Optimising Web Rendering Performance

Group 3 Christian Aluani, Herbert Fuchs, Alexander Schwaiger, and Peter Schögler

706.057 Information Visualisation SS 2016
Graz University of Technology

22 Nov 2016

Abstract

As technology evolves and the world moves faster, the need for speed and performance of web pages rises dramatically. Loading times and responsiveness of web services (even with a slow connection) are key factors for a satisfying user experience (UX). This properties can be enhanced by optimising rendering times in the web browser to prevent high rebound-rates of users.

This survey covers the most essential parts of rendering performance optimisation, but does not include all methods that are related to reduce the download time and network overhead of resources. In the first section the Critical Rendering Path (CRP) is explained in detail, to provide a better insight into the rendering process of modern web browsers. Additionally some basic tools and methods to measure and analyze performance of web pages and their most important use cases are introduced. The last chapter covers concrete methods to improve CSS and Javascript files to minimize their impact on the loading times.

Finally a Case Study puts all the presented theoretical aspects into a realistic setting and the actual results are recorded and compared to each other.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

What is the Critical Rendering Path?

The critical rendering path is a sequence of steps to convert HTML, CSS and JavaScript to pixels on the screen. If this path gets optimised, the page-speed will be increasing in a enormous way and the user will have a better experience when the web page becomes visited. To obtain a basic understanding of the context of this sequence, every step will be discussed in detail in the next sections.

1.1 Overview

As already mentioned the critical rendering path is needed to display pixels on the screen. Therefore the Document Object Model and the CSS Model have to be constructed. After that, these two models have to get combined to a render tree. Afterwords the Layout step calculates the sizes and alignments of the render tree objects. In the last step the browser paints pixel by pixel onto the screen. This is shown in Figure 1.1. [*Designing for Performance: Weighing Aesthetics and Speed*]

1.2 Constructing the Document Object Model

To construct a Document Object Model (DOM), the browser has to read the raw-bytes from the HTML file. Then these bytes get converted into characters, depending on the encoding type of the file. After that, these characters get tokenized, which means, that the characters will be converted into HTML-Tags for example like <body>, <div>or <p>. Afterwards these tags are translated into nodes. In a usual webpage, there are three different types of nodes, which are the element-node, the attribute-node and the text-node. In the last step these objects are getting combined with each other, depending on their relationships, so that a tree structure will be created. This steps are shown in figure 1.2.
After the construction, the browser is not able to display the webpage, because there are only the relations between the objects defined. That's why the CSS object model is needed. [*DOM Scripting: Web Design with JavaScript and the Document Object Model*]

1.3 From CSS Code to the CSS Object Model

The CSS Object Model (CSSOM) defines how the objects should look on the display. The construction of such a model is similar to the DOM. The browser reads the raw-bytes from the CSS-file and convert this bytes to characters, depending on the encoding type. In the following step these characters are translated into tokens, then into nodes and in the last step, these objects getting combined to the CSSOM.

As seen in figure 1.3, each child inherits the attributes of the parent object. The browser starts with the root node and gives all objects these attributes. Afterwords the browser goes recursively step by step deeper into the tree structure to style every single object. [*Constructing Model Objects*]

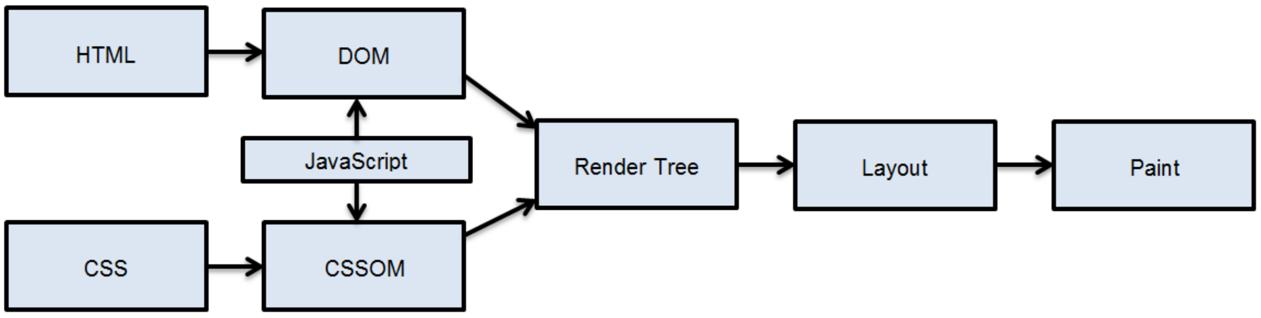
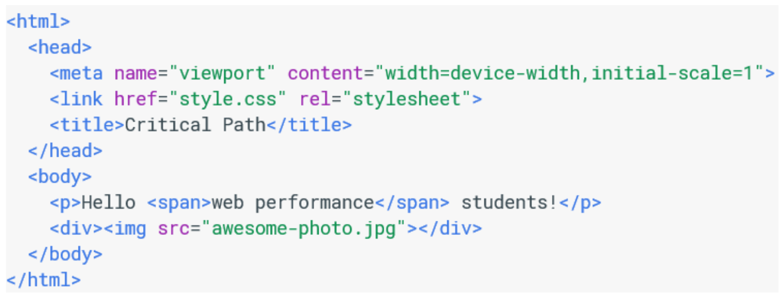
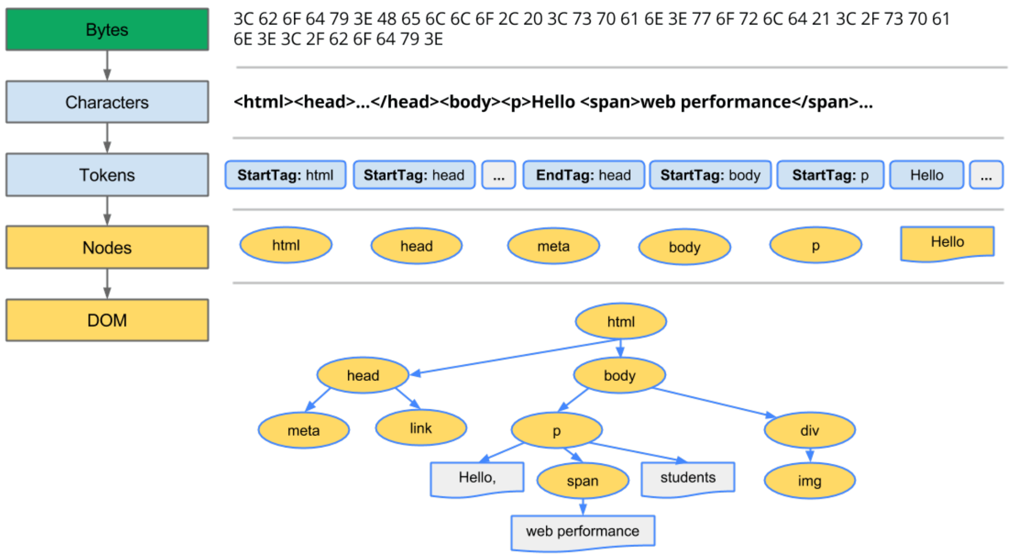


Figure 1.1: Overview of the critical rendering path
Redrawn from Grigorik [*Deciphering the Critical Rendering Path*]



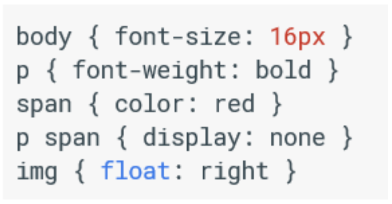
```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```

(a) A simple HTML source code to demonstrate the construction of a Document Object Model.



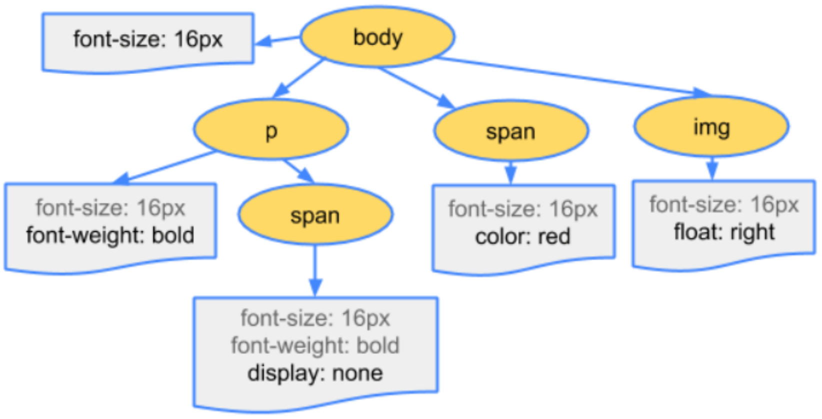
(b) The five construction steps, which a browser needs to construct a Document Object Model from a receiving HMTL file.

Figure 1.2: Schematic representation of the construction steps of a Document Object Model. (a) shows the HTML source code. (b) shows the construction steps.
Image extracted from Grigorik [*Constructing Model Objects*] - Creative Commons Attribute 3.0 (CC BY 3.0) licence



