Master Thesis
Computer Science

Radboud University Nijmegen

Client-side performance profiling of JavaScript for web applications

Author:
Thomas Nägele
thom@snagele.nl

Supervisor Radboud University:
prof. dr. Jozef Hooman
hooman@cs.ru.nl

Supervisors Topicus:
Remco Zigterman
remco.zigterman@topicus.nl

Mark Brul
mark.brul@topicus.nl

24th June 2015

Abstract

Modern web applications greatly rely on JavaScript to be executed on the computer of the client. Since every user runs the application on their own computer, there are many different hardware and software combinations. From the developers perspective, it is therefore difficult to identify performance issues that may occur at run-time. This thesis describes the development of a generic profiler for JavaScript in web applications. Client-side performance in the context of web applications is defined and contributing factors to this performance are explained. Before constructing a profiling method, first an evaluation method is constructed to quantitatively compare different methods. A few of methods are proposed and evaluated, after which one profiling method is implemented. Next, an integration tool is developed to enable easy integration with any web application. As expected, the profiler has some impact on the performance of the application, but it was found to be accurate, easy to use and portable.

Acknowledgements

First of all, I would like to thank Topicus for giving me the opportunity to carry out this research. In particular, I would like to thank Remco Zigterman and Mark Brul for their guidance and support throughout the process. Also, many thanks to my supervisor, Jozef Hooman, who provided me with loads of feedback on every draft I sent. His feedback and ideas have helped me enormously. I also would like to thank Joost Visser for taking the time to act as second reader. Finally, last but not least, thanks to my friends and family who supported me, read bits and pieces of this thesis and provided additional feedback. Thank you.

Contents

1 Introduction **1**
1.1 Background information . . . 1
1.2 Problem statement . . . 2
1.3 Approach . . . 3
1.4 Scientific context . . . 5

2 Web applications **7**
2.1 General architecture . . . 7
2.2 Page rendering . . . 8
2.2.1 Web application workflow . . . 9
2.3 JavaScript . . . 10
2.3.1 Benchmarks . . . 11
2.3.2 Profilers . . . 12
2.3.3 Static program analysis . . . 13
2.3.4 Frameworks . . . 13
2.3.5 AngularJS . . . 15
2.4 Interact CTG . . . 19
2.4.1 CTG . . . 19
2.4.2 Architecture . . . 19

3 Profiling performance **22**
3.1 Definition of client-side performance . . . 22
3.1.1 Classical definition of performance . . . 22
3.1.2 End-user definition of performance . . . 24
3.1.3 Technical definition . . . 24
3.2 Performance analysis method properties . . . 25
3.3 Method evaluation . . . 26
3.3.1 Accuracy . . . 27
3.3.2 Impact . . . 31
3.3.3 Usability . . . 31
3.3.4 Portability . . . 32

4 Methods for performance profiling **33**

4.1 Manual profiling 33
4.1.1 Pre-evaluation 33
4.2 Manual logging 34
4.2.1 Pre-evaluation 35
4.2.2 Implementation 35
4.2.3 Post-evaluation 37
4.2.4 Issues 39
4.3 Manual Logging improved 40
4.3.1 Implementation 40
4.3.2 Post-evaluation 45

5 Integration 48
5.1 Aspect Oriented Programming 48
5.2 Integration tool 49
5.2.1 Implementation 49
5.2.2 Improvements 50
5.2.3 Limitations 51
5.2.4 Concluding grade 51

6 Conclusion 53
6.1 Discussion 54
6.2 Related work 55
6.3 Future work 55

Appendix A Benchmark results 59
A.1 Dromaeo JavaScript Tests 59
A.2 Sunspider 1.0.2 60
A.3 Kraken 1.1 60
A.4 Octane 2.0 61

Appendix B Accuracy test 62
B.1 Sources 62
B.1.1 Array 64
B.1.2 String 65
B.1.3 Date 66
B.1.4 Json 68
B.1.5 Regexp 68
B.2 Results from built-in JavaScript profiler 69
B.3 Results from Manual Logging method 69
B.4 Results from improved Manual Logging method 69

All sources mentioned in this thesis can be found in the following repository:
`https://bitbucket.org/phpnerd/jsprofiler/`

Chapter 1

Introduction

1.1 Background information

The problem statement for this Master's thesis stems from a question raised by Topicus Healthcare. Topicus Healthcare develops an application, Interact CTG[1], that monitors the birth of a baby. This application is based on a server-client architecture. The server component receives data from a cardiotocograph (CTG) produced by another party and processes this. A client is able to fetch this data from the server and display it to the user in any form.

A CTG is a machine that records the foetal heartbeat and the uterine contractions during pregnancy. With a CTG the health status of the unborn child is monitored. It can also be used to perform smaller investigation during the last phase of pregnancy can be performed to check whether the child is healthy or not.

The client is a web application which should run in all common browsers and is the front-end of the whole system. The client is written in AngularJS[2] and communicates with the REST [1] server written in Java. Figure 1.1.1 displays a high level overview of the system and its connections.

The server, written in Java, functions as a REST server. REST (Representational State Transfer) is a design principle used for requesting, storing and deleting data on the server and is used for the Hypertext Transfer Protocol (HTTP) and commonly used for building Application Programming Interfaces (APIs).

[1] `http://www.topicus.nl/zorg/tweede-lijn-en-transmuraal/verloskunde/product/interact-ctg`

[2] `https://angularjs.org/`

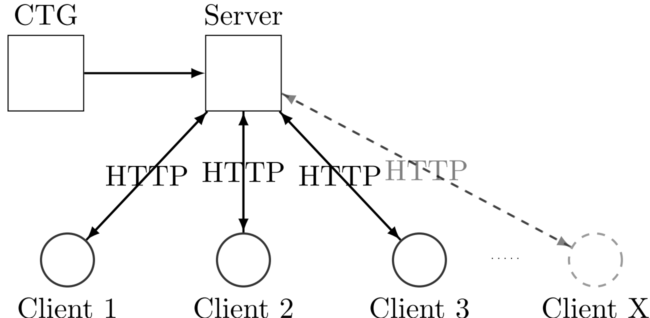

Figure 1.1.1: High level overview of the Interact CTG system.

1.2 Problem statement

Because the front-end is a web application which can run from any browser, Topicus does not have any influence on its environment. There are quite some different browsers that may load the application, running on numerous different operating systems. In addition to this problem, the hardware on which the application runs can also be any configuration you may think of. The number of configurations is therefore practically infinite and the web application can never be tested on all those configurations.

Also, when a client contacts the Topicus support with a complaint about the application being too slow, it is not always possible to rerun the scenario as described by the client in a similar – or preferably identical – configuration. For debugging purposes it is, however, very important to precisely locate the problem in the source code.

The front-end of Interact CTG uses AngularJS, which is a JavaScript framework for creating dynamic web pages. JavaScript is interpreted and executed by a JavaScript engine in the browser. Because there are quite a few JavaScript engines and each browser uses a different JavaScript engine, the performance of these browsers when processing JavaScript may vary. It is therefore possible that the users of a web application experience different performance when running the application.

Hence, the aim is to create a method to measure the performance of a web application at the client. This measurement should not only measure the performance of the web application, but should also enable localisation of parts of code that have a negative impact on the performance. With these results it may be possible to improve the speed of the web application in future releases. This analysis may be specific for AngularJS, but may be applicable for JavaScript in general as well. This analysis corresponds, in contrast to static code analysis, to a real-world performance scenario.

To be able to measure the performance of an application, it is important to define performance for this application. There are a couple different possible measures for performance in the field of web applications. Measures like latency, page render time and user interaction speed all contribute to the overall performance of the web applic-

ation and can therefore be seen as potential subjects of complaints from clients. For a complete analysis of the performance of the web application, a well-defined list of measures for performance should be made.

Research questions

The main target of this research is to create a method to measure the client-side AngularJS performance of a web application, considering that there are many different configurations possible as client system. The main question for this research can be formulated as follows.

> How to measure the client-side AngularJS performance of a web application in such a way that performance problems can be diagnosed?

This question may be split up in the following parts.

1. What is performance in the context of AngularJS and web applications in general?
    - What causes differences in performance of the same web application when running on different machines or web browsers?
2. How to analyse AngularJS performance?
    - How much does analysing the AngularJS performance affect the results?
    - Is the method generalisable to JavaScript?
3. How can the results be validated?
    - Are the results correct?
4. Is the method easy to extend to other projects?
    - What should be done to use this method for another project?
    - For what kind of projects is this method usable or suitable and for what projects it is not?
5. What actions should the end-user perform to report a problem diagnosis with the developed method?
    - How to minimise the effort the client has to do?

1.3 Approach

First of all, it is important to define performance in the context of the existing web application. For this, literature should be found on front-end performance of web

applications in general. In addition to this, user experiences can be obtained, leading to a common definition of performance.

Additionally, it is useful to know what causes the differences in performance on different web browsers or on different systems. The results of investigation should lead to examples of weaknesses and strong points of specific web browsers or systems. With these examples, small web applications can be developed as some sort of test set to help measuring the performance.

Next, research needs to be done on how to measure the performance as defined in the first stage of the research. Existing techniques for this may apply or small prototypes can be developed in this stage to do some kind of analysis on small pieces of code. Multiple methods can be tested. These methods are then evaluated on a set of properties to determine the usability of the method. These properties are the following.

- **Accuracy**
  How correct the method is in measuring the performance of the web application. This can be measured by testing the analysis on the set of sample programs as mentioned before. These sample programs contain known weaknesses for different browsers and try to exploit those weaknesses. If the analysis of the results succeeds to pinpoint these weaknesses within the programs, it is a correct measurement.
- **Impact**
  The impact of the method on the actual performance of the web application. This can be calculated by subtracting the execution time without this method from the execution time when applying this method for analysis.
- **Usability**
  How usable this method is for the end-user of the application. Should the user perform all kinds of operations or is it just one click on a button? To be able to compare this property of the methods, for each method a list should be created with the actions to be performed by the end-user.
- **Portability**
  Is the method relatively easy to apply to different project? What should be done to apply this method on another web application?

Every method is validated for these properties to determine its suitability for a web application. This phase should lead to answers on the second question. Whether this method is generalisable for JavaScript in general can possibly be determined later. The most suitable method is selected and a larger prototype implementing this method for the Interact CTG web application will be developed.

Halfway developing this larger prototype for the Interact CTG web application, this prototype should be reviewed and evaluated to determine whether this is still the most suitable method. Depending on the results either a new prototype based on a different

method can be built or the current prototype should be finished.

Once this prototype is implemented for this particular web application, it is useful to check whether this method can be generalised for JavaScript in general. By using methods that are in compliant with JavaScript itself instead of using AngularJS-specific methods, the method is more likely to be applied to a more general context. Also, it is important to determine whether this method is portable, e.g. can be used for other projects and, if applicable, what projects are suitable to perform this analysis on. To achieve this, some other projects can be selected to apply this method to. Of course, it is convenient to start thinking about this in an early stage, because it is always less work to start early on working on a more general solution, if possible.

1.4 Scientific context

JavaScript has become increasingly popular over the last couple of years. This is partly caused by JavaScript slowly moving from a typical web programming language to a more general purpose language [2]. For web applications, dozens of JavaScript frameworks are available to enhance the programmability, usability and visibility of the web application. Examples of some widely used frameworks are jQuery[3], Dojo[4], Prototype[5] and AngularJS[6]. These frameworks are being used on millions of websites.

The measurement and improvement of performance of web applications has received considerable attention for as long as web applications exist. Most performance improvements are done at the side of the server by increasing scalability [3] of the back-end or making the application more efficient, e.g. by using a smart cache [4].

The performance of a web application at the client side is often not measured or just tested for usability. There are, however, some widely used JavaScript benchmark suites for measuring the performance of web browsers. Unfortunately, these benchmarks fail to represent the performance of a browser on a typical website [5]. For this reason, research [6] was done to construct new JavaScript benchmarks that are able to measure performance like it would be on a real website.

The main focus of such a benchmark is not to measure the performance of a web application itself, but to measure its performance within a specific browser [7]. It is therefore still not useful for debugging purposes, because no detailed analysis on the web application itself is generated. Also, a benchmark performed at the client can only confirm or deny whether the performance of the web application is as reported by the client. That is why a new method of measuring and analysing the JavaScript performance of a web application need to be developed.

[3] `http://jquery.com/`
[4] `http://dojotoolkit.org/`
[5] `http://prototypejs.org/`
[6] `https://angularjs.org/`

This thesis contains two chapters with background information and definitions, two chapters that describe implementation efforts and one final concluding chapter. The first chapter introduces the problem and formulates the research questions and scientific context. Then, chapter 2 describes web applications and JavaScript. The third chapter then defines performance and constructs an evaluation method. Chapter 4 proposes a couple of method for performance profiling and describes the implementations. Chapter 5 then describes possible integration methods and implements one of them. The final chapter draws a conclusion for this thesis, adds a discussion and proposes some possible future works.

Chapter 2

Web applications

A web application is any application that runs in a browser [8]. Web applications are downloaded from a server and are often capable of interacting with the server. With the emergence of AJAX (Asynchronous JavaScript And XML [9]) and responsive web design [10, 11], many old-fashioned applications were converted to web applications for use on desktops, tablets and mobile phones, rather than on one specific system.

2.1 General architecture

A web application must be built in a language – or languages – that the web browser understands. Languages like HTML [12, 13], CSS [14] and JavaScript [15] are widely used for the development of web applications. HTML stands for HyperText Markup Language and describes the structure and content of a web page. CSS (Cascading Style Sheets) contain the layout of individual components of the HTML, e.g. text colour, font-size and page width. Together, HTML and CSS describe the layout and content of a web page. Many web applications also use Java applets [16], Flash components [17] and Silverlight[1] to add interactive content. These web applications are nowadays being replaced with web applications built with only HTML, CSS and JavaScript, because these older technologies are often platform-dependent, thus difficult to maintain for running on different devices.

Modern web applications typically use AJAX to communicate with the server without the need of reloading the whole page on every user action. AJAX is used to send an asynchronous request to the server and uses callback functions for handling the response from the server. This response is then interpreted and used for adding content to the page. A web application often depends on these requests for its flexibility. Note that not all modern web applications use AJAX: Java, Flash, Silverlight and page-reloads

[1] http://www.microsoft.com/silverlight/

are still being used.

2.2 Page rendering

To render a web page, a browser consists of two separate engines [18]. The layout engine creates a visual representation of the HTML together with the stylesheets, both in CSS and built-in. The JavaScript engine processes and executes the JavaScript code.

The browser itself manages the Document Object Model (DOM [19]), which holds the contents of the web page in a structured model. This DOM is created from the HTML code of the web page. The layout engine applies the available stylesheets – defined in the CSS – to the DOM to complete the page and to finally display it to the user.

The JavaScript engine interprets and executes the included JavaScript code. To enable the interactive behaviour of a web application, the execution includes the attachment of event handlers to the DOM. With these event handlers, the DOM can be altered by the JavaScript code after the whole page has been rendered and displayed to the user, thus providing interactive behaviour.

Because JavaScript is interpreted and executed within the browser of the end-user, performance also dependens on both the layout- and JavaScript engine at the client-side. Since both engines are usually developed independently from the browsers, browser manufacturers have a choice of which layout- and JavaScript engine to include with their browser. Therefore, all browsers perform differently when rendering or interacting with a web site. Table 2.2.1 gives an overview of the five most popular browsers [20] as well as their layout- and JavaScript engines. For all browsers, the current stable builds are used as reference. For convenience, the version number is also displayed.

			Engines	
Browser	**Version**	**Manufacturer**	**Layout**	**JavaScript**
Chrome[2]	40	Google	Blink[3]	V8[4]
Firefox[5]	35	Mozilla	Gecko[6]	SpiderMonkey[7]
Internet Explorer[8]	9+	Microsoft	Trident[9]	Chakra
Opera[10]	27	Opera	Blink	V8
Safari[11]	8	Apple	WebKit[12]	Nitro[13]

Table 2.2.1: An overview of the five most popular browsers and the engines that are included with them.

