsidans innehåll och CSSOM representerar webbsidans stil. Webbläsaren kombinerar DOM och CSSOM till ett renderingsträd som innehåller endast de synliga noderna från DOM:et och motsvarande stil från CSSOM [7]. Därefter räknar webbläsaren storleken och positionen för varje nod i renderingsträdet. Slutligen målar webbläsaren resultatet på skärmen och webbsidan har laddats.

Kort nedladdningstid för det kritiska innehållet ger ett intryck av en omedelbar nedladdning fastän webbsidan skulle fortsätta ladda resurser i bakgrunden. Optimering av det kritiska innehållet är speciellt viktigt för stora webbsidor och nedladdning av webbsidor med dålig anslutning. Det finns flera olika sätt att mäta nedladdningstiden av webbsidor. I denna avhandling delas nedladdningstiden i fyra delar. Första delen är tiden från att webbläsaren skickar en HTTP-begäran tills användaren ser första pixeln på skärmen. Denna tid kallas tid till första målning (time to first paint). Den följande är tid till första innehållsrika målning (time to first contentful paint), när användaren ser något betydelsefullt på skärmen, såsom text eller en bild. Den tredje är tid till första meningsfulla målning (time to first meaningful paint), där användaren ser något betydelsefullt såsom en produktbeskrivning och produktbild i en webbutik. Den sista mätaren kallas tid till interaktivitet (time to interactive), där webbsidan är användbar.

Flera faktorer påverkar nedladdningstiden av en webbsida. Nätverkslatens är tiden som det tar för webbläsaren att få HTTP-respons från webbservern. Varje förfrågan dröjs av nätverkslatens. Distansen mellan klienten och servern är den största orsaken till nätverkslatens. Klientens nätverksanslutning påverkar också nedladdningstiden. Genomsnittliga nätverksanslutningar orsakar ingen flaskhals för laddandet av webbsidor av medelstorlek. Medelstorleken för en webbsida är 2 MB [2] medan genomsnittliga mobila nätverksanslutningen har hastigheten 30 Mbps [12]. HTML-filen som webbservern skickar som HTTP-respons för att visa en webbsida på browsern, innehåller ofta externa Javascript- och CSS-filer. Ordningen som dessa filer laddas i och på vilket sätt dessa laddas påverkar nedladdningstiden. Externa filer kan laddas synkroniskt och asynkroniskt. Det synkroniska laddandet är långsammare än det asynkroniska eftersom koden hämtas och körs innan browsern kan fortsätta parsningen. Då externa resurser laddas asynkroniskt, hämtas de parallellt med parsningen. Dock är inte resursen användbar förrän den har laddats fullständigt. Webbservern där webbsidan ligger spelar också stor roll i nedladdningshastigheten. Apache HTTP-server är den mest allmänna webbservern som används på 38,9 % av alla webbsidor [15]. I webbserverns konfigureringsfiler kan man ändra på flera inställningar som gör att en webbsidas nedladdningstid blir kortare.

Optimeringsmetoder

I denna avhandling delas optimeringsmetoderna i två delar. Första delen är allmänna optimeringsmetoder och andra delen fokusera på optimeringsmetoder för det kritiska innehållet.

Varje HTTP-förfrågan som skickas från webbsidan ökar på nedladdningstiden. Minskandet av antalet HTTP-förfrågningar genom att kombinera resurser är en effektiv allmän optimeringsmetod. Följande optimeringsmetod är minimering av dessa resurser, där alla onödiga tecken tas bort ur resursen. För att göra resurserna ännu mindre kan de komprimeras med en metod som kallas gzip, vilket minskar signifikant på filstorleken. Följande metod är optimerandet av bilder, där bildstorleken optimeras för den skärm där webbsidan visas. Femte allmänna optimeringsmetod är caching. Resurser kan sparas i webbläsarens cacheminne, vilket minskar på laddningstiden då webbsidan besöks följande gång. De två sista allmänna optimeringsmetoder som behandlas i denna avhandling är CSS-leverans och Javascript-leverans. Både CSS och Javascript kan levereras som externa filer eller direkt i HTML-koden. Asynkroniskt laddande av dessa filer minskar ofta på nedladdningstiden, men kan orsaka problem i och med att resurserna är oanvändbara innan de har laddats ner.