```
body { font-size: 16px }
p { font-weight: bold }
span { color: red }
p span { display: none }
img { float: right }
```

(a) A simple CSS source code to demonstrate the construction of a CSS Object Model.



(b) A simple CSS Object Model.

Figure 1.3: Schematic representation the relation between the source code and the CSS Object Model. (a) shows the CSS source code. (b) shows the CSS Object Model.
Image extracted from Grigorik [*Constructing Model Objects*] - Creative Commons Attribute 3.0 (CC BY 3.0) licence

1.4 Combine Object Models to Render Tree

In the third step of the critical rendering path, the Document Object Model and the CSS Object Model are getting combined to a render tree. This render tree is then used for the calculations in the layout step. This tree structure only includes objects, which are really necessary for rendering. So the browser starts with the root node of the DOM. Then every object gets checked, whether it is visible. As seen in figure 1.4 the meta and the link objects are not visible on the screen, that's why they will not be in the render tree. If it is a visible object, the attributes of that will be checked in the CSSOM and then gets into the render tree. There is one exception. If the object has the attribute display:none, it will not be getting into the render tree. [*Render-tree Construction, Layout, and Paint*]

1.5 The Layout Stage or Reflow

This section und its subsections are based on the Work of Ilya Grigorik [*Render-tree Construction, Layout, and Paint*] and Smith [*Professional Website Performance: Optimizing the Front-End and Back-End*].

1.5.1 The Render Tree as Base for the Layout

As illustrated in the previous section, the render tree is constructed from the Document Object Model and the CSS Object Model by combining HTML nodes with their associated styles. All nodes that are not visible in the view-port because of their styles are then omitted from the render tree. "Not visible" in the context of 'Layout' means that a node does not take up space on the screen - for example nodes with "display: none" are not used, but elements with "visibility: hidden" are. The resulting structure can now be used for the next step in browser rendering - the layout.
At this point the browser has information about visible nodes and their computed styles, but the rendering tree does not give away any clues on the element sizes or the exact alignment on the viewport, which is needed to paint the pixels onto the screen. Right here is were the layout process steps in.

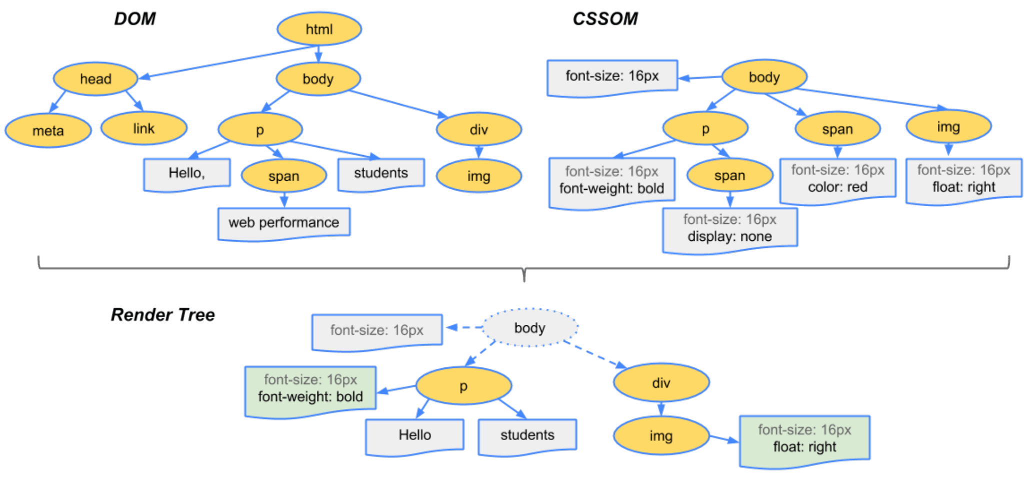


Figure 1.4: Schematic representation of a render tree creation from a CSSOM and a DOM. Image extracted from Grigorik [*Render-tree Construction, Layout, and Paint*] - Creative Commons Attribute 3.0 (CC BY 3.0) licence

1.5.2 A Simple Layout Example

Consider a simple web page with two nested elements inside the HTML Body. Both have inline-styles setting their respective width to 50%.

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Critial Path: Layout Example</title>
  </head>
  <body>
    <div class="parent" style="width: 50%">
      <div class="child" style="width: 50%">Hello world!</div>
    </div>
  </body>
</html>
```

Listing 1.1: Simple HTML Layout Example

In the code snippet above there is one line in the header that is crucial for the layout process to work as expected. The tag "`<meta name="viewport" content="width=device-width">`" tells the browser, that the screen-width of the current device should be used as base for the viewport. If this option is not set, the computed relative node styles for width refer to a standardized value (in most cases 880px), which can lead to unexpected behavior.
With the "meta-viewport" option enabled, the body consumes 100% of the device width and height, even without any explicit styling.

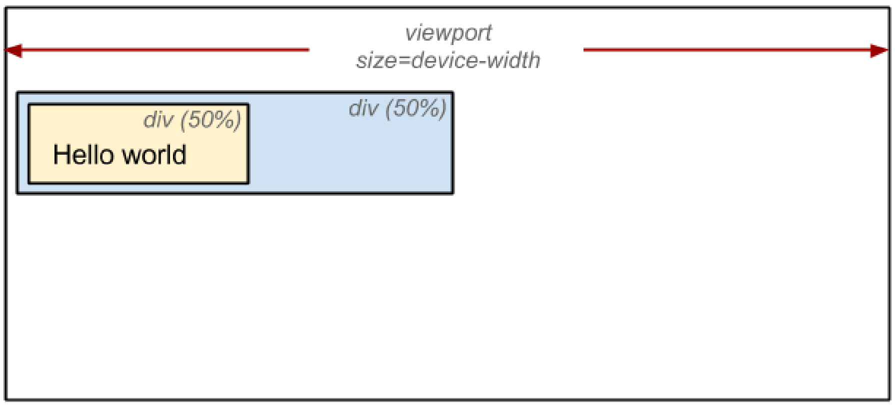


Figure 1.5: Box Model for layout of the simple example
Image extracted from Grigorik [*Render-tree Construction, Layout, and Paint*]
Creative Commons Attribute 3.0 (CC BY 3.0) license

The simple example uses two nested containers inside the body. The `"width:50%"` of the `<div class="parent">` is relative to the body and therefore also to the viewport-size. In contrast to that, the style for the inner `<div class="child">` is interpreted as relative to the "parent" container, which means that is assigned half of 50% of the viewport, that is 25Ḃecause there is no floating specified, all elements will be automatically aligned on the left-hand side. The ouput of this whole layout step is called "Box Model", which is illustrated in Figure 1.5. To be actually able to use this model for painting, the browser needs to convert the relative sizes to absolute pixel values.

1.5.3 Bottleneck for Rendering Peformance: Layout Trashing

As in every step of the critical rendering path, there are some potential bottlenecks concerning the layout step, that can slow down the whole browser rendering process. To detect this possible 'stoppers' it is important to evaluate all types of triggers that force the execution of the layout process. This subsection is based on the work of Wilson Page [*Preventing 'Layout Trashing'*].

Layout is executed when...

- the rendering tree is initially created (first page load)
- the viewport-size changes (window resize or orientation change)
- **state of a node is changed via JavaScript (layout invalidation)**
  - node is added
  - node is removed
  - the visibility in the viewport (display) of a node changes
  - attributes of a node are overwritten

The initial render tree generation on a page load and a change of the viewport size can not be prevented and their effects can not be omitted using rendering optimization techniques.
As opposed to this, changes to the state of a node with JavaScript can be controlled by a software engineer and is therefore subject to performance optimization. If badly written JavaScript code "violently writes, then reads from the DOM multiple times causing document reflows", it produces ***"Layout Trashing"***. This phenomenon is demonstrated in Listing 1.2.

```
// Read
var h1 = element1.clientHeight;

// Write (invalidates layout)
element1.style.height = (h1 * 2) + 'px';

// Read (triggers layout)
var h2 = element2.clientHeight;

// Write (invalidates layout)
element2.style.height = (h2 * 2) + 'px';

// Read (triggers layout)
var h3 = element3.clientHeight;

// Write (invalidates layout)
element3.style.height = (h3 * 2) + 'px';
```

Listing 1.2: Layout Trashing Example

When the size of a node is changed by JavaScript code (in Code 1.2 the height of 3 nodes is changed), the whole layout gets invalidated and needs to be recalculated (document reflow). This operation would normally take place after the current frame completes, which is after the code is completely executed. If the overwritten value is used in the same frame (with a read) the layout recalculation is triggered immediately. This slows down the whole process since code is render blocking, leading to Layout Trashing.
A solution to prevent Layout Trashing and still be able to keep the structure of the code logically in place is demonstrated in chapter 3.3.6

1.6 Painting the layout to the screen

This section and subsections are based on the work of Colin McAnlis [*CSS Paint Times*] and Ilya Grigorik [*Render-tree Construction, Layout, and Paint*].
After the render tree is transformed to the layout box-model and all the relative size measures are recalculated to absolute pixel values, the last step of the browser rendering is spot-on: The painting. In this process the actual content is "printed" to the viewport by the browser, using a "layer model".

