KTH Computer Science
and Communication

Modular responsive web design

Allowing responsive web modules to respond to custom criteria instead of only the viewport size by implementing *element queries*

LUCAS WIENER
lwiener@kth.se

Master's Thesis at CSC

Supervisors at EVRY AB: Tomas Ekholm & Stefan Sennerö
Supervisor at CSC: Philipp Haller
Examiner: Mads Dam

TRITA xxx yyyy-nn

Abstract

Abstract goes here.

Referat

Modulär responsiv webbutveckling

Sammanfattning ska vara här.

The colored rectangles that appear throughout this thesis are comments and feedback (presented as todos), which mean that they are not part of the actual content. It should be noted that all named todo items are not quotations of the persons that it was received from. Instead, all todo items represent my own interpretation of the feedback given to me.

Contents

Todo list

Chapter 1

Introduction

Write about the targeted browser compatibility. Started writing about this but put it on hold in the appendix.

Tomas: Show which cite's are used for which parts of the sections better. Could still be done in the metatext by just describing what the cited resource address. Or perhaps just citing the resource again where neccesary in the text.

Tomas: Don't think that the intro should be so "selling", write about what "will" be done and hypothesis and such. Write instead what "have" been done, what you have personally done and stuff like that.

Add note that the thesis is best viewed in color.

Write that glossary words will be emph on first appearence?

Should the subtitle be criterion instead of criteria?

Philipp: I would change the meta text font style to be the same as the regular text. There's no need to typeset this introductory text differently.

Philipp: You can add more information about why it's interesting to the reader (meta text). Suggestion: turn each of the sentences starting with "Section 2.1" and "Section 2.2" into paragraphs.

Philipp: Yes, "see Section 3.1" is the right way to do it (normally, one capitalizes "Section"). When referring to subsections and subsubsections, I would also call them "Section", indeed, because their number already says what kind of subsection it is.

1.1 Targeted audience

This thesis is targeted for *web* developers that wish to gain a deeper understanding of element queries and how one could solve the problem today. Heavy use of web terminology is being used and intermediate web development

knowledge is beneficial. For readers that are not familiar with web terminology there is a glossary at the end of the document.

Philipp wanted this to be more general.

1.2 Problem statement

By using Cascading Style Sheets (CSS) *media queries*, developers can specify different style rules for different *viewport* sizes. This is fundamental to creating *responsive* web applications, as shown in section 2.1.2. If developers want to build modular applications by composing the application by smaller components (*elements*, scripts, styles, etc.) media queries are no longer applicable. Modular responsive components should be able to react and change style depending on the size that the component has been given by the application, not the viewport size. The problem can be formulated as: *It is not possible to specify conditional style rules for elements depending on the size of an element.* See the included document in section B.1 of the appendix for a more practical problem formulation.

Tomas: The reader could misundestand this as modular design is about putting all scripts with scripts, styles with styles, etc.

Do I need to have a reference for this?

Participants, which include both paying members and the public, of the main international standards organization for the web, the World Wide Web Consortium (W3C) are interested in solving the problem and possible solutions are being discussed. However, they are still at the initial planning stage [25] so a solution will not be implemented natively in the near future. Additionally, implementing element queries imposes circularity and performance problems, as shown in chapter 3, which needs to be resolved before writing a standard. While awaiting a *native* solution it is up to the developers to implement this feature as a *third-party* solution. As presented in chapter 6, efforts have been made to create a robust third-party solution, with moderate success. It should be noted that no effort of implementing a native solution have been made. Since all current third-party solutions have shortcomings, there is still no de facto solution that developers use and the problem remains unsolved.

Tomas: Maybe not third-party? Third-party refers to something that has not been developer in-house. third-party here should be substituted to the negation of native.

1.3 Objective

The main objective of this thesis is to design and develop a third-party non-native implementation of element queries. To do this, it is needed to research existing solution approaches in order to understand and analyze the advantages and shortcomings of the approaches. It is also necessary to be aware of the premises, such as *browser* limitations and specifications that one must conform to. In addition, research will be done about the problems of implementing element queries natively, to get a deeper understanding of the

Marcos: I will research the problems of.

potential shape of a standardized Application Program Interface (API). Solving the main problem will require research and empirical studies of many subproblems. Examples of such subproblems that needs to be addressed are:

- Should circularity be handled? If yes, should it be detected at runtime or parsetime, and what should happen on detection?
- How can one observe element size changes without any native support?
- How can a custom API be crafted that will enable element queries and still conform to the language specifications? Is it possible to create an API that feels natural to web developers and works in tandem with other tools and libraries? For instance, if the API requires custom CSS syntax then CSS preprocessors, linters and validators cannot be used anymore. Likewise, if the API requires a special element handling process, it could be hard to use with popular libraries.
- Is it possible to solve the problem with adequate performance for large applications that make heavy use of responsive modules?
- How can adequate browser compatibility with legacy browsers be achieved without native support?

