Challenges of Web Application Development: How to Optimize Client-Side Code

Chris Mulder
S0326631

August 16, 2011

Contents

1 Introduction **3**
- 1.1 Problem Statement . . . 4
- 1.2 Research Goals . . . 5
- 1.3 Research Methodology . . . 5
- 1.4 Structure of the Thesis . . . 5

2 Description of Problem Domain **6**
- 2.1 History . . . 6
  - 2.1.1 The Invention of the Web . . . 6
  - 2.1.2 Browser Wars . . . 7
  - 2.1.3 The HTML Standard . . . 8
  - 2.1.4 JavaScript . . . 10
  - 2.1.5 CSS . . . 13
- 2.2 Five Emerging Issues . . . 13
  - 2.2.1 History/Back Button . . . 14
  - 2.2.2 Search Engine Optimization . . . 14
  - 2.2.3 Modularization, Code-generation . . . 15
  - 2.2.4 Hiding Differences in Target Platforms . . . 16
  - 2.2.5 Profiling/Benchmarking and Optimization . . . 17
- 2.3 Choice and Motivation . . . 18

3 Action Research **20**
- 3.1 Research Setup . . . 20
- 3.2 Project Kickoff . . . 20
- 3.3 Research Phase #1: Tools Overview . . . 21
- 3.4 Research Phase #2: First Trials . . . 24
  - 3.4.1 Plain JavaScript Approach . . . 24
  - 3.4.2 Speed Tracer Approach . . . 25
- 3.5 Development Phase: Chosen Approach . . . 26
  - 3.5.1 Navigation Timing . . . 26
  - 3.5.2 Custom Measurements . . . 27
  - 3.5.3 Triggering The Benchmark . . . 28
  - 3.5.4 The Beacon . . . 29
  - 3.5.5 Automation of Benchmarking . . . 30

3.6 Overview of Implementation 31
3.7 Results 31

4 Discussion **33**
4.1 Challenges in Web Engineering Research 33
4.2 Course of the Process 33

5 Conclusion **35**

6 Furture Work **37**
6.1 Additions to Current Implementation 37
6.2 Optimization 37
6.3 Common Good Practices in JavaScript 38
6.4 Implications for other Websites using Client-Side Techonology . . 39
6.5 Implications for Social Web Applications 39

Chapter 1

Introduction

More and more time is spent on the Internet. People use Internet for on-line banking, planning a holiday, on-line gaming et cetera. One of the most popular websites are social networking websites. A social networking website is a website on which you can connect with people and communicate with them in many different ways such as sending messages and sharing photo's. The company I work for is Hyves, the largest social networking website of the Netherlands. Since the founding of Hyves in 2004, it was always trying to be ahead or at least up to date in the industry by using the most cutting edge technologies available at that moment. This because of the other social networks that were emerging. This means Hyves is continuously evolving and the biggest innovation is the transformation from a website to a web application.

The traditional website consist of static web pages on which every interaction such as clicking on a link or submitting a form resulted in another static page. This results in a slow user experience in comparison with/to desktop applications whose interface is able to directly respond to user interaction and therefore in general have a better user experience. With the arrival of techniques like AJAX, faster JavaScript, CSS3, HTML5, this slow user experience for websites is becoming something from the past. Using these techniques a web developer has the same possibilities to react on user actions as native application developers have. Thanks to AJAX, data can be retrieved from and send to the server without a full page rendering (see figure 1.1) and CSS3 provides a way of creating beautiful layouts and interfaces combined with hardware accelerated transitions and animations. All these developments have resulted in a new phenomenon: web applications. Web applications, also called web apps, are applications that are use technologies as JavaScript, CSS3, and HTML5. They are built to run in the browser. A lot of websites turning into web applications because of ability to improve the user experience by enhancing the responsiveness.

These web applications bring new challenges and since the use of these new techniques has become very popular, the new challenges that arise become more relevant. One of these challenges is optimizing the client-side code. In this

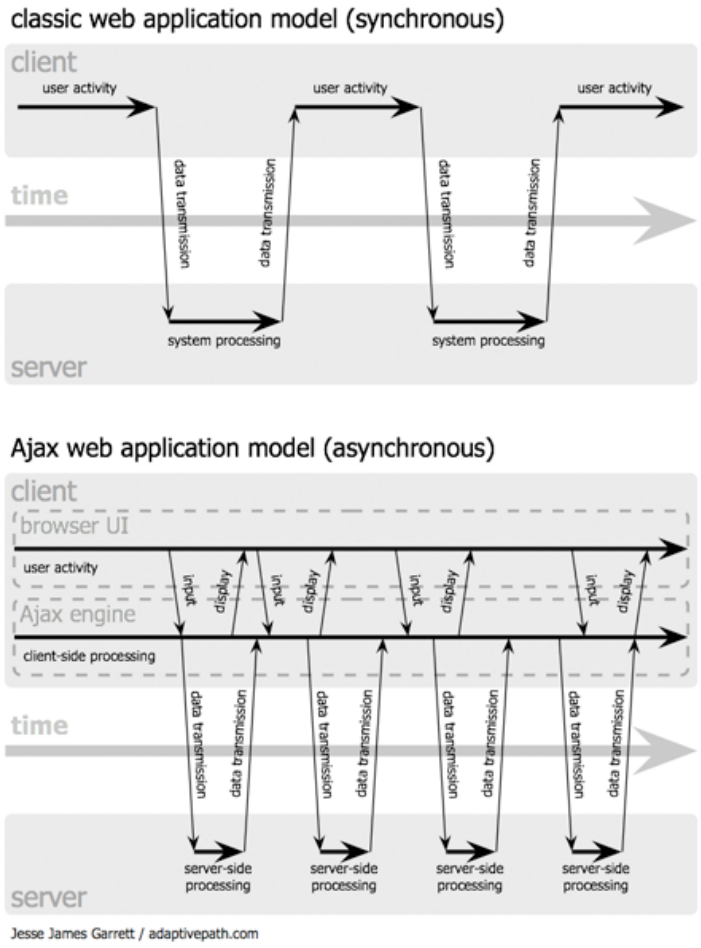

Figure 1.1: "classic" vs. "AJAX" model

source: adaptivepath.com

thesis, I will research how to do this, but before we start looking at optimizing the code, there should first be a way to benchmark the code in order to indicate the performance improvements.

1.1 Problem Statement

In this thesis I will discuss the challenges that arise with the shift from a traditional page-based website to an interactive web application. To have a better understanding of the evolution of web applications, I will start with describing

the history and evolution of the World Wide Web and HTML itself. After this I will discuss the some of the newly arisen challenges and pick one of them for a more elaborate investigation on how this can be dealt with for a web application.

1.2 Research Goals

My goals is to describe the challenges that arise when developing a web application. As part of the development team of the social network company Hyves I want to find a solution of one of these challenges. How to optimize client-side code?

1.3 Research Methodology

For the implementation part of this thesis the action research method is used. Action research is an iterative process in where you diagnose the problem, then do an action intervention and at last you reflect and learn from this change [1]. In this case the action intervention corresponds to the tried approaches. During these first trials, we came to new insights which can be seen as the reflection and learning phase. After this phase we re-thought the the problem definition and started making changes again (action intervention).

1.4 Structure of the Thesis

This thesis can be split up into four main parts. The first part is the general introduction in which the subject and the goal of this thesis is described. Then there is the second part which consists of an overview of the history of the World Wide Web. In the third part talks about the emerging issues that you get from turning a website into a web application. Finally, the last part will be about implementing a solution for one of the emerging issues.

Chapter 2

Description of Problem Domain

2.1 History

2.1.1 The Invention of the Web

In 1989 Tim Berners-Lee proposed the concept of the Web at CERN, the European Laboratory for Particle Physics in Geneva. The concept described a way of structuring documents such that you could link them to each other, where ever they were and thereby making a web of documents wherein it is easy to browse through them. This was very useful for scientific documents that were stored on servers all over the world. It used to be difficult to see what information was already available, but this new possibility of referring and direct linking from one document to another, even if that other document was on a computer on the other side of the world was a huge improvement. He created the first web browser in 1990 called WorldWideWeb to view the web pages. This was also the year the first web page was created. [2] The document format that Berners-Lee created for web pages was called HTML, which stands for HyperText Mark-up Language. The concept of HyperText derives from an application for the Macintosh which was called Hypercard. The application provided a way to create virtual cards that could contain text and images and trough which you could navigate by clicking on links. The format and syntax of HTML was based on a existing format, SGML (Standard Generalized Mark-up Language). This mark-up language what already used and recognized by the international scientific community. SGML contained tags for formatting text, such as tags to created lists, paragraphs, different heading levels, etc. The only big addition that HTML had in comparison to SGML at first was the anchor tag. The anchor tag is used to create a hypertext link to another HTML document. This additional tag was the key feature of what made the Web. It contains a part that is called the URI (Uniform Resource Identifier). This URI consists of the protocol,

domain name and the path. The simplest form of the URI syntax is as follow: `protocol://domain.name/path`. An URI that identifies a document resource is also called an URL. (Uniform Resource Locator). The protocol that Berners-Lee created to provide and retrieve HTML was HTTP (HyperText Transfer Protocol). It describes how the server and the client should communicate to transfer HTML documents.

```
<html>
<head>
    <title>Example page</title>
</head>
<body>
    <h1>Heading</h1>
    <p>This is a paragraph, with a
        <a href=http://www.example.com>hyper link</a>
    </p>
</body>
</html>
```

Another important development which made the Web possible was the domain name system, invented in the middle of the 1980s. The domain name system is a system that gives people the possibility to use domain names, which are letters separated by points, instead of IP addresses, and therefore are much easier to remember. Domain names are mapped to IP addresses by a program called DNS (Distributed Name Service). Domain names made computers all over the world easier to connect to and therefore are an important part of the URL. All the ingredients like domain names, SGML, Hypercard and the Internet where there and Berners-Lee combined all of them and thereby created the Web. [3]

2.1.2 Browser Wars

In 1993 a new web browser was released by the National Center for Supercomputing Applications (NCSA): The Mosaic web browser. This browser supported extra features of HTML which it originally did not support, like nested lists and one of the most important ones: images. The NCSA also developed the NCSA HTTPd, which was the first HTTP server that supported the ability to use CGI. CGI stands for Common Gateway Interface and it defines a way for web servers to use executables to generate web pages. The executables can make use of parameters given to it by the web server which retrieves these parameters from the URL and from data posted by forms. After execution, the result is served by the web server to the client as an HTML web page.