The layer model consists of the following steps:

- Split DOM into layers
- Paint each layer into software bitmaps
- Convert bitmaps into textures and load them to the GPU
- Combine all layers to final screen image

Every time the layout of a page has to be recalculated because a document reflow is triggered, the browser also needs to repaint the pixels on the screen. This recurring actions can be categorized as **paint cycles**. After taking a closer look at the behaviour, a clear implication between layout calculations and paint cycles can be observed. Using this dependency, minimizing the need to reflow a document also optimizes the frequency of paint cycle executions. It is advisable to reduce paint calls, since the web browser needs to repaint all layers

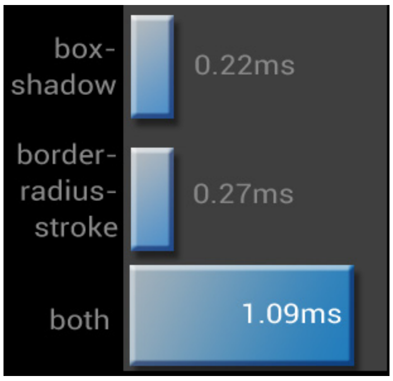


Figure 1.6: Comparison of the impact on paint times by three different CSS styling specifications.
Image extracted from McAnlis [*CSS Paint Times*]
Creative Commons Attribute 3.0 (CC BY 3.0) license

that are changed in the current animation frame. Keeping this in mind, the performance loss of the whole web page is likely to grow exponentially with its node count.
Another important aspect are the actual paint times for a specific layer. Different CSS styling attributes for a DOM node, can lead to great differences in how long the browser needs to paint the whole element. This phenomenon can be observed best, looking at a simple example.

```
// Only Border Radius
.example-object {
  border-radius: 50%;
}
// Only box-shadow
.example-object {
  box-shadow: 1px 2px 3px 4px black;
}
//Both
.example-object {
  border-radius: 50%;
  box-shadow: 1px 2px 3px 4px black;
}
```

Listing 1.3: CSS Paint Times Example

If each styling definition is applied to the same node and their paint times are compared the following inside can be obtained:
After breaking down different results, it is obvious that distinct CSS attribute combinations result in higher paint times (see Chapter 4 for further observations).

Chapter 2

Measuring and Analyzing Browser Rendering Performance

2.1 Measuring Rendering Performance

One of the most important things to maintain the user experience is to measure the rendering performance and to target specific benchmarks.

2.1.1 The RAIL Model

This section and is based on the Google Developers Guide [*Measure Performance with the RAIL Model*]. To get some references for user specific timing benchmarks, the RAIL Model can be used. The RAIL Model is a user-centric performance model which defines 4 essential aspects:

Response

One of the most important point for users is the response of a user action. If there is too much time between action and reaction, the user won't continue and may leave the website or Web-App. There are specified response time intervals influencing the user experience:

- 0-100ms
  - optimal time to respond on user interaction
  - user won't recognize lags
- 100-300ms
  - acceptable for most users but they will notice a lag
- 300-1000ms
  - feels like a page reload for the user
- `>1000ms`
  - user loses focus on their actual tasks

Animations

Animations are high pressure points. The aim is render frames every 16ms (60 frames per second). This is especially important for scrolling and touch moves on mobile devices.

Response
Animation
Idle
Load

Figure 2.1: Schematic representation of the RAIL model. Image extracted from Google [https://developers.google.com/web/fundamentals/performance/rail] - Creative Commons Attribute 3.0 (CC BY 3.0) licence

Idle

Use idle time to load additional datasets and keep preloaded data very small to keep the response time under 100ms.

Load

A loading time longer than 1 second tempts the user to leave the page or makes the user assume that the request is broken.

2.1.2 Chrome DevTool

Chrome's DevTool is a very powerful analyzer tool to measure performance, timing and user interactions. There are three main features which are used in this case study:

Network

Chrome supports a complete image of start and end time of loading, XHR times and loaded file sizes.

Timeline

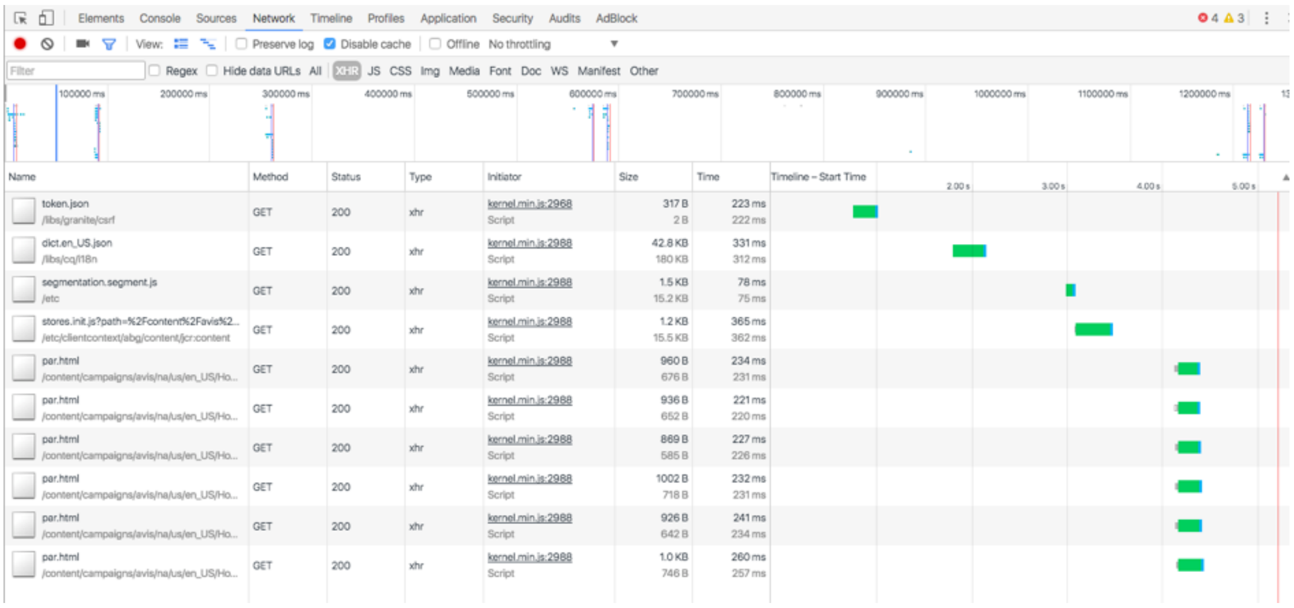
In the timeline tab it's possible to record specific actions and measure response time of user interactions or animation time. It's used to analyze JavaScript and CSS optimizations in the case study.

Recording tips:

- Keep recording short to get a suitable timeline
- Disable browser caching
- Disable Chrome extensions

Profile

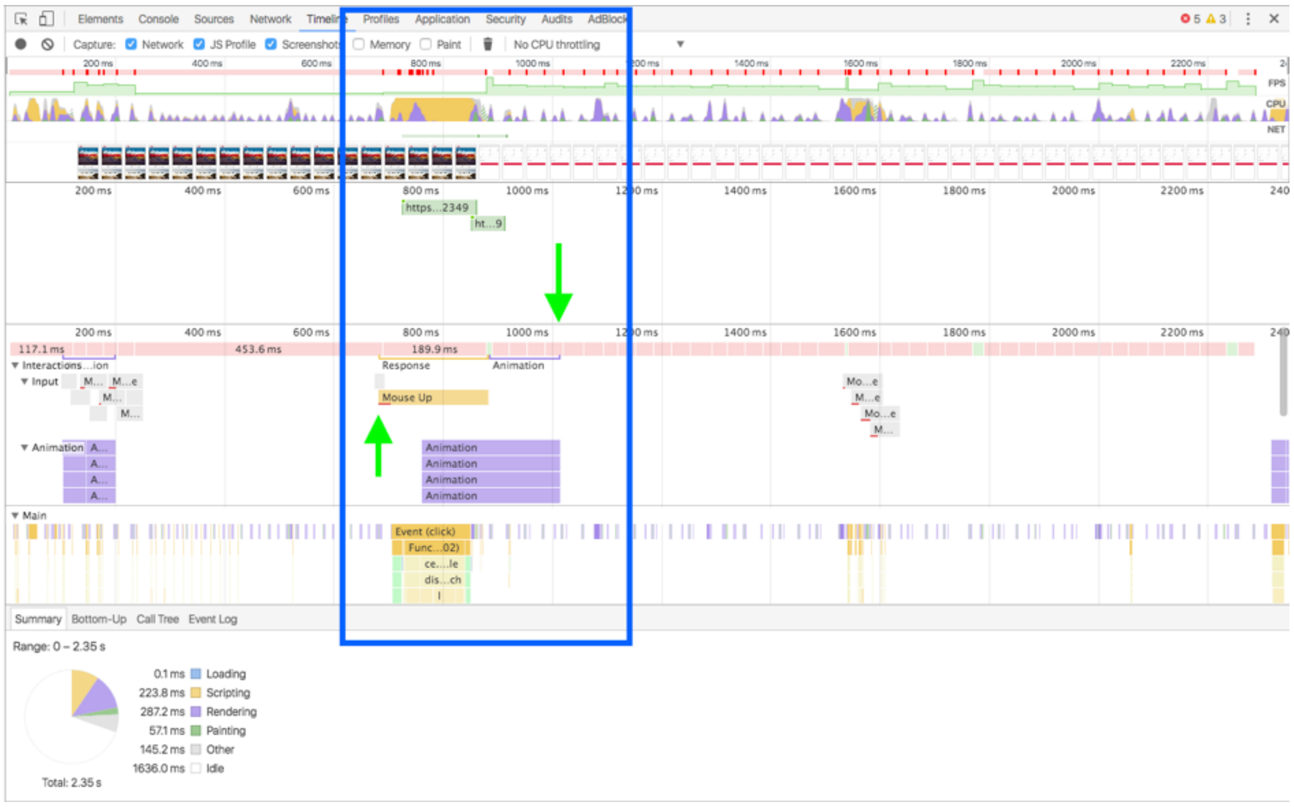
Records the JavaScript CPU profile and verify expensive JavaScript Files.