Should I give examples like Angular React Backbone Bootstrap?

The scientific question to be answered is if it is possible to solve the problem without extending any language specification of the web. The hypothesis is that the problem can be solved with high reliability and adequate performance by developing a third-party implementation. The goal of this thesis should be considered fulfilled if a solution was successfully implemented or described, or if the problems hindering a solution are thoroughly documented.

Tomas: Remove this paragraph. Should state what have been done instead.

1.4 Significance

Tomas: Maybe more general or more specific. If details have to be presented, then the reader has to be able to understand it.

Many libraries and techniques are being used in web development to keep the code from becoming an entangled mess. Creating modules facilitates the development and increases the reusability, as shown in section 2.1.3. Unfortunately, it is currently impossible to create *encapsulated* responsive modules since conditional styles cannot be applied to elements by element size criterion. Without element queries, responsive modules force the application to style them properly depending on the viewport sizes, which defeats the purpose of modules. Modules should be encapsulated and may not require the

user to perform some of the module logic in order to work. Another option would be to make modules context-aware so they can style themselves according to the viewport, but then they would not be reusable (since they assume a specific context). Also, changes to the application layout would then require to rewrite the media queries of the modules to take the new layout into account. Clearly, no sound option exists for having responsive modules.

The last couple of years a lot of articles have been written about the problem and how badly web developers need element queries [2, 6, 71, 39, 77, 49, 55, 15, 22, 59, 53, 31]. As already stated in section 1.2, third-party implementation efforts have been made with moderate success. The W3C receives requests and questions about it, and the Responsive Issues Community Group (RICG) have started writing a draft [26] about element queries use cases.

Solving this problem would be a big advancement to web development, enabling developers to create truly modular responsive components. By studying the problem, identifying approaches and providing a third-party solution the community can take a step closer to solve the problem natively. If the hypothesis holds, developers will be able to use element queries in the near future, while waiting for the W3C standardize a solution. The outcome of this thesis can also be helpful for the W3C and others to get an overview of the problem and possibly get ideas how subproblems can be handled.

Would like to reformulate this to not being so confident that a standard will definately be developed

Tomas: The hypothesis should be valid or invalid at the time the report is handed in.

1.5 Scope

The focus of this thesis lays on developing a third-party library that realizes element queries. All theoretical studies and work will be performed to support the development of the library.

1.5.1 What will be done

- A third-party implementation of element queries will be developed.
- The problems of implementing element queries natively will be addressed.
- Theory about *layout engines*, programming languages of the web, responsive web design and modularity will be given to fully understand the problem.

1.5.2 What will not be done

- No efforts will be made to solve the problems accompanied with a native solution.
- No API or similar will be designed for a native solution.

- Graphical design of end-user interfaces will not be addressed, other than necessary for understanding the problem.

1.6 Outline

This is now totally outdated.

Finish the Outline when the other parts have been written.

Chapter 2

Background

↪ *This chapter aims to present sufficient background to understand why element queries is a desired feature and why it is hard to implement natively in browsers. Section 2.1 will present a brief history of web development and motivate why responsive web design and modular development are desired concepts in modern web development. Section 2.2 describes briefly how layout engines operates, with focus on the layout process, to later present how they can be parallelized in section 2.2.6. The chapter ends with section 2.2.7 that defines layout thrashing and shows how it can be avoided. Expert readers may want to skip directly to chapter 3 where element queries is described.*

Write that 2.2.6 will be needed to understand the problems of element queries?

Write that 2.2.7 will be extensively referenced from implementation chapter?

2.1 Web development

↪ *Browsers are the far most popular tools for accessing content on the web, which makes them very important in the modern society. In the dawn of the web, browsers were simply applications that fetched and displayed text with embedded links. Today, browsers act more like an operating system (on top of the host system) executing complex web applications. There even exist computers that only run a browser, which is sufficient for many users. This section will describe the transition from browsers rendering simple documents to being hosts for complex applications in section 2.1.1. It will also describe two key evolution points in web development — responsive web design in section 2.1.2 and modular development in section 2.1.3.*

2.1.1 From documents to applications

↪ *This section will describe the transition from browsers rendering simple documents to being hosts for complex applications. Since web development trends are not easily pinned to exact dates, this section will only present dates as guidance and should not be regarded as exact dates for the events. This section is a summary of* [75, 10, 8, 80, 35, 40].

