Degree project

Design and implementation of a next generation Web Interaction SaaS prototype
Abstract

Web applications are getting more and more complicated with the extensive growth of the Internet. In order to cope with user demands, that are constantly increasing, a special attention should be paid to performance optimizations. While a lot of attention is devoted to back-end optimization, front-end is often overlooked and therefore is a fertile ground for performance bottlenecks.

This thesis is destined to investigate a set of well-established front-end optimization techniques in order to find out those, that are the most efficient.

The thesis primarily focuses on an examination of a limited set of techniques, that can be applied to static web resources. Some of the techniques are: resources consolidation, minification, compression and caching. The measurements used during the examination are based on four metrics, such as the Page Size, the Page Load Time, the Page Start Render Time and the Number of Requests the page made.

The results show which methods impact performance most. In particular, the results revealed, that the resource compression technique alone brings significant performance improvements, the page size was reduced by 79% and the page load time by 72%, respectively. Despite that, it is evident that the best results can be achieved by a combination of different techniques. All optimization techniques combined made a serious difference, helping us reduce the page load time from 24 seconds down to just one second.

Keywords

Web applications, front-end performance, page load time, performance optimization
Content

1. Introduction...................................................................................................................1
   1.1 Background...........................................................................................................1
   1.2 Problem...............................................................................................................2
   1.3 Goals.....................................................................................................................2
   1.4 Outline..................................................................................................................2
2. Background...................................................................................................................3
   2.1 Web performance optimization.............................................................................3
   2.2 Front-end optimization........................................................................................3
3. Optimization methods in action...............................................................................7
   3.1 Setting up the test environment...........................................................................7
   3.2 Defining metrics....................................................................................................8
   3.3 Applying optimization methods...........................................................................9
   3.4 Baselining..............................................................................................................9
   3.5 Rearrange resources..........................................................................................10
   3.6 Bundle resources................................................................................................11
   3.7 Bundle and minify resources..............................................................................16
   3.8 Bundle and compress resources..........................................................................19
   3.9 Bundle, minify and compress resources..............................................................20
   3.10 Cache resources.................................................................................................21
4. Discussion....................................................................................................................23
   4.1 Page Load Time metric.......................................................................................23
   4.2 Page Size metric...................................................................................................24
   4.3 Number of Requests and Start Render Time metrics..........................................25
   4.4 Optimization methods applicability....................................................................26
5. Conclusion and future work......................................................................................27
6. References...................................................................................................................28
1. Introduction

The following chapter explains the importance of web optimization from the user and the business standpoint, briefly describes web optimization techniques used over the years and presents compelling reasons as to why the front-end should be optimized.

1.1 Background

Studies done in 1997 by Jakob Nielsen (1997, 2010), an expert in the field of web usability, pointed out how people hated web pages that loaded slowly and how much they prefer and demand faster and more responsive web sites.

Recent researches indicate that a user expects a web page to load in two or less seconds, and 40% of users would abandon a web site if it takes more than three seconds for a web page to load. Interestingly enough, in 2006 the expected web page load time was four seconds, and eight seconds in 2003. The same researches for commercial sites found out that 79% of online shoppers, who visited an underperforming web site, were likely not to buy any goods from it, and eight out of ten people would never return after a disappointing user experience (Bixby 2010, Akamai 2009).

Not only faster pages equal more revenue. Speed is also a great delight for people. There is a real world example from the consumer electronics field, that indicates that the speed pleases people. Today SSDs (Solid-State Drive) grow in popularity, and the reason behind that is not because they are more reliable or more durable (because they are absolutely not), but largely because they work faster: they wake up a computer from sleep faster, as well as they copy files faster. And people seem to like it despite the higher prices.

At the time Jakob Nielsen conducted his research, the situation was different: the Internet was in its infancy, connection were slow and computers were underpowered. Optimization methods in the web were different too. Essentially, developers had to keep page sizes smaller by conservative use of graphics and multimedia. After more than 15 years both computers and the Internet changed significantly. Today’s faster CPUs (Central Processing Unit), more affordable memory and broadband availability drove the Internet to where it is today.

If we take a look at the optimization techniques from back then and compare them with what we have now, they may appear to be quite different, yet they had the same purpose. No matter when, today or in the nineties, web optimization attempts always shared a common goal, to display the information the user requested as fast as possible.

A growing movement towards SaaS (Software as a Service, also referred as an "on-demand software") and RIA (Rich Internet Applications) makes web pages heavier and introduces a lot of heavy lifting both for servers and clients. The answers to the majority of web site performance problems are often the database scheme optimization, the server upgrades or the back-end application code refactoring. There is a lot of attention devoted to these very approaches. At the same time, researches conducted by Yahoo! and Google performance engineers revealed, that for the majority of websites, the back-end performance accounts for only 10-20% of the overall time required for a page to load. This time is spent generating an HTML (HyperText Markup Language) content and transmitting it from a server to a client (a browser). The remaining 80-90% of the time is spent downloading the web page static resources, such as styles, scripts and images.

Considering the importance of faster pages we would like to leverage the existing front-end performance optimization techniques in the project our team is working on.
1.2 Problem
The process of a page creation involves both server-side and client-side steps: starting from the moment the client requests it, until the moment the content is loaded. There is a number of different approaches to the web optimization and they primarily address the server-side. The main problem is that front-end optimizations are often obscure and therefore underrated or, at the worst, entirely disregarded.

Why is this a problem? As we mentioned before, whereas the vast majority of the efforts of web performance optimization are spent on the back-end, recent researches show that the front-end is where users wait the most. The back-end optimization is a standard practice and a variety of tools help with it, but the front-end optimization is a newer field and it still lacks in relevant expertise and solutions. Moreover, the front-end optimizations are often more complicated, as they lie in the intersection of the client-side and the server-side development, web usability, behavioral psychology, etc.

The questions the thesis is set to answer are as follows. Why is the front-end slow? What kind of front-end optimization methods are used now? Will our application perform better when these methods are applied? In order to answer the questions I will study, survey and evaluate some of the front-end optimization practices used today, and this is explained in the upcoming section.

1.3 Goals
The thesis pursues the following goals:

- To review the existing optimization techniques, understand their pros and cons and when they can be and cannot be applied.
- To apply a set of optimization techniques in the actual web application.
- To analyze the results to see which method or a set of methods yield the best results.

1.4 Outline
The thesis contains five chapters. Chapter 1 highlights the problem of the web performance optimization. Chapter 2 gives some general knowledge in order to understand optimization techniques deeper. Chapter 3 describes the test environment, tools and methods in the beginning and later demonstrates our experiments and measurements. In Chapter 4 we discuss the results obtained in Chapter 3 and conclude the experiments. Chapter 5 draws a conclusion based on the experiments and presents the potential for our future work.
2. Background

This chapter is an introductory chapter into the subject of web optimization with the emphasis on the front-end. This chapter explains issues that negatively impact performance of web pages and suggests a number of methods that can mitigate some of them.

2.1 Web performance optimization

Performance of a web application is a complex matter and can be described by the following characteristics:

- Responsiveness.
- Throughput.
- Reliability and availability.
- Scalability and stability.