[2] https://www.google.nl/chrome
[3] http://www.chromium.org/blink
[4] https://code.google.com/p/v8/
[5] https://www.mozilla.org/nl/firefox/new/
[6] https://developer.mozilla.org/en-US/docs/Mozilla/Gecko
[7] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

Web applications are usually only tested on – a subset of – these five browsers. This is caused by the total market share of these browsers, which is more than 95% [20].

2.2.1 Web application workflow

All web applications are built on the same principles and therefore share a common architecture. When a web application is loaded in the client's browser, the web application starts interacting with the user and the server. Figure 2.2.1 shows a typical interaction architecture for a web application after the main page has been loaded.

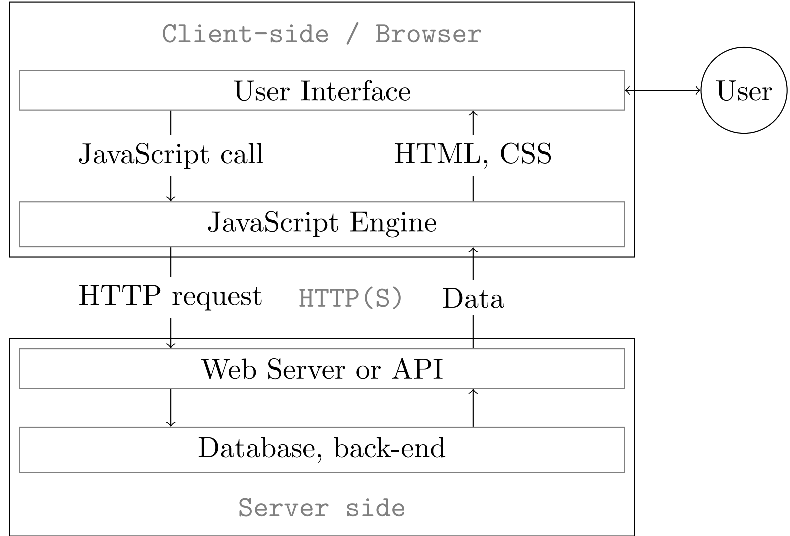

Figure 2.2.1: Architectural overview of a typical web application. Based on Figure 1 by J. Garett [9].

Since web applications are much alike on an architectural level, a common workflow for web applications can be defined as follows. This workflow is based on [9].

1. The browser requests a page from the server by sending an HTTP request.
2. The server returns the requested page to the browser.
3. This page (HTML, CSS) is interpreted by the layout engine, which generates a visual representation of the page. Additional resources, such as scripts, images and style sheets, linked from the initial page are fetched with HTTP requests as well.

[8] http://www.microsoft.com/windows/ie/
[9] https://msdn.microsoft.com/en-us/library/aa741317.aspx
[10] http://www.opera.com/
[11] https://www.apple.com/safari/
[12] http://www.webkit.org/
[13] http://trac.webkit.org/wiki/JavaScriptCore

4. The script engine executes the scripts (JavaScript) included with the web page and attaches handlers to DOM objects.
5. When a user action triggers a handler, the handler function is executed. This function may send an AJAX call to the server or update the DOM itself. If an AJAX request is made, a function is attached to the request to handle the response. Otherwise, the DOM is updated and the workflow ends.
6. The server processes the AJAX request and responds with a result. While processing the request, the server may need to fetch data from a database or an external system. A structured response is sent back to the client.
7. The attached AJAX listener handles the result from the server and updates the DOM.

Steps five to seven are repeated for every action the user requests in which an AJAX call is required. Not all user actions require an AJAX call to the server to update the DOM. If no AJAX call is sent, only step five is repeated for the user action.

2.3 JavaScript

JavaScript [21] is a dynamic, object-oriented scripting language, mostly used to create web applications. It was developed for Netscape in 1996 by Brendan Eich [22, 23]. At first, the language was named LiveScript, which evolved to JavaScript within a year. JavaScript was supposed to incorporate more advanced web applications in the browsers without the need for Java applets.

ECMA (European Computer Manufacturers Association[14]) standardised JavaScript in 1997, making JavaScript one of the implementations of ECMAScript [24]. Other implementations of the ECMA standard are ActionScript[15] and JScript[16]. The latest stable release of JavaScript, version 1.8.5, is compliant with ECMAScript 5. This version will also be the default version for this research.

Like Java, JavaScript code is compiled to bytecode, which is comparable with machine instructions, and then executed on a virtual machine, which acts like a sandbox and offers memory protection. These virtual machines often implement a garbage collector to clean up unused objects, functions and variables, thus freeing memory. This virtual machine is implemented within the JavaScript engine, which most commonly resides within a web browser.

Whereas old-fashioned JavaScript engines were completely integrated in a web browser and were not interchangeable with any other application, modern engines are developed

[14] `http://www.ecma-international.org/`
[15] `http://help.adobe.com/livedocs/specs/actionscript/3/`
[16] `https://msdn.microsoft.com/library/hbxc2t98.aspx`

as standalone applications and are therefore capable to run JavaScript code outside a browser [21]. This evolution also boosted the performance of the JavaScript engines.

Because JavaScript can now be executed outside a browser and performs well on modern systems, more applications are being developed. Node.js[17], for example, is a JavaScript framework designed to run standalone JavaScript applications, or more specifically: servers and services written in JavaScript.

2.3.1 Benchmarks

Since browsers have become more important over the last few years, also the performance of a browser becomes more important. Also, browsers have become more accurate in interpreting and displaying web pages, causing all browsers to display a web page equally good. Therefore, it is increasingly difficult to distinguish the browsers from one another.

Therefore, benchmarks have become important tools to compare browsers: Browser developers nowadays use results of benchmarks to promote their product. Browser performance can significantly affect the user experience and is therefore an important aspect of a browser. Because this is so important, AreWeFastYet[18] automatically monitors the JavaScript performance of different browsers by executing benchmarks on the latest versions.

A benchmark [25, 26] is a program designed to measure the performance of hardware or software. Most benchmarks only yield one or more scores, representing the performance of the system. These scores provide an easy comparison method when comparing the performance of two different systems with each other.

Quite some benchmarks are available for browser testing. Some of them are focused on benchmarking the full browser. These benchmarks measure the performance of the browser in terms of DOM modification, JavaScript execution and page rendering. Examples of such benchmark suites are Peacekeeper[19] and Browsermark[20].

Additionally, there are some pure JavaScript benchmark tools. These benchmarks only measure the JavaScript execution performance of the browsers or stand-alone JavaScript engines. Some of these benchmarks are a direct offspring of a JavaScript engine. Those benchmarks often are being used during development of the JavaScript engine for performance optimisations. Table 2.3.1 provides an overview of the most popular benchmarks for browsers and JavaScript engines available.

The column 'JS engine' specifies what JavaScript engine is also being developed by the same developer as the benchmark. If the benchmark is 'full', the benchmark is meant

[17] `http://nodejs.org/`
[18] `http://www.arewefastyet.com/`
[19] `http://peacekeeper.futuremark.com/`
[20] `http://browsermark.rightware.com/`

for measuring the performance of the whole browser. If it is not, the benchmark only measures JavaScript execution performance.

Name	**Developer(s)**	**JS engine**	**Full**	**Notes**
Benchmark.js[21]	Mathias Bynens, John-David Dalton	-	No	
Browsermark	Rightware	-	Yes	
Dromaeo[22]	Mozilla	SpiderMonkey	No	
JSLitmus[23]	Robert Kieffer	-	No	
Kraken[24]	Mozilla	SpiderMonkey	No	
Octane[25]	Google	V8	No	Successor of the V8 benchmark[26]
Peacekeeper	FutureMark	-	Yes	
SunSpider[27]	WebKit	WebKit	No	

Table 2.3.1: An overview of popular benchmarks for web browsers.

Although there are quite a few benchmarking applications, previous research [27] shows that these benchmarks do not reflect the performance of actual web applications well: Benchmarks run faster with just-in-time compilation [28] techniques, but real-world web applications showed an increased execution time; benchmarks use less anonymous functions than real-world web applications do; the `eval`-function, which is used for executing code from a string, is used much more in real-world web applications. In addition, these benchmarks only measure rather static behaviour of a JavaScript application, whereas real-world web applications deal with interactive user interfaces and unknown behaviour from the side of the user. Therefore, the benchmarks do not represent web applications in terms of performance and do not seem to be suitable as comparator for browser performance for web applications.

2.3.2 Profilers

A profiler [29] is a tool for monitoring the dynamic execution behaviour of a program. A profiler measures the memory usage, execution time and typical instructions during the execution of a program. Therefore, profiling is a form of dynamic program analysis and is often used to check or benchmark programs during development to locate possible bugs or inefficiencies in the source code.

[21] http://benchmarkjs.com/
[22] http://dromaeo.com/
[23] http://www.broofa.com/Tools/JSLitmus/
[24] http://krakenbenchmark.mozilla.org/
[25] https://developers.google.com/octane/
[26] http://v8.googlecode.com/svn/data/benchmarks/v7/run.html
[27] http://www.webkit.org/perf/sunspider/sunspider.html

A profiler provides insight in the execution behaviour of a program for debugging purposes. This analysis is also used to improve the performance or decreasing the memory footprint of an application. An analysis can show which methods consume excessive amounts of memory or CPU-time. With this analysis, a developer can try to improve the function and therefore improve the performance of the application.

Browsers often have built-in developer tools to provide the developers of web applications with a toolbox for debugging web applications. One of the components of such toolboxes is a JavaScript profiler. This profiler is often capable of measuring both the CPU execution time and usage of the heap. Additionally, most browser support external developer tools – including profilers – as well. A popular example for Firefox is Firebug[28].

2.3.3 Static program analysis

Static program analysis [30] is an analysis method that analyses an application without running the application. The analysis is based on the machine code, bytecode or source code of the application. This form of analysis is often performed during development. Software Development Kits (SDKs) usually check the syntax of the code during development and warn the developer if any errors are found before compiling or running.

For JavaScript, numerous forms of static analysis have been implemented. These analyses often cover type checking [31] or security [32]. Static program analysis is also used for estimating code maintainability and performance. This method strongly resembles a performance benchmark for applications of which the application execution is predictable. Since JavaScript is usually event-driven, the application flow of the application cannot be predicted: it is unknown which parts of the code are executed. Therefore, static code analysis is not capable of predicting the JavaScript performance in a web application.

2.3.4 Frameworks

For JavaScript, numerous frameworks have been built, especially for building web applications. These frameworks are all built to increase development speed and obtain better source code in terms of maintainability. Since JavaScript engines can be executed stand-alone as well, some stand-alone server-side JavaScript frameworks, such as Node.js[29], were created.

Most JavaScript frameworks, however, focus on the client-side of web applications. These frameworks add more convenient DOM selectors to JavaScript, add interactive functionality and provide methods for AJAX requests. The most popular client-

[28] `http://getfirebug.com/`
[29] `http://nodejs.org/`

size JavaScript frameworks, according to W3Techs [33], are jQuery[30], Modernizr[31], MooTools[32] and Prototype[33].

The code in listings 2.1, 2.2 and 2.3 all contain equal functionality, but use regular JavaScript, jQuery and MooTools respectively. In each fragment, the object with *id* equal to 'component' is fetched from the DOM and its visibility is toggled. If the object was visible, it will become hidden and vice versa.

```
1 var element = document.getElementById("component");
2 if (element.style.display == "none") {
3   element.style.display = "block";
4 } else {
5   element.style.display = "none";
6 }
```

Listing 2.1: JavaScript code to toggle 'component' without using any framework.

```
1 var element = $("#component");
2 element.toggle();
```

Listing 2.2: JavaScript code to toggle 'component' with jQuery.

```
1 var element = $$('#component');
2 element.toggle();
```

Listing 2.3: JavaScript code to toggle 'component' with MooTools.

Both jQuery and MooTools include an easy-to-use `toggle`-function to toggle components. Also, the code to fetch the element from the DOM is much shorter. Another widely used JavaScript feature is making an AJAX request to the server through the XMLHttpRequest-object. Listing 2.4, 2.5 and 2.6 display how an AJAX-request is done without framework, with jQuery and MooTools respectively.

```
1 var xmlhttp;
2 if (window.XMLHttpRequest) {
3   xmlhttp = new XMLHttpRequest();
4 } else {
5   xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
6 }
7 xmlhttp.onreadystatechange = function() {
8   if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
9     document.getElementById("placeholder").innerHTML = xmlhttp.
        responseText;
10  }
11 };
12 xmlhttp.open("GET", "response.html", true);
13 xmlhttp.send();
```

Listing 2.4: A sample AJAX request sent without the use of any framework.

[30] `http://jquery.com/`
[31] `http://modernizr.com/`
[32] `http://mootools.net/`
[33] `http://prototypejs.org/`

```
$.ajax({
  url: "response.html",
}).done(function(response) {
  $("#placeholder").html(response);
});
```

Listing 2.5: A sample AJAX request with jQuery.

```
var request = new Request({
    url: 'response.html',
    method: 'get',
    onSuccess: function(response) {
        $$("#placeholder").set('html', response);
    }
});
request.send();
```

Listing 2.6: A sample AJAX request with MooTools.

Frameworks provide all sorts of functionality in a structured and more readable way. The resulting code is better readable, maintainable and shorter compared to the code when no framework is used. The overhead caused by the need to download the framework first is very small, since many of these widely used frameworks are already cached by a browser, because they are accessible via a shared CDN (Content Delivery Network).

2.3.5 AngularJS

AngularJS is an open-source JavaScript framework used to develop web applications [34]. It is written in JavaScript and uses common web technologies like HTML, CSS and, of course, JavaScript and focuses on building single-page web applications. AngularJS aims to simplify the development of web applications by separating DOM manipulation from the controllers by using a Model-View-Controller (MVC) architectural pattern.

One of the advantages of AngularJS is its implementation of directives. AngularJS itself contains some built-in directives, but the developer can also create his or her own directives. A directive is an implementation of a custom HTML element. This can be an element, attribute, class or comment and basically extends native HTML. Directives allow the developer to write small parts of functionality that can be reused in other web applications. It is also possible to override the browser-specified behaviour of an HTML element.

Additionally, AngularJS clearly separates models, views and controllers. Controllers are defined within `script`-tags or in external JavaScript files. The view is represented with the HTML-document, to which data from the model can be bound. The model holds all variable data for the page and can be accessed from anywhere within the same scope. AngularJS features two-way data binding, so that when the view changes, the model changes accordingly and when the model is changed, the view updates.

Two-way data binding is provided with dirty-checking. A so-called digest loop checks whether a field has been modified – is dirty – and updates the values that depend on it. This includes other objects or fields within the model and the view in which it is all displayed. In modern browsers, this loop is being replaced with the `Object.observe()`-method[34] for the sake of performance. This method provides native browser support for observing objects instead of using a custom-made dirty-checking loop, which is some sort of busy waiting.

AngularJS also offers a method to create services to add more persistent functionality to the web application. These services are loaded once and only when they are required. AngularJS includes a number of built-in services, such as the `$http`-service, which supplies easy-to-use methods to communicate with the back-end of an application through AJAX-calls.

An AngularJS app starts loading once the DOM is completely loaded by the browser. It then creates the components required to run, e.g. a `$compile`-service and a `$rootScope`. The `$compile`-service finds all directives and links them to the `$rootScope`. Once the app is running, AngularJS catches events with its own digest loop and handles them.