In October 1994, one of the developers of the Mosaic browser, Marc Andreessen made a press release of the first Netscape Navigator browser. It was based on Mosaic, but published by the company Netscape Communications with commercial possibilities in mind. When in September 1995 Netscape Navigator

2.0 beta was released, JavaScript was introduced. JavaScript gave web pages the possibility to execute code within the browser, making web pages more interactive without doing an HTTP request.

Microsoft began to see the potential of the WWW but had problems with licensing the Netscape Navigator and for that reason they turned to Spyglass. Spyglass planned to make a browser based on NCSA's Mosaic browser which they licensed in May 1994. [4] Microsoft used Spyglass's Mosaic browser as the foundation of Internet Explorer 1.0 which the released in August 1995. Internet Explorer 1.0 was bundled in the Microsoft Plus! package which was available for their operating system called Windows 95.

This was the start of the first big browser war. At this moment, Netscape's Navigator was by far the largest browser on the market (80%) and this made Microsoft feel threatened. Because Microsoft had a lot of financial resources, they could give their browser away for free to all their user, whereas Netscape charged commercial use of their browser. Next to that, Microsoft used it market leadership of its operating system to pressure computer manufacturers not to pre-install Netscape Navigator on the machines, which actually led to the United States v. Microsoft anti trust case [5]. However, these moves from Microsoft had success and its market share kept rising to a staggering 96% in 2002.

In 2004 Mozilla's Firefox was introduced, which can be seen as a beginning of a second browser war. It was an open source browser based Netscape's Navigator. Since Microsoft neglected to innovate their dominant browser, more and more people started using alternative browsers, like Firefox. Also the Opera browser became a free-ware browser, meaning its free version did not had advertisement anymore, but despite being innovating with features like mouse gestures, Opera never reached an significant market share.

Firefox reached it's biggest market share in 2010, where it had 31% of the market. Even though Internet Explorer's share keeps decreasing, Firefox's share also is decreasing, which is a result of a new upcoming browser, Google Chrome. In 2008 the search engine giant, Google, first released their Google Chrome browser. It is based on WebKit, the rendering engine behind Apple's Safari browser, which was also around for a while, but did not have a big user base since it was mainly used on Apple's Mac OS. Just like Mozilla Firefox, Google Chrome is available for Window, Mac and Linux. Mainly because of it's clean and simple interface combined with high performance Google Chrome is winning territory. Also because of the lack of support of HTML5, Internet Explorer was loosing grounds, but with their latests releases, 8 and 9, they made a big effort in being more compliant with the updated HTML, JavaScript and CSS standards. All the new releases of the major browsers now have significant support for HTML5, but there still exist differences.

2.1.3 The HTML Standard

The first version of HTML, 1.0 was just a stripped down version of the existing SGML extended with the anchor tag. This anchor tag was Tim Berners-Lee's invention. Since browsers like Mosaic were becoming more popular, which all

implemented their own new HTML features, the World Wide Web Consortium (W3C) was established in October 1994, to keep an eye on HTML as an open standard and other Web standards and to Lead the Web to Its Full Potential. This led to the creation of the specification of HTML 2.0, which included the added features used by Mosaic and other browsers. HTML 2.0 was officially published on November 24th 1995 as an IETF RFC. The Internet Engineering Task Force (IETF) is an organized activity of the Internet Society (ISOC). This task force has the goal to make the Internet work better and does this by providing technical documents which contain standards and guidelines called RFCs (Request For Comments). After publication the HTML 2.0 specification was still being updated till January 1997. Some of the new important features of this version where, images, tables and meta information.

Because the growth of the Web was enormous and the new browsers vendors like Netscape and Microsoft were competing for market share, they kept adding their own invented tags to make their browsers more fancier, but this polluted the Web with non-standard HTML.

In January 1997 HTML 3.2 was first version which was published as a W3C Recommendation instead of a IETF RFC because IETF shut down its HTML Working Group. A W3C Recommendation is a technical standard document of the W3C. This document was now also being written with the input of the commercial browser vendors, because the W3C acknowledged the fact that they could not define the standard them self anymore due to the influence of the browser vendors on the use of HTML. HTML 3.2 did not get support for all proprietary tags, such as Microsoft's `<marquee>` tag and Netscape's `<blink>` tag [2] but adopted support for other visual markup tags by Netscape including the `<font>` tag.

The problem with these newly added tags for visual markup was that they interfered with the initial function of HTML, which was to bring structure to a document, so it could be interpreted by any computer system, even if it did not have a graphical interface. HTML was not intended to contain information about how the document should be displayed. [6]

When in December 1997 HTML 4.0 was published most of the visual markup tags were made deprecated because at that time a new system for styling was finally becoming mature, Cascade Style Sheets (CSS). CSS is a standard way of telling the browser how to display certain tags. With CSS you can, for example, say: all paragraph tags (`<p>`) should get a margin of 10 pixels. The browser than knows how display paragraph tags. In HTML 4.0 you could set the mode of your document to strict, which would not let you use the deprecated tags anymore. If you still wanted to use them, you were able to use the transitional mode. [7] In the end of 1999 HTML 4.01 was officially published with contained some minor modifications.

In the following years a new buzz word appeared, XML. While XML had it's first Working Draft in 1996, it took a few years before being used. XML stands for Extensible Markup Language and it is also derived from SGML. It has a more strict syntax than HTML but more flexibility because one can add their one tags. In 2000, W3C decided it would be nice to combine the

power of HTML 4.01, which was widely used, with the possibility of adding new elements of XML and the started the development of XHTML 1.0. But because XHTML was much more strict than HTML, browser vendors decided to add new functionality to the original HTML4 instead of pushing this new XHTML standard.

In 2004 the W3C chose not to support the evolution of the HTML standard, but to continue the XML-based XHTML and this resulted in a new group called WHATWG which was a joined initiative of Apple, Mozilla and Opera. These three browser vendors started working on a new version of HTML called HTML5. [8] Since then, this new HTML5 standard is getting a lot of love and attention. Later in 2006, the W3C got interested in the development of HTML5 after all and decided to participate since 2007 and the both groups worked together since.

HTML5 adds support for a lot of new features that were not present in earlier specifications of HTML. Among those features are support for drag-and-drop, timed media playback, the canvas element, off-line storage database and browser history management.

There are also some new technologies that are referred to in the media as being HTML5 but are actually not officially part of the HTML5 specification, but have their own specification [9]. One of those technologies is Web Workers, which allow a developer to execute CPU intensive code in a separate thread, so that other parts of the page keep working, for example the user interface. Also the Geolocation API is such an example and is already implemented in a few browsers. This API (application programming interface) gives developers the opportunity to retrieve the users physical location, through for example GPS.

2.1.4 JavaScript

Netscape's Brendan Eich created Mocha, which was later called LiveScript. It was first shipped in a beta release of Netscape Navigator 2.0. Later it was renamed to JavaScript in Netscape Navigator 2.0B3, which was released in December 1995. [10] The name suggests that the language was derived from Java, but this was not the case, some people say it comes from a deal Netscape had with Sun Microsystems, the creator of the Java platform, to bundle the Java runtime environment with Netscape Navigator. Others say is was to let JavaScript lift on the success of Java which was the hot programming language for the Web at the time. Microsoft added support for JavaScript in their Internet Explorer 3.0B1 release in May 1996. Microsoft version of JavaScript was actually called JScript, but was almost identical and partially compatible.

JavaScript made it possible for web page developers to write scripts to interact with the HTML mostly by manipulating the Document Object Model (DOM). The Document Object Model is a object representation of the HTML document which is available in the JavaScript code. Manipulating this object results in changes in the presentation of the HTML document itself. [11]

The first implementation of JavaScript was not very powerful. This was due to the fact that only a small part of the DOM was available. This was later called

Level 0 DOM. This, not officially backed by the W3C, specification was invented by Netscape to make it possible to interact with the HTML from JavaScript. Level 0 DOM was limited in the fact that you only had access to the following type of elements: images, forms, links and anchors. After the document was completely loaded, these elements where appended to a corresponding array, for example an image would be appended to the array `document.images[]`. The Level 0 DOM model was also adopted by Microsoft in the Internet Explorer 3 browser because of their need to compete with Netscape and the fact that people were already using the model. In a later versions of the browsers Netscape and Microsoft added a way to access layers of the page to give JavaScript developers the opportunity to move these layers around, which was called Dynamic HTML (DHTML), the big buzz word around the year 1998. Netscape invented `document.layers` and Microsoft `document.all` for this single use. [12]

In October 1998 the W3C published the first version of the Document Object Model (DOM) Level 1 specification. It defines 'a platform- and language-neutral interface that allows programs and scripts to dynamically access and update content, structure and style of documents' [13] . With this model it was possible to access all element in a document, instead of a limited set which was the case in Netscape's version. The DOM Level 1 consists of two parts: Core and HTML. The Core part describes the low-level interfaces for representing any kind of structured document and for XML documents. The HTML part is a more high-level interface which is based on the Core but adds more support for HTML documents in specific. The nice thing of this specification that is was adopted by both Netscape and Microsoft.

Because of the fact that the DOM is a representation of a HTML document and a HTML document has a tree structure - a tag can contain one or more children tag and those tags can also contain one ore more children tags and so on - the DOM also has a tree structure. The DOM Level 1 specification gave developers a way to walk through this DOM tree and modify it. All elements in the DOM, also called nodes in the context of a tree, have properties like `firstChild`, which returned the nodeś first child and `childNodes`, which was an array of all the children of the node. Furthermore there are methods like `document.createElement` which created a new element. This new element could then be appended to a existing node with this node's method `appendChild(newElement)`.