From the user perspective, responsiveness is the essential characteristic, because users perceive a web application from the client-side user interface. The architecture of an application, the network configuration and the server-side data processing are not visible to the end user.

A process of delivering a page to a user is complicated as it involves many different steps. These steps can be loosely split into back-end and front-end actions. The back-end ones include accepting and processing a request from a user and generating the page that will be returned to the user. This part might involve security control checks, executing server-side code, fetching data from the database and more, up to the moment when the page content is ready to be served. The front-end includes everything, that follows the page creation. This means delivering the page, including its resources, over the network to the user who requested it, as well as rendering and post-processing it in the browser. The complete process of speeding up a web site, including both back-end and front-end, is usually referred to as web performance optimization.

Before we get our hands to any performance optimizations it is important to know where users spend their time wait the most. Steve Souders (2007, 2009) coined the term “performance golden rule”, which says that there is a huge split between the back-end and the front-end time. The back-end time is the time the first byte reaches the client, and the front-end time is everything else that happens afterwards. Based on the analysis of the top 50000 popular websites, the back-end time is just 13% of the total response time (Souders 2012). Although our project is immature and is still being developed, time measurements show that the back-end time is only 2% of the total time (0.49 seconds of the total of 24.29 seconds).

2.2 Front-end optimization

Why the front-end is painfully slow? Various trends contribute to this problem.

First of all, web pages are getting bigger and heavier. Web pages have increased sevenfold in size over the last few years. This is the result of adding more and more dynamic content, functionality and media components, that aim to satisfy users. Bigger pages are, naturally, slower to download and more difficult for browsers to process and render. This makes any rich functionality of contemporary web applications a fertile ground for performance bottlenecks.
Second, a shortage of the front-end optimization tools and expertise, as compared to the back-end optimization, has the impact as well. Website owners default to spend their time on the back-end, and do not know how or even if they should speed up the rest.

Front-end optimization is the optimization of the page loading process – the process, performed by the client application, a browser. Front-end optimization is a generalized title for a series of established techniques used to speed up front-end components. The techniques vary greatly, but aim to reduce a page download time.

It is appropriate to examine optimization techniques from the page load time viewpoint. The process of loading a page can be loosely split into two sequential steps:

1. Downloading the page and its resources.
2. Processing and rendering the page: constructing the DOM (Document Object Model) tree, computing styles and processing scripts.

The following approaches are suggested in order to speed up the former step (the process of downloading the page):

1. Reduce the payload.
2. Reduce the number of HTTP (Hypertext Transfer Protocol) requests made to the server by the client.
3. Speed up resources.

In order to speed up the second step, the subsequent processing and rendering, the page resources should be optimized. A developer should have an absolute understanding of how different browsers work and how they handle page resources.

Let us discuss some of the approaches mentioned above.

The first approach is to reduce the payload. This is an attempt to reduce the amount of data being sent over the network. The payload includes the HTML content of the page itself and static resources, such as styles (CSS, Cascading Style Sheets), scripts (JS, JavaScript), images and other multimedia objects. Table 2.1 below briefly describes techniques used in this approach.

<table>
<thead>
<tr>
<th>Technique name</th>
<th>Objects affected</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minify resources</td>
<td>HTML, CSS, JS</td>
<td>Reduce static page resources in size by removing unnecessary characters and running other algorithms that make the source code more compact. <strong>Pros</strong>: browser and programming language independent <strong>Cons</strong>: may affect functionality (valid for JS resources)</td>
</tr>
<tr>
<td>Compress resources</td>
<td>HTML, CSS, JS</td>
<td>Reduce the source code significantly by applying gzip compression. <strong>Pros</strong>: very efficient <strong>Cons</strong>: lack of support in legacy servers and browsers</td>
</tr>
<tr>
<td>Compress images</td>
<td>Images</td>
<td>Apply image compression algorithms. <strong>Cons</strong>: worse quality of the images and possible compression artifacts</td>
</tr>
<tr>
<td>Externalize resources</td>
<td>HTML</td>
<td>Take away inlined styles and scripts to external files in order to reduce the</td>
</tr>
</tbody>
</table>
**Pros**: (relatively) easy to implement, works across all environments  
**Cons**: external resources would require additional HTTP requests

Use semantic HTML  
**Pros**: a website is more accessible to screen readers and search engines  
**Cons**: lack of support in legacy browsers

Use cookie-less domains  
**Pros**: (relatively) easy to implement  
**Cons**: external resources would require additional HTTP requests

<table>
<thead>
<tr>
<th>Technique name</th>
<th>Objects affected</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consolidate resources</td>
<td>CSS, JS</td>
<td>Merge resources into a bundle and serve to the client at once. <strong>Cons</strong>: it is difficult to manage resources in complex web application</td>
</tr>
<tr>
<td>Use CSS sprites</td>
<td>Images</td>
<td>Combine all images into a single file and serve it to the client once. CSS property background-position is used to access an image. <strong>Cons</strong>: difficult to manage image files</td>
</tr>
<tr>
<td>Cache resources</td>
<td>All resources</td>
<td>Keep resources on the client so that they are not requested when a page is refreshed. <strong>Pros</strong>: (relatively) easy to implement <strong>Cons</strong>: occupies the disk space on the client; a risk that the resources might not be up-to-date</td>
</tr>
<tr>
<td>Avoid redirects</td>
<td>All resources</td>
<td>Do not redirect a request when a resource is requested. <strong>Cons</strong>: heavily depends on how servers and frameworks implemented</td>
</tr>
</tbody>
</table>

Table 2.1. Techniques used to reduce payload

The next approach is to reduce the number of HTTP requests a page makes. External resources can occupy up to 90% of the total payload, but what is more important is that each resource needs an HTTP request, and each HTTP request contributes to the total overhead costs. Table 2.2 below presents you with some techniques used in this approach.

<table>
<thead>
<tr>
<th>Technique name</th>
<th>Objects affected</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache resources</td>
<td>All resources</td>
<td>Keep resources on the client so that they are not requested when a page is refreshed. <strong>Pros</strong>: (relatively) easy to implement <strong>Cons</strong>: occupies the disk space on the client; a risk that the resources might not be up-to-date</td>
</tr>
</tbody>
</table>

Table 2.2. Techniques used to reduce the number of HTTP requests
The third approach is to speed up the process of handling resources. It has to do with how browsers process an HTML content and scripts and render a page. The main goal is to optimize the page rendering time, so the HTML content and styles should be loaded very quickly. Some techniques are briefly described in Table 2.3.

<table>
<thead>
<tr>
<th>Technique name</th>
<th>Objects affected</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rearrange resources</td>
<td>CSS, JS</td>
<td>Put styles in the <code>&lt;head&gt;</code> tag and scripts in the very end of the page before the closing <code>&lt;body&gt;</code> tag. <strong>Pros:</strong> easy to implement</td>
</tr>
<tr>
<td>Defer loading resources</td>
<td>JS, images</td>
<td>Load scripts when the page is ready. Besides, load scripts and images on demand <strong>Pros:</strong> decreases the payload in addition to the rendering time <strong>Cons:</strong> lack of support in legacy browsers, non-trivial and requires the third-party libraries</td>
</tr>
<tr>
<td>Serve resources from CDN (Content Delivery Network)</td>
<td>All resources</td>
<td>Use a CDN to store resources, so the resources will get to the end user faster <strong>Cons:</strong> CDN service cost</td>
</tr>
</tbody>
</table>

Table 2.3. Techniques used to speed up resources loading

In addition the methods we mentioned above, there are also certain rules on how styles and script should be written. For instance, inefficient CSS instructions should be avoided, CSS selectors should be optimal, and JavaScript code should be efficient. This contributes to less time spent to process and render a page.