Listing 2.7 shows a sample application written in AngularJS. This application contains some basic functionality of AngularJS. The application requests the user to enter his or her name and displays this name in a heading. When the button after the input field for the name is clicked, an alert message is displayed to the user. If a name is provided, a table containing all roles as defined in the controller is shown. These roles are ordered by name, descending. If the role is admin (admin field is set to true), the second column displays 'yes', otherwise 'no'.

```
1  <!DOCTYPE html>
2  <html data-ng-app="app" data-ng-controller="Test3Controller">
3  <head lang="en">
4      <meta charset="UTF-8">
5      <title>AngularJS Example</title>
6      <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/
          angular.min.js"></script>
7  </head>
8  <body>
9  Please enter your name:
10 <input type="text" data-ng-model="fullname">
11 <button data-ng-click="clickEvent()">Click here!</button>
12 <h3>Welcome {{fullname}}!</h3>
13 <table data-ng-show="fullname.length > 0" border="1" cellspacing="0"
      cellpadding="5">
14   <tr>
15     <th>Name</th>
16     <th>Admin</th>
17   </tr>
18   <tr data-ng-repeat="role in roles | orderBy:'-name'">
19     <td>{{role.name}}</td>
```

[34] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/observe

```
20      <td>{{role.admin ? "yes" : "no" }}</td>
21    </tr>
22  </table>
23  <script>
24  var app = angular.module('app', []).
25      controller('Test3Controller', function ($scope) {
26        $scope.roles = [
27          { name: "Administrators", admin: true },
28          { name: "Teachers", admin: false },
29          { name: "Students", admin: false }
30        ];
31
32        $scope.clickEvent = function() {
33          alert($scope.fullname + " clicked the button!");
34        };
35      });
36  </script>
37  </body>
38  </html>
```

Listing 2.7: A small example application written with AngularJS.

Below, the relevant lines for AngularJS in this web application are briefly explained. The code itself can be copied into a file and tested in any browser.

Line 2 tells AngularJS that the whole document should be handled as an app with name 'app', using the 'Test3Controller' controller.

Line 6 includes the AngularJS framework into the web application.

Line 10 defines a model *name* within the `$scope`.

Line 12 contains an AngularJS expression, which can be used by adding {{*expression*}} to the HTML-document. This line binds the *name*-model from the `$scope` to the view and displays its value.

Line 11 binds a click event to the button. `clickEvent()` is now triggered when the button is clicked.

Line 13 uses the `ng-show` directive, which decides whether the table is shown or not. This depends on the length of the value of the *name* model.

Line 18 initialises a loop over all elements in *roles* and orders them. This `tr`-tag is repeated as well in the loop.

Lines 19-20 contain the body of each row. The name of a *role* is displayed in the first cell, whether this *role* has administrative rights in the second.

Line 24 creates a new module called 'app'. This module is then linked to the app directive as created on line 2.

Line 25 creates a controller called 'Test3Controller' which is used by the application and defines a `$scope`.

Lines 26-30 define an array of *roles* within the `$scope`.

Lines 32-34 define a function `clickEvent()` that displays an alert message.

Figure 2.3.1 shows the web application as it is interpreted and displayed by Google Chrome. In this figure, 'Jimmy' has been entered as *name*. This *name* is also visible in the welcome message. With two-way data binding, the welcome message changes when modifying the *name* value instantly. Also, the condition whether to show the table or not is re-evaluated when the *name* is changed. This re-evaluation causes the table to show or hide when entering a *name*. Figure 2.3.2 shows the interface when the button is clicked.

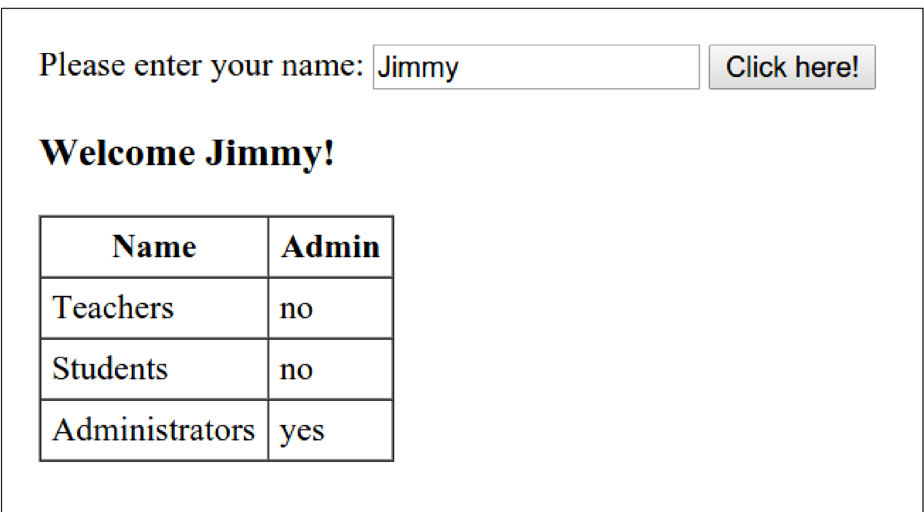

Figure 2.3.1: Our example AngularJS application as it is displayed by Google Chrome.

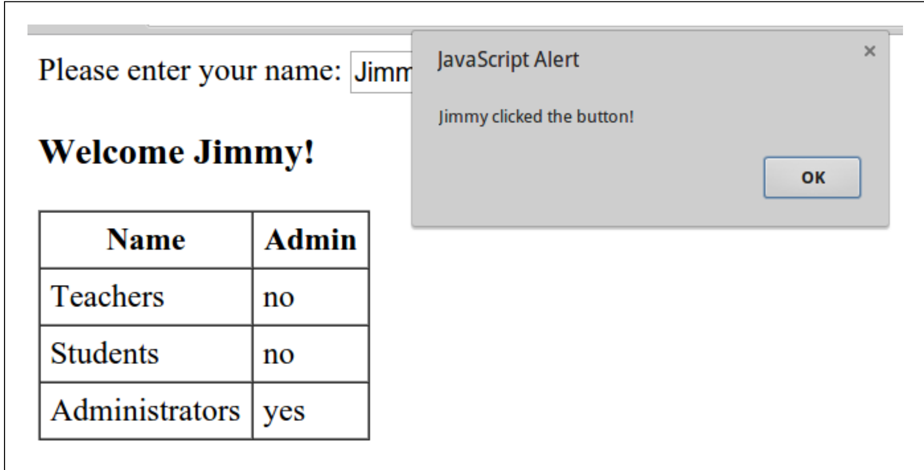

Figure 2.3.2: Our example AngularJS application when the button is clicked.

2.4 Interact CTG

Interact CTG[35] is an application that provides an interface to a cardiotocograph (CTG). It is being developed and maintained by Topicus. The application can be connected to Topicus' Electronisch Patiëntendossier (EPD) for data sharing, but can also be used as a stand-alone application.

Interact CTG receives and stores data from a CTG and displays the data in a web interface. Within the interface, it is possible to save traces and add comments. It is also possible to answer predefined questions regarding the state of the patient.

2.4.1 CTG

A cardiotocograph [35] is a measuring device to monitor the foetal heart rate and uterine contractions. The device is used during childbirth as well as for regular checkups on the foetus during pregnancy. CTG devices usually print the results of their measurements on paper or display it on a screen. For Interact CTG, the signals are captured, converted and stored in a database. These measurements are used to check the health status of the child, e.g. by estimating the oxygen level based on the heart rate.

2.4.2 Architecture

Interact CTG consists of two REST [1] APIs and a database. One API is the Back-end API, which receives and processes all data measured by the CTG. The second API is the Front-end API, which handles the requests from the web application.

The output of the CTG is converted to XML [36] format by a custom-built conversion module and then sent to the Back-end API as JSON [37]. This API parses and stores the data to the database. Since the measuring frequency of the CTG is 4 Hz, a set of measurements is only sent to the Back-end API every two to three seconds.

The Front-end REST API serves content to its clients. This API receives requests from the web browser in which the web application is loaded, executes or queries them and replies in JSON format. This API is also connected to the database, but queries data as well as it stores or modifies data. The web application itself can be hosted within the same application as the front-end API or in an external HTTP server, such as Apache[36].

Both REST APIs are built using Spring[37]. Spring is an open source Java[38] framework

[35] `http://www.topicus.nl/zorg/producten/product/interact-ctg`
[36] `http://httpd.apache.org/`
[37] `http://spring.io/`
[38] `http://www.oracle.com/nl/java/overview/index.html`

with extensions to JavaEE[39]. The APIs both run on the Jetty[40] web server. For handling large amounts of data, MongoDB[41] is used as database server. The web application communicates with the front-end API with JSON over HTTP(S).

The Interact CTG application can be deployed as a standalone application or as an additional module to Topicus' EPD solution. With both applications installed, data can be exchanged between the applications to provide a more complete overview of the status and history of a patient. The EPD stores its data in its own database and is capable of exchanging data over Java Messaging Services (JMS[42]). Figure 2.4.1 shows all components of Interact CTG and its connections, including the EPD.

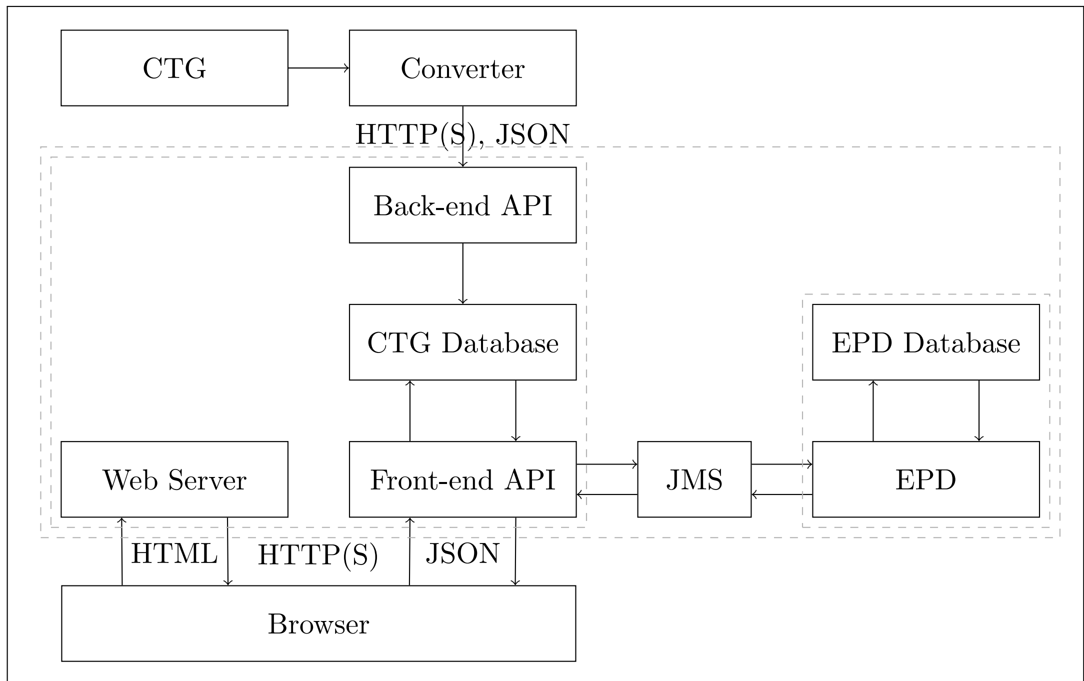

Figure 2.4.1: Detailed overview of the architecture of Interact CTG.

Both databases do not necessarily need to be installed on the same machine as the applications run from. Also, the web server may run on another machine. It is valid to run all applications and databases from a single machine as well.

The web application fetches new data from the Front-end API every two seconds. Because the CTG converter sends data to the Back-end API every two to three seconds, the total theoretical latency is four to five seconds. In practice, some additional time is lost on processing and the network. The absolute minimum time consumed for a

[39] http://www.oracle.com/technetwork/java/javaee/overview/index.html
[40] http://www.eclipse.org/jetty/
[41] https://www.mongodb.com/
[42] http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html

measurement to reach the CTG web application is four seconds, while the maximum is anything above four seconds that is still acceptable by the configuration. This is, however, acceptable, since the network- and processing latency is usually not very high and measurements are used to detect patterns instead of direct threats.

Chapter 3

Profiling performance

To analyse the client-side performance of web applications, first a definition of performance should first be established. Section 3.1 defines client-side performance within the context of a web application. Section 3.2 describes the properties to test this method on. Finally, Section 3.3 describes how these properties are used to evaluate the performance analysis method.

3.1 Definition of client-side performance

3.1.1 Classical definition of performance

Since performance itself is not a generally applicable metric, the performance of a computer system is typically determined from a set of metrics for performance. There are numerous performance metrics that measure parts of the performance of a computer system or the performance of an application as experienced by the user. However, not all metrics are usefully applicable to any system. Which metrics are useful depends on the purpose and nature of the system and is to be determined for each system specifically. Widely used performance metrics for both hardware- and software systems are listed below [38, 39].

Execution time
: The execution time is the time it takes a program to complete its task. This task may be a complete execution of the program or a smaller sub-task. The execution time is typically expressed in (milli)seconds, but can also be expressed in clock-ticks to provide a more CPU-independent unit measure.

Throughput
: Throughput is typically expressed by a certain amount of successfully completed jobs per second. Examples are bits per second, documents per second or redirects

per second. This metric directly relates to the speed of the hardware the software runs on.

Resource consumption
Resource consumption is the amount of hardware resources used at a certain point in time. Examples are CPU-utilisation, memory footprint or bandwidth usage. Typically, a lower resource consumption is better.

Availability
Availability is used to express the amount of time a computer system is actually available for clients in relation to the amount of time the system is expected to be available by the customer. It is also used outside the context of computer systems and basically tells something about the reliability of the system. The higher the availability, the more stable the system is. For hardware, availability is often influenced by hardware failures, such as broken components and power failure. Within the context of software, bugs are of significant influence on the availability.

Bandwidth
Bandwidth expresses the maximum data flow in and out of a computer system or component. It is usually expressed in Mbps (Megabits per second) or Gbps (Gigabits per second). Bandwidth is a typical metric for hardware performance, since software bandwidth is determined by the hardware bandwidth. In the context of a network, bandwidth typically specifies the maximum speed of communication between different machines. On an internal level, bandwidth specifies the maximum speed of communication between hardware components, such as the CPU and memory.

Response time
Response time is the time taken by a round-trip from the client to the server. This includes the request being made, sent, processed and a response being sent back again. The response time is typically expressed in milliseconds, but may also be expressed in any other time unit. Response time is a metric which is typically used in the context of network applications, but response time may also be used outside the context of a network: in the context of one machine, the response time equals the execution time. A lower response time typically results in a higher performance.

These metrics are used to define performance for computer systems. Often, the general performance of a system is defined by using multiple metrics. Additionally, these metrics are used to define minimum requirements for computer systems. For example, a company may require an availability of at least 99% for a computer system. Each system may have other combinations of performance metrics that are important.

3.1.2 End-user definition of performance

For web applications, user experience is one of the most important properties used to determine the usability of the application. Especially for interactive web applications, it is important to deliver a smooth user experience. The user experience is highly affected by the time taken to update the page with new content, also known as waiting- or loading time [40, 41]. Longer loading times typically lead to a decrease in the user satisfaction.

From the perspective of a user, a web application provides the user with visual information through a screen. This screen is typically the only way to access information that is of interest to the user. By requesting new information from the server and loading it into the screen, the user navigates through the web application and its data. While loading new content, the user typically waits until loading is completed and the content is displayed on the screen. When this process takes more time than expected by the user, this negatively affects the user experience.

The user experience is often closely related to the perception of performance of a web application by a user. A fast-loading web application is believed to be a good performing application, thus having a high performance. As a result, for the end-user performance strongly correlates to the speed at which content is presented.

3.1.3 Technical definition

The user-perceived performance depends on a couple of processes invisible to the user. As depicted in Figure 2.2.1, the user only interacts with the user interface, but the actual content is created by the layers below. To fetch new content from the server, the following layers are active.

JavaScript engine
The engine typically handles the user input by calling a sequence of functions upon an event triggered by the user. After sending a request for data to the web service, it usually awaits the response to process it and update the user interface.