Specifications for DOM Level 2 and 3 followed. The DOM Level 2, which consisted of several parts had a few additions worth mentioning. The famous and most used one is `document.getElementById(id)`. With this function you can retrieve an element with a given value of its id attribute. Nowadays this function is used a lot of retrieving a specific element on the page. Another important specification was that of the events model. Netscape and Microsoft had already support for events, but they had their own different models, because of their ongoing battle. This new third event model created by the W3C and based on Netcape's event model, was supposed to replace the existing two but it took Microsoft more than 10 years till they adopted the model in their latest released browser Internet Explorer 9. [14]

One of the most important developments in the evolution of JavaScript is AJAX. AJAX is a term first used by Jesse James Garrett in 2005 and stands for Asynchronous JavaScript and XML. It describes a way to request data from the server from within JavaScript. It is asynchronous because after triggering the HTTP request, the JavaScript continues to execute and a provided callback function is called in case of a HTTP response of the server. The XML part comes from the fact the data was originally in XML format, therefore the object responsible for doing the HTTP request was called XMLHttpRequest object. See figure 2.1.

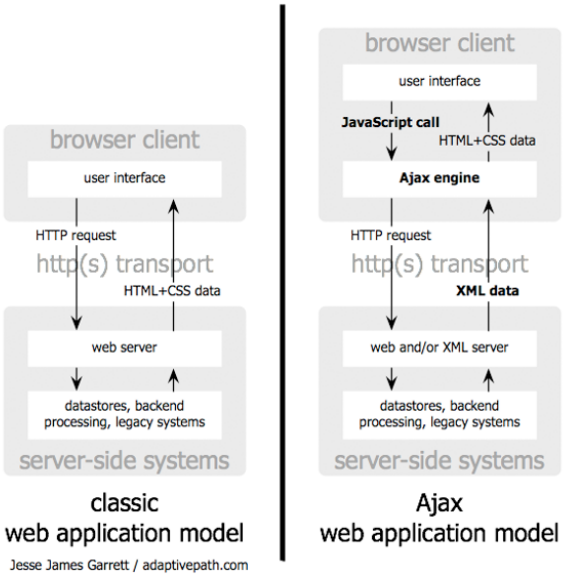

Figure 2.1: "classic" vs. "AJAX" model

source: adaptivepath.com

The origin of the XMLHttpRequest object lies at Microsoft's Outlook Web Access which was the web interface for Microsoft Exchange. Developers at that project needed a way to exchange data with the server with JavaScript without reloading the whole page. They created an ActiveX component for Internet Explorer which was released with Internet Explorer 5 in 1999, called XMLHTTP because it was part of the XML library. [15]. Later other browsers implemented a version of this object called XMLHttpRequest and in 2006 the W3C published their first Working Draft on the object. AJAX became the big buzz word of the following years. It also revived the use of JavaScript, because it showed a way of creating web applications that resemble the flow and responsiveness of native desktop applications.

2.1.5 CSS

Cascading Style Sheets (CSS) provide a way of adding style to a HTML document hence defining the presentation of the document[16]. This can be done by formulating rules that describe how certain tags should look like, in respect to color, margins, font size, etc.

The first specification of CSS was published in December 1996 [17] but CSS has also evolved in the years. CSS2 was developed and gave web designers more control over the positioning of elements. CSS2.1 followed, which was a cleaned up version of the previous one. Parts of the specification of CSS2 that were poorly supported were removed and some new browser-invented features were added [18].

The latest big step is CSS3. CSS3 adds more advanced features for customizing the presentation. It for example lets you use gradients as background, give boxes rounded corners but also it brings support for transitions in CSS. The concept of transitions is completely new in CSS. 'CSS Transitions allows property changes in CSS values to occur smoothly over a specified duration.' [19]. The problem with CSS3 is the lack of overall support. There is not one browser that support every feature of the CSS3 specification.

2.2 Five Emerging Issues

Websites used to be a bunch of pages linked to each other using hyperlinks. Every click on a link would invoke a new HTTP request and resulted in a completely new HTML document that was retrieved and rendered in the browser. The biggest action one one page was when it would contain a form. This form could then be filled in by the user, and sometimes there would be some input validation done using JavaScript, but after submitting the form the whole page disappeared and the resulting page would be rendered from scratch.

With the increasing power of JavaScript thanks to the introduction of technologies like the XMLHttpRequest object, a single page can behave like an application, not needing to be fully retrieved and re-rendered anymore. Such pages can be called web apps. Most of the changes in the interface of the application during interaction can be done with JavaScript and some CSS and when there is a need to exchange data with the server, this can also be invoked asynchronously by JavaScript, keeping the user on the same page and giving them the feeling of working in a native desktop application.

Creating web apps, instead of traditional websites brings new challenges. At Hyves, the biggest social network site of the Netherlands, developers are constantly improving and innovating the website. While improving the front-end, Hyves more and more turns into a web app. During this transition, the following challenges emerged:

- How do you keep the browser's history and back button working
- Search engine optimalization (e.g. what does Google see v.s. what the user will see)

- Modularization, code-generation (Model Driven Architecture)
- Hiding differences in target platforms (HTML5 desktop browser v.s. an enormous range of mobile platforms)
- Profiling and optimizing

These are some of the challenges web application developers face. As part of describing the domain of the paradigm shift within web development and the problems is accompanies, I will now describe these five challenges and pick on of them for a more in-dept analysis.

2.2.1 History/Back Button

When the state of a AJAX based web application changes, the user stays on the same page and therefore the web browser does not recognize this as a different state. The problem with this is that there is no history of the states in the browser, meaning the back an forward button become useless. Another related problem is that when a user bookmarks the address and revisits the bookmark later on, the web application we be in its initial state, instead of the state in which the user bookmarked it.

There are different libraries available that provide a solution for these problems. For example the YUI Browser History Manager[20] which is part of Yahoo's JavaScript libraries. Another well known JavaScript framework that gives you support for this is the dojo toolkit with its dojo.back library [21].

All the solutions utilize hash fragments. The hash fragment is the part of the URL after the hash (#) sign. It is meant for URLs to be able to point to a section within a page. It is ignored by the server but it is originally used by the browser to scroll to corresponding part of the page and most importantly is available within JavaScript so thus in the web app. Because the browser records URLs with different hash fragments as different URLs, it stores them in the history. The hash fragments can also be used by the web app to restore the corresponding state.

Besides from JavaScript implementations to support history for web apps, the draft version of the new HTML5 specification also contains a History API [22]. The suggested API does provide a solution for this problem, but it still is just a draft version and not all web browsers support it already.

2.2.2 Search Engine Optimization

Websites that use AJAX to load the content are not crawlable by search engine bots. Search engine bots look at the initial HTML that is served by the web server and they do not execute any of the JavaScript that is included in the page. Therefore it will not see the content that is obtained or generated with JavaScript.

For pointing to an exact state of an web app, there is the 'hash fragment' method that is described earlier. The only problem that remains is that since

the crawl bot does not execute JavaScript and therefore will not see the actual state that is referred to.

Google now offers a solution to make web apps be able to be indexed[23]. The key is to use #! instead of # for the URLs. The Google bot will then convert the 'hash fragmented' URL, or as how Google calls it an pretty URL to an URL containing a query parameter called _escaped_fragment_ which Google calls the ugly URL. The _escaped_fragment_ contains the hash fragment from the original pretty URL, but because it is now passed through as an query parameter, it is available for the server. The server is now responsible to return the HTML snapshot of the page as of how it would look like when the hash fragment was given and the JavaScript was executed.

This can be done by executing the JavaScript server-side by loading the page with a headless browser. A headless browser is a browser without a graphical user interface which you can control using a programming language. It can interact with the HTML document as a normal browser is able to and also supports JavaScript. With a headless browser you simulate what a normal JavaScript supported browser would return when going to the 'hash fragmented' URL and save this generated HTML, which is called the HTML snapshot. This snapshot can be returned to the Google bot when it requests the ugly URL. The most common headless browser is the Java component HtmlUnit and is mostly used with Java's testing framework JUnit.

An other way to create the HTML snapshot is by having a smart mapping between the client-side JavaScript and a server-side scripting language. if you have all the building blocks of the page available server-side, you may be able to use them to create the HTML that otherwise would be generated by the client-side JavaScript. This may be hard to accomplish for a web app that is already built, but if you take it into account when you start designing the architecture, it is not to difficult.

After the server returns the HTML snapshot, the Google bot will index it and will use the pretty URLs for referring to it in the search results. That way the users will get the full AJAX experience and not the HTML snapshot that are intended for the Google bot only.

2.2.3 Modularization, Code-generation

Because of the shift from server-side programming to client-side programming with JavaScript, more code should be written in JavaScript. The fact that JavaScript is getting more important role in the architecture demands it to be written in a more modular fashion. In this way we can better apply the principle of DRY, don't repeat yourself. Next to satisfying the DRY principle, modularization of code keeps the code nice and prevents global namespace pollution.

Code-generation is where source code is automatically derived from abstract models, for example UML models. This process is called Model Driven Architecture, MDA [24]. The biggest benefit of MDA is the fact that the models that describe the business logic are platform independent. So one designed model

could be used to generate source code for different platforms, being it application code for a native application or JavaScript for a web application.

The principle of modularization is already commonly used in other programming languages, but is quite new for JavaScript. Since JavaScript does not natively support namespaces and object inheritance, there is not one official way to do this. Because of JavaScript's flexibility, multiple solutions for this problem exists. The most commonly supported approach for JavaScript objects and modules is the following is called the JavaScript Module Pattern and here is an example of it:

```
var MODULE = (function () {
    var my = {},
        privateVariable = 1;

    function privateMethod() {
        // ...
    }

    my.moduleProperty = 1;
    my.moduleMethod = function () {
        // ...
    };

    return my;
}());
```

In the example above MODULE is a global module with a private variable `privateVariable` and private method `privateMethod()` because they are defined within the scope of the module itself. The property `moduleProperty` and the method `moduleMethod()` are also defined in the same scope, but references to this property and methodo are returned by the module making them accessible from outside the module meaning they are public. Ben Cherry has a good description of this pattern [25].