Preserve log | ☑ Disable cache | Offline | No throttling

Name	Method	Status	Type	Initiator	Size	Time
token.json /libs/granite/csrf	GET	200	xhr	kernel.min.js:2968 Script	317 B 2 B	223 ms 222 ms
dict.en_US.json /libs/cq/i18n	GET	200	xhr	kernel.min.js:2988 Script	42.8 KB 180 KB	331 ms 312 ms
segmentation.segment.js /etc	GET	200	xhr	kernel.min.js:2988 Script	1.5 KB 15.2 KB	78 ms 75 ms
stores.init.js?path=%2Fcontent%2Favis%2... /etc/clientcontext/abg/content/jcr:content	GET	200	xhr	kernel.min.js:2988 Script	1.2 KB 15.5 KB	365 ms 362 ms
par.html /content/campaigns/avis/na/us/en_US/Ho...	GET	200	xhr	kernel.min.js:2988 Script	960 B 676 B	234 ms 231 ms
par.html /content/campaigns/avis/na/us/en_US/Ho...	GET	200	xhr	kernel.min.js:2988 Script	936 B 652 B	221 ms 220 ms
par.html /content/campaigns/avis/na/us/en_US/Ho...	GET	200	xhr	kernel.min.js:2988 Script	869 B 585 B	227 ms 226 ms
par.html /content/campaigns/avis/na/us/en_US/Ho...	GET	200	xhr	kernel.min.js:2988 Script	1002 B 718 B	232 ms 231 ms
par.html /content/campaigns/avis/na/us/en_US/Ho...	GET	200	xhr	kernel.min.js:2988 Script	926 B 642 B	241 ms 234 ms
par.html /content/campaigns/avis/na/us/en_US/Ho...	GET	200	xhr	kernel.min.js:2988 Script	1.0 KB 746 B	260 ms 257 ms

Figure 2.2: Google Chrome DevTool Network example. Recorded the loading of https://www.avis.com/en/home.



Range: 0 – 2.35 s

0.1 ms	Loading
223.8 ms	Scripting
287.2 ms	Rendering
57.1 ms	Painting
145.2 ms	Other
1636.0 ms	Idle

Total: 2.35 s

Figure 2.3: Google Chrome DevTool Timeline example. Recoreded the button click "Select my Car" on https://www.avis.com/en/home. The green arrows point at the start of request and the end of response.

2.1.3 PageSpeed Insights

Google PageSpeed Insights analyses the performance of a website regarding to mobile devices and desktop computers.

The tool analyzes:

- Server Configuration
- HTML Structure
- External Resources
  - Images
  - JavaScript Files
  - CSS Files

The analysis results in a site rating separated in Speed and User-Experience. A suggestion summary shows weak points and links to explanations how to fix the issues. For details see: https://developers.google.com/speed/pagespeed/insights/?h

2.2 Analyzing The Rendering Performance

This section is based on the work of Ilya Grigorik [*Analyzing Critical Rendering Path Performance*].

In chapter 1 the theoretical aspects of the Critical Rendering Path (CRP) and its individual components are introduced to enforce a basic knowledge for the inner workings of a web browser. Additionally, tools to measure, capture and compare rendering performance are covered in section 2.1 of this chapter. In the following section the interaction of the CRP components and their individual impacts on the rendering performance are analyzed and illustrated using the HTML example below.

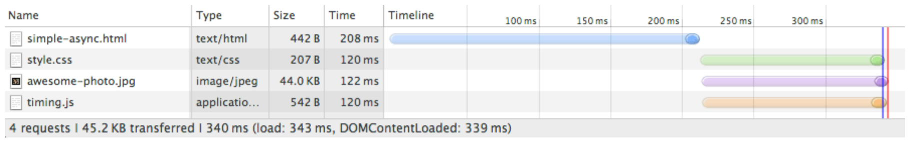
```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js"></script>
  </body>
</html>
```

Listing 2.1: HTML Analyzing Example

2.2.1 Unoptimised Browser Rendering

The example page in Listing 2.1 consists of a simple HTML markup, an external image and external CSS and JS File. Using the Google DevTools timeline to inspect the CRP of this sample page shows some interesting results.

The "DOM Loaded" Event fires after 339ms, which is very long considering the small size of the page. Images do not block the rendering, they are loaded asynchronously. In contrast to that, the external CSS file blocks the page rendering, the JS blocks the parsing of the DOM. Following this observations the CSS and JS



Name	Type	Size	Time	Timeline
simple-async.html	text/html	442 B	208 ms	
style.css	text/css	207 B	120 ms	
awesome-photo.jpg	image/jpeg	44.0 KB	122 ms	
timing.js	applicatio...	542 B	120 ms	

4 requests I 45.2 KB transferred I 340 ms (load: 343 ms, DOMContentLoaded: 339 ms)

Figure 2.4: Google DevTools timeline output for unoptimised CRP
Image extracted from Grigorik [*Analyzing Critical Rendering Path Performance*]
Creative Commons Attribute 3.0 (CC BY 3.0) license

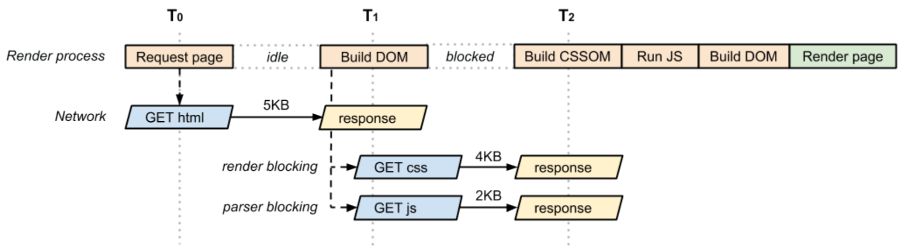


Figure 2.5: Waterfall model for unoptimised CRP
Image extracted from Grigorik [*Analyzing Critical Rendering Path Performance*]
Creative Commons Attribute 3.0 (CC BY 3.0) license

files can be described as **critical resources**, because they can slow down the initial rendering of the page. In addition the HTML file itself is always a critical resource, resulting in a total of three. The **critical path length** (number of roundtrips needed to fetch all critical resources) for this model is **two or more** and there are at least **11kb** of **critical bytes** to load for the current state.
If this whole process and its dependencies is illustrated as waterfall model it would look like Figure 2.5.

2.2.2 Optimised Browser Rendering

To get a better user experience it is crucial to optimize the whole CRP by minimizing the critical resources, the critical path length as well as the number of critical bytes. In chapters 3.1 and 4 all necessary steps to reach is goal are described and explained with code samples. This section only provides an overview of the final impact on the total performance, if the CRP is optimised.
If the example from listing 2.1 is re-evaluated and the loading process of both CSS and JavaScript files is modified by making them non-render and non-parse blocking, the waterfall model changes accordingly.
The number of critical resources is reduced to only one (the HTML file), the total critical path length is exactly one round-trip (from T0 to T1) and the number of critical bytes is reduced to 5kb. This changes lead to a improved rendering time by approximately 120-130ms for a very simple page. The DOMLoaded() event fires now after 212ms instead of 339ms before the optimisations. Applying this pattern to more complex pages, can lead to an even bigger gain of rendering speed and drastically enhance user experience.