Network
The requests sent by the JavaScript engine and received from the web service travel over the network. Since the distance between sender and receiver is typically large, passing through this layer consumes a significant amount of time.

Web service
The web service receives, processes and responds to a request from a client. To process a request, the service should sometimes request additional data from a back-end or database and construct a response based on this data.

Optionally: Back-end
When the web service should request data from a back-end or database, this layer

requires some time to receive and process the request and eventually respond to it. If the back-end does not run on the same machine as the web service, some additional time is spent on an additional network layer.

These layers all contribute to the performance as it is experienced by the user, since all layers are processed for every request (implicitly) made by the user. From the perspective of the user, these layers can be categorised as an internal layer (JavaScript engine) and an external layer (network, web service and back-end). The actions in the internal layer take place on the machine of the client, whereas the actions performed in the external layer take place at an external party or server.

Since the focus for this research is on the client-side performance of web applications, the performance of the internal layer must be measured. From the perspective of the internal layer, the external layer acts like a black box en should be measured as such. Therefore it is out of scope to measure throughput, availability or bandwidth at the server-side or in the network. Consequently, time, as both execution time (internal) and response time (external), is selected to provide a useful indication of performance.

To obtain clear directions for the developer, it is convenient that the execution times of parts of the program. Internally, all functions executed are measured to provide the developed some insight on the execution speed. The external performance can be expressed by measuring the response times of the server from the client-side code. By doing so, the server is treated as a black box. Therefore, this paper uses the following definition of performance.

- The execution time of all separate functions and the total execution time of the web applications together form the client-side performance of the web application.
- As part of the client-size performance, also the response time from the server. This response time implicitly includes network traffic, latency and processing time at the server-side.

Performance measurements at the client-size provide the required guidance for a developer to find and solve performance issues in the front-end. This should lead to better – or faster – web applications without the explicit need for help from the customer to locate an issue.

3.2 Performance analysis method properties

To be able to compare different performance analysis methods, a set of properties is defined. This set of properties is used for determining the overall quality of a method and helps to find the best suitable method for an application. The properties used in this paper are the following.

Accuracy
The accuracy of the method is the most obvious property to test for. The method

should provide information to the developer that is correct and helps to locate the performance issue in the source code. A fixed set of sample web applications functions as benchmark for this accuracy. The sample applications all contain known performance issues, which should be located by applying the method. More about these sample applications is described in Section 3.3.1.

Impact
In order to correctly measure the performance of the web application itself, the analysis should not have a significant impact on the performance, thus on the results. If the analysis does have an impact on the performance, the impact must be constant: for each item that is measured, a constant amount of analysis time must be subtracted. By doing so, the resulting performance is not affected by the overhead of the analysis itself.

Usability
The method should be easy to use or trigger by both the user and developer or administrator. The customer should not be asked to perform series of actions in order to create a useful performance analysis on his or her system. The developer or administrator should not do this either. The usability is used to grade the amount of work that is expected from both the customer and developer or administrator to deliver a useful performance analysis.

Portability
For software companies, it is important to have a set of standardised methods that can easily be applied in different projects. The portability grades the amount of work needed to be done to use the performance analysis method for another web application.

These four properties cover the most important aspects of the method to analyse performance of a web application.

3.3 Method evaluation

In order to grade methods for performance evaluation, the properties described in section 3.2 are used. This section describes how these properties will be graded to compare performance analysis methods. A method is graded for each of these four properties, after which a total grade for the method can be calculated. This grade is a weighted average of the four sub-grades. Definition 1 displays the formal definition.

Definition 1. $G_{total} = \frac{2}{5} \cdot G_{accuracy} + \frac{1}{10} \cdot G_{impact} + \frac{3}{10} \cdot G_{usability} + \frac{1}{5} \cdot G_{portability}$

In Definition 1, G_{total} expresses the total score as a grade between 0 and 10. The grade is constructed from the grades of the four properties described in this section. The

weights of the components can be changed to match the wishes of the customer or developer testing the analysis method. The following list explains the weights.

- 40% weight for accuracy, because it is the most important goal of the analysis. If the method fails to collect usable information for the developer, the method should not be considered to be a good method.
- 10% weight for impact, because the method should not influence the performance of the web application significantly, but it is more important to collect correct results. Additionally, the performance of the application may drop significantly, but only if the performance is decreased by a constant factor. As long as the performance decrease is a constant time, the impact only accounts for 10% of the total grade.
- 30% weight for usability, because the customer should not be requested to perform a whole set of actions to analyse the performance and submit the results to the developer. Additionally, the developer should not be asked to set up a whole testing environment either. Therefore it is acceptable if the customer and developer should only perform a limited set of actions.
- 20% weight for portability, because the developer should not spend large amounts of time on a method which is used occasionally. Since only manually entered lines of code are considered in this property, this directly reflects the effort for the developer to include the method in a new project. Because this action is only performed once for each project, the property's weight is less than those of actions that are repeated more often.

Additionally, a developer is free to add requirements to a method. For example, a method that receives an insufficient grade (< 5) for either one of the properties may be marked as unsuitable. For this research, a method is not acceptable if any grade is lower than 5 or if the analyses from clients cannot be processed automatically, e.g. when manual contact by phone or email to send the analyses to the developer is required.

3.3.1 Accuracy

Because browsers use different JavaScript engines, web applications written in JavaScript may perform differently across browsers. In order to establish a measure for accuracy, a test is developed. This test contains pieces of JavaScript code that is known to perform quite different on specific browsers. If the performance analysis method is able to direct the developer to all these weaknesses and assigns correct priorities to performance issues, the accuracy is high.

In order to develop such a test application, the performance differences between browsers need to be located. To do this, we used the benchmarks that are available on the web and evaluated the results. The benchmarks were all executed in a virtual machine running on Windows 7 SP1 with 1GB of memory and one CPU-core (Intel Core i5-3337U,

1,7∼2,7 Ghz). Each benchmark was executed three times. Appendix A explains the benchmarks that were executed and contains the results achieved. The benchmark results for Dromaeo, Sunspider, Kraken and Octane can be found in Appendix A.1, A.2, A.3 and A.4 respectively.

Based on these results, some tests are selected for each browser to measure the accuracy. The lists below show the benchmark topics that were significantly slower for specific browsers. If a topic is listed for a browser, this the browser did not perform well on this test.

- Chrome
  - String, Sunspider
  - 3D, Sunspider
- Firefox
  - Arrays, Dromaeo
  - Json, Kraken
  - Regexp, Octane
  - Mandreel, Octane
- Opera
  - String, Sunspider
  - Date, Sunspider
  - 3D, Sunspider
- Internet Explorer
  - Json, Kraken
  - Date, Sunspider
  - Regexp, Octane
  - Control Flow, Sunspider

The benchmark results clearly show similarities between Chrome and Opera, caused by the fact that both browsers use the Blink and V8 engines. A sample application containing some of these elements is constructed to measure the accuracy of a method constructed.

Accuracy test construction

The benchmarks described above revealed some JavaScript subjects that would show clear performance differences between browsers. From these subjects, five commonly used subjects were selected and implemented in an application comparable to the benchmarks used to find these subjects. All tests were aligned around an execution time of 600 ms in Google Chrome 41.0.2272.118 on the developer machine. All tests were then run on the virtual machine which was also used to find the subjects for the accuracy test.

Arrays

The array test (Appendix B.1.1) is based on the array test as found in Dromaeo and initialises arrays by using two different methods and runs `splice`, `shift`,

`unshift`, `push` and `pop` operations. Every test is repeated 10.000 times with an array of length 30.000.

Strings
The string-test (Appendix B.1.2) initialises string objects and performs a number of string methods. Strings of length 220 are constructed from words of length 11, which are generated randomly by `concatenation`. Even characters are set to `uppercase` characters, odd characters are set to `lowercase`. Finally, the word is `trim`med before being concatenated with the long word (220 characters). For every long word, the `split`, `substring` and `charAt` methods are applied. This process is repeated until a total of 300.000 (small) words is generated.

Dates
The date-test (Appendix B.1.3) is based on the date-test as found in Sunspider and extends the JavaScript Date object with a `formatDate` method. This method is very similar to the PHP implementation of `date`[1]. A date is formatted in a way as specified by the user by passing a string. This argument contains characters corresponding to elements of the date to be printed. For example, 'H' corresponds to the hour. This test initialises a Date object and starts a loop. Every cycle of this loop formats the date and adds a large number of milliseconds to it. This loop contains 35.000 cycles.

JSON
The JSON-test (Appendix B.1.4) consists of a large JavaScript array of around 70 kB. This array contains 65 objects with 22 attributes each. The function first performs the `stringify` operation on the object for 700 times. Then, the resulting string is `parse`d another 700 times.

Regular expressions
Before the regular expression test (Appendix B.1.5) starts, a random string of one million characters is generated. Hereafter, eleven regular expressions are executed on the random string for 100 times each.

Results

To verify whether these tests are suitable to test the accuracy of a method for analysing the front-end performance of a web application, the tests were executed as benchmarks on the system used in Section 3.3.1 to find performance differences between browsers. The same (versions of) browsers are used and the hosting machine remains unchanged.

Figure 3.3.1 displays the normalised execution times as measured by the accuracy evaluation script. Lower is faster. The averaged values measured are displayed in Appendix B.2.

[1] `http://php.net/manual/en/function.date.php`

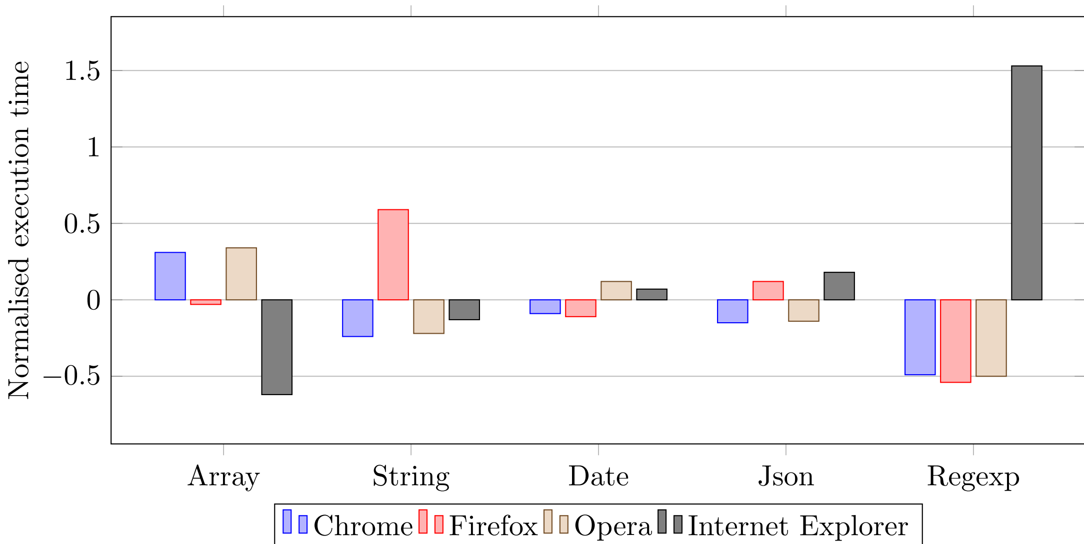

Figure 3.3.1: Normalised results for the accuracy test for evaluation. Lower is faster.

The results of the accuracy test confirm the results from the benchmarks used to identify the test subjects. The results clearly show performance differences between the browsers, although these differences are not always as expected by the previous benchmarks on which these tests are based. However, the results are very useful to test the accuracy. From the results, the following statements can be formed.

- Internet Explorer is very fast at array operations.
- Firefox is slow at string operations.
- Opera is slower than Chrome at date operations.
- Chrome and Opera are fastest at JSON operations.
- Internet Explorer is slow at executing regular expressions.

To check a method for its accuracy, these five statements should be verified. It is not important to what degree the method matches the execution times of this reference test as long as the statement can be confirmed from the results. The number of confirmed statements for each method determines the grade for accuracy that is given. This number shall be called $n_{confirmed}$, where $0 \leq n_{confirmed} \leq 5$.

Since accuracy is an important property of the method evaluation, a grade for this property cannot be calculated linearly. A method that can confirm only three out of five statements should already be graded as insufficiently accurate. Therefore, an exponential decrease in grade is used to create a definition for the accuracy.

Definition 2 (Accuracy)**.** $G_{accuracy} = \sqrt[5]{11^{n_{confirmed}}} - 1$

3.3.2 Impact

The impact can be measured by comparing the execution time of the application with and without performance analysis. Since the execution time is the main metric for the performance analysis, this value alone should be sufficient to calculate the impact.

Definition 3 (Impact)**.** $G_{impact} = 10 - \frac{10 \cdot (t_{with} - t_{without})}{t_{without}}$

In Definition 3, G_{impact} is the grade for the impact on performance of the method, where a negative number for G_{impact} ($G_{impact} < 0$) is set to $G_{impact} = 0$. t_{with} is the execution time of the web application in milliseconds with the performance analysis running. $t_{without}$ is the execution time of the application without the analysis. $t_{without}$ can be measured by using the built-in profiler of a browser. The constant factor 10 is used to express the grade degradation factor: the grade would be 0 when the application takes two times as much time to execute compared to the version in which the analysis method is not applied. A higher constant factor would lead to faster grade degradation, e.g. a lower grade for an equal impact.

The impact can only be calculated once the method is applied to a project. Since each viable method is implemented only as a prototype, it would be unnecessary to apply the method to a whole project. Therefore, each viable method is applied to a small part of the CTG project. One of the most JavaScript intensive functions built by Topicus itself is a jQuery plug-in for rendering CTG graphs. This plug-in will therefore be the part of the project to which the method is applied.

The impact is measured upon opening the *Dashboard* of the CTG application, in which four graphs are rendered. For each browser, the *Dashboard* is opened eight times, after which the results are measured across all browsers to calculate the average execution time of the rendering of the graphs. This process is repeated twice: once without the analysis method, once with the method implemented.

3.3.3 Usability

The usability of the method is expressed by a set of categories. The developer or customer should determine or estimate the amount of work required to perform a useful performance analysis using the method developed. The amount of work corresponds to one of the categories which results in a single grade. Table 3.3.1 lists and defines the categories that express the usability.

This table should be used to estimate the usability for the end-user of an analysis method and result in a single grade.

Grade	Title	Amount of work (m) in minutes
0	Large effort	$m > 60$
2	Quite large effort	$20 < m \leq 60$
4	Moderate effort	$10 < m \leq 20$
6	Little effort	$5 < m \leq 10$
8	Very little effort	$0 < m \leq 5$
10	No effort	$m = 0$

Table 3.3.1: Usability categories definition for the performance analysis methods.

3.3.4 Portability

Portability should be measured by estimating the amount of work a developer has to do to include the performance analysis method in a new project. Since Lines Of Code (LOC) is a commonly used metric to estimate the size of a project or the amount of work that has been put into a project, portability will be measured by using LOC as a metric, also resulting in a single grade. For JavaScript web applications, only the number of lines of JavaScript code will be considered in this metric.

Definition 4 (Portability)**.** $G_{portability} = 10 - \frac{25 \cdot (LOC_{with} - LOC_{without})}{LOC_{without}}$

In Definition 4, $G_{portability}$ is the grade for portability, where a negative number ($G_{portability} < 0$) is set to $G_{portability} = 0$. LOC_{with} is the number of lines of code for the application with the addition of the performance analysis method, where $LOC_{without}$ is the number of lines of code without the method. Only the lines of code that should be manually typed by the developer should be counted, because only those lines of code are to be typed by the developer. Less lines of code to add to a project results in a higher grade, because it is typically less work for the programmer, resulting in a higher grade. The factor 25 expresses the grade degradation speed. The larger the number, the more expensive additional lines of code will be in terms of the grade: 40% additional lines of code results in a 0 as grade for portability. This degradation can be manipulated by selecting a different factor.