The given methods certainly do not cover all existing optimization techniques, but they all aim to provide the better user experience and satisfaction by speeding up the page load time.

This thesis covers a subset of these techniques: resources rearranging, consolidation (bundling), minification, compression and caching. The selected methods are marked gray in the tables above.
3. **Optimization methods in action**

The following chapter describes the test environment, metrics, stipulations and assumptions that were made prior to performing experiments. Besides, here we describe and apply some of the optimization methods that we mentioned in the previous chapters, and compare the performance enhancement before and after the optimizations.

3.1 **Setting up the test environment**

The essence of any test environment is a tool which is measuring data. In order to measure a page performance as accurate as possible, we should use a suitable web page performance measuring tool. There are many performance tools available and we have been choosing between the most popular ones, such as WebPagetest, Firebug with YSlow (Firefox extension), Google Chrome Developer Tools with YSlow and Google PageSpeed (Google Chrome extension).

The choice was made in favor of WebPagetest for the following reasons:

- It is a web-based tool and can be run in any browser and operating system.
- It allows us to choose where tests should be performed, e.g. where the test machine will be located geographically. Client machines to choose from are located all over the world.
- It is possible to choose a browser where tests will be performed. The tool offers all popular browser to choose from.

In addition, the tool gives the high-level information about a page using page-level and request-level metrics. Besides, the tool is equipped with a network monitor and is capable of building HTTP waterfall charts.

A waterfall chart is a diagram that visualizes data that is generated sequentially during the process of downloading and rendering a web page to the client. In our case waterfalls will represent a series of actions that a browser does when it loads a page. A waterfall chart includes a list of resources that a page contains, activities that happen once a resource is delivered, the page load time, the time the rendering started, and other things. The example of a waterfall chart is show in Figure 3.1.

![Figure 3.1. A waterfall chart example](image.png)
Tests that we are about to run may look synthetic, because the test environment conditions are different from the real-world usage. The differences concern both client and server configurations (e.g. computer hardware, software, browsers) and network connection settings (e.g. bandwidth, client location). Thus, it is reasonable to divide factors that influence a page performance into two categories: factors that are controllable and uncontrollable.

The controllable factors are the factors that we can change (the server configuration and server software). The test application is deployed on an Amazon EC2 (Amazon Elastic Compute Cloud) micro-instance (the least-powerful instance available on Amazon EC2). The application is written on Java and Groovy programming languages, uses Grails web framework, and is deployed on a server with the Apache Tomcat web server. Therefore the set of tools we can make use of is limited.

As for the uncontrollable factors, which are the factors that we cannot control in the real world, we created the following conditions for our test: the micro-instance is running in the us-east-1 (Virginia, US) region and the client machine running WebPagetest is located in Stockholm, Sweden. All tests are performed in the Microsoft Internet Explorer 8 browser. All the tables in the subsequent chapters below contain the mean result of 20 test runs for each metric. Of those 20 tests, 10 tests were performed with empty cache and 10 with primed cache.

### 3.2 Defining metrics

There is a number of different metrics that can be measured. In the thesis we are going to measure four major metrics:

- Page Load Time.
- Page Size.
- Number of Requests.
- Start Render Time.

The first metric, the most common and important performance metric, from the front-end perspective, is the Page Load Time. What is the load time? This is a period of time between the moment a user requests a page and the moment when the user gets a complete page in her browser. Several factors influence this metric, such as a server-side latency, a network delay and a client-side latency. The elements this metric is made up of are shown on Figure 3.2. As discussed in the previous chapter, a number of factors (such as how fast the internet connection is and so on) is beyond the control of developers.

![Figure 3.2. The anatomy of the Page Load Time metric](image)

The next metric is the Page Size. It is rather obvious, that the smaller the payload (the page size) is, the less time it would take to transfer the HTTP response across the network to the client (the load time). This metric is made up of the HTML document and external resources, like scripts, styles and images, that the page contains.
Apart from the Page Load Time and the Page Size, there are two more metrics that we will be paying attention to. The metrics are the Number of Requests and the Start Render Time. The Number of Requests metric is the second factor, that has the highest correlation to the page load time. It represents the number of requests a client makes to transfer all external resources that the page includes. The Start Render Time metric represents the time span from the very first request that was sent from a client until the time the first element appeared in the browser window.

Our main priority is to optimize the value of the Page Load Time metric. This is because a user perceives the speed of a page through the time it takes the page to load. The next two metrics are influencing the load time and therefore they should be taken care of as well. Finally, the last metric, despite not being of highest importance, should not be neglected, as the shorter the time it takes for a browser to start rendering a page, the better the first impression of a web site is.

3.3 Applying optimization methods
The main focus of the thesis is on the client-side, and the research covers the static resources management part, i.e. we are aiming to apply a number of methods that are going to affect the static resources on a web page. Important clarification about the optimization methods is that even though they influence the client-side, the majority of the methods, all but one, to be precise, are implemented on the server-side. It is the server, that helps out with the static resources, using a number of Grails resources plugins.

As it was mentioned before, the research has been focused on the static resources management. Tests were run in the following order:
- Baselining (original page prior to applying any optimization).
- Rearrange resources.
- Bundle resources.
- Bundle and minify resources.
- Bundle and compress resources.
- Bundle, minify and compress resources.
- Caching resources.

All tests ran twice: first time with empty and the second time with primed cache. Empty cache means that the cache space has nothing loaded into it yet. Primed cache is a cache that contains some data from the source location from the previous visits.

3.4 Baselining
All tests have been performed on a single heaviest page in the project that contained 14 JavaScript files, 8 CSS files and 16 images. Content breakdown is shown in the Table 3.1 below.

<table>
<thead>
<tr>
<th>MIME Type</th>
<th>Number of Requests</th>
<th>Size, Mb</th>
</tr>
</thead>
<tbody>
<tr>
<td>text/html</td>
<td>1</td>
<td>0.02</td>
</tr>
<tr>
<td>text/css</td>
<td>8</td>
<td>0.31</td>
</tr>
<tr>
<td>application/javascript</td>
<td>14</td>
<td>3.62</td>
</tr>
<tr>
<td>image/gif, image/png</td>
<td>16</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Table 3.1. Page content breakdown
Table 3.2 shows results for the original page with empty cache prior to applying any optimization methods.