As further described in section A.2 of the appendix, browsers were initially applications that displayed *hypertext* documents with the ability to fetch linked documents in an user friendly way. Static content was written in HyperText Markup Language (HTML), which could include hyperlinks to other hypertext documents or hypermedia. Different stylesheet languages were being developed to enable the possibility of separating content styling with the content. In 1996 the W3C officially recommended CSS, which came to be the preferred way of styling web content. Since HTML is only a markup language it is not possible to generate dynamic content, which was sufficient at the time HTML was only used for annotating links in research documents.

The need for generated dynamic content grew bigger, and the National Center for Supercomputing Applications (NCSA) created the first draft of the Common Gateway Interface (CGI) in 1993, which provides an interface between the web server and the systems that generate content. CGI programs are usually referred to as scripts, since many of the popular CGI languages are script languages. Generating dynamic content on the server is sometimes referred to as *server-side scripting*. This enabled developers to generate dynamic websites, with different content for different users for instance. However, when the content is delivered to the client (browser) it is still static. There was no way for the server to change the content that the client has received, unless the client requests another document.

Around 1996, client side scripting was born. The term Dynamic HTML (DHTML) was being used as an umbrella term for a collection of technologies used together to make pages interactive and animated, where client side scripting played a big role. Examples of things that were being done with DHTML are; refreshing pages for the user so that new content is loaded, give feedback on user input without involving the server and animating content. Plugins also started to exist during this time that enabled browsers to handle and execute embedded programs. *Java applet*[1] and *Flash*[2] are examples of browser embedded programs requiring browser plugins to execute. In con-

[1] A Java applet is a small application which is written in Java and delivered to users in the form of bytecode. See `www.java.com`

[2] Flash is a multimedia and software platform for producing cross platform interactive animations. See `www.flash.com`

cept, users have to install the plugin runtime in order for the browser to be able to execute the plugin programs, which is undesired since it adds a barrier between users and the content. However, plugins enabled developers to access some capabilities that were lacking in browsers (e.g., video playback, programmable graphics, and interactive animations). Some Operating Systems (OSs) and browsers ships with preinstalled plugins, while others do not support plugins at all.

With the increase of smart devices (such as phones, televisions, cars, game consoles, etc.) that includes browsers with limited third-party runtimes, plugins quickly decreased in popularity. Additionally, as the web platform was improved and users being discoraged by browsers from installing plugins due to security issues, the use of plugins decreased further.

As *JavaScript* and HTML supported more features, websites turned into small applications with user sessions and rich Graphical User Interfaces (GUIs). Still, parts of the applications were defined as HTML pages, fetched from the server when navigating the site. When the *XMLHttpRequest* API was supported in the major browsers, pages no longer needed to reload in order to fetch new content as XMLHttpRequest enabled developers to perform asynchronous requests to servers with JavaScript. This opened up for the Asynchronous JavaScript and XML (AJAX) web development technique which became a popular way of communicating with servers "in the background" of the page. Developers pushed browsers and HTML to the limit when creating applications instead of documents that they were originally designed for. In a W3C meeting in 2004 it was proposed to extend HTML to ease the creation of web applications, which was rejected. The W3C was criticized of not listening to the need of the industry, and some members of the W3C left to create the Web Hypertext Application Technology Working Group (WHATWG). The WHATWG started working on specifications to ease the development of web applications which came to be grouped together under the name *HTML5*. In 2006, the W3C acknowledged that WHATWG were on the right track, and decided to start working on their own HTML5 specification based on the WHATWG version. HTML5 is an evolutionary improvement process of HTML, which means that browsers are adding support as parts of the specification is finished.

A new era of APIs and features came along with HTML5 and *CSS3*, which truly enabled developers to create rich client side applications. With the new features developers could utilize advanced graphics programming, geolocation, local and session storage, advanced input, offline mode, and much more.

Marcos: Remember that AJAX is a total misnomer :) It has little to do with XML, or JS really, + many people made sync requests, as allowed by XMLHttpRequest. You could say that asynchronously fetching of data over HTTP to then dynamically update presentation was colloquially referred to as "AJAX".

Write that traditional applications are threatened by web applications, since the reach and availability of the web is superior to any other distribution platform. Also, no installation is required with web applications. Also, updates and patches can be applied to all users instantaneously and enforced.

Tomas: Maybe write about the difficulties in supporting the standards (fragmented features)

2.1.2 Responsive web design

↪ *A few years ago, web developers could make assumptions about the screen size of user devices. Since typically only desktop computers with monitors accessed web sites they were designed for a minimum viewport size. If the size of the viewport was smaller than the supported one, the site would look broken. This was a valid approach in a time when tablets and smartphones were unheard of. Today, another approach is needed to ensure that sites function properly across a range of different devices and viewport sizes. This section is a summary of* [47, 61, 51].