The portability is, like the impact, calculated from the application of the analysis method on the graphs plug-in for the CTG project. The number of lines of code with and without the method are counted and used to calculate a grade.

Chapter 4

Methods for performance profiling

This chapter describes several methods to analyse the client-side performance of a web application. For every method, a pre-evaluation is done. If this is done with positive result, the method is implemented and post-evaluated.

4.1 Manual profiling

Manual profiling is the most basic method to analyse the client-side performance of a web application. The built-in JavaScript profiler of the browser is used to measure the performance of the application. The results are saved and sent to the developer, who can use the results to locate the performance issue. The following list shows a general workflow for this method.

1. The user reports a suspected performance issue to the developer.
2. The developer sends an email explaining the procedure and a document explaining how to profile the application using the built-in profiler.
3. The user uses the profiler to profile the application, then stores and sends the result to the developer.
4. The developer imports and analyses the profile.

4.1.1 Pre-evaluation

Accuracy: 10

Because all JavaScript code is executed in the JavaScript engine within the

browser, the profiler is able to directly measure execution time of the JavaScript functions. Therefore, the results of this method are expected to locate performance issues correctly, consequently grading it with 10 points.

Impact: 10
Since the profiler is a layer built in the JavaScript engine, it is expected to be one of the fastest possible methods, resulting in 10 points.

Usability: 2
The total amount of work is estimated between 30 to 60 minutes total. The user must first learn how to work with the profiler, which takes most of the time. Hereafter, a profile should be made, which cannot be done during regular working activities, because the parallel use of the profiler significantly decreases working efficiency. Usability is therefore graded 2 points.

Portability: 10
From the perspective of the developer, this method is applicable to any project regarding a web application. No work from the developer is required during development, because it is a standalone method to analyse the performance.

The estimated final grade for this method is $G_{total} = 7,6$. However, this method probably violates one of the additional requirements: all grades must be higher or equal to 5. Since the estimated grade for usability is 2, this method is probably not suitable for performance analysis.

Additionally, the assumption that all browsers contain JavaScript profilers does not hold. Mobile browsers, for example, usually do not include a JavaScript profiler. Also, it is difficult to automatically process all profiles recorded by users, because every profiler may store its profile in a different format. Consequently, this method appears not to be suitable as method to analyse client-side performance of web applications.

4.2 Manual logging

Manual logging involves the addition of code to automatically construct an execution-profile for the web application. The developer should add one logging statement at the beginning of a function and one at the end. These statements should start and end the measuring of the function and store the result in a profiler-object.

Since enabling the profiler for all users of the web applications would eventually lead to inconvenience for the users, this method requires a more targeted approach towards the users. The resulting code after adding this method would be larger and therefore slower in execution than the code without this method. Therefore, also an interface should be created which allows the developer to switch on the profiler for a specific user. Additionally, a function to send the analysis to the developer should be developed. A typical workflow for this method follows.

1. The user reports a suspected performance issue to the developer.
2. The developer replies and enables the profiler for the user.
3. The user uses the web application as usual, results are automatically sent to the developer.
4. After some time, the developer disables the profiler again and analyses the results.

4.2.1 Pre-evaluation

Accuracy: 10
Since all functions are measured by adding statements to the function itself, it is expected that this method can identify all performance issues within the code.

Impact: 7
Because this method involves the addition of two methods per function, the execution time is likely to increase. However, the methods to be executed for each function call should not be computation intensive, resulting in an estimated increase in execution time of 30%.

Usability: 8
Both the user and developer have very little work to do to perform such a performance analysis. The estimated time is less than 5 minutes, since it only involves a simple email conversation and turning a feature on and off for one specific user.

Portability: 5
Manually adding lines of code for performance analysis to every function within a project requires some additional effort from the developer. Assuming a function contains an average of 10 lines of code, the total amount of lines is increased by 20%.

The estimates final grade for this method is $G_{total} = 8,1$, which is slightly higher than manual profiling. This method also meets the additional requirements: all grades are equal or higher to 5 and no manual transmission of analysis results are required.

4.2.2 Implementation

Manual logging is implemented by adding a JavaScript package to the web application, in which a *Profiler*-object is created. The *Profiler* both holds all profiles of functions and is used to start and stop profiling of a function. The data structure of the resulting profile is tree-like. Functions called during the execution of others are displayed as child-functions and also account for parts of the total execution time for their parent function.

To profile specific functions within the application, `start` and `stop` methods from the profiler should be called. `Start` requires a single argument containing the name of the function. This is used to identify the function being called to the developer. The profiler calculates the execution time of the function and stores the function-profile in the tree.

Listing 4.1 shows a small – partial – implementation of a confirmation dialog in JavaScript. This code contains statements to profile the functions executed. Every function, including the anonymous functions, starts with a function call to the profiler to notify the profiler a new function has started executing. At the end of the function execution, the profiler is again notified about the function being terminated.

```
1  function showConfirmDialog(message, okCallback, cancelCallback) {
2    profiler.start('showConfirmDialog');
3    document.getElementById('okButton').onclick = okCallback;
4    /* --- do stuff --- */
5    profiler.stop();
6  }
7
8  function hideConfirmDialog() {
9    profiler.start('hideConfirmDialog');
10   document.getElementById('confirmDialog').style.display = 'none';
11   profiler.stop();
12 }
13
14 function deleteItem() {
15   profiler.start('deleteItem');
16   showConfirmDialog('Are you sure you want to delete this item?',
         function() {
17     profiler.start('anonymous function 1');
18     alert('Deleted!');
19     hideConfirmDialog();
20     profiler.stop();
21   }, function() {
22     profiler.start('anonymous function 2');
23     hideConfirmDialog();
24     profiler.stop();
25   });
26   profiler.stop();
27 }
28
29 deleteItem();
```

Listing 4.1: A sample application that uses the profiler.

The application contains three main functions, which are basically explained below.

showConfirmDialog receives a message and two callbacks from its caller. The function injects the message into the DOM and attaches the event handlers to the *onclick*-events on the buttons. Then, it sets the dialog display mode to 'block'.

hideConfirmDialog hides the *confirmDialog* by setting its display mode to 'none'.

deleteItem calls the `showConfirmDialog` and defines the anonymous functions which are attached to the buttons within the dialog. The OK-button displays an alert-message with the text 'Deleted!' and hides the dialog; the Cancel-button only hides the dialog.

After defining these functions, the `deleteItem`-function is called by default. When the execution of this code completes, the profiler can be closed, after which the profile can be displayed or saved to a database.

Since this application does not contain large amounts of functionality, the profile recorded from running the application is small. Profiles are returned in the JSON-format in a tree-structure and contain all function calls with their execution times and possible functions called within. A possible output for the sample application in Listing 4.1 is displayed in Table 4.2.1.

Main	6562 ms execution time, of which 5631 ms idle time
`deleteItem`	5 ms
`showConfirmDialog`	3 ms
`anonymous function 1`	925 ms
`hideConfirmDialog`	0 ms

Table 4.2.1: Possible result from the profiler.

From this profile, it is clear that `deleteItem` is executed first. This function calls `showConfirmDialog` before returning, which implies the execution time of `showConfirmDialog` is included in the execution time of `deleteItem`. The callback-function attached to the OK-button is eventually executed, which calls `hideConfirmDialog`. Some fast functions execute too fast for the profiler to accurately measure. These functions are typically reported to execute 0 to 1 millesecond, since 1 millisecond is the most accurate a profiler written in JavaScript can be. This is limited by the maximum accuracy of the built-in *Date*-object in JavaScript.

4.2.3 Post-evaluation

To establish whether this method is suitable to analyse the performance of a web application, the method is implemented in the accuracy test. This implementation is graded, based on the four properties defined.

Accuracy: 10

Figure 4.2.1 displays the results of the accuracy test for this implementation (Appendix B.3). The results show a clear similarity with the results achieved in the reference benchmark. The five identifying accuracy properties as described in Section 3.3.1 should be checked to calculate a grade for accuracy.

- Internet Explorer is very fast at array operations: confirmed.

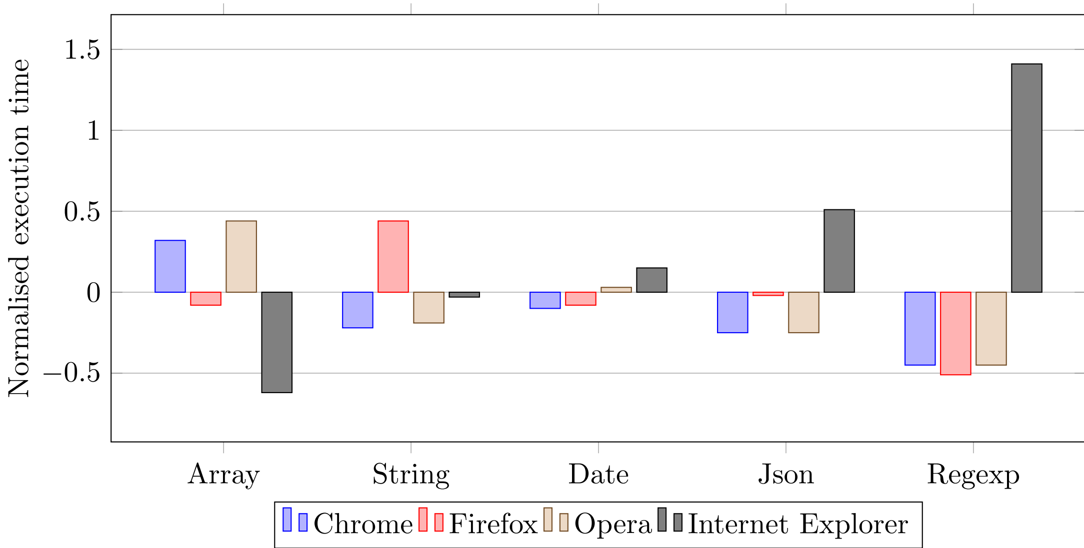

Figure 4.2.1: Normalised results for the accuracy test for evaluation of the Manual Logging method. Lower is faster.

- Firefox is slow at string operations: confirmed.
- Opera is slower than Chrome at date operations: although the difference is small, it can be confirmed.
- Chrome and Opera are fastest at JSON operations: confirmed.
- Internet Explorer is slow at executing regular expressions: confirmed.

Since all five statements can be confirmed by the accuracy test, the resulting grade is a 10.

Impact: 5,9

The impact is calculated by measuring the total execution time of the graphs being drawn to the screen in the CTG application with and without performance analysis implemented. Table 4.2.2 displays the average execution times across the browsers for each test, including the sum of execution times. The execution time difference is expressed in the column 'Difference', which is based on the execution time without analysis method. The grade for Impact can than be calculated as follows: $G_{impact} = 10 - \frac{10 \cdot (496 - 353)}{353} = 5,9$.

Usability: 8

As long as the process of sending and receiving the profile which is created by the profiler is automated, this profiler requires little effort from both the developer and the user. The amount of work to be performed together is expected to be less than 5 minutes, resulting in $G_{usability} = 8$.

	Without	**With**	**Difference**
Chrome	209,2	351,0	67,78%
Firefox	292,6	387,8	32,54%
Opera	235,6	348,2	47,79%
Internet Explorer	673,8	895,2	32,86%
Average	352,8	495,6	40,46%

Table 4.2.2: Total execution time of the execution test with and without the performance analysis.

Portability: 8,4

After counting the manually written lines within the CTG project after implementing this performance analysis method and comparing it with the number of lines without the method, a grade for portability can be calculated. Table 4.2.3 shows the number of lines of code before and after the implementation of the profiler in the project.

graph.js	**Lines of code**
Without profiler	1297
With profiler	1381
Difference	84

Table 4.2.3: The number of lines of code within the scope with and without the performance analysis method.

The grade for portability can now be calculated as defined: $G_{portability} = 10 - \frac{25 \cdot (1381-1297)}{1297} = 8,4$.

With these four grades the final grade for this method is calculated: $G_{total} = \frac{2}{5} \cdot 10 + \frac{1}{10} \cdot 5,9 + \frac{3}{10} \cdot 8 + \frac{1}{5} \cdot 8,4 = 8,7$. Since all sub-grades are equal or higher 5, the process around the method is viable to be automated and the resulting final grade is $8,7$, this method is likely to be useful for web applications.

4.2.4 Issues

Although the evaluation shows this is likely to be a suitable method for performance analysis for web applications, there are still issues to overcome. Two of the biggest issues that should be solved are the following.

- When running the profiler for a while, the profile can become too large to handle. Since the String test within the accuracy test performs a large amount of `random()` function calls, the profile grew to a size not all browsers could handle.

To overcome this issue, this function is not profiled. A more permanent solution is required for this method to be used in production.

- Asynchronous function calls will break the call hierarchy displayed by the profile being recorded, since these functions are started within a function, but will usually return after their parent function is finished. AJAX-calls will therefore show up incorrectly in the profile and corrupt the profiles of other function calls.
- Non-empty return statements may result in incorrect profiles. If a return statement calls a function, this function is not recorded at the correct position in the profile, since the profiling of the parent function has already finished. This child function is therefore displayed at the same depth as its parent in the profile tree. Additionally, if the return statement contains a large computation or a function call, the profile does not contain the complete execution time, because the return statement is not included.
- It is not convenient for the programmer to manually add statements to each function to start and stop the profiler. Therefore, a more developer-friendly method should be developed. However, the method implies the manual addition of lines of code to the source, which makes this issue trivial.

4.3 Manual Logging improved

Since manual logging is a viable method to analyse client-side web application performance, this method is improved. This section describes the changes and improvements made to the original method from Section 4.2.

The main focus was to address the issues described in Section 4.2.4, except for the last issue, since this issue is addressed in Chapter 5. The improvement of the method, however, is done with future automation in mind. Since the pre-evaluation does not differ from the post-evaluation of the first manual logging method described, this section is discarded.

4.3.1 Implementation

The implementation of the original manual logging method is largely unchanged. The most important changes are explained below.

Smaller profile object

Since large amounts of functions caused the profile object to increase significantly over time, the first version of the profiler became very slow when saving or displaying profiler. Hence, it was important to decrease the size of the profile. This is achieved

by grouping equal function calls within the same scope. A scope consists of a set of functions executed sequentially with the same parent function. For every grouped function call, the following relevant additional attributes are stored.

calls: the number of times this function has been called in this scope.

averageTime: the average time each function executed. This is calculated from by dividing the totalTime by the number of function calls.

minTime: the minimum execution time for the function.

maxTime: the maximum execution time for the function.

Note that if a function call occurs only once in a scope `averageTime` equals `minTime`, `maxTime` and `totalTime`.

Grouping takes place upon `start`ing a function. After its scope is found, `findElement` checks whether this function already exists in the profile for the scope. If not, the function is added to the scope; if it does exist, its attributes are updated. A similar function runs upon `stop`ping the function.

The resulting profile is potentially significantly smaller than when using the original manual logging method. For example, the `formateDate` function in the *date* test (Appendix B.1.3) created for testing the accuracy of a method runs 35.000 times on an object. This execution is performed sequentially within the same scope (`runDate`). With the original manual logging method, this scope would have 35.000 child function, each containing 10 children. With the improved method, `runDate`'s scope only contains 1 child. This child contains only 8 children, since two functions are executed twice. In total, the profile object contains 9 children below `runDate` instead of 350.000. Since this is a rather extreme example – it is a benchmark – the gain is not always this significant.

Since grouping results in a significant overhead for each function call, it is expected to have a higher impact on the overall performance when the profiler is enabled.

Asynchronous function calls

Asynchronous function calls, such as AJAX requests, require callback functions to be passed. These callback functions are executed upon a certain event, such as receiving a response from the server. These functions then handle the response. Since their parent function, in which the function is attached to the listener, has already been terminated, the execution of the callback function is profiled in another scope than their parents scope. The profile therefore resembles the actual execution order of the functions, but does not provide a hierarchical overview of function calls to the developer.