<table>
<thead>
<tr>
<th>Load Time</th>
<th>Start Render Time</th>
<th>Page Size</th>
<th>Number of Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>24,29 s</td>
<td>23,59 s</td>
<td>4,0 Mb</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 3.2. Baseline metrics

3.5 Rearrange resources

Properly arranging resources on a web page impacts the time interval from the moment the browser gets the HTML document until the time the browser starts rendering the page. The reason for that is that browsers have to perform a number of steps before rendering and displaying anything. So, the rendering is blocked until the following steps are completed:

1. Receive an HTML document.
2. Receive all external CSS files, extracted from the document.
3. Receive all external JavaScript files defined inside the \texttt{<head>} tag.
4. Receive all external JavaScript files inside the \texttt{<body>} tag, that are located before the page element the browser is currently processing.

The quote below explains why all browsers do that:

\textit{If stylesheets are still loading, it is wasteful to construct the rendering tree, since you don’t want to paint anything at all until all stylesheets have been loaded and parsed. Otherwise you’ll run into a problem called FOUC (the flash of unstyled content problem), where you show content before it is ready.}

(Souders 2007, p. 43)

Because JavaScript code may change the contents of a page, browsers would stop rendering the page elements that follow the JavaScript code, until the script is loaded completely and executed. In addition to this, a special attention should be paid to inlined styles and scripts. The content of a \texttt{<style>} tag may trigger a repaint event and change the page layout considerably.

So, in order to avoid the blank white screen and prevent a browser from repainting a page, styles declarations should be moved to the top in the document’s \texttt{<head>} and scripts should go to the bottom of the page. Table 3.3 demonstrates the order of resources, in before-and-after fashion.

<table>
<thead>
<tr>
<th>Original resources placement</th>
<th>Rearranged resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{&lt;head&gt;</td>
<td>\texttt{&lt;head&gt;</td>
</tr>
<tr>
<td>\texttt{&lt;link rel=&quot;stylesheet&quot; href=&quot;core.css&quot; type=&quot;text/css&quot;&gt;}</td>
<td>\texttt{&lt;link rel=&quot;stylesheet&quot; href=&quot;core.css&quot; type=&quot;text/css&quot;&gt;}</td>
</tr>
<tr>
<td>\texttt{&lt;script type=&quot;text/javascript&quot; src=&quot;jquery.js&quot;/&gt;}</td>
<td>\texttt{&lt;link rel=&quot;stylesheet&quot; href=&quot;mobile.css&quot; type=&quot;text/css&quot;&gt;}</td>
</tr>
<tr>
<td>\texttt{&lt;script type=&quot;text/javascript&quot; src=&quot;jquery-ui.js&quot;/&gt;}</td>
<td>\texttt{&lt;link rel=&quot;stylesheet&quot; href=&quot;ext-all.css&quot; type=&quot;text/css&quot;&gt;}</td>
</tr>
<tr>
<td>\texttt{&lt;link rel=&quot;stylesheet&quot; href=&quot;mobile.css&quot; type=&quot;text/css&quot;&gt;}</td>
<td>\texttt{&lt;/head&gt;}</td>
</tr>
</tbody>
</table>
Table 3.3. The example of where resources on a page can be and should be placed

In the Table 3.4 below the measurements are summarized for after resources were rearranged. The measurements are compared with the baseline values.

<table>
<thead>
<tr>
<th></th>
<th>Load Time</th>
<th>Start Render Time</th>
<th>Page Size</th>
<th>Number of Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured</td>
<td>24.06 s</td>
<td>6.41 s</td>
<td>4.0 Mb</td>
<td>37</td>
</tr>
<tr>
<td>Improvement</td>
<td>0.23 s (1%)</td>
<td>17.18 s (73%)</td>
<td>(no change)</td>
<td>(no change)</td>
</tr>
</tbody>
</table>

Table 3.4. Rearranged resources: difference compared to the baseline values

As we can observe, after styles were moved to the top and scripts to the bottom of the page, the Start Render Time metric was significantly improved, from 23.5 seconds down to 6.4 seconds, which is almost 73% faster than the original result in the baseline. Also, there was a very small enhancement of the Page Load Time, while the Page Size and the Number of Requests, naturally, remained the same.

The location of styles and scripts did not affect the Page Load Time very much, but improved the rendering considerably. In the following methods we are going to focus on improving the Page Load Time.

3.6 Bundle resources

Combining separate files into a single file reduces the number of HTTP requests, but this does not mean that stylesheets should be combined with scripts. Multiple stylesheets across a page should be in a single stylesheet, whereas multiple scripts should be in a single script. So, what should happen in theory, if we bundle resources?

The time spent on loading a single resource via HTTP is made up of the following steps:

1. Request time $T_{\text{Request}}$ (the time spent on sending a request from the client to the server);
2. Request processing time on the server-side. We can neglect this value, since we deal with static resources, such as styles, scripts and images;
3. Response time $T_{\text{Response}}$, which is a sum of the network latency $T_{\text{Latency}}$ (the time for the first TCP header to reach the client) and the time spent on downloading the resource $T_{\text{Download}}$ (the time for the last byte in the payload to reach the client). $T_{\text{Download}}$ value is proportional to the size of the resource;
4. Resource parsing, rendering (for HTML and CSS) or execution (for JavaScript) time $T_{\text{Parse}}$.

Let us assume we have an HTML page with 3 external resources. The time for the page to load completely is a sum of time spans spent on each external resource. At the highest level of abstraction we have the following timings:

- The HTML page is requested;
- The HTML page is received;
- Resource $R_1$ is requested: $T_{\text{Request}}$;
- Resource $R_1$ is downloaded and parsed: $T_{\text{Latency}} + T_{\text{Download}}(R_1) + T_{\text{Parse}}(R_1)$;
- Resource $R_2$ is requested: $T_{\text{Request}}$;
- Resource $R_2$ is downloaded and parsed: $T_{\text{Latency}} + T_{\text{Download}}(R_2) + T_{\text{Parse}}(R_2)$;
- Resource $R_3$ is requested: $T_{\text{Request}}$;
- Resource $R_3$ is downloaded and parsed: $T_{\text{Latency}} + T_{\text{Download}}(R_3) + T_{\text{Parse}}(R_3)$.

A waterfall chart for this example is shown in Figure 3.3.

![Waterfall chart](image)

Figure 3.3. The page resources that are loaded separately and sequentially

All in all, we get the time described the formula below (assume that $T_{\text{Request}}$ and $T_{\text{Latency}}$ times are constant values for each resource):

$$3 \times (T_{\text{Request}} + T_{\text{Latency}}) + T_{\text{Download}}(R_1 + R_2 + R_3) + T_{\text{Parse}}(R_1 + R_2 + R_3)$$  \hspace{1cm} (1)

If we bundle resources then, naturally, we decrease the number of requests and the timing becomes the following:

- The HTML page is requested;
- The HTML page is received;
- Resource $(R_1+R_2+R_3)$ is requested: $T_{\text{Request}}$;
- Resource $(R_1+R_2+R_3)$ is downloaded: $T_{\text{Latency}} + T_{\text{Download}}(R_1+R_2+R_3)$;
- Resource $(R_1+R_2+R_3)$ is parsed: $T_{\text{Parse}}(R_1+R_2+R_3)$.