Marcos: Remember that developers might be also targeting actual browsers, not just "mobile" or "desktop".

According to *StatCounter*, 37% of the web users are visiting sites on a mobile or tablet device. No longer is it valid to not support small screens. Furthermore, it is understood that sites need to be styled differently if they are visited by touch devices with small screens or mouse-based devices with large screens. Since web developers were not ready for this rapid change of device properties, they resorted to using the same approach that they had done before — making assumptions about the user device based on the server request. When a browser requests a resource, an *user agent string* is usually sent with the request to identify what kind of browser the user is using. By reading the user agent string on the server-side, a mobile-friendly version of the site could be served if the user was sending mobile user agent strings and the desktop version could be served otherwise. The mobile version would be designed for a maximum width, and the desktop would be designed for a minimum width.

Explain user agent, that all browsers are user agents but not the other way around.

This was a natural reaction since no better techniques existed, but the approach has many flaws. First, developers now have multiple versions of a site to maintain and develop in parallel. Second, this approach doesn't scale well with new devices entering the market. For instance, tablets are somewhere in the middle of mobile and laptops in size, which would require

Marcos: Not only devices (also browser).

another special version of the site. Further, when desktops support touch actions and smartphones support mouse actions, even more versions of the website needs to be developed in order to satisfy all user devices. Third, the desktop site assumes that desktop users have big screens (which usually is true). However, there is no guarantee that the browser viewport will be big just because the screen is big. Users might want to have multiple browser windows displayed at the same time on the screen, which would break the assumptions about the layout size available for the site. Clearly, a better approach was needed.

With the release of CSS3 media queries new possibilities opened up. Media queries enabled developers to write conditional style rules by media properties such as the viewport size. See listing 2.1 for an example of how media queries can be used to style elements differently with relation to the viewport size. This can be used to tailor a site for a specific medium or viewport size at runtime. Responsive Web Design (RWD) refers to the approach of having a single site being responsive to different media properties (mainly the viewport size) at runtime to improve the user experience. With RWD it is no longer needed to maintain several versions of a site, instead the site adapts to the user medium and device.

```
@media screen and (max-width: 600px) {
  body {
    background-color: blue;
  }
}

@media screen and (min-width: 601px) {
  body {
    background-color: yellow;
  }
}
```

Listing 2.1. The above CSS styles the body of the website blue if the viewport is less or equal to 600 pixels wide, and yellow otherwise.

Perhaps give an example of a responsive module/view so that the reader understands well why responsiveness is important. This module/view could then act as a testbed for the different EQ approaches.

2.1.3 Modularity

↪ *As the web was a platform for hypertext documents that quickly transitioned to serving complex applications, few techniques existed for writing modular code. The last couple of years as applications grew bigger, techniques and libraries have been developed to ease modular development. Modular development is an old concept, but somewhat newborn in the web scene. This section aims to describe what modular development really is, and why it is such an important success factor to software development. This section is partly based*

Tomas: Applications did not grew bigger. Instead, they transitioned from being generated at server side, to living entirely on the client side

on [44, 28, 29].

By creating modules that can be used in any context with well-defined responsibilities and dependencies, developing applications is reduced to the task of simply configuring modules (to some extent) to work together which forms a bigger application. It is today possible to write the web client logic in a modular way in JavaScript. The desire of writing modular code can be shown by the popularity of libraries that helps dividing up the client code into modules. The ever so popular libraries *Angular*, *Backbone*, *Ember*, *Web Components*, *Requirejs*, *Browserify*, *Polymer*, *React* and many more all have in common that they embrace coding modular components. Many of these libraries also help with dividing the HTML up into modules, creating small packages of style, markup and code.

Tomas: And to be finished in time?

A big challenge to software development is to be able to write software that is reliable (i.e., should not have bugs) and easy to change. What keeps developers from producing such software is often complexity, which hinders developers from reasoning about the software. The word "complex" can be defined as *something consisting of interconnected or intertwined parts*. A quote from Rich Hickey:

> So if every time I think I pull out a new part of the software I need to comprehend, and it's attached to another thing, I had to pull that other thing into my mind because I can't think about the one without the other. That's the nature of them being intertwined. So every intertwining is adding this burden, and the burden is kind of combinatorial as to the number of things that we can consider. So, fundamentally, this complexity, and by complexity I mean this braiding together of things, is going to limit our ability to understand our systems.