At the moment there are not a lot of solutions for MDA in web applications. One of the few solutions out there is WebRatio [26]. WebRatio is a commercial product which converts your models into Java web applications. However, WebRatio generates server-side code instead of client-side JavaScript and HTML. Therefore it is not the solution we are looking for. A real client-side MDA solution which produces JavaScript and HTML is still missing. Creating such a solution can be a very useful. However, making it so that the generated code is would work in all the different browsers can be a tough job.

2.2.4 Hiding Differences in Target Platforms

The nice thing about web applications is that every computer platform has a web browser these days, so if you write a web app, you should be able to run it on all platforms instantly. Unfortunately, this is not the case. Most browsers

handle HTML, JavaScript en CSS slightly different. For example, there are a lot of web browsers for mobile phones that do not support JavaScript.

There are two solutions to handle the different target platforms. The first one is to create separate interfaces for each platform. The advantage is that you do not have to take other platforms into account. You can focus on one single browser. The other side is that you must create an new interfaces for every different platform. This is not only a lot of work to create, but much harder to test and maintain.

The second solution is make one front-end that then can be used on all platforms. In that case it is important to take care of graceful degradation. Graceful degradation in this context means that when a certain feature is not available on a platform this will not result in errors and in some cases a alternative solution can be provided. The nice thing about this approach is that all the users can get almost the same experience and there is only one interface to maintain. The disadvantage is that because of the big range of different browsers and devices, keeping the interface compatible with all of them is a difficult task.

There are libraries available to detect which features are available in the current browser the user is using, such as Modernizr [9] [27]. This library does rely on JavaScript, so it will not work on all mobile browsers. To decide which browser features can used by the clients with lack of JavaScript support, you can only rely on the user agent string of the browser. This is a string that a browser sends with its request to the server identifying which kind of a browser it is. The user agent string contains the name of the browser vendor and the browser version. This is certainly not the preferred way of feature detection, because you must know exactly which browser version has which features.

2.2.5 Profiling/Benchmarking and Optimization

Profiling is the process of measuring CPU and memory usage of an application during execution mostly for finding places where the application can be optimized. Since more of the application logic is moving from the server to the client, profiling at the client-side level becomes more important. There are browser plug-ins that provide the ability to profile a website, for example Google Chrome's Speed Tracer [28] extension and the Firefox's add-on Firebug [29] combined with Yahoo's YSlow [30]. Another great package is dynaTrace [31] which supports Internet Explorer en Firefox.

All these tools provide interesting insights in what are the bottlenecks for your website. The problem with these tools are that there is not one tool that can cover all platforms, this means to see how your site is performing on different platforms you must use different tools. An other problem is that the different tools do not give you the same type of output, which makes the output results difficult to compare.

2.3 Choice and Motivation

Hyves is a social networking web site with more than 10 million users. They are responsible for a staggering 9 million page views per hour. A page view can be seen as interaction on the site where there is data being send or retrieved from the web server, in other words an HTTP request. To be able to deliver these requests quick enough, Hyves has a server park of about 3000 servers.

But the high performance of Hyves is not only due to the big server park. A lot of effort is put in optimizing the code. Profiling utilities are built into the code to monitor which parts of the code are slower than others and these slower parts are being optimized.

There is also a lot of caching, using packages like Memcached, to minimize database lookups and hard disk reads, which are slow operations. Memcached keeps data in memory so it can be fetched very fast. This requires lots of RAM memory and is not persistent, meaning you will lose the data when rebooting the server. Therefore databases are still needed.

An other type of big optimization on the server-side is the use of Facebooks HipHop for PHP. Hyves, like a lot of other web sites, is mostly written in PHP. Since PHP is a scripting languages which is being compiled on-the-fly, it is not the fasted language. The problem is that switching to a compiled language like C++ or an interpreted one like Java, is not an option mostly because of the huge code base and the lack of resources in the form of developers. Also, a scripting language has the advantage of a shorter development cycle, because you do not need to compile it for every change you make. Facebook also coped with these problems and developed a program that transforms PHP code into C++ code. This C++ code can then be compiled into an executable. The web server then uses that executable to process the request and generate the response. Because the code is already compiled instead compiled on-the-fly for each request, the speed up is enormous and Facebook is now able to serve 70% more pages with the same hardware thanks to HipHop [32].

Since the possible performance improvements at the server-side at Hyves are becoming smaller and Hyves more and more turned into an application that runs in the browser instead of on the server, performance of the client-side code is becoming an increasingly important part of the total speed of the web site. After the page is generated at the server-side it gets served to the client, the browser. From this moment the perceived performance of the page depends on the time it takes for the browser to retrieve related content and to render and execute all this content. At the moment, not much research is done on client-size optimizations for Hyves. That is why a lot of results can be gained.

The speed of a website is of big importance, or to quote Jeff Atwood: "Performance is a feature [33]." In 2006 Google noticed that when they increased the number of search results per page from ten to thirty. Instead of making people happy, the traffic dropped by 20%. After searching for the reason of this decrease, they stumbled upon an uncontrolled variable, the rendering time. The page with 10 results used to load in 0.4 seconds and the new version, with 30 results, took 0.9 seconds to load. This 0.5 second increase of load time trans-

lated into a 20% drop in traffic, which directly corresponds in a 20% drop in revenue [34].

Mordern web applications in contrast to classic websites consist for a major part of client-side code. Until recently the performance of client-side code was not taken into account but since speed has become a big issue and that there could be a lot result gained at the client-side, I chose to concentrate on how the front-end of Hyves can be optimized.

Chapter 3

Action Research

3.1 Research Setup

Due to my work at Hyves as a developer I got the change do research into how the front-end of a big web application can be optimized. I was teamed up with a colleague who is specialized in front-end development. The research is setup as an action research. Action research is an iterative process of analyzing the problem, make changes and reflect on how these changes affected the problem. In this specific research we have a problem which is the need for a way to optimize the front-end of a web application. We will then try available tools and implement possible solutions. After these changes we reflect on the results that our changes have made and when the results are not good enough, the process starts all over again. We reshape our problem definition, make new changes and reflect on those changes.

3.2 Project Kickoff

The project started with a kickoff meeting to generally discuss what we wanted to achieve in the subject of front-end performance. What kind of benchmarking data would be nice to have about the the front-end? Which pages are most critical performance-wise? The meeting lead to a few action points:

- Investigate how other big websites are doing front-end benchmarking
- What tools are available for measuring front-end performance
- Determine the most critical pages / regions of the site in means of traffic
- Determine what kind of metrics are relevant
- Look into setting up a front-end lab
- Create a wiki page on the company's internal Wiki

The most extensive research after this meeting what was done was for the available tools for measuring the front-end performance. Important is that the such a tool can be used in an automated environment so it can be integrated within the current automated testing system that is set up at Hyves so that when someone makes a change to the code base which has a negative impact on the client-side performance, this can be detected automatically, just like already happens when a bug is introduced that generates an error. This progress is also known as continuous integration, which is used in software development where multiple members of the same team work on the same product. With continuous integration you automatically integrate the code changes of the members on a regular basis and perform unit and functionality tests on the integrated code to see if product still works as it should be [35].

Furthermore, the tool should give as detailed information as possible, considering the huge complex code base we are dealing with. An overall loading time is nice, but a more in depth view at a function level would be a lot better.

Now I will discuss some of the tools that we found:

3.3 Research Phase #1: Tools Overview

Tool	Desired metrics	Cross Browser	Able to Automate
dynaTrace	yes	partly	no(*)
Speed Tracer	yes	no	yes
YSlow	partly	partly	no
Page Speed	partly	partly	no
Boomerang	partly	yes	yes

Table 3.1: Comparison of the tool

*the free edition of dynaTrace is not easily automated

dynaTrace

dynaTrace is an elaborate profiler for web applications. It gives you the possibility to discover almost every bottleneck in the progress starting with requesting a web page up to interacting with the website itself. To gather its information, it hooks into Microsoft Internet Explorer browser or the Mozilla Firefox browser. This information gathered can be put into the following sections:

- HTTP requests
- JavaScript calls
- Page rendering

dynaTrace is pretty complete product. It gives detailed information about the time it took to render the page and to execute JavaScript calls but due to the fact that it is not an open source but a commercial product it has a free edition called dynaTrace AJAX edition and multiple paid editions. The biggest disadvantage of the free edition is that it lacks any kind of documentation. Despite the fact that dynaTrace states on their website that the free edition supports integration into automation frameworks, how this is done keeps a mystery since to be able to access the documentation and forums a premium account is required which costs a significant amount of money.

The dynaTrace AJAX edition is a good tool for manually analyzing a website because it gives a large amount of interesting data, however, the fact that it lacks any documentation on how to integrate it into a automation framework this tool is not an option for this project.

Speed Tracer

Speed Tracer is a Google Chrome extension, which allows you to monitor everything what happens when loading a website. It does this in a similar fashion as dynaTrace does. Speed Tracer gives you metrics that are gathered from low level instrumentation points inside the browser' [28]. These metrics give you an insight which parts of the website are slowing done the performance. The collected data is about the retrieval of network resources, painting events of the browser, parsing of HTML, style recalculation, garbage collection and more, so it is very elaborate. Speed Tracer sets all those events on a time-line giving a nice graph which gives you a good overview of the performance of the site.

Another nice thing of the Speed Tracer project is that is also contains a headless version of the extension. This headless version records the same data as the normal version except it does not provide the graphical user interface, but instead it can post the collected data to a given server. In this way you can collect the data automatically, which makes this version of the extension great for use in combination with an automation framework like Selenium.

The fact that Speed Tracer is free and open-source is also a big advantage compared to dynaTrace.

YSlow

Yahoo's YSlow started as a Firefox add-on, but today is also available as an extension for Google Chrome and as a bookmarklet which makes it available in other browsers as well [30]. YSlow will analyze your website and will return a list of things that are slowing down the website and tips how to deal with the found issues and gives your site a performance rating. The rules that are used to rate the performance of your site is based on a number of best practices for speeding up a website [36].