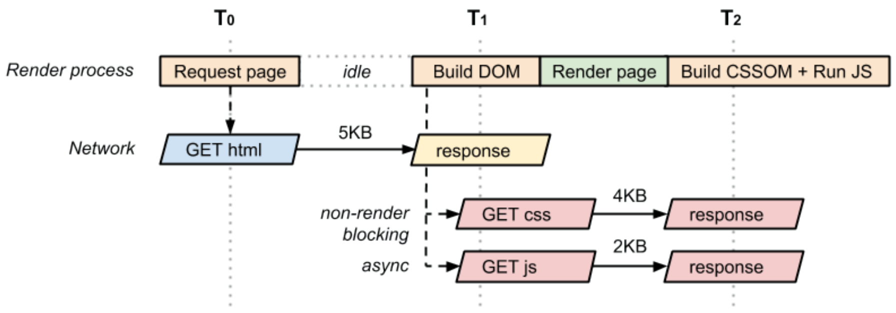
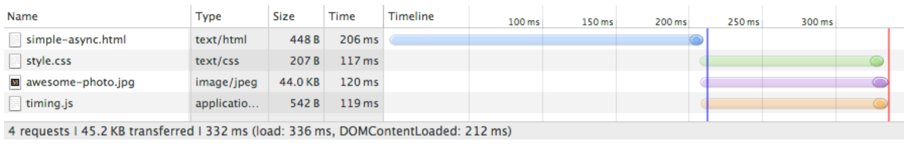


Figure 2.6: Waterfall model for an optimised async CRP
Image extracted from Grigorik [*Analyzing Critical Rendering Path Performance*]
Creative Commons Attribute 3.0 (CC BY 3.0) license



Name	Type	Size	Time	Timeline
simple-async.html	text/html	448 B	206 ms	
style.css	text/css	207 B	117 ms	
awesome-photo.jpg	image/jpeg	44.0 KB	120 ms	
timing.js	applicatio...	542 B	119 ms	

4 requests | 45.2 KB transferred | 332 ms (load: 336 ms, DOMContentLoaded: 212 ms)

Figure 2.7: Google DevTools timeline output for optimised async CRP
Image extracted from Grigorik [*Analyzing Critical Rendering Path Performance*]
Creative Commons Attribute 3.0 (CC BY 3.0) license

Chapter 3

Optimise Rendering Time and Content Efficiency

Web Applications are more popular than they have ever been before. New environments to develop rich applications, such as Angular, React or NodeJS, have been showing up in the past years, giving developers all around the world new possibilities to develop their applications and software.
All these technologies enrich the products but cause problems to the rendering time and efficiency. Users expect applications or websites to load instantly, as shown in the RAIL model, but also to provide more and more functionality and features. Developers take advantage of HTML5, SCSS and JavaScript which provide a variety of features such as animations, transitions and effects. While files should be kept as small as possible, the trend shows exactly the opposite as indicated by the graphic underneath. Although the internet keeps getting faster in various areas of the world, some areas still lack high speed internet and therefore cause a bad user experience.
While developers keep retrieving more and more ways to develop powerful and complex applications, causing larger files and content to be downloaded, the challenge of optimising rendering time becomes more important than ever as IIya Grigorik from Google says: "...we often have to fetch dozens, and sometime hundreds, of various resources, all of which can add up to megabytes of data and must come together in hundreds of milliseconds to facilitate the instant web experience we are aiming for" [*Evolution of Download Sizes*]
In order to provide users the best experience by simultaneously taking advantage of all new environments, some steps can be performed which will be explained in the following sections.

3.1 Render-Blocking CSS and JavaScript

The critical rendering path requires that both the DOM and the CSSOM are ready in order to construct the render tree. CSS is, as well as HTML, a render blocking resource. This means, that content will not be rendered until the CSSOM has been constructed. Developers should write clean and powerful CSS code and make sure it is available as quickly as possible. Typically all websites or web applications have the problem of render-blocking content. The more important it is to improve or eliminate this issue. After all CSS resources have been downloaded, which is always the case, no matter if the resource is blocking or not, developers can exclude or mark statements in their CSS code as not render blocking. This can be achieved by using media queries or media types. The following example shows how this could be implemented:

```
<link href="style.css" rel="stylesheet">
<link href="print.css" rel="stylesheet" media="print">
```

Listing 3.1: Render-blocking and non-blocking includes

The first statement is completely render-blocking as there is no media query. The second statement has the media query print which states that this styles should only be applied when the user wants to print the document. Using such queries can help improving the performance of the site being rendered as only the necessary stylesheets are being used. The first time to render the page will improved and therefore enhances the user experience. Furthermore stylesheets should be directly important in the html document head as the browser quickly recognises the link tag and schedules the request for CSS directly. Despite that, imports should be avoided whenever it is possible. Imports are a way of importing stylesheets into other stylesheets. However, they cause additional roundtrips in the critical path.
To improve performance, CSS code can also be directly inlined in the html document using the style tag.

```
<html>
  <head>
    <style>
      .body {
        color: red;
      }
    </style>
  </head>
  <body>
  </body>
</html>
```

Listing 3.2: Inline CSS code using the style tag

Inlining CSS code avoids additional roundtrips in the critical rendering path and improves the performance of rendering the page the first time.

3.2 CSS optimisation

This chapter explains how developers can improve writing CSS code and what they should focus on when writing code for their projects. Computed style calculation is the process of calculating the styles for the given DOM. In the document object model all elements, like divs, spans and many more, are collected. By changing the DOM through removing, adding or changing styles on various elements, the browser is forced to recalculate the styles and the layout of the whole page. To understand how this browser process works, two parts have to be mentioned. The first part consists of analysing and checking all elements and see if there are pseudo-selectors, classes or elements with an ID which apply to the given element. Viewing it from a developer perspective we could compare this process with an for-loop using conditional statements to identify the elements we are looking for. The second part of the browser style calculation is to calculate the final styles for each element. The browser goes through each element and invokes the styles, then the final styles have to be figured out and applied. Although browsers get faster accomplishing these tasks, developers should be warned about using too much elements and too complex CSS addressing.

3.2.1 Reduce the complexity of styling selectors

Developers tend to build complex CSS code by default using complex addressing, many elements and pseudo-selectors. The deriving consequences for the browser should be considered as they effect the performance in an important way. The following code snippet visualises a typical code snippet which can be found in projects as they grow in size and complexity.

```
<style>
.box:last-of-type .wrapper:nth-last-child .content {
  color: red;
}
</style>
```

Listing 3.3: Complex CSS example

This snippet simply changes the colour of the content element in the last wrapper element child in the last box type. The task for the browser is a bit more difficult. The browser needs to look for the box element and then find the last one in the DOM, then needs to look for the wrapper element and find the last child in it. Finally the content element is searched and the styling applied. If there are hundreds of selectors, like the example above, the page will be drastically slowed down in rendering performance.
The correct way of addressing elements, like the one mentioned above, would be to assign such elements unique classes. In the provided example, the content element could have a unique class for specially targeting the element. The challenge of creating a simple and clean structure of elements and classes caused a need for developers to come up with a way to improve this process, which is called BEM.

3.2.2 Block-Element-Modifers

BEM stands for Block-Element-Modifier and is a guiding scheme on how to assign classes and pseudo selectors. It is build on three basic principles, blocks, elements and modifiers. It is an approach that is based on components or modules. Like all modern and famous web development frameworks, like Angular or React, are build on the idea of using components, BEM integrates this approach into styling HTML elements.

A block is a unique and standalone DOM component which can be reused or nested. It is identified by the class attribute and describes the purpose of the element. Developers should not make use of the id attribute as it is not compatible with the BEM standard. Furthermore a block should not change the layout around it, it has to be considered as a unique and independent component. An example for a block is a search form that maybe consists different input fields.

```
<form class="search-form"></form>
```

Listing 3.4: Input Form HTML example

As indicated above, blocks can also be nested inside each other.

```
<form class="search-form">
  <div class="box"></div>
</form>
```

Listing 3.5: Nested blocks

One should consider that a block inside another block should not be styled or affected in terms of layout by the parent block. Blocks are standalone and not bound to each other.

Elements represent the next layer of the BEM model. Elements are completely bound to blocks, they cannot be used separately in the code. They answer the question What is this? and can as well be nested inside each other. Developers signal an element by typing two underscores after the block name, `block-name__element-name`. Referring to the example earlier, an input field could be an element of the search form block and be addressed in the following way:

```
<form class="search-form">
  <input class="search-form__text-box" type="text"></input>
</form>
```

Listing 3.6: Block with one Element

Elements can be nested inside each other with no limitation on the number of them. What should be considered is that elements do not style each other. There is no addressing as `block__element1__element2`. The benefit of this guide can be shown by a simple code snippet.

```
<div class="block-element">
  <div class="block-element__element1">
    <div class="block-element__element2">
      <div class="block-element__element3">
      </div>
    </div>
  </div>
</div>
```

Listing 3.7: Block with nested elements

Styling all the elements in the snippet may look difficult but it is very simple and effective:

```
<style>
.block-element {}
.block-element__element1 {}
.block-element__element2 {}
.block-element__element3 {}
</style>
```

Listing 3.8: CSS Styling with BEM

The last layer in the BEM model are modifiers. As the name states, modifiers define the appearance, style or behaviour of an element or block. A modifier is indicated by a single underscore after the elements name. The correct way of addressing is therefore: `block__element_modifier`. Developers should use them to support different variants of elements and blocks. They could be used for different sizes, colours or any property. As modifiers are completely dependent on their parent, they cannot be used alone.

BEM introduces an effective and powerful way to structure CSS code and reduce complexity of selectors. In the last chapter of this survey, BEM will be used and compared with bad and complex written code in order to demonstrate effectiveness.