De allmänna optimeringsmetoderna minskar också på nedladdningstiden av det kritiska innehållet. Följande optimeringsmetoder fokuserar enbart på minskandet av nedladdningstiden av det kritiska innehållet. Den första är minimering av antalet kritiska resurser. Endast de resurser som det kritiska innehållet kräver för interaktivitet och stil ska nedladdas. Ju färre resurser webbläsaren behöver ladda, desto kortare blir nedladdningstiden. Den andra optimeringsmetoden för det kritiska innehållet är minimering av antalet kritiska HTTP-förfrågningar. Optimalt ska detta antal vara noll, då all kritisk Javascript och CSS är i HTML-koden.

Den sista optimeringsmetoden för det kritiska innehållet är tudelad, optimering av kritiskt CSS och optimering av kritiskt Javascript. Det kritiska CSS:et identifieras från CSS-filerna och tillsätts till HTML:et. Därefter laddas externa CSS-filer asynkroniskt. På samma sätt identifieras de kritiska delarna av Javascript-filerna och resten av Javascript-koden laddas direkt i HTML:et.

Experiment

Följande allmänna optimeringsmetoder testas på en icke-optimerad experimentwebbsida: minimering av CSS, minimering av Javascript, minskandet av antalet HTTP-förfrågningar, komprimering med gzip och optimering av bilder. Alla dessa metoder testas enskilt på experimentwebbsidan varefter de testas tillsammans.

Optimeringsmetoderna för det kritiska innehållet, det vill säga optimering av kritiskt Javascript och optimering av kritiskt CSS, testas på en version av experimentwebbsidan som har alla ovannämnda allmänna optimeringar. Slutligen testas optimeringsmetoderna tillsammans för en sammansatt effekt. Experimenten körs fem gånger på experimentwebbsidan.

Experimenten har följande mål: evaluera hurdan påverkan varje optimeringsmetod har på nedladdningstiden, evaluera hur mycket arbete implementeringen av varje optimeringsmetod kräver och evaluera hurdan påverkan alla optimeringsmetoder har tillsammans på nedladdningstiden.

Experimenten körs på en Apache/2.4.39 webbserver som ligger på en Macbook Pro 2018. Google Lighthouse används för att köra testerna med simulerad nätverksstrypning som motsvarar en dålig mobil 4G-förbindelse. En icke-optimerad experimentwebbsida skapades för dessa experiment. Webbsidan saknar alla optimeringar och har en väldigt lång nedladdningstid på över 80 sekunder. Webbsidan består av en HTML-fil, sju stora Javascript-filer, sex stora CSS-filer och fem bilder. Den totala storleken på webbsidan är 21,4 MB vilket motsvarar en relativt stor webbsida, då medelstorleken på en webbsida är endast 2 MB.

Resultat

Alla allmänna optimeringsmetoder minskade på nedladdningstiden. Metoden komprimering av resurser gav bästa resultat av dessa optimeringsmetoder.

Table 6.1: Jämförelse av olika optimeringsmetoders inverkan på nedladdningstiden

Allmänna optimeringsmetoder			
Strategi	FCP	FMP	TTI
Före optimeringar	80,1 s	80,2 s	80,4 s
Minimering av CSS	73,3 s	73,4 s	73,7 s
Minimering av JS	54,3 s	54,4 s	54,7 s
Optimering av bilder	80,0 s	80,0 s	80,2 s
Komprimering av resurser	14,6 s	14,7 s	14,7 s
Minimering av antalet förfrågan	78,1 s	78,1 s	78,5 s
Sammansatt	8,7 s	8,7 s	9,4 s

Den sammansatta effekten av alla de ovannämnda optimeringsmetoderna är väldigt bra. Nedladdningstiden minskade från 80,1 sekunder till 8,7 sekunder. Storleken på webbsidan minskade från 21,4 MB till 9,1 MB och antalet HTTP-förfrågningar minskade från 20 till 8.

Optimeringsmetoderna för det kritiska innehållet implementerades ovanpå de ovannämnda allmänna optimeringsmetoderna. Resultaten visas i tabellen nedan.