For example, Listing 4.2 contains the function `fetchData` that performs a basic AJAX request to the server and alerts the response to the user. To achieve this, an `xmlhttp`

object is created and an `onreadystatelistener` function is attached (line 4). Line 11 and 12 send the request for *models/model_1.json* to the server.

```
1  function fetchData() {
2    profiler.start('fetchData');
3    var xmlhttp = new XMLHttpRequest();
4    xmlhttp.onreadystatechange = function() {
5      profiler.start('anonymousFunction1');
6      if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
7        alert(xmlhttp.responseText);
8      }
9      profiler.stop();
10   };
11   xmlhttp.open("GET", "models/model_1.json");
12   xmlhttp.send();
13   profiler.stop();
14 }
```

Listing 4.2: An example of a function performing an AJAX request.

The five possible values for the `readyState` are displayed in Table 4.3.1. With every transition of the `readyState` to the next, the listener is called, resulting in four calls to the listener function. In this case, this function only performs some additional actions once the `readyState` equals 4 – when the response is ready. Since the first call to the listener only initialises and sends the `xmlhttp` request, this is the only time the function is called within its parent function. All other calls to the listener are performed asynchronously and are therefore profiled outside its parents scope.

Value	**State**
0	UNSENT
1	OPENED
2	HEADERS_RECEIVED
3	LOADING
4	DONE

Table 4.3.1: Possible XMLHttp `readyState` values [42].

1.	Main
2.	`fetchData`
3.	`anonymousFunction1`
4.	`anonymousFunction1`
5.	`anonymousFunction1`
6.	`anonymousFunction1`

(a) Ordered tree

1.	Main
2.	`fetchData`
3.	`anonymousFunction1`
4.	`anonymousFunction1`
5.	`anonymousFunction1`
6.	`anonymousFunction1`

(b) Hierarchical tree

Table 4.3.2: Two possible profile trees for the execution of `fetchData` in Listing 4.2.

Table 4.3.2a displays the resulting profile when using the profiler as described in Section 4.2. The listener – anonymousFunction1 – is called four times, due to changes of the `readyState` value.

For the developer, a more convenient way of constructing the tree would be a hierarchical one, as displayed in Table 4.3.2b. This tree constructs its profile in a way that is consistent with the source code of the program. All four calls to the listener should therefore be child functions of `fetchData`. The original manual logging method is not able to construct hierarchical profiles, but can only construct profiles based on the execution order.

To overcome this issue, callback functions should be marked as such by the developer. The profiler will then store the current scope of this function upon execution. When the function later in time executes, the scope in which it should be profiled is fetched, after which the hierarchical structure that exists in the code is maintained. It is, however, not possible to automatically detect which – sometimes anonymous – functions are executed asynchronously. Therefore, the developer should manually specify these functions. Because the integration process should eventually be automated, this is not a viable solution.

Another theoretical solution is to extend the XMLHttp object, which is used for AJAX requests to the server, with code to start and stop the profiler on the right time to store the callback function in the right hierarchical scope. Since not all browsers allow the extension of the XMLHttp object, this solution would not be compatible with all browsers, which is also an important requirement for the methods.

Considering that there is no viable solution for profiling the asynchronous function calls, this method cannot place callback methods in their hierarchical scope. Instead, the profiler gives an overview of the order in which functions are executed.

Non-empty returns

JavaScript functions may return values, statements or functions. The original manual logging method cannot handle non-empty returns. Therefore, a function containing a non-empty return statement should be stopped just before the return statement. This, however, does not profile the whole function, since the return statement may also contain calculations or function calls. Possible recursive function calls would also become sequential function calls according to this profiler.

The improved version supports an optional second argument for the `stop` method. This argument contains the return statement, which is executed before stopping the measurement for its parent function. The resulting profile is therefore more correct (recursive functions are displayed correctly in the tree) and accurate (since the return statement is measured as well).

```
1 function rand(max) {
2   profiler.start('rand');
3   profiler.stop();
4   return Math.floor((Math.random() * max) + 1);
5 }
```

Listing 4.3: A `rand` function using the original manual logging method.

```
1 function rand(max) {
2   profiler.start('rand');
3   return profiler.stop('rand', Math.floor((Math.random() * max) + 1));
4 }
```

Listing 4.4: A `rand` function using the improved manual logging method.

Listing 4.3 uses the original manual logging method, which starts and stops the profiling of this function without actually measuring something, because the actual content of the function is placed inside the `return` statement. Listing 4.4 shows how the improved manual logging method can be used to measure the statements in the `return` statement as well. Also, the function name is placed in the first argument of `stop` to enable grouping as described in Section 4.3.1.

Listings 4.5 shows a function `fact` that uses recursion to calculate the factorial of a given number with the original manual logging method applied.

```
1  function fact(n) {
2    profiler.start('fact');
3    if (n <= 0) {
4      return 0;
5    }
6    if (n === 1) {
7      return n;
8    }
9    profiler.stop();
10   return n * fact(n-1)
11 }
```

Listing 4.5: A `fact` function using the original manual logging method.

The resulting profile tree when calling `fact(4)`, will be like displayed in Table 4.3.3

Main
`fact(4)`
`fact(3)`
`fact(2)`
`fact(1)`
`fact(0)`

Table 4.3.3: Profile from the original profiler for running `fact(4)`.

This tree is incorrect, as every call of the recursive function `fact` is the child of its parent. These calls should therefore be recorded as children of each other, resulting in a nested instead of sequential structure. With the improved manual logging method, this function should be changed to the code displayed by Listing 4.6.

```
1  function fact(n) {
2    profiler.start('fact');
3    if (n <= 0) {
4      return 0;
5    }
6    if (n === 1) {
7      return n;
8    }
9    return profiler.stop('fact', n * fact(n-1));
10 }
```

Listing 4.6: A `fact` function using the improved manual logging method.

This code would yield the profile as displayed in Table 4.3.4 for a function call to `fact(4)`. This profile contains a nested structure, and therefore shows the recursive behaviour of `fact` correctly.

Main
fact(4)
fact(3)
fact(2)
fact(1)
fact(0)

Table 4.3.4: Profile from the improved profiler for running `fact(4)`.

4.3.2 Post-evaluation

Accuracy: 10

Figure 4.3.1 shows the results of the accuracy test for the improved implementation of the Manual Logging method. Again, the results are similar to those of the reference benchmark, but the five identifying accuracy properties should be checked to calculate the accuracy.

- Internet Explorer is very fast at array operations: confirmed.
- Firefox is slow at string operations: confirmed.
- Opera is slower than Chrome at date operations: confirmed, although the differences are very small.
- Chrome and Opera are fastest at JSON operations: confirmed, although the differences between the four browsers are smaller.

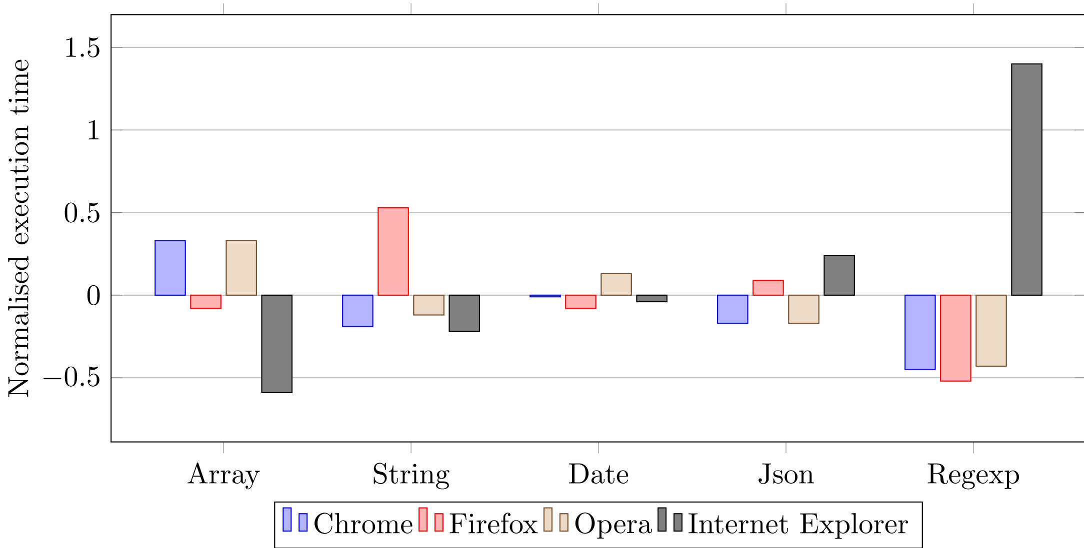

Figure 4.3.1: Normalised results for the accuracy test for evaluation of the improved Manual Logging method. Lower is faster.

- Internet Explorer is slow at executing regular expressions: confirmed.

Since all five properties can be confirmed from the accuracy test, the grade for accuracy for this method is a 10.

Impact: 5,2

Table 4.3.5 shows the average execution times of the rendering of the graphs on the dashboard page of the CTG application. The grade can be calculated as follows: $G_{impact} = 10 - \frac{10 \cdot (495-335)}{335} = 5,2$.

	Without	**With**	**Difference**
Chrome	226,4	365,0	61,22%
Firefox	262,4	355,2	35,37%
Opera	223,6	384,0	71,74%
Internet Explorer	629,2	876,6	39,32%
Average	335,4	495,2	47,64%

Table 4.3.5: Total execution time of the execution test with and without the performance analysis.

Usability: 8

Since this method only involves a technical improvement to the profiler, the expected total time the end-user and developer need to spend on creating an analysis is less than 5 minutes. Therefore, the grade for usability remains the same as before: $G_{usability} = 8$.

Portability: 8,4

The number of lines of code within *graph.js* is used to calculate the grade for portability of this method. The amount of code that needs to be typed by the developer has increased a little, since the name of the function must be given as argument to `stop` the function. However, this does not affect the number of lines the developer should type. The number of lines is slightly decreased, because `return` statements can now be joined with the `stop` method. Table 4.3.6 displayed the number of lines of code of *graph.js* with and without the profiler applied. The grade for portability is computed as follows: $G_{portability} = 10 - \frac{25 \cdot (1379 - 1297)}{1297} = 8,4$.

graph.js	**Lines of code**
Without profiler	1297
With profiler	1379
Difference	82

Table 4.3.6: The number of lines of code within the scope with and without the performance analysis method.

With these four grades, the final grade is calculated: $G_{total} = \frac{2}{5} \cdot 10 + \frac{1}{10} \cdot 5,2 + \frac{3}{10} \cdot 8 + \frac{1}{5} \cdot 8,4 = 8,6$. This final grade is slightly lower than the original method. Since all properties are grades at least 5 points and secondary requirements are met, this method is suitable for use in a project.

Although the final grade for the improved method is slightly lower than that of the original method, the improved method is more suitable. The resulting profile contains all relevant information, but is significantly smaller and therefore more usable for processing. To keep the profile small, similar items are grouped by the profiler. Grouping requires some additional logic in the profiler, which decreases execution speed of the profiler itself, as shown by the grade for impact. However, when processing or transferring the profile, the costs for a huge profile will be significantly higher than those of the additional logic in the profiler to keep the profile small.

Chapter 5

Integration

This chapter describes methods that can be used to integrate the performance analysis method described in Section 4.3 with any web application project. It is important that the addition of the profiler integrates with a build process in a working environment.

5.1 Aspect Oriented Programming

Aspect Oriented Programming (AOP, [43]) is a programming paradigm that adds additional behaviour to source code without the need to modify the original code. A commonly used method for this is the addition of annotations to the code. Hence, AOP is regularly used to add logging statements to functions. Doing so, the developer does not need to add the logging statements manually to the functions. Also, the code is better readable, since the functions only define the behaviour that is expected from the function, without the additional logging statements.

Since JavaScript does not support annotations, AOP for JavaScript is implemented like an extension of an existing function that adds behaviour and executes the original function. By overloading every function, profiling statements could be added to every function. This creates a large overhead for the developer. Additionally, there are frameworks available[1][2][3] to use AOP in JavaScript. However, every function should manually or automatically be modified to enable profiling statements to be execution upon function calls. This can be done by prepending the function name or adding comments, depending on the framework used. Since this method depends on a framework that should be added to the project and it also involves some overhead for the developer, the use of frameworks for AOP in JavaScript is not suitable.

[1] http://aspectjs.com/

[2] https://github.com/raganwald/YouAreDaChef

[3] http://mulli.nu/2010/05/07/aop-js/

Additionally, the developer can pick specific functions to profile by overloading the function. While this increases flexibility, it also decreases the correctness of the profiles obtained from the profiler, since it is not always the case a full profile is recorded. In order to overload all functions within the project – including frameworks that are used – the developer should spend a large amount of time. Automatic injection is possible, but creates a significant overhead in the code. Therefore, Aspect Oriented Programming is not suitable for integrating the profiler in a project.

5.2 Integration tool

The statements that need to be added to profile the JavaScript execution of a web application are very predictable. These function calls to the profiler object only require the name of the function that is being profiled and need to be placed consistently on the correct location within the code. Therefore, the insertion of these profiler statements to existing source code could relatively easy be automated. Therefore, a tool was developed that allows the integration of the profiler statements within an existing project or file.

5.2.1 Implementation

Our tool parses the input source(s) and inserts the statements that are required to profile all functions. It can also store the code of the profiler itself to a given destination. The tool handles three types of files.

JavaScript files: The extension of these files is `.js`. These files are parsed, after which profiling statements are inserted and the file is written to the given output.

Markup files: The extension of these files is `.htm`, `.html` or `.php`. These files may contain JavaScript code within `script`-tags. The files are read and searched for `<script type="text/javascript">` tags. The contents of these tags are processed as JavaScript files, after which the resulting file is written to the given output.

Other files: These files do not match any format of the above and are copied to the given destination.

JavaScript code – both from files and embedded code snippets – is parsed by the Mozilla Rhino[4] JavaScript parser to obtain an Abstract Syntax Tree (AST). This AST is then modified by inserting profiler statements to enable the functions to make use of the profiler. The following steps are performed on the AST.

1. Find all function nodes in the tree.

[4] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

2. Add a `profiler.start` call in front of every function body.
3. Recursively call step 1 with the function body as tree.
4. Find all return statements within the scope of this function and rewrite them to stop the profiler for this function.
5. If the function does not end with a `return` statement, add a `profiler.stop` to the end of the function body.

This is the algorithm to apply the profiler to a given script. This routine is repeated for every script that is found in the input directory. Additionally, the tool can store the source containing the profiler itself in a given directory. Once the tool detects that files will be removed or overwritten, it prompts the user whether to continue or not. Table 5.2.1 provides an overview of the arguments that are accepted by the tool.

Argument	**Description**
`-f`	When set, the user will not be prompted when files are being removed or overwritten.
`-fullpath`	When set, the function name in the call to the profiler is set to the relative path instead of only the name of the source file.
`-help`	When set, a help message is displayed to the user.
`-if` *file*	Provides an input file or directory to apply the profiler to.
`-of` *file*	Provides an output file or directory to write the output to. If this argument is not set, the tool will overwrite the input file with its output.
`-pf` *file*	When set, the source of the profiler itself is written to this file. If a directory is given, *Profiler.js* is written in this directory; if a file (*.js*) is given, the source is written to this file.

Table 5.2.1: Accepted arguments for the integration tool.

The tool was developed in Java 8[5], since this language included support for both Mozilla Rhino and Apache Commons CLI[6]. Also, the default developing environment of Topicus is based on Java 8.

5.2.2 Improvements

Although the improved manual logging method is capable of reporting performance statistics to the developer rather clearly by displaying the name of the function, these results may yield ambiguous results when used in a large project. Since function names may be used multiple times throughout a project in different contexts, these may eventually not be unique, making them unsuitable as main identifier.