3.2.3 Complexity of Style Calculations

Web applications or web pages consist of layouts. Layouts define the geometric information for all the HTML elements a page exists of. Depending on all the different style assignments, elements will have layouts which need to be rendered and painted. Pages that contain a large number of complex elements need much more time to be rendered and should therefore be avoided.
There are lot of layout affecting styling commands which should not be used whenever it is possible. Such include width, height, display and many more. A list of layout triggering commands can be found at CSS Triggers. As layout is scoped and bound to the whole document, it takes a lot of time if elements change.

In order to analyse the performance of a page, developers should use the Google Chrome Timeline option. Using the timeline feature of the browser provides feedback on how many elements were in need of layout, how long it took to construct or repaint them as well as the layout tree size. An important way of using layout have been width, height, float or the position trigger. Using the chrome timeline feature shows that using the new trigger flex box provides much more performance than the old ones. More information on how much this effects the performance of the page will be given in the case study chapter.

3.3 JavaScript Optimizations

While writing Web-Apps there are a lot of possible methods, design pattern and conventions how to write the logic in JavaScript. Since the focus goes on mobile applications, performance is a very important component. According to [*JavaScript Best Practices*] there are some base concepts to write performant, compact JavaScript Code:

- Use local variables instead of global
- Be careful when and how to use Closures
- Treat numbers, strings and booleans as primitive values – not objects!
- Don't use new Object()

3.3.1 Closures

In JavaScript, Closures are a very powerful and dangerous construction containing an outer and an inner function:

```
var incElementCount = function() {
var elementCount = 0;

return function() {
  return elementCount += 1;
}
}();
```

Listing 3.9: Iterating trough Objects with knwon length

If incElementCount gets invoked, the variable elementCount gets initialized and is accessible in the inner function, so the return value will be 1. The next time incElementCount gets invoked, the outer function is not executed anymore, but the variable elementCount and it's previous value is still accessible. So the second return value will be 2. This means, that the variables declared in the outer function still exist after the outer function returns, because of the use in the inner function. This also means, that these variables won't get cleared by the Garbage Collector and stay in memory. There is the so called "Lexical Environment", which means, that if there is a variable in the outer function and any inner functions refers to that variable (even if the function is unused), the variable stays in memory. As meteor programmers [*An interesting kind of JavaScript memory leak*] found out, this could cause memory leaks:

```
var theThing = null;
var replaceThing = function () {
  var originalThing = theThing;
  var unused = function () {
    if (originalThing)
    console.log("hi");
  };
  theThing = {
    longStr: new Array(1000000).join('*'),
    someMethod: function () {
    console.log(someMessage);
  }
};
setInterval(replaceThing, 1000);
```

Listing 3.10: Memory Leak Szenario

In that case, the function "setInterval()" calls "replaceThing" every second. The variable "originalThing" refers to "theThing" which creates a big variable. If the function "unused" does not exist, the Garbage Collector would clear "originalThing", because it's not used in any Closure. But now, even if the function "unused()" is never called, the Garbage Collector just recognizes, that there is a reference to the variable "originalThing", so it stays in memory because of the concept of the Lexical Environment. The Garbage Collector detects the use of the variable "originalThing" in one inner function, so it has to stay in memory. This leads to a memory leak of 1mb per second. Chrome's DevTool records that leak in the timeline:

There are also performance issues when using Closures. According to Gregory Baker and Erik Arvidsson [*Optimizing JavaScript code*]: "Creating a closure is significantly slower than creating an inner function without a closure, and much slower than reusing a static function".

3.3.2 Objects vs. Arrays

The most common containers in JavaScript are Arrays and Objects. While Objects are perfect for storing data of different types, Arrays are comfortable to store for example numbers. As already mentioned, don't declare these types with the keyword "new" but rather like:

```
var myArray = [];
var myObject = {};
```

Listing 3.11: Declaration of Arrays and Objects

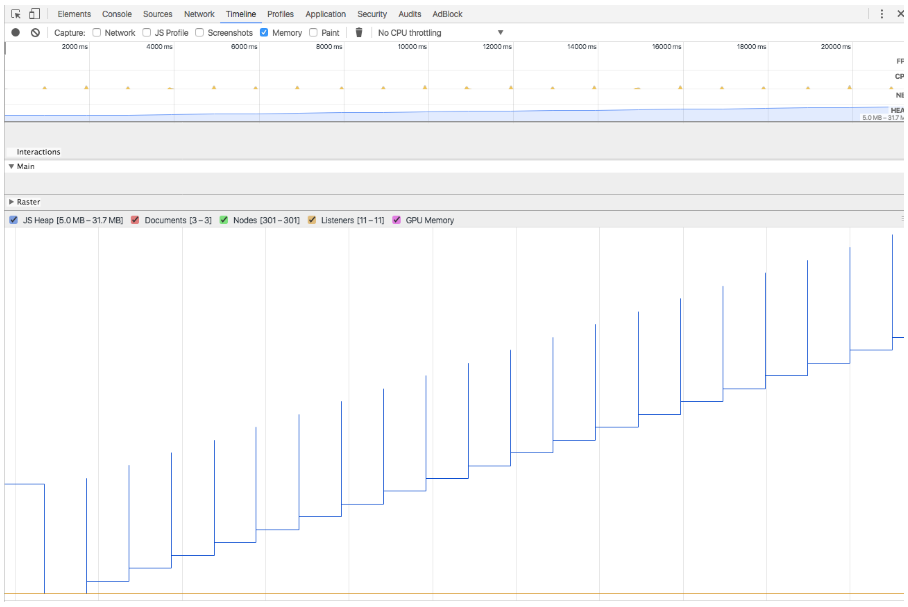


Figure 3.1: Schematic representation the record of the Closure Memory Leak

Data Access and Iteration

In JavaScript, Objects are represented as associative Arrays. In Chrome's V8 there are some additional hidden classes to optimize data access in Objects [*A tour of V8: object representation*].

There are generally two ways to access the data:

- Get value via index (integer indexed Arrays)
- Get value via key (Objects and associative Arrays)

In general, there are two ways to iterate through these containers:

```
Iterating through integer-indexed Array:

for (var index = 0; index < myIntArray.length; ++index) {
  var value = myIntArray[index];
}

Iterating through Object with string-based keys:

for (var key in myStringObject) {
  var value = myStringObject[key];
}
```

Listing 3.12: Iterating trough Arrays and Objects

As demonstrated in section 4.2.1, there are performance drawbacks on iterating trough arbitrary Objects with the for..in scheme. The determining point is the length attribute of the array. So what about iterating performance if the Object keys were also integers and the length was known:

```
for (var index = 0; index < containerSize; ++index) {
  var stringKey = index.toString();
  var value = myIntObject[index];
}
```

Listing 3.13: Iterating through an Object with the assumption, taht the keys are known integers

As also shown in section 4.2.1, the performance compared to integer-based Arrays is nearly the same. So is it just the for-loop or does the integer-based vs. string-based key access also influence the performance? The following example shows the timing difference between these methods:

```
for (var index = 0; index < containerSize; ++index) {
  var stringKey = index.toString();
  var value = myStringObject[stringKey];
}
```

Listing 3.14: Iterating through an Object with the assumption, that the keys are in range '1' to '1000'

The example in chapter 4.2.1 shows, that altough the for-loop is the same, the data access via a string-key shows performance drawbacks of about 8%.

3.3.3 Initializing instance variables

According to [*Optimizing JavaScript code*], the main thing to take care when initializing instance variables is, that fixed valued primitives shouldn't be declared in the constructor. The problem is, that this code must always be executed when calling the constructor. Instead of that, declare it on the Prototype:

```
function a() {
  this.obj = {};
}

a.prototype.sameValue = 3;
```

Listing 3.15: Iterating trough Objects with knwon length

According to definition of [*JavaScript Object Prototypes*], every JavaScript Object has a prototype and the prototype is also an Object. Further, all JavaScript Objects inherit all properties and methods from their prototype. So instance variables of an object can be declared on the prototype of the Object. These variables are declared once (outside the constructor) and are accessible in every new instance of the Object.

3.3.4 Defining Classes and Class Methods

There are some pattern how to define classes and declare class methods.

Module Pattern

The so called Module Pattern defines a class with instance variables and methods:

```
MyClass = function () {
  var studentCnt = 0;

  var addStudent = function () {
    studentCnt +=1;
    console.log(studentCnt);
  };
  var kickStudent = function () {
    studentCnt -=1;
    console.log(studentCnt);
  };

  return {
    addStudent: addStudent,
    kickStudent: kickStudent
  }
}

var testClass = MyClass();
testClass.addStudent();
testClass.addStudent();
testClass.kickStudent();
```

Listing 3.16: JavaScript Module Pattern

On every new instance of MyClass, the code including the instance variables and methods has to executed.