While giving an interesting insight in the performance of the website, in particular the overall load time, YSlow does not give you the detailed browser metrics about JavaScript execution and render time, which tools like dynaTrace

and Speed Tracer do give you. YSlow gives you some handy tips that you should try to apply as much as possible for every kind of website, but it does not help with pinpointing slow peaces of JavaScript and CSS itself.

Aside from the lack of detailed profiling information, YSlow is not suitable for automation. For automation you would like the tool to run automatically - not needing any interaction to start - and send its collected data to a server so it can be saved and analyzed afterwards. YSlow does not provide support for this.

Page Speed

Google's Page Speed can be installed as a Google Chrome extension as well as an Firefox add-on [37]. Just like YSlow, Page Speed analyzes the pages loading time by looking at common mistakes and it will give you tips on how to take care of these mistakes. Things that these kind of tools checks for are minified JavaScript, CSS and HTML.

Minifying is stripping all unnecessary bytes from the code, such as white space characters, like spaces, line breaks, tabs. With minifying JavaScript, variables names within the scope of a function are also being shortened. Minified files are smaller and therefore can be downloaded faster by the browser.

Page Speed also looks at the usage of images. Are the images compressed good enough? Compression means smaller images sizes, which leads to shorter download times. Are the dimensions of the images set in the image tag? This can speed op the rendering of the page because the render engine knows at the start how big of a box it should reserve for the image, so it is not needed to re-render after the images is loaded. Are CSS-sprites being used? A CSS-sprite is one image file, the sprite, containing multiple images next to each other. When wanting to use one of the images in the sprite, in the CSS the sprite itself is used in combination with the the top offset, left offset, width and height of the image within the sprite. The big advantage of using CSS-sprites is that only one image needs to be downloaded, instead of an image for every image in the sprite. This saves HTTP request, which saves time.

Other things that tools like YSlow and Page Speed look at are related to caching, for example by setting the correct HTTP headers. Static resources, like JavaScript files and CSS files tend not to change very often. For such files, the web server can send an additional HTTP header with the file as it is being served, containing how long the file can be cached by the browser, preventing it to download the resource every time the page is being visited.

Page Speed resembles YSlow, in that it gives you a nice list of points on which you could optimize your website, but it also lacks the possibility of integrating it in an automation framework and the possibility of sending the data to a beacon server.

Boomerang

Boomerang, created at Yahoo! is JavaScript-based script for measuring user perceived performance [38]. The script can be included on the website that you want to investigate. It can gather information about the load time, measured from the moment the user clicks a link on your site until the `onload` event is triggered by the browser. It sets this starting point by setting a cookie with the current time at the moment the user clicks a link. In the resulting page it reads this time from the cookie and calculates how much time was passed.

If the cookie is not present, this can be in the case when the user comes from a different website and therefore the cookie was not set, Boomerang will try to use the new Navigation Timing API. This API is still work in progress at the W3C (Wang, 2011) and gives a lot of measurements concerning navigating over the web. It for example gives the time when the previous page had its `unload` event. The `unload` event is called when leaving a web page and the time of this event can be used as a starting point in case the cookie does not exist.

This passed time since the starting point that is measured can be seen as the loading time, or how the Boomerang team calls it: round trip time.

Because the `onload` event does not always indicate when the user can interact with the page and Boomerang is all about user perceived loading time, it is possible to specify this event as a developer by calling a certain boomerang method.

Another interesting feature of Boomerang, that does not exists in the other tools that are mentioned, is the ability to measure the bandwidth and network latency. The latency is measured by downloading a 32 byte image for 10 times in a row and looking at the time it takes to download them. The first download time is not taken in to account because it is more expensive, but the over the other 9 download times, the arithmetic mean is taken and this can seen as a measure of latency. For the bandwidth Boomerang measures the download time of images that differ in size. The median, standard deviation and the standard error from these download speeds is send back to the beacon as a value for the bandwidth.

3.4 Research Phase #2: First Trials

3.4.1 Plain JavaScript Approach

Tools like Speedtracer and dynaTrace give an detailed overview of what parts of the page contribute what to the overall loading time of the page. But the biggest problem was that there was no measurement tool which would work across all platforms.

In an attempt to create pure JavaScript-based measurement tool that could time some basic important events on all platforms, it appeared that some browsers start executing the JavaScript at the top of the page before rendering the page and other browsers do not. This caused the JavaScript implementation to fail, because there was not one uniform starting point in time and therefore

it was not possible to retrieve an approximation of the first byte time. But not only is the moment of executing the first bytes of JavaScript different across browsers, but Internet Explorer on Windows XP and Windows Vista have the problem that the internal timer of the browser is only updated once in 15 milliseconds. This means an average deviation of 7.5 milliseconds. Timing a piece of JavaScript using native JavaScript would look as followed:

```
var start = (new Date).getTime();
/* Run code that you whould like to measure */
var diff = (new Date).getTime() - start;
```

But because of the deviation, this measuring method is not reliable [39].

3.4.2 Speed Tracer Approach

When we realized that there no way of profiling the client-side performance on each platform in a uniform way we focused on which tool would give us the most useful data possible and next to that is the most flexible. Of all the tools we looked into, Google Chome's Speed Tracer was the best option. It returns accurate and specific timing metrics and it gives you the ability to set your own timeline marks using WebKit's Logger API, the marks will be available in the generated report.

Another key feature of Speed Tracer which was necessary in order to be able to use it for this project was that there was a headless version. This headless Speed Tracer function gathers the same data except it does not have a graphical interface but it could be triggered from the JavaScript on the page, or by adding special query string to the URL of the website. The fact that you don not have to interact with it by clicking on buttons, but with JavaScript of query string parameters makes it very suitable the use with an automation framework.

One side effect of the fact that Speed Tracer headless extension gives a lot of data is that it is very hard to compare these data sets in a automated fashion while getting meaningful values, because they do not always have all the same attributes. This because the website can be altered and this can change the order of HTTP request, page renderings, etc. and this will therefore change the attributes of the data object containing all the collected metrics.

Another problem we walked into concerning the Speed Tracer headless extension is that it appeared to be a bit unstable. Unlike the normal Speed Tracer extension, the headless version is not available in the Chrome Extension Gallery. To obtain the headless version you first need grab a checkout of the Speed Tracer code and compile it. But when when trying to use it we stumbled upon a strange problem. The extension only reported data in a few of the many attempts of loading a page. When debugging the extension, a JavaScript error was detected, but after this was fixed, the problem kept occurring. After restarting the the Chrome browser, the extension seemed to work again. The problem appeared to be that the extension only works the first time after starting up Google Chrome. This problem, which could be avoided by restarting

Chrome for every single measurement test, made us doubt the reliability of the extension in general. Also the fact we could not find an answer or reference to such a major bug anywhere on the Internet lead us to believe the headless extension is not used a lot and maybe also not maintained anymore.

3.5 Development Phase: Chosen Approach

3.5.1 Navigation Timing

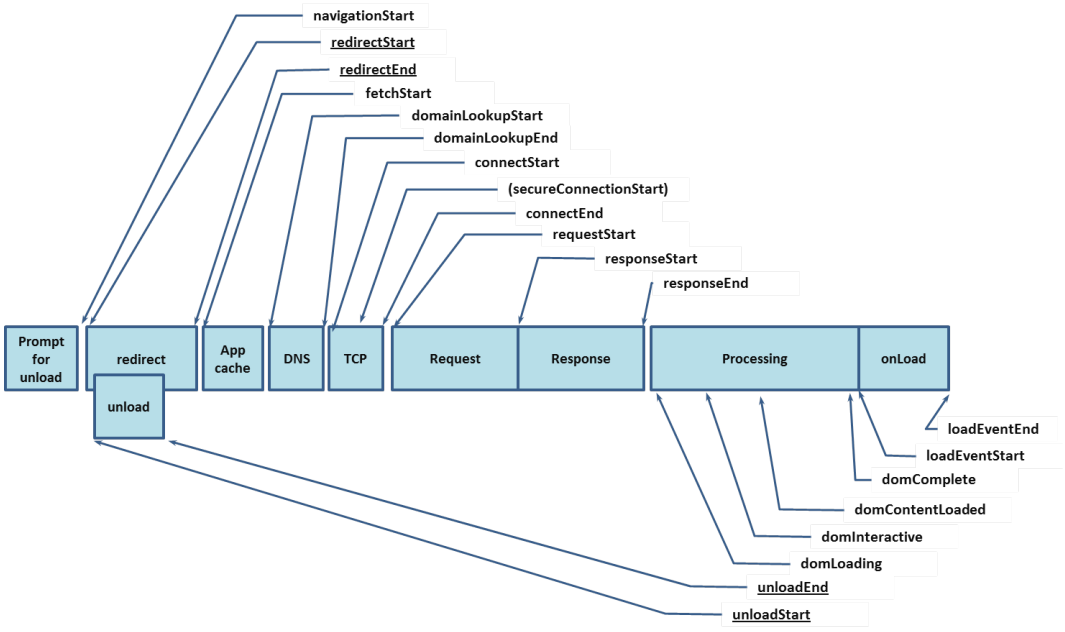

Figure 3.1: All timing metrics of the PerformanceTiming and the PerformanceNavigation interface

source: w3.org

The journey with the Speed Tracer brought us to the insight that we need simple, but reliable metrics, which can be collected with the PerformanceTiming interface which is described in the Navigation Timing W3C Candidate Recommendation [40].