A waterfall chart is show in Figure 3.4.
The total time would be the following (assuming that time to render each resource individually is the same as all resources in bunch):

\[ T_{\text{Request}} + T_{\text{Latency}} + T_{\text{Download}} (R1+R2+R3) + T_{\text{Parse}} (R1+R2+R3) \]  

(2)

So the difference is \( 2 \times (T_{\text{Request}} + T_{\text{Latency}}) \) and it becomes even more significant if we had more resources. For instance, if we had 20 resources on a page, the difference would be \( 19 \times (T_{\text{Request}} + T_{\text{Latency}}) \).

What happens in reality, though?

Let us assume that we have pages P1, P2 and P3 that are loaded in a consecutive order. The page P1 uses resources A, B, C, the page P2 uses resources A, B, D and the page P3 uses A, B, E, F. Table 3.5 shows what resources a browser requests when different approaches to consolidate resources are used.

<table>
<thead>
<tr>
<th>How resources are bundled?</th>
<th>What resources are requested?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Page P1</td>
</tr>
<tr>
<td>Not bundled</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td>Single bundle</td>
<td>(A+B+C+D+E+F)</td>
</tr>
<tr>
<td>Bundle for each page</td>
<td>(A+B+C)</td>
</tr>
<tr>
<td>Bundle for each page with</td>
<td>(A+B)</td>
</tr>
<tr>
<td>core resources singled out</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 3.5. How bundles change a browser behaviour (pages P1, P2 and P3 are loaded sequentially one after another)

In case we have a single bundle, the resources for the pages P2 and P3 will be fetched from the cache, but the time spent to download resources for the page P1 is significantly increased.

If we create a separate bundle for each page, then every page with an empty cache will load slightly slower: even though the number of requests is decreased, all resources still have to be loaded. Therefore, this way we do not gain any benefits, when pages are loaded in consecutive order.
Lastly, we can identify a core, or a set of generic resources used on each page. In this case, the order in which we request pages does not matter and we benefit by using the core resources bundle \((A+B)\) from cache.

Obviously, that the last approach is the best. The more resources we can identify and single out to the core bundle, the better results can be achieved.

The bundles were created using Grails resources plugin, which addresses multiple issues:

- Correctly places resources on the page (as discussed in the chapter 3.5);
- Performs resources dependency management (when one bundle depends on another bundles);
- Loads resources in the right order;
- Unifies all processes related to resources optimization.

The Grails terminology prefers the name “module” over “bundle” to describe consolidated resources. In our test page we were able to single out several resources and create two modules called core and test. After we applied this approach, the 22 static resources on the page were combined into just 4 resources. Each module contains a bundle with CSS files and a bundle with JavaScript files. The example is shown in Table 3.6.

<table>
<thead>
<tr>
<th>Original resource file</th>
<th>Bundled resource file</th>
</tr>
</thead>
<tbody>
<tr>
<td>core.css</td>
<td>bundle-core_head.css</td>
</tr>
<tr>
<td>mobile.css</td>
<td></td>
</tr>
<tr>
<td>ext-all.css</td>
<td></td>
</tr>
<tr>
<td>login.css</td>
<td>bundle-test_head.css</td>
</tr>
<tr>
<td>teams.css</td>
<td></td>
</tr>
<tr>
<td>tender.css</td>
<td></td>
</tr>
<tr>
<td>jquery.js</td>
<td>bundle-core_defer.js</td>
</tr>
<tr>
<td>jquery-ui.js</td>
<td></td>
</tr>
<tr>
<td>ext-all.js</td>
<td></td>
</tr>
<tr>
<td>tiny_mce.js</td>
<td>bundle-test_defer.js</td>
</tr>
<tr>
<td>project.js</td>
<td></td>
</tr>
<tr>
<td>notification.js</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6. The example of resources files bundled into two modules

Before modules can be used on a web page, they should be declared using a simple DSL (Domain Specific Language) with the Groovy syntax, where the names of modules, the resources within modules and the dependencies between modules are defined. For example, the following code defines our core module:

```groovy
core {
    dependsOn 'jquery'
    resource url: '/css/style.css'
    resource url: '/js/script.js'
}
```

This module can be reused on any page using a Grails tag:

```groovy
<r:require module="core"/>
```
The measured values and the improvement compared to the baseline are shown in Table 3.7.

<table>
<thead>
<tr>
<th></th>
<th>Load Time</th>
<th>Start Render Time</th>
<th>Page Size</th>
<th>Number of Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured value</td>
<td>23.79 s</td>
<td>5.83 s</td>
<td>4.0 Mb</td>
<td>20</td>
</tr>
<tr>
<td>Improvement</td>
<td>0.49 s (2%)</td>
<td>17.76 s (75%)</td>
<td>(no change)</td>
<td>17 (46%)</td>
</tr>
</tbody>
</table>

Table 3.7. Resources bundled: savings compared to the baseline

As we can see, there is not much of a benefit that we got using this approach. Despite the Number of Requests value, which was reduced almost twice, we have a very small improvement of the Load Time. The Start Render Time value still looks good.

Furthermore, by reducing the Number of Requests we cut down back-end work load: now the server would have fewer requests to handle and therefore would have more resources to start handling the next request it receives, so that the response can be dispatched faster.

A few words about why we did not succeed with the current optimization method. There are a few misconceptions in the formula (1): it does not take into account the fact that resources are downloaded in parallel, and persistent connections between a client and a server.

The given formula assumed that the number of connections to a server equals one. The number of connections determines how many resources can be downloaded in parallel, and in the worst case (some legacy and mobile browsers (Browser scope c.2012)) a browser would open two connections per hostname. So, we might end up in a situation where a page makes fewer HTTP requests, but it takes more time to load it, as shown on Figure 3.5.

![Figure 3.5. Page load complete: separate and bundled resources loaded in parallel](image)

This is, of course, not necessarily the case for every page. A recent statistics shows that a web page in average has 84 external resources: 14 script files, 4 styles and 53 images (Httparchive 2012), so reducing the number of requests should pay off on heavy pages.

In addition to the parallel downloads, modern browsers do not necessary open a new connection for each request. When a browser sends a new request, it would use the existing connection to transfer the next resource, rather than establish a new one. This solves the problem with repeated DNS (Domain Name System) lookups (the process of looking up an IP address associated with a hostname) cost, which takes about 20-120
milliseconds. This is achieved with persistent connections, introduced in HTTP/1.1, or alternatively, using the Connection: Keep-Alive header in HTTP/1.0.

The better results could have been achieved if we had bundled the rest of the resources too. Not only stylesheets and scripts can be combined. Several images can be merged into a single image – a technique called “image sprites”. Furthermore, some images can be inlined, using a data URI scheme (URI: Uniform Resource Identifier). These techniques are left for the future work.

### 3.7 Bundle and minify resources

We have had a progress optimizing the Start Render Time metric, and now let us focus on enhancing the Page Load Time and the Page Size metrics. Bundles, that we created in the previous chapter, will be stepping stones for the further optimization methods.

Generally speaking, a page load time value is directly proportional to the size of the page, but since network expenses included in the load time are beyond the control of developers, any further optimizations should focus on the size of the page and address a question of how can we preserve functionality of the page, while delivering smaller payload to the client.

In order to reduce the amount of data that needs to be transferred, we can apply one of two methods: refactor our back-end code, or compress the data that we transfer.