Prototype Methods

The second enhanced way is to declare the class methods on prototypes:

```
MyClassProt = function() {
  this.studentCnt = 0;
};

MyClassProt.prototype.addStudent = function() {
  this.studentCnt +=1;
  console.log(this.studentCnt);
};

MyClassProt.prototype.kickStudent = function() {
  this.studentCnt -=1;
  console.log(this.studentCnt);
};


var testClass1 = new MyClassProt();
testClass1.addStudent();
testClass1.addStudent();
```

```
testClass1.kickStudent();
```

Listing 3.17: JavaScript Prototype Methods Pattern

In this pattern, the class methods are declared once. So the code does not need to be executed on every new class instance.

Module Pattern vs Prototype Methods

As performed in the case Study 4.2.2 the performance difference is clear. Prototypes lead to a performance boost of about 60%. The drawback while using Prototype Pattern is the readability of code. In many cases the Module Pattern is more readable and looks more structured than the faster Prototype Pattern.

3.3.5 Adding DOM elements with DocumentFragment

One of the most expensive operations for the browser is the redraw of the DOM. According to Nicholas Zakas [*Speed up your JavaScript, Part 4*], there are various points which trigger a so called reflow:

- add or remove a DOM node
- apply style dynamically (e.g. element.style.width = "50
- retrieve a measurement like client height of a scrollable container

The DocumentFragement is a "lightweight" document object to process off DOM and avoid or minimize the expensive reflow. The aim is to create a new fragement off the DOM and insert it into the DOM after complete creation of the object. Iteratively inserting new elements into the DOM produces a reflow on every call of appendChild(). While using DocumentFragements, the reflow gets minimized remarkable as demonstrated in the case study in section 4.2.3.

3.3.6 Preventing Layout Trashing

As described in section 1.5 modifying node sizes and the triggering a document reflow by reading the new value leads to layout trashing, hence causing performance problems.
There are two main approaches to fix this issue:

- Reordering of the code in a way that does not lead to premature reflow
- Using asynchronous frames

A quick fix to this problem would be the code reordering, but this practically only works in a perfect environment (without the need for outsourced or logically grouped code). This solution is not applicable for most projects, so a better way is using the asynchronous function `requestAnimationFrame(...)`, which allows moving code execution to a separate frame and therefore unblocks the rendering.
The code snippet below demonstrates this procedure on the example from code 1.2.

```
// Read
var h1 = element1.clientHeight;

// Write in seperate frame -> no invalidation
requestAnimationFrame(function() {
  element1.style.height = (h1 * 2) + 'px';
```

```
    // We may want to read the new
    // height after it has been set -> could be modified
    // by another frame afterwards
    var height = element1.clientHeight;
  });

  // Read
  var h2 = element2.clientHeight;

  // Write
  requestAnimationFrame(function() {
    element2.style.height = (h2 * 2) + 'px';
    var height = element2.clientHeight;
  });
```

Listing 3.18: Preventing Layout Trashing

Applying this method to all sources of possible layout trashing can result in a dramatic performance boost (speeds rendering up to 25%), without destroying program logic or code structure.

Chapter 4

Case Study

In order to demonstrate the effectiveness and usefulness of the suggestions and conclusions derived from this survey, a case study has been completed. Both CSS and JavaScript optimisations have been performed and being compared with bad and complex code. The following sections give a brief insight into the code that has been written and the changes performed. For measuring the optimisation, the Google Timeline feature and the browser Chrome have been used.

4.1 CSS Optimisation

The CSS study features the usage of the BEM standard versus complex and bad element addressing as well as the advantages of new technologies for creating efficient layouts. Therefore a block has been created which can be seen in the image. It consists of two elements, an image and a content block. Both blocks have an equal width. The content, including both the headline and the sub-line, are horizontally and vertically aligned inside their block. To demonstrate differences, more than a thousand blocks have been used.

4.1.1 Complex and Slow Code

The following code snippet is the basis block which is used for this experiment and a typical example for unstructured HTML which has been coded without the usage classes. Although the code is fine, one should not use such code.

```
<div class="box">
<div>
<img src="images/forest.jpeg">
</div>
<div>
<h1>Überschrift</h1>
<span>Das ist eine Beschreibung</span>
</div>
</div>
```

Listing 4.1: Bad labeled HTML code

In order to achieve the layout and style, CSS code has been written making use of complex and bad performance selectors, like last-of-type or nth-child. Even tough the code achieves the expected styling, it causes a much higher rendering time.

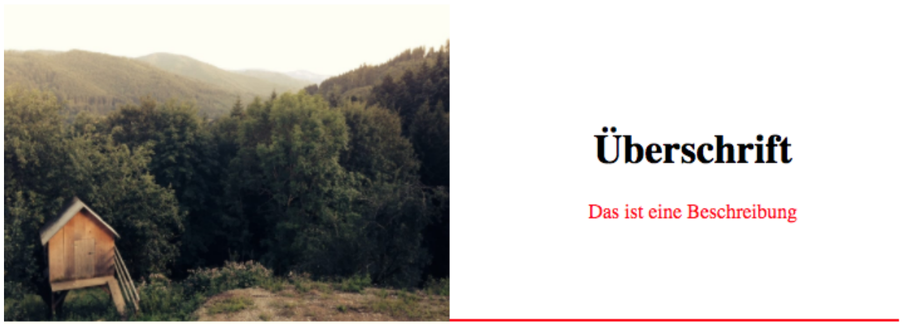


Figure 4.1: Base HTML Block

```
<style>
.box {
position: relative;
float: left;
display: inline-block;
width: 100%;
height: 250px;
border-bottom: 2px solid red;
}

.box > div {
position: relative;
float: left;
width: 50%;
display: block;
height: 100%;
}

.box > div:last-of-type > div:nth-child(1) > span {
float:left;
width: 100%;
text-align:center;
color: red;
padding: 0 15px;
}

.box > div:last-of-type > div:nth-child(1) > h1 {
padding: 0 15px;
float:left;
width: 100%;
text-align:center
}

.box > div > img {
position: relative;
float: left;
width: 100%;
overflow: hidden;
height: auto;
}

.box > div:last-of-type > div:nth-child(1) {
display: block;
position: absolute;
top: 50%;
transform: translateY(-50%);
width: 100%;
text-align: center;
}
</style>
```

Listing 4.2: Bad and complex written CSS code

The performance has been measured using the Google Chrome timeline tool. Loading the page gave the following rendering time:

Rendering Time: 177.0ms

4.1.2 Using Block-Element-Modifier

As explained in previous chapters, the Block-Element-Modifier is a powerful tool for developers to write clean CSS code. Therefore the HTML structure of the defined block used in this project has been extended by classes. Afterwards all the assignments in the stylesheet have been replaced using the new classes obtained by the BEM model.
The new rendering time clearly showed a better result compared to the first test run:

Rendering Time: 139.5ms

4.1.3 Using Flexbox for Layout

The last phase of the CSS experiment is dominated by the usage of the new layout technology flexbox. Flexbox can be used to layout elements and blocks in an efficient and powerful way. One does not need floating assignments or absolute positioning anymore. Developers always struggled with the vertical alignment of elements. Previously the display option could be used, otherwise absolute positioning and transformation did the job. With the new flexbox, vertical alignment is achieved with a single line. In order to demonstrate the effectiveness, the complete CSS code has been rewritten using BEM and flexbox. The following code is much more clean and by far not as complex as the old one.

```
<style>
box {
width: 100%;
display: flex;
flex-direction: row;
height: 250px;
border-bottom: 2px solid red;
}
.box__element {
width: 50%;
}
.box__element_image {
width: 100%;
height: 100%;
}
.box__element_last {
display: flex;
justify-content:center;
align-items:center;
}
.box__element_content_red {
color:red;
}
.box__element_content_padding {
padding: 0 15px;
}
</style>
```

Listing 4.3: Clean and structured CSS code

The performance check gave the following rendering time:

Rendering Time: 108.0ms

4.2 JavaScript Optimization

4.2.1 Objects vs Arrays

The following example shows the performance difference of looping through integer-based Arrays, integer-based Objects and string-based Objects:

```
/******************************************************/
// Arrays vs. Objects
/******************************************************/

var myIntArray = [];
var myIntObject = {};
var myStringObject = {}
var containerSize = 1000;

//fill contaiuner
for (var index = 0; index < containerSize; ++index) {
myIntArray[index] = index + 1;
myIntObject[index] = String(index);
myStringObject[String(index)] = index;
}

function intArrayTest() {
for (var index = 0; index < myIntArray.length; ++index) {
var stringKey = index.toString(); //dummy
var value = myIntArray[index];
}
}

function stringObjectTest() {
for (var key in myStringObject) {
var stringKey = index.toString(); //dummy
var value = myStringObject[key];
}
```

test name	duration[ms]	
intArrayTest	0.03220	fastest
stringObjectTest	0.07126	121% slower
intObjectTest	0.06349	97% slower
intObjectIndexedTest	0.03992	24% slower
stringObjectIndexedTest	0.04267	32 % slower