Table 6.2: Jämförelse av optimeringsmetoder för det kritiska innehållet

Optimeringsmetoder för det kritiska innehållet			
Strategi	FCP	FMP	TTI
Före optimeringar	8,7 s	8,7 s	9,4 s
Kritisk JS	5,0 s	5,0 s	9,6 s
Kritisk CSS	4,8 s	6,8 s	9,4 s
Sammansatt	0,6 s	0,6 s	9,7 s

Båda optimeringsmetoderna för det kritiska innehållet gav goda resultat. Den sammansatta effekten av optimeringsmetoderna var väldigt bra, då nedladdningstiden minskade till 0,6 sekunder.

Diskussion

Komprimering av resurser orsakade största minskningen av nedladdningstiden av experimentwebbsidan. Nedladdningstiden minskade med 65,5 sekunder. Implementeringen av denna metod var väldig enkel och gjordes i Apache-webbserverns konfigurationsfil. Metoden saknar negativa sidoeffekter och den används av över 75% av alla världens webbsidor.

Minimering av Javascript orsakade att storleken av Javascript-filerna minskade med 61%. Nedladdningstiden minskade med 25,8 sekunder vilket är betydligt. Att implementera denna metod är lätt med allmänt tillgängliga verktyg. Minimering gör koden i praktiken oläslig och därför rekommenderas det att minimering automatiseras och utförs först när produkten är klar för leverans.

Minimering av CSS förkortade nedladdningstiden med 6,8 sekunder. Implementeringen av denna metod var enkel med allmänt tillgängliga verktyg. Eftersom denna metod minskar på filstorleken och på nedladdningstiden, är implementering av metoden att rekommendera. Metoden saknar negativa sidoeffekter.

Minimering av antalet förfrågningar förkortade nedladdningstiden med endast två sekunder. De externa resurserna laddades ned från samma webbserver vilket gjorde metoden mindre effektiv. Ifall resurserna hade laddats från en annan server skulle nätverkslatensen ha varit större och således skulle optimeringsmetoden ha minskat nedladdningstiden mer. Metoden är lätt att implementera och har inga negativa sidoeffekter.

Optimering av bilder förminskade storleken av webbsidan från 21,4 MB till 15,4 MB. Trots detta blev nedladdningstiden bara 0,1 sekunder kortare. Bilderna laddades efter att de externa resurserna hade hämtats och exekverats. Implementerandet av metoden förbättrade på användarupplevelsen i och med att hela banner-bilden laddades vid första meningsfulla målningen. Eftersom metoden förbättrar användarupplevelsen är det att rekommendera att implementera denna metod. Metoden har inga negativa sidoeffekter.

Optimering av kritiskt CSS förkortade på nedladdningstiden från 8,7 sekunder till 4,8 sekunder. Då metoden implementeras manuellt krävs det mycket arbete

men det finns verktyg för implementeringen. Ifall de kritiska CSS-reglerna inte raderas från de externa CSS-filerna laddas kritiska CSS:et två gånger. Metoden saknar andra nackdelar.

Optimering av kritiskt Javascript förkortade nedladdningstiden till 5,0 sekunder. Som vid optimering av kritiskt CSS, kräver implementeringen av denna metod mycket manuellt arbete ifall ett verktyg inte används.

Genom att kombinera optimeringsmetoderna minskade nedladdningstiden från 80,1 sekunder till 0,6 sekunder. Tiden till interaktivitet blev 9,7 sekunder vilket är en relativt långt tid. Trots detta var webbsidan användbar efter 0,6 sekunder.

Slutsats

I denna avhandling behandlades sju olika metoder för optimering av nedladdningstiden på webbsidor. Dessa metoder delades i två delar: allmänna optimeringsmetoder och optimeringsmetoder för det kritiska innehållet. Alla de allmänna optimeringsmetoderna implementerades enskilt på en experimentwebbsida. Optimeringsmetoderna för det kritiska innehållet implementerades på en version av experimentwebbsidan där de allmänna optimeringsmetoderna var implementerade.

Slutligen implementerades alla optimeringsmetoder för en optimerad webbsida. Resultatet var en väldigt signifikant förkortning av nedladdningstiden. Den slutsats som kan dras av dessa experiment är att det kritiska innehållet även på stora webbsidor kan laddas på en kort tid för god användbarhet.