[5] `http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html`
[6] `https://commons.apache.org/proper/commons-cli/`

To overcome this issue, the file name and line number where the function is declared are now passed as arguments to the profiler and stored in the profile. The profiler itself has been modified to support these additional properties.

5.2.3 Limitations

Although our integration tool is capable of handling most common JavaScript sources and snippets, some assumptions have been made that limit its functionality. The limitations of the tool are listed below.

- Functions defined in `eval` expressions will not be subject to the application of the profiler statements. Since these functions are defined as text, this text should first be parsed again in search for function definitions. This introduces a possible infinite recursion and is considered to be a bad practice and makes the tool unnecessarily complex. Therefore, function declarations using `eval` expressions will not be injected with profiler statements.
- Definition of functions by initialising the `Function` object are not supported, since this is considered to be an implicit `eval` expression. The function body is passed as string argument to the `Function` object, making it unnecessarily complex.
- Files not having the the extension `.js`, `.htm`, `.html` or `.php` are considered not to contain JavaScript content. The source of the tool, however, allows rather easy extension to other file types when required.
- Files or folders cannot be selected or excluded individually for application of the profiler within a single run of the tool. The tool can only operate on one file or a whole folder. A workaround is to run the tool for each file separately.

5.2.4 Concluding grade

Since the original profiler has been changed to store both the file name of the source and the line number on which the function is declared, the resulting profile is expected to become larger than with the improved version of the method. A larger profile is likely to be handled slower, resulting in a lower grade for impact. Also, the developer should add significantly less lines of code to a project to apply the profiler to it, resulting in a higher grade for portability. Therefore, these two grades are revised and the resulting final grade of the method including the integration tool is calculated.

Impact

Table 5.2.2 shows the execution times in milliseconds to draw four CTG graphs on the dashboard of the CTG application. As in the previous impact tests, the test was conducted five times on each browser for both with and without a profiler applied to the *graph.js* file.

	Without	**With**	**Difference**
Chrome	238,0	364,8	53,28%
Firefox	275,6	365,2	32,51%
Opera	220,0	353,0	60,45%
Internet Explorer	609,4	826,2	35,58%
Average	335,8	477,3	42,16%

Table 5.2.2: Total execution time of the execution test with and without the performance analysis.

The results may be a little surprising, since the impact of the profiler appears to be smaller than before. The profiler, however, has been extended to store both the file name of the source file and the line number on which the function is declared, which was expected to be slightly slower. Since the difference is only 5%, it is more likely to be related to environmental causes such as overall CPU load than to the profiler itself. Altogether, the extension does not have any measurable influence on the impact on the performance of the profiler. Since the grade has been changed, however, the impact grade is recalculated: $G_{impact} = 10 - \frac{10 \cdot (477-336)}{336} = 5,8$.

Portability

Since this tool is built to decrease the amount of work for the developer, it is expected that the portability grade will increase. With this tool, the developer does not need to manually add profiler statements to the source, resulting in a significant decrease of the amount of work for the developer. Instead, the developer should perform one command to apply the profiler on the project. Therefore, no manually typed code needs to be added to the project sources, resulting in a high grade: $G_{portability} = 10 - \frac{25 \cdot (1297-1297)}{1297} = 10$.

Because the functionality of the method has not changed, both accuracy and usability are assumed to remain unchanged. The resulting final grade for the method with integration tool can then be calculated: $G_{total} = \frac{2}{5} \cdot 10 + \frac{1}{10} \cdot 5,8 + \frac{3}{10} \cdot 8 + \frac{1}{5} \cdot 10 = 9,0$. Since the combination of the profiler and the integration tool results in the highest graded method, this combination is very suitable to use for client-side performance analysis.

Chapter 6

Conclusion

This thesis describes the steps that were taken towards a method to measure the client-side performance of web applications. A definition of client-side performance within the context of web applications was formulated and properties on which to evaluate possible methods were defined. With these properties, an evaluation method was developed. Two profiling method were suggested, of which one method was eventually implemented and improved in the next step. Finally, an integration method to apply the profiling method to a project was implemented.

In addition to the overall speed of the hardware of a computer, the browser has an impact on the performance of a web application running on a computer. The differences between browsers are caused by the different JavaScript engines that are used. From the perspective of a user, a web application performs badly when it is slow or unresponsive. This is caused by slow execution of JavaScript by the JavaScript engine, which handles the execution of the actions triggered by the user. Therefore, performance of a web application is mainly determined by the execution time of functions in JavaScript and the total execution time of JavaScript code within the application.

To select suitable methods, four measurable properties were introduced to conduct verifiable evaluations on the profiling methods: accuracy, impact, usability and portability. These properties were translated to four metrics which are combined into one final grade for each profiling method. This grade expresses the suitability of the method to measure the performance of a web application at the client-side.

A profiling method is presented that involves adding statements to function declarations in JavaScript to create a profile tree at run-time. This profile tree holds information about every executed function, function name, location and execution time. The profiler should be included in the web page to be used and is written in pure JavaScript, independent of any JavaScript framework that could be included. It is therefore applicable to any web application written in JavaScript. To increase the portability of the method, an integration tool is constructed to integrate the profiler in a file or a

project. This tool automatically adds profiling statements to the source code, which significantly speeds up the integration process for the developer.

The method was evaluated with positive result. It is accurate, user friendly and portable. The impact on the performance of the application, however, is significant, as it decreases by roughly 45%. Due to this impact, the method should only target specific users for performance analyses. Use of the profiler in production environments targeting all users would slow the application down significantly.

Altogether, the method can be applied to any web application that is built with JavaScript and has the ability to provide the developer with detailed information about the execution speed at the end-users.

6.1 Discussion

In Section 3.3, an evaluation method is constructed containing four quantitative metrics and two additional conditions that should be met by a profiling method. While defining profiling methods, it became clear that this definition was not sufficient to determine the suitability of a method for application in a project. Additional requirements were necessary.

Section 4.2.4 describes issues with the original manual logging method. One of the issues is that the resulting profile is too large to send over the internet and slows down the browser significantly. The size of the profile is, however, never mentioned in the evaluation method. This property appeared to be of importance for the evaluation method, but was not included. In Section 4.2.3, the final grade for the method turned out to be lower than the method proposed earlier. The method that received a lower grade, however, was improved. This decision seems to ignore the results from the evaluation method. These examples show that the proposed evaluation method needs refinement.

Section 3.3.1 describes the construction of an accuracy test for the evaluation method. Although the goal of this paper is to develop a method to measure the performance of JavaScript execution in any browser, the accuracy test is constructed based on results of only the four most popular browsers on the market. Consequently, there may be more performance differences between browsers that have not been considered in this paper. These differences may not be identified with the method presented.

The benchmarks carried out in Section 3.3.1 were executed in a virtual machine. Therefore, these benchmarks may not result in very trustworthy results, as the amount of CPU time depends on the overall load of the system as a whole instead of only the browser running the benchmark. The results, however, were usable, since they were only used to compare the browsers with each other.

6.2 Related work

When it comes to general JavaScript execution behaviour, the works of both Martinsen, Grahn [44] and Martinsen, Grahn and Isberg [27] should be noted. These papers investigate the execution behaviour of JavaScript in a large set of popular web applications. This research, however, does not result in a generic method to record the execution behaviour of any web application.

Little scientific research has been carried out towards an implementation of a generic JavaScript profiler that could be included in any web application. However, some solutions have been developed, including AngularJS Batarang[1], Benchpress[2] and Zone.js[3]. These tools, however, are meant to be used to debug specific parts of the source code by the developer, instead of using the tool in a production environment. Also, these tools depend on frameworks to be included with the application.

6.3 Future work

Although an integration tool has been developed to easily integrate the JavaScript profiler in a project, more work is needed to be able to use the approach. To be able to store and analyse the results of the profiler, a connector should be created for the back-end to which the client can push the profile that was collected. Also, a mechanism should be created that enables the developer to target specific users to run the profiler. Finally, the profiler should include methods that start and stop the profiler at specified moments and sends the profile to the back-end.

The current implementation of the profiler outputs the recorded profile as JSON string. This string contains a structured tree of all the functions that were executed during the profiling process. This tree, however, is not yet convenient for the developer to use, as it does not incorporate any visual aspect. To make the tree better readable for the developer, a visualisation tool should be developed or found that visualises the JSON string.

As mentioned in Section 6.1, the evaluation method requires some more refinement before it can be used on similar problems. However, the requirements and metrics that can be added, depend on the application on which the method should be applied. The evaluation method could therefore be extended with addition requirements and metrics. While developing a method, those that are useful can then be selected.

Finally, a case study could be done to the profiling method applied to an application. This study should provide an answer to the question whether the profiling method is useful in practice or not.

[1] `https://github.com/angular/angularjs-batarang`
[2] `https://github.com/angular/benchpress`
[3] `https://github.com/btford/zone.js/`

Bibliography

[1] R. T. Fielding, *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[2] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs", *SIGPLAN Not.*, vol. 45, pp. 1–12, June 2010.

[3] B. Stone, "Monitoring and control engine for multi-tiered service-level management of distributed web-application servers", Nov. 23 2004. US Patent 6,823,382.

[4] J.-C. Bolot and P. Hoschka, "Performance engineering of the world wide web: Application to dimensioning and cache design", *Computer Networks and ISDN Systems*, vol. 28, no. 7, pp. 1397–1405, 1996.

[5] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications", in *Proceedings of the 2010 USENIX conference on Web application development*, pp. 3–3, USENIX Association, 2010.

[6] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated Construction of JavaScript Benchmarks", *SIGPLAN Not.*, vol. 46, pp. 677–694, Oct. 2011.

[7] J. Nielson, C. Williamson, and M. Arlitt, "Benchmarking modern web browsers", in *2nd IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2008.

[8] D. Nations, "Web Applications: What is a web application?". `http://webtrends.about.com/od/webapplications/a/web_application.htm`. Accessed: 2015-02-18.

[9] J. J. Garrett, "Ajax: A new approach to web applications". `https://courses.cs.washington.edu/courses/cse490h/07sp/readings/ajax_adaptive_path.pdf`, 2005. Accessed: 2015-02-18.

[10] E. Marcotte, *Responsive web design.* Editions Eyrolles, 2011.

[11] K. Natda, "Responsive Web Design", *Eduvantage*, vol. 1, no. 1, 2013.

[12] D. Raggett, A. Le Hors, and I. Jacobs, "Html 4.01 specification", *W3C recommendation*, vol. 24, 1999.

[13] World Wide Web Consortium, “Html5 specification”, *Technical Specification, Jun*, vol. 24, p. 2010, 2010.

[14] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs, “Cascading style sheets, level 2 CSS2 specification”, 1998.

[15] B. Eich, “JavaScript at Ten Years”, *SIGPLAN Not.*, vol. 40, pp. 129–129, Sept. 2005.

[16] D. Kramer, “The Java Platform”, *White Paper, Sun Microsystems, Mountain View, CA*, 1996.

[17] J. Gay, “The history of Flash”, *Adobe Systems Inc*, 2001. `http://www.adobe.com/macromedia/events/john_gay/index.html`, Accessed: 2015-02-18.

[18] A. Grosskurth and M. W. Godfrey, “A reference architecture for web browsers”, in *Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on*, pp. 661–664, IEEE, 2005.

[19] G. Nicol, L. Wood, M. Champion, and S. Byrne, “Document object model (DOM) level 3 core specification”, 2001.

[20] StatCounter, “StatCounter Global Stats: Top 5 Desktop, Tablet & Console Browsers from Feb 2014 to Feb 2015”. `http://gs.statcounter.com/#browser-ww-monthly-201402-201502`. Accessed: 2015-03-13.

[21] D. Flanagan, *JavaScript: the definitive guide*. ”O’Reilly Media, Inc.”, 2006.

[22] “A Short History of JavaScript”. `https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript`. Accessed: 2015-02-20.

[23] Brendan Eich, “Brendan Eich”. `https://brendaneich.com/`. Accessed: 2015-02-20.

[24] ECMAScript, ECMA and European Computer Manufacturers Association and others, “ECMAScript Language Specification”, 2011.

[25] K. M. Dixit, “The SPEC benchmarks”, *Parallel Computing*, vol. 17, no. 1011, pp. 1195 – 1209, 1991. Benchmarking of high performance supercomputers.

[26] J. L. Henning, “SPEC CPU2000: Measuring CPU performance in the new millennium”, *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[27] J. K. Martinsen, H. Grahn, and A. Isberg, “A comparative evaluation of JavaScript execution behavior”, in *Web Engineering*, pp. 399–402, Springer, 2011.

[28] J. Aycock, “A brief history of just-in-time”, *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003.

[29] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof: A call graph execution profiler”, in *ACM Sigplan Notices*, vol. 17, pp. 120–126, ACM, 1982.

[30] P. Louridas, "Static code analysis", *Software, IEEE*, vol. 23, no. 4, pp. 58–61, 2006.

[31] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript", in *Static Analysis*, pp. 238–255, Springer, 2009.

[32] S. Guarnieri and V. B. Livshits, "GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code.", in *USENIX Security Symposium*, vol. 10, pp. 78–85, 2009.

[33] W3Techs, "Usage of JavaScript libraries for websites". `http://w3techs.com/technologies/overview/javascript_library/all`. Accessed: 2015-02-27.

[34] A. Lerner, *Ng-Book - the Complete Book on Angularjs*. Fullstack.io, 2013.

[35] Z. Alfirevic, D. Devane, G. Gyte, *et al.*, "Continuous cardiotocography (CTG) as a form of electronic fetal monitoring (EFM) for fetal assessment during labour", *Cochrane Database Syst Rev*, vol. 3, no. 3, p. CD006066, 2006.

[36] World Wide Web Consortium and others, "Extensible markup language (XML) 1.1". `http://www.w3.org/TR/2006/REC-xml11-20060816/`, 2006. Accessed: 2015-02-27.

[37] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format". `http://tools.ietf.org/html/rfc7159.html`, 2014. Accessed: 2015-02-27.

[38] R. Jain, *The art of computer systems performance analysis*. John Wiley & Sons, 2008.

[39] D. Menasce, "QoS issues in web services", *Internet Computing, IEEE*, vol. 6, no. 6, pp. 72–75, 2002.

[40] F. F.-H. Nah, "A study on tolerable waiting time: how long are web users willing to wait?", *Behaviour & Information Technology*, vol. 23, no. 3, pp. 153–163, 2004.

[41] F. Culwin and X. Faulkner, "Brewsing the Web: Delay, Determination and Satisfaction", in *2013 46th Hawaii International Conference on System Sciences*, vol. 5, pp. 5018–5018, IEEE Computer Society, 2001.

[42] M. D. Network and individual contributors, "Xmlhttprequest". `https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest`. Accessed: 08-06-2015.

[43] G. Kiczales, J. Lamping, C. Lopes, J. Hugunin, E. Hilsdale, and C. Boyapati, "Aspect-oriented programming", Oct. 15 2002. US Patent 6,467,086.

[44] J. Martinsen and H. Grahn, "A methodology for evaluating javascript execution behavior in interactive web applications", in *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*, pp. 241–248, Dec 2011.

Appendix A

Benchmark results

This appendix contains the results of the benchmarks that were executed in a virtual machine running Windows 7 SP1. The virtual machine was assigned 1GB of memory and one CPU-core (Intel Core i5-3337U, 1,7∼2,7 Ghz). Each benchmark was executed three times, after which the results were averaged.

The following versions of the browsers were used.