Table 4.1: Objects vs. Arrays performance results

```
}

function intObjectTest() {
for (var key in myIntObject) {
var stringKey = index.toString(); //dummy
var value = myIntObject[key];
}
}

function intObjectIndexedTest() {
for (var index = 0; index < containerSize; ++index) {
var stringKey = index.toString(); //dummy
var value = myIntObject[index];
}
}

function stringObjectIndexedTest() {
for (var index = 0; index < containerSize; ++index) {
var stringKey = index.toString();
var value = myStringObject[stringKey];
}
}


function iterationTest() {

var testFunctions = [intArrayTest, stringObjectTest, intObjectTest,
    intObjectIndexedTest, stringObjectIndexedTest];

for(testFunction in testFunctions) {
var t0 = performance.now();
for(var i = 0; i < 1000; ++i) testFunctions[testFunction]();
var t1 = performance.now();
console.log(testFunctions[testFunction].name + ": " + ((t1 - t0)/1000).
    toFixed(5) + " ms.")
}
}
```

Listing 4.4: Test Array and Object iteration and data access performance

The result of the performance tests shows the following average of the different access methods:

4.2.2 Module Pattern vs. Prototype Pattern

The following case study measures the performance of creating 100 objects with the Module Pattern compared to the Prototype Pattern as follows:

```
/********************************************************/
// Module Pattern vs. Protoype Pattern
/********************************************************/

ModuleClass = function () {
var studentCnt = 0;

var addStudent = function () {
studentCnt +=1;
//console.log(studentCnt);
};
var kickStudent = function () {
studentCnt -=1;
//console.log(studentCnt);
};
return {
addStudent: addStudent,
kickStudent: kickStudent
}
}

PrototypeClass = function() {};

PrototypeClass.prototype.studentCnt = 0;

PrototypeClass.prototype.addStudent = function() {
this.studentCnt +=1;
//console.log(this.studentCnt);
};

PrototypeClass.prototype.kickStudent = function() {
this.studentCnt -=1;
//console.log(this.studentCnt);
};

var patternTestIterations = 100;
var patternTestObjectsToCreate = 1000;

function modulePatternTest() {
console.time("modulePatternTest");
var objects = [];
for (var cnt = 0; cnt < patternTestIterations; ++cnt) {
var object = ModuleClass();
object.addStudent();
object.addStudent();
object.kickStudent();
}
console.timeEnd("modulePatternTest");
}
```

test name	**duration[ms]**	
prototypePatternTest	0.02624	fastest
modulePatternTest	0.04290	about 60% slower

Table 4.2: Module Pattern vs. Prorotype Pattern results

```
function prototypePatternTest() {
console.time("prototypePatternTest");
var objects = [];
for (var cnt = 0; cnt < patternTestIterations; ++cnt) {
var object = new PrototypeClass();
object.addStudent();
object.addStudent();
object.kickStudent();
}
console.timeEnd("prototypePatternTest");
}

function patternTest() {
var testFunctions = [modulePatternTest, prototypePatternTest];

for(testFunction in testFunctions) {
var t0 = performance.now();
for(var i = 0; i < patternTestIterations; ++i) testFunctions[
   testFunction]();
var t1 = performance.now();
console.log(testFunctions[testFunction].name + ": " + ((t1 - t0)/
   patternTestIterations).toFixed(5) + " ms.")
}
}
```

Listing 4.5: Module Pattern vs. Prototype Pattern performance test

These test cases show the following results on creating 1000 objects with the Module Pattern in comparison to the Prototype Pattern:

4.2.3 DocumentFragment Performance Test

In the following test case, the use of DocumentFragment is compared to iterative inserting child nodes into the DOM. The performance difference in that case is for example very interesting in Apps with large lists or chat applications where data or child nodes are get appended in the DOM. The performance test in that case is done as follows:

```
/*******************************************************/
// DocumentFragment
/*******************************************************/

var elementsToInsert = [];
var numOfElementsToInsert = 20;
var numOfIterations = 10;
```

```
for (var cnt = 0; cnt < numOfElementsToInsert; ++cnt) {
var newElement = document.createElement("div");
newElement.innerHTML = "child" + String(cnt);
elementsToInsert.push(newElement);
}

function addIteratively() {
console.time("addIteratively");
for (var iterator = 0; iterator < numOfIterations; ++iterator) {
var containers = document.getElementsByClassName("parent");
//console.log(containers);
for(var containerIndex = 0; containerIndex < containers.length; ++
   containerIndex) {
var leftElements = 0;
while(leftElements < numOfElementsToInsert) {
var newElement = elementsToInsert[leftElements].cloneNode(true);
containers[containerIndex].appendChild(newElement);
leftElements += 1;
}
}
}
console.timeEnd("addIteratively");
}

var fragToInsert = document.createDocumentFragment();
for (var cnt = 0; cnt < numOfElementsToInsert; ++cnt) {
fragToInsert.appendChild(document.createElement("div"));
}

function addFragement() {
console.time("addFragement");
for (var iterator = 0; iterator < numOfIterations; ++iterator) {
var containers = document.getElementsByClassName("parent");
//console.log(containers);
for(var containerIndex = 0; containerIndex < containers.length; ++
   containerIndex) {
var frag = document.createDocumentFragment();
var leftElements = 0;
while(leftElements < numOfElementsToInsert) {
var newElement = elementsToInsert[leftElements].cloneNode(true);
frag.appendChild(newElement);
leftElements += 1;
}
containers[containerIndex].appendChild(frag);
}
}
console.timeEnd("addFragement");
}
```

Listing 4.6: DocumentFragment Performance Test

The test adds 20 child nodes in a DOM structure. This is done 10 times, while with each iteration, the tree gets more complex. This specific test case shows the following performance comparing iterative insertion and DocumentFragments:

test name	**duration[ms]**	
addFragement	7	fastest
addIteratively	16	about 120% slower

Table 4.3: Iterative Insertion vs. DocumentFragments Insertion Results with 10 iterations of 20 child node insertions

4.3 Conclusion

The CSS study showed, that both the usage of BEM and new layout technologies can drastically improve performance. The rendering time was nearly double as fast compared to the old code. The experiment only featured a small amount of elements, compared to a web application or website, but the results confirm and justify the methods presented in this survey.

Coming to a conclusion, developers should make use of the Block-Element-Modifier tool and avoid the usage of selectors like nth-child or last-of-type. Despite that, new layout technologies should be considered too in order to achieve an optimal rendering performance.

Bibliography

Conrod, Jay [2013]. *A tour of V8: object representation*. 2013. http://jayconrod.com/posts/52/a-tour-of-v8-object-representation (cited on page 21).

Glasser, David [2013]. *An interesting kind of JavaScript memory leak*. Aug 2013. http://info.meteor.com/blog/an-interesting-kind-of-javascript-memory-leak (cited on page 20).

Gregory Baker, Erik Arvidsson [2016]. *Optimizing JavaScript code*. Feb 2016. https://developers.google.com/speed/articles/optimizing-javascript (cited on pages 20, 22).

Grigorik, Ilya [2016]. *Analyzing Critical Rendering Path Performance*. Nov 2016. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp (cited on pages 12–14).

Grigorik, Ilya [2016]. *Constructing Model Objects*. Nov 2016. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model (cited on pages 1–3).

Grigorik, Ilya [2016]. *Deciphering the Critical Rendering Path*. Nov 2016. http://calendar.perfplanet.com/2012/deciphering-the-critical-rendering-path/ (cited on page 2).

Grigorik, Ilya [2016]. *Evolution of Download Sizes*. Nov 2016. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/ (cited on page 15).

Grigorik, Ilya [2016]. *Render-tree Construction, Layout, and Paint*. Nov 2016. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction (cited on pages 3–6).

Hogan, L.C. [2014]. *Designing for Performance: Weighing Aesthetics and Speed*. O'Reilly Media, 2014. ISBN 9781491903735. https://books.google.at/books?id=QPixBQAAQBAJ (cited on page 1).

Kearney, Meggin [2016]. *Measure Performance with the RAIL Model*. Nov 2016. https://developers.google.com/web/fundamentals/performance%/rail (cited on page 9).

Keith, J. [2006]. *DOM Scripting: Web Design with JavaScript and the Document Object Model*. Apress, 2006. ISBN 9781430200628. https://books.google.at/books?id=LBTQ83bAz6QC (cited on page 1).

McAnlis, Colt [2016]. *CSS Paint Times*. Apr 2016. https://www.html5rocks.com/en/tutorials/speed/css-paint-times/ (cited on pages 6–7).

Page, Wilson [2013]. *Preventing 'Layout Trashing'*. Sep 2013. http://wilsonpage.co.uk/preventing-layout-thrashing/ (cited on page 5).

Smith, P.G. [2012]. *Professional Website Performance: Optimizing the Front-End and Back-End*. Wiley, 2012. ISBN 9781118551721. https://books.google.at/books?id=MHLJ1UfXV4QC (cited on page 3).

W3Schools.com. *JavaScript Best Practices*. http://www.w3schools.com/js/js_best_practices.asp (cited on page 19).

W3Schools.com. *JavaScript Object Prototypes*. http://www.w3schools.com/js/js_object_prototypes.asp (cited on page 22).

Zakas, Nicholas C. *Speed up your JavaScript, Part 4*. https://www.nczonline.net/blog/2009/02/03/speed-up-your-javascript-part-4/ (cited on page 24).