As described earlier, Navigation Timing is a Candidate Recommendation of the World Wide Web Consortium. A Candidate Recommendation is the second phase in the progress of becoming an official W3C Recommendation, which means that the W3C sees it as a widely reviewed document which only requires some implementation experience before getting sent to the W3C Advisory Com-

mittee for final endorsement after which it will become an Recommendation [41]

The document describes how browsers should implement a few interfaces for JavaScript developers that provide detailed information about the client-side latency. The most important of these interfaces is the PerfomanceTiming interface. This interface provides timing information about all the key events that occur when a website is loaded, starting at the unload event of the previously opened website. To give you an idea of these timing metrics, here is a short description of a few important ones:

- `navigationStart`. This corresponds with the time when the unload event of the previous document has ended. In case there is no previous document it is the same as fetchStart.
- `fetchStart`. This is the moment when the document is being requested, before the browser checks its cache.
- `domContentLoadedEventStart`. This values represents the time when the DOMContentLoaded event is triggered. This is a very important metric, because it is the first time when the DOM of the document is ready for manipulation.
- `domComplete`. This is the moment when all the JavaScript that is included is executed. This is also the moment the load event of the document is triggered.
- `loadEventEnd`. The moment when the document is completely loaded.

The metrics as described above are some of the key measurements we would like to have. To get a good overview of the metrics, see figure 3.1. The PerformanceTiming object can be found in the browser's window object, to be exact: `window.performance.timing`.

There is, however, one big disadvantage of the PerformanceTiming interface and that is the fact that it is only supported by Google Chrome (since version 6) and Internet Explorer 9, but the fact that this is such nice comparable data is a big enough of a reason to still use it. To collect this data, it is send to a beacon on the server.

3.5.2 Custom Measurements

Next to collecting the overall measurements that the PerformanceTiming object provides us, we also want a way of profiling specific pieces of JavaScript within the code base. This could be done by invoking a global function which would be called something like `startMeasuring(name)` and can be stop by invoking the `stopMeasuring(name)` function. In this fashion, the difference between the start and the corresponding stop time can be calculated which gives us the time it took to execute the measured code. These recorded measurements will be stored in an JavaScript object and will be send, in combination with the PerformanceTiming object, to the beacon

As what was discussed earlier, measuring JavaScript by encapsulating it with `(new Date).getTime();` to register the time was not accurate in Internet Explorer [39]. Whoever, Internet Explorer 9 does not have this problem anymore according to the following test:

```
<script>
    var aTimes = [];

    // every 1 millisecond: push current time in array
    interval = window.setInterval(function() {
        aTimes.push((new Date).getTime());
    }, 1);

    // after 100 milliseconds: stop recording time and output array
    window.setTimeout(function() {
        clearInterval(interval);
        document.getElementById('output').innerHTML = aTimes.join('<br />');
    }, 100);
</script>
<div id="output"></div>
```

3.5.3 Triggering The Benchmark

Benchmarking will not be done every time a page is executed. To enable benchmarking for a certain page, a few query string parameters must be set:

- `benchfrontend`. This enables the benchmark. It does not have to have a value, but is just have to be there
- `env_id`. This contains the environment ID. The environment ID stands for the machine on which the benchmark is performed, in our current setup, this will be the slow or fast machine.
- `test_id`. This contains the test ID. The test ID is the name of the testcase that initiates the benchmark. For example, there is a testcase for benchmarking the logged in homepage, this testcase is called loggedinHomepage. This name would be the test ID.

To give an example of how the URL would look like when invoking a benchmark, this is the URL when running the logged in homepage test on the slow benchmark machine:

```
http://www.hyves.nl/?benchfrontend&env_id=slow&test_id=loggedinHomepage
```

These given parameters will also be sent to the beacon, so that the data can be labeled correctly.

3.5.4 The Beacon

What It Does

To be able to collected the data from the PermanceTiming object and all the custom measurements, a server-side beacon is set up, to which all this data can be send.

This beacon is a page where data can be send to from JavaScript with an XHR (XMLHttpRequest) call. After the beacon collects the data it pre-processes it and makes sure it gets stored.

How It Stores The Data

For storing we first used a simple MySQL database table, but because the amount of data can grow very fast, we were advised by a system engineer at Hyves to use a different architecture for storing the metrics, Graphite [42]. Graphite is a tool for storing numeric values over time and will also generate graphs for this data. The reason that using Graphite is better than just using a standard database table is the fact that Graphite's whisper, which is its database library, works like a round robin database (RRD). A round robin database stores its data for a definite period of time. This also means it requires a fixed size of storage. When a RRD is full, the new record gets stored at the location of the oldest record. In this way, the time frame of records is constantly shifting.

Next to the fact that Graphite uses RRD-like storage, which results in a fixed footprint on the hard drive, Graphite's front-end provides an easy to use interface for generating graphs. These graphs show how the values measured change over time. By using the URL API, you can retrieve generated images by setting query string parameters in the URL. The most important parameter is target. With this parameter you specify which metrics you want to be shown in the graph. This could one, but you could also set multiple targets. Two other important parameters that can be set are the from and until time. These must be given in the AT-STYLE format, used by RRDtool. This format is a way of specifying a time offset, for example, 'two hours ago' or 'one year ago'. Some other parameters that you can set are for defining layout related settings such as colors, ranges, grid, size of image and so on. Thanks to this simple URL API it is easy to display graphs about the collected data in the company's web-based administration panel.

Instead of directly passing all the data from the beacon to Graphite we decided to use the tool StatsD made by the developers of the web shop Etsy. StatsD is a simple daemon, written in JavaScript on top op node.js, that aggregates the data it receives over a UDP socket and once per interval flushes this data to Graphite [43]. The UDP protocol has less overhead than TCP at the risk of possibly being received out-of-order or not at all and therefore more vulnerable for errors. So it is faster than TCP, but less accurate. Except of generating less network traffic when using StatsD it also has an easier API, which handles the creation of new metrics in Graphite for your, instead of having to do it yourself. Therefore we are using StatsD as an extra layer in between the

PHP-based beacon and the python-based Graphite. StatsD already provides an example PHP wrapper which encapsulates the UDP sockets with some clean functions methods which we decided to use.

3.5.5 Automation of Benchmarking

As described earlier, one key requirement of the chosen benchmarking method was that it should be able to automate it. Automation gives the power that makes us able to constantly run of benchmarks against our development environment and live environment. Since the benchmark solution that we created can be invoked by a query string parameter this can easily be done.

Machine Configuration

For testing we will setup up two lab computers on which the benchmarking can be done. One computer can be considered as the slow machine and the other as the fast machine. The slow machine will have have limited hardware resources compared to today's average computer. With limited hardware resources, a slow CPU and limited amount of working memory is meant . Also the bandwidth of the Internet connection will be limited. The fast machine will be a top off the line desktop computer, with an Internet bandwidth which is more than average, namely the one of an Internet company.

Both computers will have Microsoft Internet Explorer 9 and Google Chrome (latest stable version) installed, since these browsers support the Navigation Timing API. Furthermore, both machines have Python installed (2.7.2) with the selenium 2.1.0 package. Selenium is the framework for automating the benchmarks. I will later discuss this in more depth.

Selenium Tests

On the lab computers, different Selenium tests will constantly run, in order to run benchmarks.

Selenium is a framework for the automation of tests with browsers on different platforms [44]. It is used for quality assurance and software testing for websites. For this project we are using Selenium 2, the WebDriver approach. However, the most widely used Selenium version is version 1 which is also known as Selenium Remote Control (RC). Even though this Selenium RC supports the most platforms it has a few disadvantages in contrast to the WebDriver solution. The biggest disadvantage of Selenium RC is that its most important component is the Selenium Server, this server acts a a HTTP proxy and injects JavaScript into the website it tests to be able to invoke the commands required by the test. Since it injects JavaScript and we actually want to benchmark the JavaScript performance of the website, this method could contaminate the measurements. The WebDriver method is much cleaner and straightforward. It hooks into native browser APIs that are designed for automation. Therefore this method has less overhead and will not affect the benchmark results.

The tests themselves are written in Python. We chose Python, for its clean and efficient syntax. There is one base class called `BenchmarkTest`, this is a subclass of the `unittest.TestCase`, which is part of Python's standard testing framework [45]. Subclasses of the unittest.TestCase can override certain parent mehods, called test fixtures. Fixtures a automatically called on at certain stages in the runtime of the tests. The BenchmarkTest class contains some general methods that are useful for all the different test scenarios.

One of those general methods is `setUp`, which is one of the fixtures. It is always called at the beginning of a test and should be used for initializing the test and setting up test data. In our case `setUp` initializes the WebDriver instance.

Another useful method is `login`. This method automatically logs into the website with a test account. Because there are a lot of cases in which you should be logged in, this method was suited for the BenchMarkTest class.

3.6 Overview of Implementation

The currently implemented setup for performing front-end benchmarking consists of 3 major components: the measuring, the automated running and the beaconing. The measurements a partially taken from the Navigation Timing API, combined with custom measurements. This is all done with JavaScript running within the page that is being benchmarked. The JavaScript for measuring these metrics is invoked by the automated Selenium tests that run on two dedicated computers that are setup in the Front-End Lab and which have their Selenium tests provided to them from the Subversion repository. Finally, the measurements are being send to the beacon, which directs them to StatsD and StatsD sends them to Graphite. See figure 3.2 for a graphical overview.

3.7 Results

As described, all the gathered metrics get stored in the Graphite architecture. The machines in the front-end lab will constantly run the created Selenium unit tests to trigger the designed benchmarks. In figure 3.3 you can see how two of those measured metrics develop over time. This is a snapshot of the 5th of August till the 8th August and the shown the `domComplete` and the `domLoading` retrieved from the `window.performance` object in Internet Explorer 9. `domLoading` is the time that has passed since the DOM start to load. This is in a early phase in the rendering of the page and you can see that it is very stable. `domComplete` the time it took to load the complete DOM. This is at the end of processing the document, to get a good overview of when these metrics are recorded see figure 3.1.