The refactoring part of a web page covers the following actions:

- Optimizing the HTML document structure.
- Writing efficient CSS.
- Writing efficient JavaScript.

The data compression part includes the following actions:

- Optimize images.
- Minify scripts and stylesheets.
- Compress scripts, stylesheets.

The two latter issues are addressed in the current and the subsequent chapters. The current chapter deals with the issues of minification (and obfuscation), and the subsequent chapter focuses on scripts and stylesheets compression.

Minification is a process of removing all unnecessary characters from source code in order to reduce its size. Minification applies to HTML, CSS and JavaScript, where it eliminates redundant characters, such as whitespaces (spaces, tabs and new lines) and comments. In CSS, in addition, unneeded units of measurements, trailing semicolons and quotation marks may be stripped. In JavaScript unnecessary semicolons and block delimiters may also be removed.

Minification can also be a part of an obfuscation process. Obfuscation is an alternative optimization that can be applied to a JavaScript source code. Obfuscation usually does what minification does, but also changes the names of variables and functions and disfigures the code. Usually obfuscation is performed in order to make a source code difficult to read, understand and reverse-engineer by humans, but it also can help performance, because it reduces the size of the code in addition to what minification does. The most important rule of minification and obfuscation is that they should not change functionality.

Table 3.8 is a demonstration of how JavaScript can be minified.
### Original script before minification

(function (document, window, cookieElement) {
    var cookieDomElement = document.getElementById(cookieElement),
        cName = 'cookieTest',
        cValue = 'cookieValue',
        createCookie = function (name, value) {
            document.cookie = name + '=' + value;
        },
        readCookies = function () {
            var cookie = document.cookie;
            window.status = cookie;
            cookieDomElement.innerHTML = cookie;
            return cookie;
        };

    createCookie(cName, cValue);
    console.log(readCookies());
})(document, window, 'cookieBox');

### Script after minification

(function(a,b,c){var d=a.getElementById(c),e='cookieTest',f='cookieValue',g=function(b,c){a.cookie=b+'='+c},h=function(){var c=a.cookie;b.status=c;d.innerHTML=c;return c};g(e,f);console.log(h())})(document,window,'cookieBox')

---

**Table 3.8. JavaScript minification example**

The script “after” does not have insignificant whitespaces and semicolons. This is the result of minification. The new variable names (function arguments `document`, `window` and `cookieElement` became `a`, `b` and `c`) is the result of obfuscation. The latter is customizable and there is usually and option to omit the obfuscation step.

A number of tools are available that are aimed to achieve CSS optimization using the following techniques:

- Merge several related properties into one (background-*).
- Remove duplicated properties (.sel {width: 10px; width: 20px;}).
- Replace some values (colour codes: black is replaced by #000, units of measurement: 0px is replaced by 0).

Table 3.9 shows how CSS minification works.
As we can see, insignificant characters were removed, just like in the previous example with the JavaScript snippet. In addition, properties `background-color`, `background-image`, `background-repeat` and `background-position` were merged into a single property `background`.

There are several tools that can be used to reduce resources in size (minify and compress), and the most popular are JSMin, YUI (Yahoo! User Interface Library) Compressor and Google Closure Compiler. For our testing purposes we used `yui-minify-resources` plugin for Grails, which uses the most recent version of the YUI Compressor. This plugin runs in a combination with the `resources` plugin, described in the previous chapter.

The measured values and the improvements compared to the baseline are shown in Table 3.10.

<table>
<thead>
<tr>
<th>Load Time</th>
<th>Start Render Time</th>
<th>Page Size</th>
<th>Number of Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured value</td>
<td>14,4 s</td>
<td>5,76 s</td>
<td>2,31 Mb</td>
</tr>
<tr>
<td>Improvement</td>
<td>9,89 s (41%)</td>
<td>17,83 s (76%)</td>
<td>1,70 Mb (42%)</td>
</tr>
</tbody>
</table>

Table 3.10. Resources minified: savings compared to the baseline

The results show remarkable improvements of the Page Load Time and the Page Size values, as compared to the baseline values. Let us see how the sizes of resources have been impacted (Table 3.11).

<table>
<thead>
<tr>
<th>Content</th>
<th>Before minification</th>
<th>After minification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stylesheets</td>
<td>309 Kb</td>
<td>253 Kb (18% smaller)</td>
</tr>
<tr>
<td>Scripts</td>
<td>3,62 Mb</td>
<td>1,98 Mb (45% smaller)</td>
</tr>
</tbody>
</table>

Table 3.11. Content impacted by minification

Although, as discussed above, the basic idea behind minification is the same for CSS and JavaScript, a CSS code can not be minified as good as JavaScript, mostly because it
has less comments and whitespaces, and it is impossible to minify properties
themselves. There is no other way to make CSS constructions like "text-
decoration: underline" smaller, so CSS resources should be compact by
default.

3.8 Bundle and compress resources
Compression is another technique used to reduce the size of resources. The specification
of HTTP/1.1 standardizes three compression methods: gzip, deflate and
compress. Gzip has the widest support and is the most popular and the easiest
compression method. It is based on lossless data compression algorithm DEFLATE.

Not everything can be and should be compressed. Stylesheets, scripts, JSON, XML
and HTML documents can be compressed, while images should not be compressed at
all, because most of the image formats are already compressed: GIF (LZW lossless
algorithm), PNG (DEFLATE lossless algorithm) and JPEG (lossy).

In order to be able to work with compressed data both a server and a client (a
browser) should support gzip (or another compression method). Currently all common
servers and browsers support HTTP compression (Rack 2011). Clients indicate that they
support HTTP response in gzip format by adding the Accept-Encoding header to
the request as such:

Accept-Encoding: gzip

In turn, servers indicate that the response contains compressed data by adding the
Content-Encoding header to the response:

Content-Encoding: gzip

Resources can be gzipped using either dynamic or static compression. Dynamic,
when compression is performed on-the-fly, usually affects server performance and
introduces a little latency to each request. For instance, Apache mod_deflate module
compresses output from the server before sending it to the client (Apache Software
Foundation 2011). In this case compression expenses can be calculated using the
following formula:

\[ \text{gzip} = T_D + T_L + T_C, \]

where \( T_D \) are the file system expenses (disc input/output), \( T_L \) are the expenses on
the library initialization used for compression and other associated costs (it depends on gzip
implementation), \( T_C \) is the time spent on the actual file compression.

The zipped-resources plugin in Grails uses static compression. It compresses
resources, while deploying an application to the server. This reduces expenses on
compression, because the server would not compress resources for every request.

There are additional costs involved when working with compression on the server (to
compress resources, as discussed above) and on the client (to decompress resources).
This means that compression is a trade-off between the hardware (CPU and memory
resources), and the load time. Moreover, compressing the wrong resources may actually
take a turn to the worse. Files less than 2Kb would hardly benefit from any
compression, whereas images and PDFs should not be compressed at all (Souders
2007). A benchmark (Collin 2005) proves that in general the CPU time spent to
compress and decompress is significantly lower that the time spent to transfer an
uncompressed resource. This claim is based on results obtained from the benchmark:

1. A better-compressed file is faster to decompress;
2. Decompression performs much faster than compression;
3. It takes less than 100ms to decompress a 1.9Mb file on a low powered PC.