- Google **Chrome** 41.0.2272.101 m
- Mozilla **Firefox** 36.0.4
- **Opera** 28.0.1750.48
- Microsoft **Internet Explorer** 11.0.9600.17691

A.1 Dromaeo JavaScript Tests[1]

	Chrome	**Firefox**	**Opera**	**Internet Explorer**
Arrays	1388,46 (0,98)	652,06 (0,46)	1469,75 (1,04)	1712,49 (1,21)
Base64	1916,36 (1,35)	1146,86 (0,81)	1883,98 (1,33)	1607,78 (1,14)
Eval	444,61 (0,59)	442,04 (0,58)	488,45 (0,64)	1808,3 (2,38)
Regex	536,7 (1,69)	174,61 (0,55)	530,64 (1,67)	173,05 (0,55)
3D	2256,2 (1,38)	2146,93 (1,31)	2193,73 (1,34)	1072,58 (0,66)
Strings	3174,9 (1,30)	2848,07 (1,17)	3147,34 (1,29)	2075,54 (0,85)

Table A.1.1: Average Dromaeo benchmark results in runs per second and normalised with the average results of the browsers. Higher is faster.

[1] `http://dromaeo.com/?dromaeo`

A.2 Sunspider 1.0.2[2]

	Chrome	Firefox	Opera	Internet Explorer
3D	75,43 (1,11)	64,13 (0,94)	74,17 (1,09)	58,67 (0,86)
Access	23,37 (0,68)	18,6 (0,54)	23,7 (0,69)	72,13 (2,09)
Bitops	18,5 (0,63)	11,33 (0,39)	24,9 (0,85)	62,83 (2,14)
Control Flow	2,9 (0,41)	2,57 (0,36)	3,3 (0,46)	19,63 (2,77)
Crypto	28,63 (0,78)	25,77 (0,70)	36,57 (0,99)	56,13 (1,53)
Date	32,6 (0,90)	33,27 (0,92)	37,13 (1,03)	41,67 (1,15)
Math	28,07 (0,86)	16,93 (0,52)	28,07 (0,86)	57,87 (1,77)
Regex	9,47 (0,95)	10,67 (1,07)	9,7 (0,98)	9,87 (0,99)
String	120,63 (1,08)	95,23 (0,86)	134,5 (1,21)	94,73 (0,85)

Table A.2.1: Average Sunspider benchmark results in milliseconds and normalised with the average results of the browsers. Lower is faster.

A.3 Kraken 1.1[3]

	Chrome	Firefox	Opera	Internet Explorer
AI	168,43 (0,81)	129,93 (0,62)	168,03 (0,80)	370,53 (1,77)
Audio	847,1 (0,88)	862,33 (0,89)	841,27 (0,87)	1316,57 (1,36)
Imaging	432,6 (0,85)	377,03 (0,74)	427,93 (0,84)	801,97 (1,57)
Json	136,97 (0,87)	162,1 (1,02)	140,87 (0,89)	192,73 (1,22)
Stanford	593,37 (0,80)	583,6 (0,79)	609,13 (0,82)	1170,5 (1,58)

Table A.3.1: Average Kraken benchmark results in milliseconds and normalised with the average results of the browsers. Lower is faster.

[2] `http://www.webkit.org/perf/sunspider-1.0.2/sunspider-1.0.2/driver.html`
[3] `http://krakenbenchmark.mozilla.org/kraken-1.1/driver.html`

A.4 Octane 2.0[4]

	Chrome	Firefox	Opera	Internet Explorer
Richards	21384 (1,14)	20940 (1,12)	21831 (1,16)	10857 (0,58)
Deltablue	28409 (1,31)	16715 (0,77)	33281 (1,54)	8172 (0,38)
Crypto	19180 (1,10)	19171 (1,10)	18583 (1,07)	12520 (0,72)
Raytrace	40512 (1,22)	39945 (1,20)	42246 (1,27)	10270 (0,31)
EarleyBoyer	29523 (1,22)	24056 (0,99)	30212 (1,25)	13270 (0,55)
Regexp	3037 (1,31)	1747 (0,75)	3046 (1,31)	1452 (0,63)
Splay	11707 (1,25)	11260 (1,21)	11363 (1,22)	3008 (0,32)
SplayLatency	12201 (1,31)	8708 (0,93)	14428 (1,55)	2001 (0,21)
NavierStokes	18883 (0,93)	24434 (1,20)	19074 (0,94)	19165 (0,94)
pdf.js	10269 (1,21)	7991 (0,94)	9901 (1,17)	5784 (0,68)
Mandreel	11584 (1,33)	4043 (0,46)	11069 (1,27)	8087 (0,93)
MandreelLatency	9182 (0,92)	12125 (1,21)	8601 (0,86)	10165 (1,01)
GB Emulator	30739 (1,19)	31437 (1,22)	28227 (1,09)	12770 (0,50)
CodeLoad	11028 (1,02)	13407 (1,23)	9885 (0,91)	9122 (0,84)
Box2DWeb	15982 (1,67)	6386 (0,67)	10562 (1,10)	5371 (0,56)
zlib	41085 (1,23)	39119 (1,17)	41700 (1,25)	11316 (0,34)
Typescript	15787 (1,04)	12575 (0,83)	13903 (0,91)	18680 (1,23)

Table A.4.1: Average Octane benchmark results in points assigned by the benchmark suite and normalised with the average results of the browsers. Higher is faster.

[4] `http://octane-benchmark.googlecode.com/svn/latest/index.html`

Appendix B

Accuracy test

B.1 Sources

```
1  <!DOCTYPE html>
2  <html>
3  <head lang="en">
4    <meta charset="UTF-8">
5    <title>Method Accuracy Evaluation</title>
6    <style>
7      button {
8        width: 200px;
9        display: block;
10       margin: 15px;
11     }
12     .resultsHolder {
13       width: 300px;
14       border: 1px solid #000;
15       border-radius: 5px;
16       padding: 3px 10px;
17     }
18   </style>
19   <script type="text/javascript">
20     var text;
21     var runArray = function() {},
22       runString = function() {},
23       runDate = function() {},
24       runJson = function() {},
25       runRegexp = function() {};
26     function run(type) {
27       if (type === "Regexp")
28         text = String.random(1e6);
29       console.log("Running " + type + "..");
30       var startTime = new Date();
31       eval("run" + type + "()");
32       var duration = new Date() - startTime;
33       console.log("Finished " + type + " (" + duration + " ms).");
```

```
34        var node = document.createElement("li");
35        node.innerHTML = type + ": " + duration + " ms";
36        document.getElementById("results").appendChild(node);
37      }
38      function runAll() {
39        run("Array");
40        run("String");
41        run("Date");
42        run("Json");
43        run("Regexp");
44      }
45    </script>
46  </head>
47  <body>
48  <h3>Method Accuracy Evaluation for Master Thesis</h3>
49
50  <button onclick="run('Array')">Run Array functions</button>
51  <button onclick="run('String')">Run String functions</button>
52  <button onclick="run('Date')">Run Date functions</button>
53  <button onclick="run('Json')">Run Json functions</button>
54  <button onclick="run('Regexp')">Run Regexp functions</button>
55  <button onclick="runAll()">Run All functions</button>
56  <div class="resultsHolder">
57    <h4>Results</h4>
58    <ul id="results"></ul>
59  </div>
60
61  <script type="text/javascript" src="js/array.js"></script>
62  <script type="text/javascript" src="js/string.js"></script>
63  <script type="text/javascript" src="js/date.js"></script>
64  <script type="text/javascript" src="js/jsonData.js"></script>
65  <script type="text/javascript" src="js/json.js"></script>
66  <script type="text/javascript" src="js/regexp.js"></script>
67  </body>
68  </html>
```

Listing B.1: HTML code that contains the layout and scripts for accuracy evaluation.

B.1.1 Array

```
var ARR_LENGTH = 3e4;
var ARR_ROUNDS = 1e4;
var arr, tmp, a;

function runArray() {
  for(var i = 0; i < ARR_ROUNDS; i++) {
    arr = [];
    arr.length = ARR_LENGTH;
  }

  for(var i = 0; i < ARR_ROUNDS; i++)
    arr = new Array(ARR_LENGTH);

  arr = [];
  for(var i = 0; i < ARR_ROUNDS; i++)
    arr.unshift(ARR_LENGTH);

  arr = [];
  for(var i = 0; i < ARR_ROUNDS; i++)
    arr.splice(0, 0, ARR_LENGTH);

  a = arr.slice();
  for(var i = 0; i < ARR_ROUNDS; i++)
    tmp = a.shift();

  a = arr.slice();
  for(var i = 0; i < ARR_ROUNDS; i++)
    arr.splice(0, 1);

  arr = [];
  for(var i = 0; i < ARR_ROUNDS; i++)
    arr.push(i);

  a = arr.slice();
  for(var i = 0; i < ARR_ROUNDS; i++)
    tmp = a.pop();
}
```

Listing B.2: JavaScript code that performs array operations.

B.1.2 String

```
var STR_ROUNDS = 3e5;
var STR_LENGTH = 11;
var STR_WORDLEN = 220;

var chars = [ "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l"
    , "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z
    ", " " ];

function rand(max) {
  return Math.round(Math.random() * max) % max;
}

String.randomC = function(length) {
  var s = new String();
  var c;
  for(var i = 0; i < length; i++) {
    c = chars[rand(chars.length)];
    c = i % 2 === 0 ? c.toUpperCase() : c.toLowerCase();
    s += c;
  }
  return s.trim();
}

function runString() {
  var str = new String();
  var words = 0;
  var s;
  for(var i = 0; i < STR_ROUNDS; i++) {
    s = String.randomC(STR_LENGTH);
    str = str.concat(str, s);
    if (str.length > STR_WORDLEN) {
      str.split(" ");
      str.charAt(765);
      str.substring(333, 666);
      str = new String();
      words++;
    }
  }
  if (words < STR_ROUNDS * STR_LENGTH / STR_WORDLEN)
    console.error("Invalid number of words! (" + words + ")");
}
```

Listing B.3: JavaScript code that performs string operations.

B.1.3 Date

```
1  var DATE_ROUNDS = 3.5e4;
2  var DATE_STEP = 652918732;
3
4  Date.prototype.formatDate = function(input,time) {
5    var methods = [ "Y", "m", "M", "d", "D", "H", "i", "s" ];
6    var months = [ "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
         "Sep", "Oct", "Nov", "Dec" ];
7    var days = [ "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun" ];
8
9    function Y() {
10     return self.getFullYear();
11   }
12
13   function m() {
14     var m = self.getMonth() +  1;
15     return m < 10 ? "0" + m : m;
16   }
17
18   function M() {
19     return months[self.getMonth()];
20   }
21
22   function d() {
23     var d = self.getDate() +  1;
24     return d < 10 ? "0" + d : d;
25   }
26
27   function D() {
28     return days[self.getDay()];
29   }
30
31   function H() {
32     return self.getHours() < 10 ? "0" + self.getHours() : self.getHours
           ();
33   }
34
35   function i() {
36     return self.getMinutes() < 10 ? "0" + self.getMinutes() : self.
           getMinutes();
37   }
38
39   function s() {
40     return self.getSeconds() < 10 ? "0" + self.getSeconds() : self.
           getSeconds();
41   }
42
43   var self = this;
44   if (time) {
45     var prevTime = self.getTime();
46     self.setTime(time);
47   }
```

```
48
49    var spl = input.split("");
50    var res = [];
51    for(var c = 0; c < spl.length; c++ ) {
52      if (methods.indexOf(spl[c]) >= 0)
53        res[c] = eval(spl[c] + "()");
54      else
55        res[c] = spl[c];
56    }
57
58    if (prevTime)
59      self.setTime(prevTime);
60
61    return res.join("");
62  };
63
64  function runDate() {
65    var date = new Date("2015-04-10 09:21:54");
66
67    for(var i = 0; i < DATE_ROUNDS; i++) {
68      date.formatDate("D d M Y (d-m-Y), H:i:s");
69      date.setTime(date.getTime() + DATE_STEP);
70    }
71  }
```

Listing B.4: JavaScript code that performs date operations.

B.1.4 Json

The JSON test requires a large JSON object. Since this file is too big to include in the paper, it is provided as an external resource. *jsonData.js* contains a single variable declaration for the variable *jsonData* and assigns a JSON object of roughly 70 kB.

```
1  var JSON_ROUNDS = 700;
2  var tmp;
3
4  function runJson() {
5    for (var i = 0; i < JSON_ROUNDS; i++)
6      tmp = JSON.stringify(jsonData);
7    for (var i = 0; i < JSON_ROUNDS; i++)
8      JSON.parse(tmp);
9  }
```

Listing B.5: JavaScript code that performs JSON operations.

B.1.5 Regexp

```
1  var REGEX_ROUNDS = 100;
2
3  var regex = [
4    /(htrs([b]+)lkjc|([a]*)([b]+)([a]+))/i,
5    /((a|b)+)/i,
6    /sxpl([a-zA-Z]{3-8})lp/,
7    /ln?mas(_{2})/,
8    /tra*khba/,
9    /\s(abc|def|ghi|jkl|mno|pqr|stu|vwx|yz)\s([a-f]{1-3})/i,
10   /^[\s\xa0]+|[\s\xa0]+$/g,
11   /(((\w+):\/\/)([^\/:]*)(:(\d+))?)?([^#?]*)(\?([^#]*))?(#(.*))?/,
12   /  /,
13   /^\s*(([a]+)|([b]+)+)(\S*(\s+\S+)*)\s*$/,
14   /^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}/i
15 ];
16
17 String.random = function(length) {
18   var s = "";
19   for(var i = 0; i < length; i++)
20     s += chars[rand(chars.length)];
21   return s;
22 };
23
24 function runRegexp() {
25   regex.forEach(function(rex) {
26     for (var i = 0; i < REGEX_ROUNDS; i++)
27       rex.exec(text);
28   });
29 }
```

Listing B.6: JavaScript code that executes regular expressions on text.

B.2 Results from built-in JavaScript profiler

	Chrome	Firefox	Opera	Internet Explorer
Array	510,8 (1,31)	380,3 (0,97)	522,3 (1,34)	149,0 (0,38)
String	697,5 (0,76)	1460,4 (1,59)	713,0 (0,78)	792,1 (0,87)
Date	369,5 (0,91)	358,8 (0,89)	454,3 (1,12)	433,3 (1,07)
Json	797,3 (0,85)	1053,4 (1,12)	807,5 (0,86)	1107,8 (1,18)
Regexp	661,3 (0,51)	598,8 (0,46)	659,6 (0,50)	3310,6 (2,53)

Table B.2.1: Results for the accuracy test for evaluation as read from the built-in JavaScript profiler. Lower is faster.

B.3 Results from Manual Logging method

	Chrome	Firefox	Opera	Internet Explorer
Array	575,1 (1,32)	397,5 (0,92)	623,8 (1,44)	139,9 (0,32)
String	1225,3 (0,78)	2264,6 (1,44)	1279,5 (0,81)	1536,3 (0,97)
Date	1030,6 (0,90)	1061,8 (0,92)	1184,0 (1,03)	1323,1 (1,15)
Json	783,4 (0,75)	1020,9 (0,98)	783,9 (0,75)	1573,0 (1,51)
Regexp	647,1 (0,55)	583,9 (0,49)	650,4 (0,55)	2851,3 (2,41)

Table B.3.1: Results for the accuracy test for the evaluation as read from the Manual Logging method (Section 4.2). Lower is faster.

B.4 Results from improved Manual Logging method

	Chrome	Firefox	Opera	Internet Explorer
Array	539,5 (1,33)	372,1 (0,92)	539,3 (1,33)	167,5 (0,41)
String	1247,8 (0,81)	2364,4 (1,53)	1350,3 (0,88)	1201,6 (0,78)
Date	1020,8 (0,99)	955,5 (0,92)	1165,6 (1,13)	998,9 (0,96)
Json	783 (0,83)	1029,3 (1,09)	778,9 (0,83)	1170,5 (1,24)
Regexp	706,9 (0,55)	609,5 (0,48)	734,4 (0,57)	3075,3 (2,4)

Table B.4.1: Results for the accuracy test for the evaluation as read from the improved Manual Logging method (Section 4.3). Lower is faster.