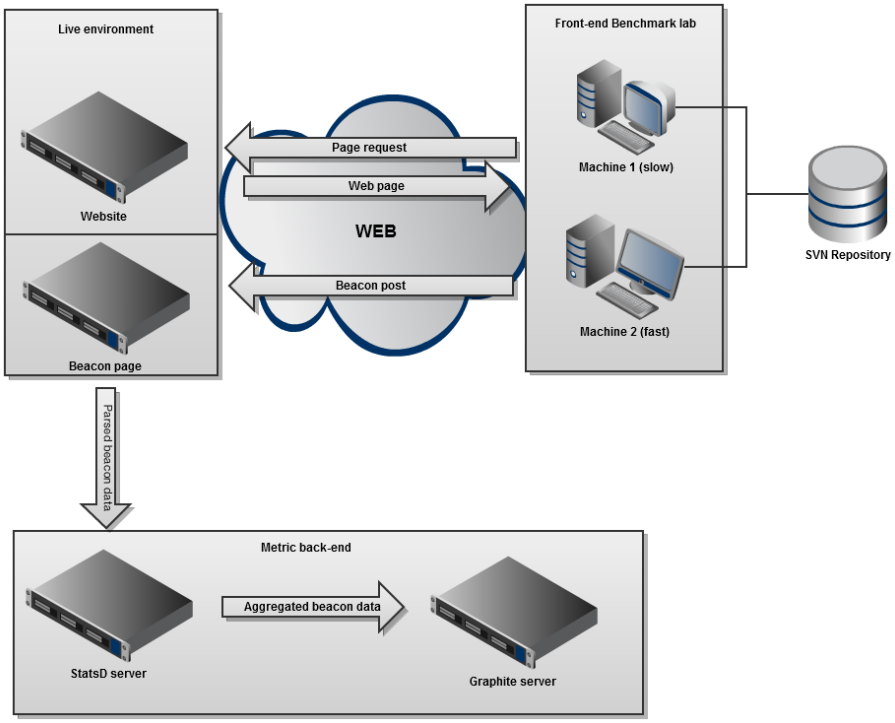

Figure 3.2: Overview of implemented benchmarking setup

Figure 3.3: Graphite graph of two gathered metrics

Chapter 4

Discussion

4.1 Challenges in Web Engineering Research

The Web is a rapidly changing environment. Now, more than ever, new web standards emerge and browsers become more powerful. Because of this fast evolution research that is done now, can be outdated tomorrow. This means that some of the discussed tools and techniques in this thesis could have become irrelevant or updated during the creation of the thesis.

Also since this is a fairly new research area and its innovation mostly comes from real world problems, there are not a lot of scientific articles about this subject. For this reason, almost all of our research is done on the Internet, for example there are interesting blogs written by people from the industry, some of whom can be considered real experts. Among those experts there are people like John Resig, the creator of the most popular JavaScript library at the moment, jQuery and author of the book called "Pro JavaScript Techniques". Another famous blogger is Jeff Atwood, he is the creator of stackoverflow.com, the Q & A site for programmers. Next to sources like blogs, there are W3C documents, that define how all the standards that have to do with HTML and JavaScript should work and be implemented. Also the documentation of all the tools we found can be considered as important sources.

4.2 Course of the Process

My research started with the question from the development department of Internet company Hyves, about how to tackle issues that arise during the paradigm shift from the classic web model to the modern web application model. This question already included the five issues that I have discussed and I had decided to explore all of them and to choose one that I would research in-dept and look for a possible solution for this problem. To explain were this paradigm shift comes from and to describe the problem domain I start with an overview of how the World Wide Web was invented and evolved along the way before

discussing the arising issues. Since a lot of developments lead to the Internet as we know it today, it took a lot of time deciding which ones to include and to which extend.

During the kickoff of the project, I was teamed up with a colleague and we had divergent approach with which we looked at all the areas of front-end performance of websites. All the information and tools that could be of any interest were examined and explored.

Since we started with a broad approach in the first research phase, we had gathered a lot of possibly interesting tools that we could use. Tools that stood out, such as Google Chrome's Speed Tracer, we examined more in-dept in the second research phase and also partly integrated in the development environment. But after actually using it in a more advanced way we stumbled on some unwanted behavior which led us to drop this tool in our final implementation. Other tools like Yahoo's YSlow only required some basic investigation to determine that they were not suitable for the kind of setup we had in mind.

At last in the development phase we finally choosing for the solution using the Navigation Timing API, our approach went from divergent to convergent. It became more and more clear what kind of data we were able to collect and how this could best be done. When we knew the tools we were going to use it was a matter of connecting them all together to get on integrated system for automatically benchmarking test scenarios. Connecting these tools required batch scripting, python, PHP and JavaScript.

Chapter 5

Conclusion

The Web is a very dynamic and rapidly changing environment. This makes it difficult to always be on top of the game. Since the invention of the Web by Tim Berners-Lee, the HTML specification evolved to HTML5 and the power and speed of browsers grew enormously. The shift from static web pages where all the programmatical logic, if any, was at the server-side to web applications that almost entirely run within the client's browser brought a lot of changes in the way these 'web pages' should be developed.

At the social networking website Hyves, where I work, we are in the transition of a website to a web application. In this journey new challenges arise. I have described these challenges and proposed possible solutions. One of the challenges is optimizing client-side web applications and on this topic I did extensive research. This research was setup according to the action research methodology. The initial research problem was to find ways of optimizing the front-end of Hyves. This problem statement changed in implementing a system to automatically benchmark the front-end of Hyves. Multiple iterations of defining the problem, implementing changes and reflecting on the outcome of these changes took place. Now I will shortly discuss all the challenges and what our final implementation is.

These changes in web development included for example making sure the browser's back-button and history functionality keep working. Because web applications tend to keep on the same page, the browser does not recognize transitions and therefore clicking on it's back-button would result in returning to the previous page and thereby possibly exiting the web application. For this problem there a multiple solutions available, most of them JavaScript libraries, but the new HTML5 specification also brings support for manipulating the browser history.

Search engine optimization of web applications is also one of the new challenges since search engine bots do not execute JavaScript and are therefore not able to crawl through your entire application. Thankfully, the biggest search engine, Google provides a way for web developers to serve the Google bot the pages of your application as if they rendered by JavaScript so the Google bot is

able index page's content.

For modularization of JavaScript there is an nice coding pattern called the JavaScript module pattern. This let you write clean modules that can easily be reused and do not pollute the global namespace. In terms of code-generation, at the moment there does not exist a tool that convert an abstract model to working JavaScript and HTML code. It could be an interesting project to develop such a tool but it would be difficult to support all the different browser.

Another big challenge when making an web application is hiding the differences in the target platform. Since not all web browsers have support for the same level of JavaScript and CSS, this can be a painful task. One approach can be to create different interfaces for different target platforms, for example one for mobile phones with a touch screen and one for mobile phone without a touch screen. The problem with this approach is the number of different interfaces can big and it could become a complicated job to keep them all up-to-date. The other approach is to detect which features the client supports and have one interface that gracefully degrades, meaning it falls back on other techniques if certain features are not available. Modernizr is a useful library for implementing this. This approach however can also be difficult to maintain because of the different ways browsers implement JavaScript and CSS.

The last new challenge that is discussed is the need to be able to profile and optimize client-side code. In the search of a way to fulfill this need we stumbled on quite a few tools that could be able do that. I discussed all of these tools and it turned out that none of them was suited for the way we wanted to benchmark the Hyves website. This was mostly due to insufficient possibilities for using the tools in an automated environment, because of they could not be run automatically and could not beacon their collected data to a server.

To be able to get the measurements we want we decided to write our own piece of JavaScript that could be invoked on the page that you would like to benchmark. The piece of JavaScript takes metrics from the Navigation Timing API, a W3C specification, combined with custom measurements that can be gathered in any part of the page and sends this data to a beacon. This beacon eventually to Graphite, via StatsD to be stored. The data can be visualized with Graphite's front-end.

Web application development brings new challenges, since this field is relatively new, standard solutions do not exist yet. One solution on how to profile a web application is discussed here. Because of the huge popularity of web applications, more and better tools and frameworks will be created. Web standards will be elaborated and better implemented by browsers.

Chapter 6

Furture Work

6.1 Additions to Current Implementation

There is a lot that still can be added to the current implemented front-end benchmarking setup. At the moment, we only wrote one unittest which benchmarks the logged in homepage of Hyves. Additional unit tests should be added, to benchmark other crucial parts of the websites, for example a member's profile page. With the current setup of the code of the parent unit test class `BecnhmarkTest` is should be very easy to added additional unit tests.

More data could be collected. Except from recording the metrics from the `window.performance` object and custom measurements of pieces of JavaScript other measurements could also be send to the beacon. For example data that scripts like DOM Monster [46] collect. DOM Monster is a bookmarklet which gives you information about the number of CSS rules that are loaded and the number of number of DOM elements. Metrics like these can be good indications about the performance of your website will be.

Also the checks for bad JavaScript patterns in the code base, that could be integrated in the Jenkins continuous integration system should be implemented. Developing a plugin for static code analysis of JavaScript code can be very useful for everybody that uses Jenkins for its web application development.

6.2 Optimization

In order to be able to optimize code, you should first be able to benchmark it. The initial focus of this research was on optimizing client-side code of web applications, but because we did not have a proper way of benchmarking yet, the focus of the project changed to building a system which could do automatic benchmarking. In the future this system can be used to measure how code changes influence the performance of the web application and this information can be used to optimize the client-side code.

6.3 Common Good Practices in JavaScript

Besides from implementing a benchmarking solution to detect bottlenecks it is good to look at some good practices in JavaScript. There are some common pitfalls JavaScript developers fall into when it comes to writing efficient JavaScript code. When these mistakes can be avoided it could directly lead to better code and better performance. To give you an example here are a few of those common mistakes:

One mistake is the abuse of the `document.getElementById()`. This method is the easiest way to retrieve a DOM node, but it needs to query the complete DOM tree which can be very large. Instead of using `document.getElementById()` multiple times for the same ID, it is better to store a reference to the object in a local variable.

```
// Wrong
document.getElementById('myId').innerHTML = 'Hello World!';
document.getElementById('myId').style.display = 'block';

// Right
var myObjectRef = document.getElementById('myId');
myObjectRef.innerHTML = 'Hello World!';
myObjectRef.style.display = 'block';
```

Another mistake is that the Object and Array constructors are slow. It is better to use literal notation.

```
// Wrong
var myArray = new Array();
var myObject = new Object();

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

The use of `with(){}` is also slow and cause for confusing code.