Table 3.12 shows the values of the page metrics after the resources compression was introduced.

<table>
<thead>
<tr>
<th></th>
<th>Load Time</th>
<th>Start Render Time</th>
<th>Page Size</th>
<th>Number of Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured value</td>
<td>6,87 s</td>
<td>1,76 s</td>
<td>0,85 Mb</td>
<td>20</td>
</tr>
<tr>
<td>Improvement</td>
<td>17,42 s (72%)</td>
<td>21,83 s (93%)</td>
<td>3,15 Mb (79%)</td>
<td>17 (46%)</td>
</tr>
</tbody>
</table>

Table 3.12. Resources compressed: savings compared to the baseline values

As we can see, the compression brought 17 seconds reduction of the Page Load Time, which is about 72% faster than the non-compressed baseline example. Table 3.13 shows how resources have been impacted by the compression.

<table>
<thead>
<tr>
<th>Content</th>
<th>Before compression</th>
<th>After compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stylesheets</td>
<td>309 Kb</td>
<td>34,6 Kb (89% smaller)</td>
</tr>
<tr>
<td>Scripts</td>
<td>3,62 Mb</td>
<td>0,72 Mb (80% smaller)</td>
</tr>
</tbody>
</table>

Table 3.13. Content impacted by compression

The compressed stylesheets and scripts reduce the Page Size from 4Mb down to 850Kb, a reduction by 79%. The best results so far, but let us see what we can do if we combine the two: minification and compression.

### 3.9 Bundle, minify and compress resources

Because Grails plugins, that have been used before, proved to work very well together, it is worth to try them together. Table 3.14 shows what we achieved with both minification and compression.

<table>
<thead>
<tr>
<th></th>
<th>Load Time</th>
<th>Start Render Time</th>
<th>Page Size</th>
<th>Number of Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured value</td>
<td>5,33 s</td>
<td>1,66 s</td>
<td>0,63 Mb</td>
<td>20</td>
</tr>
<tr>
<td>Improvement</td>
<td>18,96 s (78%)</td>
<td>21,92 s (93%)</td>
<td>3,37 Mb (84%)</td>
<td>17 (46%)</td>
</tr>
</tbody>
</table>

Table 3.14. Resources minified and compressed: savings compared to the baseline

As we can see, it enhanced Load Time and Page Size even more. Table 3.15 shows which method worked the best and reduced the resources in size the most.

<table>
<thead>
<tr>
<th>Content</th>
<th>Original</th>
<th>Minified</th>
<th>Gzipped</th>
<th>Minified + Gzipped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stylesheets</td>
<td>309 Kb</td>
<td>253 Kb (18%)</td>
<td>34,6 Kb (89%)</td>
<td>30,36 Kb (90%)</td>
</tr>
<tr>
<td>Scripts</td>
<td>3,62 Mb</td>
<td>1,98 Mb (45%)</td>
<td>0,72 Mb (80%)</td>
<td>0,51 Mb (86%)</td>
</tr>
</tbody>
</table>

Table 3.15. Content impacted by different compression techniques
Turns out, the most effective methods are compression and a combination of minification and compression. Minification saved additional 6% for scripts and 1% for stylesheets over compression alone. All in all, together minification and compression reduced stylesheets in size by 278Kb and scripts by 3,11Mb, which is 90% and 86% smaller respectively.

3.10 Cache resources

Browser’s caching capabilities is the best opportunity to improve performance. This is similar to the RAM (Random Access Memory) on a personal computer, that can be used as a space for application data, because it can be accessed faster compared to the hard drive.

Cache itself is a space for any arbitrary data that can stored on the client. To cache a page means that most of the page resources will be kept in a special location so that they can be accessed during the subsequent page views. A browser stores a local copy of a resource, making it faster to access the next time, instead of transferring it over the network again. This reduces the network usage, server load and the page load time. All resources are eligible for caching but it is usually applied to CSS files, JavaScript files and images, i.e. resources, that do not change very often.

A first-time visitor to a page arrives with an empty cache. On subsequent visits there will be primed cache (depending on how page resources are served).

Browsers by default would store some resources in cache after a page is loaded. But HTTP has three principal mechanisms for controlling a cache: freshness, validation and invalidation.

Freshness (or expiration-based caching) allows a cached resource to be used without its rechecking on the original server. In order to use this mechanism the server should supply a response with the Expires or the Cache-Control header. The Expires header holds a date after which the resource is considered stale, for example:

```text
Expires: Wed, 15 May 2013 18:29:06 GMT
```

The Expires header uses a specific date and thus requires synchronized clocks on the server and the client. The alternative is the max-age directive within the Cache-Control header, which overrides the Expires header. When this directive is given, the cached resource is considered stale if its age is greater than the age value obtained with a new request. The following example sets the age to 31536000 seconds (i.e. 365 days):

```text
Cache-Control:public, s-maxage=31536000, max-age=31536000
```

Validation (or validation-based caching) is used to check whether a cached resource has been modified since the time specified in the If-Modified-Since header. The server will return either a modified resource, or the 304 (Not Modified) response.

Invalidation implies that the cached resource is considered as fresh (eligible to be reused) until a non-GET request to the URL (Uniform Resource Locator) associated with this resource is performed, which triggers the resource invalidation. This method is not suited for static resources, like CSS or JavaScript unless they are requested using POST, PUT or similar method.

Grails cached-resources plugin, which has been used for our test, uses expiration-based caching and adds both Expires and Cache-Control headers set to a year in advance from the time of the request.
In this case, when a resource is not going to be validated for a year, it is crucial to ensure that the client always uses a resource which is up-to-date. A simple way to ensure that is to update the name of the resource. This is typically done using versioning, by appending a timestamp to the name of the resource: resource.js becomes resource_15052012.js and this change should be performed on every page the resource is used.

The cached-resources plugin uses another approach. It hashes files and renames gives them the name of their own hash value: when the resource content is altered, the resource name changes automatically. There is no need to change the resource name on pages manually, as the plugin will fix everything automatically.

The following tests were performed with primed cache: Test 1: Baseline, Test 2: Resources reordered, Test 3: Resources bundled, Test 4: Resources bundled and minified, Test 5: Resources bundled and compressed, Test 6: Resources bundled, minified and compressed.

Test 1 and Test 2 did not use the cached-resources plugin and used the same browser-default cache settings instead.

Tests 3-6 used the cached-resources plugin and have exactly the same values for the Page Size and the Number of Requests, because the same HTTP headers have been used:

Expires: Wed, 15 May 2013 04:34:26 GMT
Cache-Control: public, s-maxage=31536000, max-age=31536000

Table 3.16 shows measurements with primed cache; Tests 3, 4, 5, 6 are combined. Values in the parentheses indicate savings compared to Test 1.