```
// if you have those kind of namespaced variables:
ooo.eee.oo.ah_ah.ting.tang.walla.walla.bing = true;
ooo.eee.oo.ah_ah.ting.tang.walla.walla.bang = true;

// Slow. Moreover the scope is modified by 'with'
// which can create really hard to maintain code if the nested block become bigger.
// Please don't do that.
with (ooo.eee.oo.ah_ah.ting.tang.walla.walla) {
    bing = true;
    bang = true;
```

```
}

// If you need a notation shortcut, you can always declare a shorcut variable:
var o = ooo.eee.oo.ah_ah.ting.tang.walla.walla;
o.bing = true;
o.bang = true;
```

These are a few examples in of simple code modifications which can lead to better performing code. To make sure that developers stick to such good practices, checks can be build into the Jenkins Continuous Integration system [47]. Jenkins is a continuous integration system which constantly builds the the code in the code base and runs tests against it to make sure new code changes do not break the functionality. It provides a way to extend it's functionality by writing plug-ins. For checking coding styles like the ones that are mentioned above, a static code analysis plugin could be written, which would generate warnings in case wrong coding styles are used. These checks could notify the lead front-end developer of the use of code structures that are forbidden.

6.4 Implications for other Websites using Client-Side Techonology

Because of the high demands we had for this project, especially due to the requirement that all the benchmarking should be done in an automated fashion, the final setup is quite complicated. The ability to run the benchmarks automatically was crucial with a website like Hyves were multiple people work the same code base and functionality is constantly changing.

For most other websites tools like Speed Tracer and dynaTrace will be good enough. Manually benchmarking with these tools give you a lot of insight in the performance of your website. Also tools like YSlow and Page Speed should be looked at since they give you some nice practical pointers in how to optimize your website.

6.5 Implications for Social Web Applications

The Open Learning Lab of the University of Leiden is an innovative project to implement an OpenCourseWare solution in a social way, using integration of social networks like Facebook and Twitter. One of the import challenges for creating such a web application is to support different target platforms. Especially because of the open nature of the project, the website should work correctly in all the major web browsers. Modernizr [27], as discussed in section 2.2.4, is a great library for detecting which level of CSS and JavaScript support the browser of the user has.

When looking at other possible challenges, performance is always important and therefore using YSlow [30] and Page Speed [37] is also recommended. These tools give developers easy tips to enhance the performance of their website.

Acknowledgements

This thesis would not have been possible without the supervision of Christoph Johann Stettina. Christoph helped me keeping an eye on the progress of the thesis itself and how to give it a good structure. Also all the work done by my colleague Vincent Renaudin.Together we have done the research concerning front-end benchmarking and did the design and implementation of the benchmarking setup. Vincent's knowledge about front-end web development was indispensable.

References

[1] David E. Avison, Francis Lau, Michael D. Myers, and Peter Axel Nielsen. Action research. *Commun. ACM*, 42:94–97, January 1999. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/291469.291479. URL `http://doi.acm.org/10.1145/291469.291479`.

[2] D. Raggett, J. Lam, I. Alexander, and M. Kmiec. *Raggett on HTML 4*. Addison-Wesley, 1998. ISBN 9780201178050.

[3] D Connolly. A little history of the world wide web, 2011. URL `http://www.w3.org/History.html`. [Online; accessed 2-June-2011].

[4] Wikipedia. Spyglass, inc. — wikipedia, the free encyclopedia, 2011. URL `http://en.wikipedia.org/w/index.php?title=Spyglass,_Inc.&oldid=433207403`. [Online; accessed 29-July-2011].

[5] Wikipedia. United states microsoft antitrust case — wikipedia, the free encyclopedia, 2008. URL `http://en.wikipedia.org/w/index.php?title=United_States_Microsoft_antitrust_case&oldid=252504817`. [Online; accessed 2-August-2011].

[6] J. Veen. *The Art and Science of Web Design*. Pearson Education, 2000. ISBN 0789723700.

[7] Le Hors A. Raggett, D. and I. Jacobs. Html 4.0 specification, 1998. URL `http://www.w3.org/TR/REC-html32`. [Online; accessed 22-May-2011].

[8] I. Hickson. Html5, 2011. URL `http://www.w3.org/TR/html5/`. [Online; accessed 29-July-2011].

[9] M. Pilgrim. *HTML5: Up and Running*. O'Reilly Series. O'Reilly Media, 2010. ISBN 9780596806026.

[10] N Hamilton. The a-z of programming languages: Javascript, 2008. URL `http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/`. [Online; accessed 29-July-2011].

[11] Le Hgaret P. Whitmer R. Wood L. Document object model (dom), 2009. URL `http://www.w3.org/DOM/`. [Online; accessed 2-June-2011].

[12] P. Kock. Level 0 dom, 2009. URL http://www.quirksmode.org/js/dom0.html. [Online; accessed 2-June-2011].

[13] J. Sutor R Wilson C. Wood L Apparao V. Byrne S. Champion M. Isaacs S. Jacobs I Le Hors A. Nicol, G. Robie. Document object model (dom) level 1 specification, 1998. URL http://www.w3.org/TR/REC-DOM-Level-1/. [Online; accessed 2-June-2011].

[14] T. Leithead. Dom level 3 events support in ie9, 2010. URL http://blogs.msdn.com/b/ie/archive/2010/03/26/dom-level-3-events-support-in-ie9.aspx. [Online; accessed 3-June-2011].

[15] A. Hopmann. Story of xmlhttp, 2007. URL http://www.alexhopmann.com/story-of-xmlhttp/. [Online; accessed 29-July-2011].

[16] Cascading style sheets, 2011. URL http://www.w3.org/Style/CSS/. [Online; accessed 2-August-2011].

[17] H.W. Lie, H.W. Lie, and B. Bos. *Cascading style sheets: designing for the Web*. Addison Wesley, 2005. ISBN 9780321193124.

[18] Bos B. elik T. Hickson I. Wium Lie H. Cascading style sheets level 2 revision 1 (css 2.1) specification, 2011. URL http://www.w3.org/TR/CSS2/. [Online; accessed 3-June-2011].

[19] Jackson D. Hyatt D. Marrin C. Css animations module level 3, 2009. URL http://www.w3.org/TR/css3-animations/. [Online; accessed 3-June-2011].

[20] Yui 2: Browser history manager, 2011. URL http://developer.yahoo.com/yui/history/. [Online; accessed 3-June-2011].

[21] dojo.back - the dojo toolkit - reference guide, 2011. URL http://dojotoolkit.org/reference-guide/dojo/back.html. [Online; accessed 2-August-2011].

[22] I. Hickson. Html5, 2011. URL http://www.w3.org/TR/html5/history.html. [Online; accessed 2-August-2011].

[23] Making ajax applications crawlable, 2011. URL http://code.google.com/intl/en/web/ajaxcrawling/docs/specification.html. [Online; accessed 2-August-2011].

[24] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 032119442X.

[25] B. Cherry. adequately good: Javascript module pattern: In-depth, 2010. URL http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth. [Online; accessed 3-August-2011].

[26] Webratio. URL http://www.webratio.com/. [Online; accessed 3-August-2011].

[27] Modernizr, 2011. URL http://www.modernizr.com/. [Online; accessed 2-August-2011].

[28] Speed tracer - google web toolkit - google code, 2011. URL http://code.google.com/intl/en/webtoolkit/speedtracer/. [Online; accessed 3-August-2011].

[29] Firebug, 2011. URL http://getfirebug.com/. [Online; accessed 3-August-2011].

[30] Yahoo! yslow, 2011. URL http://developer.yahoo.com/yslow/. [Online; accessed 3-August-2011].

[31] Application performance management (apm) / application performance monitoring - dynatrace software, 2011. URL http://www.dynatrace.com/en/. [Online; accessed 3-August-2011].

[32] X. Qi. Hiphop for php: More optimizations for efficient servers, 2011. URL http://www.facebook.com/notes/facebook-engineering/hiphop-for-php-more-optimizations-for-efficient-servers/10150121348198920. [Online; accessed 3-August-2011].

[33] J. Atwood. Coding horror: Performance is a feature, 2011. URL http://www.codinghorror.com/blog/2011/06/performance-is-a-feature.html. [Online; accessed 3-August-2011].

[34] G. Linden. Geeking with greg: Marissa mayer at web 2.0, 2006. URL http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. [Online; accessed 3-August-2011].

[35] M. Fowler. Continuous integration, 2006. URL http://www.martinfowler.com/articles/continuousIntegration.html. [Online; accessed 3-August-2011].

[36] Best practices for speeding up your web site. URL http://developer.yahoo.com/performance/rules.html. [Online; accessed 3-August-2011].

[37] Google page speed, 2011. URL http://code.google.com/speed/page-speed/. [Online; accessed 15-August-2011].

[38] this, is boomerang, 2011. URL http://yahoo.github.com/boomerang/doc/. [Online; accessed 2-August-2011].

[39] J. Resig. Accuracy of javascript time, 2008. URL http://ejohn.org/blog/accuracy-of-javascript-time. [Online; accessed 3-August-2011].

[40] Z. Wang. Navigation timing, 2011. URL http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html. [Online; accessed 3-August-2011].

[41] I. Jacobs. World wide web consortium process document, 2004. URL http://www.w3.org/2004/02/Process-20040205/tr#maturity-levels. [Online; accessed 3-August-2011].

[42] C. Davis. Graphite - enterprise scalable realtime graphing, 2011. URL http://graphite.wikidot.com/. [Online; accessed 3-August-2011].

[43] I. Malpass. Code as craft: Measure anything, measure everything., 2011. URL http://codeascraft.etsy.com/2011/02/15/measure-anything-measure-everything/. [Online; accessed 3-August-2011].

[44] Selenium - web browser automation. URL http://seleniumhq.org/. [Online; accessed 3-August-2011].

[45] unittest unit testing framework. URL http://docs.python.org/library/unittest.html/. [Online; accessed 3-August-2011].

[46] Dom monster. URL http://mir.aculo.us/dom-monster/. [Online; accessed 3-August-2011].

[47] Jenkins ci. URL http://jenkins-ci.org/. [Online; accessed 3-August-2011].