<table>
<thead>
<tr>
<th>Test name</th>
<th>Load Time</th>
<th>Start Render Time</th>
<th>Page Size</th>
<th>Number of Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1: Baseline</td>
<td>2,53 s</td>
<td>1,81 s</td>
<td>82Kb</td>
<td>37</td>
</tr>
<tr>
<td>Test 2: Reorder</td>
<td>2,32 s (8%)</td>
<td>1,16 s (36%)</td>
<td>82Kb (0%)</td>
<td>37</td>
</tr>
<tr>
<td>Tests 3-6</td>
<td>1,26 s (50%)</td>
<td>0,57 s (68%)</td>
<td>19Kb (77%)</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.16. Measurements with primed cache

Tests 3-6 show a benefit as a result of custom caching over the browser-default mechanism. The page with resources cached makes just one request to the server, brings the Load Time reduction of 50% and the Page Size reduction of 77%.

Caching could be a challenge on handheld devices, because memory is a critical resource there: mobile browsers allocate a very small space for cache as compared to desktop browsers.

Though cache certainly brings benefits for repeated views, the first-time visitors still have to wait until all necessary resources are loaded.
4. Discussion

This chapter discusses the results we obtained in the previous chapter and makes up a conclusion about the effectiveness of the optimization that we used and how well they affect different metrics.

4.1 Page Load Time metric

Figure 4.1 and Figure 4.2 show how different optimization methods affected the Page Load Time metric.

![Figure 4.1. Page Load Time improvement](image1)

![Figure 4.2. Methods contribution to the Page Load Time improvement](image2)
The first figure (Figure 4.1) clearly shows us the importance of the resources compression. It is incredible how this method alone can speed up the page load time. Though some web developers are still very cautious about this method, because some browsers might not handle HTTP responses that contain compressed data correctly. In the worst case scenario, the page would look terribly (because CSS rules from external resources would not be available) and, the page would not function as expected (because scripts from external resources would not be available), unless, the progressive enhancement approach is used, which would allow users interact with the page and perform some actions.

The picture that follows (Figure 4.2) shows how different methods contribute to the reduction of the Page Load Time metric. The first number stands for the improvement (by how many percent did the value progress compared to the baseline value), and the second number is a contribution, or a weight, of every optimization method. Looking at the chart it can be spotted that the compression together with minification brings only a slight improvement (by 4%, 41% versus 37%) compared to the compression alone.

Tests that were performed with prime cache show that pages can load with lightning speed no matter if there are any optimizations applied or not. It does not justify the almost 25 seconds page load time with empty cache though. Obviously, a website, where it takes half a minute to load a page completely, does not stand a chance. Nobody would ever wait for so long, unless this time is spent working with big data, which would not, of course, affect the load time of the static resources anyway.

4.2 Page Size metric

The following figures (Figure 4.3 and Figure 4.4) show how the Page Size metric was improved and the share of every method in this improvement.

![Figure 4.3. Page Size improvement](image)
Previously, when we were looking at the page load time improvement we noticed that the order of resources on a page, and even bundling and serving several files as one file, did not improve the page load time. Now we can see that it did not improve the page size either. This can be explained by the fact that it simply takes time to transfer the payload that we had (around 4Mb). This is why we saw both metrics greatly improved after the minification and compression were applied. Therefore we can conclude that the Page Load Time and the Page Size metrics are closely related, and even that the Page Load Time is directly proportional to the Page Size.

Once again, the numbers on both Figure 4.3 and Figure 4.4 show how little the last method, the compression and minification together affected the page, compared to the compression alone. As discussed previously, minification through obfuscation is not always a safe operation. The primary concern about it is that there is a risk, that obfuscated code may change functionality and return incorrect data. Therefore we would suggest that if there is a possibility that the obfuscated code might lead to incorrect results the method should be ruled out.

4.3 Number of Requests and Start Render Time metrics

When looking at the numbers for the two less significant metrics, which the Number of Requests and the Start Render Time are, we can spot that there was hardly any significant difference whenever a new optimization method was applied.

As for the Number of Requests, we classified the page resources into two modules first, and then these two modules were used as a part of the rest of the optimization methods (compression and minification), which is why we had the same numbers.

It was very similar with the Start Render Time metric. Obviously, it was a huge leap forward when we reordered resources and placed styles and scripts in the right locations. Figure 4.5 shows the numbers.
Although the improvements we made after we condensed resources appear to be very impressive, they are very much related to the page size and the page load time numbers, because the moment a browser starts rendering the page, heavily depends on how quickly the external resources (styles) are transferred from the server. This, in fact, means, how many factors the compression alone affects and how important the Page Load Time and the Page Size metrics are.

4.4 Optimization methods applicability

In the end of the discussion the general applicability of the optimization techniques must be mentioned.

In general, the methods that we used, are applicable to the majority of web applications. We had the luck with to optimize our project fairly well. The waterfall graph pointed out that the responsiveness was hampered by the static resources the most: after all, it only took 1.4 seconds to execute the server-side actions and transfer the HTML response. The rest was spent downloading static resources. In our case the static resources were the low hanging fruits, as we had a whole lot of scripts and styles to download.

These methods would not always be beneficial, though. Certain things may simply need some time to carry out. For instance, complex algorithms and working with big data is likely to take more time than usual. This is not going to create any problems with the static resources and therefore the optimization would have to be applied to the back-end-related parts.
5. Conclusion and future work

This chapter ends the thesis, concludes the work that was done and contains a discussion on the future work.

In the opening chapter the problem was formulated as well as a number of questions the thesis was set to get answers to. The questions were as follows. Why is the front-end slow? What kind of front-end optimization methods are used now? Will our application perform better when these methods are applied?

During the work we tried to shed some light on why web pages are slow and what are the culprits. Besides, the world of front-end optimization techniques was studied. We looked at what usually should be measured to understand that a web page is underperforming. We discussed the benefits and drawbacks optimization techniques could bring.

Some of the methods were applied to the project our team is working on and we made a number of tests and measurements. As mentioned in the Chapter 3.1, the tests might look rather synthetic because of the conditions and assumptions that we created for our test environment. This is because the tests do not fully represent production environment, which could be much more different.

Based on the empirical analysis we revealed that the minification and compression in combination enhance performance most among the methods that we examined. The compression alone has the best impact on the page size metric. The resource caching clearly brings dramatic improvements (for repeated views) and it is desirable that a caching is thoughtfully applied to web pages throughout the Internet (with certain exceptions). Despite that, in order to make a positive first time user experience to unfamiliar visitors, it is very important to optimize pages for when cache is empty, or otherwise cache will never be full.

Therefore, all the questions of the thesis were answered.

Only some of the front-end optimization techniques were studied. That is why in the future we may want to address more optimization methods. Not only the engineering aspects of the front-end optimization should be explored. The scope of optimization techniques can be extended to some psychological aspects as well. Psychological tricks can make a page, that is not very fast, feel fast. A simple example of such a psychological trick is when a user requests a time-consuming task, then it is crucial to maintain a set of psychological rules, for instance, to provide the user with a feedback, keep the user informed of what is going on and display a progress indicator bar.

To sum up this thesis, I would say that the process of studying the web page performance optimization was a great experience for me. I want to mention the most important thing I learned is that there should not be a goal to optimize every single piece of code. Sometimes it is impossible to achieve responsiveness due to the nature of a web application. And, of course, even though the front-end optimization is critical, this does not mean that the back-end performance should be totally ignored. In order to achieve the best performance, the combination of both back-end and front-end techniques must be used.
6. References