By separating the software into well defined parts (i.e., modules) that has a single responsibility and ideally performs a single task, complexity can be reduced. Of course, splitting software up into modules alone does not help reducing complexity. The modules also need to be encapsulated, which means that they work by themselved and do not require the user of the module to write its logic. Modules should also be loosely coupled, and may not make any assumptions about the context they will be used in. It is then possible to reason about them and change the functionality locally inside the boundary of a module.

With modularity comes positive side effects. Loose coupling between modules facilitates integration testing of each module, since it is then possible to test parts of the software in isolation and independent of the system as a

whole. Developers can also work on different parts of the software in parallel without being affected by each other, as modules are not allowed to affect each other (other than by configurable options, injected strategies or directly dependent submodules). The very nature of modules not being context-aware enables them to be reused in other projects (or other parts of the same application). Reusability is not only important to speed up the development process of new software projects, it also eases managing a large code base. A single module that performs a general task is much simpler to manage than multiple modules performing the same task but being written for different contexts. When a bug arises, the patch would only need to be applied to one module instead of all similar modules that could possibly be affected. Clearly, modular development is highly desired.

Tomas: Maybe one-way dependencies?

However, since modules are allowed to be nested (i.e., modules may depend on submodules that may depend on other submodules and so on) and that the logic responsibility is moved from the application to the module, the complexity of managing them is added. API changes of a module can break modules that depend on them, and multiple versions of the same module must be maintained because the old versions may be used in parallel to the newer versions. Having a good semantic versioning convention is key to reduce the complexity of managing multiple versions of modules.

2.2 Layout engines

↪ *Before diving into the theory of element queries, it is important to understand how layout engines work. Only the layout engine of the browser will be of importance, since element queries do not affect any other browser subsystem. This section will give a brief explanation of the layout process that the engines generally perform, along with some usual optimizations that are performed to speed up the process. In section 2.2.6 insight will be given into the current state of the research of parallelizing the layout process. Last, layout thrashing will be presented and how it can be avoided in section 2.2.7. The render process will not be described, as it is not relevant for element queries.*

Have an image about the layout flow somewhere?

Marcos: The reference architecture could be outdated since the source is very old. The XML subsystem is hardly interesting anymore.

Maybe move this content to an own subsection and instead write a text about why this section is here, what the goal is (to understand parallelization and layout thrashing), etc.

2.2.1 Reference architecture and market shares

↪ *In this section, a reference architecture of layout engines will be given. In addition, the different major layout engines their market share will be presented. This section is mainly based on* [24, 21].

Browsers are complex applications that consist of many subsystems. A reference architecture of browsers has been presented by [24], see figure 2.1. It is

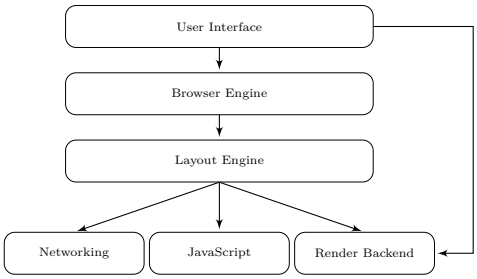

Figure 2.1. Reference browser architecture. The data persistence system, used by the browser engine and the user interface, has been omitted.

shown that the layout engine is located between the *browser engine* system and the network, JavaScript, and display backend. The browser engine acts as a high level interface to the layout engine, and is responsible for providing the layout engine with Uniform Resource Locators (URLs) of content that should be fetched and rendered. Additionally, browser engines usually provide the layout engine with layout and rendering options such as user preferred font size, zoom, etc. The main responsibility of the layout engine is to render the current state of the fetched hypertext document. Since documents may (and often do) change dynamically after parsetime it is important to keep in mind that the job of the layout engine is continuous, and is not a one time operation. The layout engine also performs the parsing of HTML. In short, layout engines perform four distinct tasks:

Explain DOM before this?

1. Fetch content (typically HTML, CSS and JavaScript) and parse it in order to construct a Document Object Model (DOM) tree.
2. Construct a *render tree* of the DOM tree.
3. Layout the elements of the render tree.
4. Render the elements of the render tree.

See section 2.2.2 and section 2.2.3 for a more in depth explanation of DOM and render trees. Browsers can typically display multiple pages at the same

time (by using tabs, multiple windows or frames) where each page has an instance of the layout engine.

There are four layout engines that the major browsers use, as presented in table 2.1. Since Blink is a recent *fork* of *WebKit* they have been grouped together as WebKit-based layout engines. This results in three distinct layout engines to consider: WebKit, Gecko and Trident. Due to Trident being closed source, only the WebKit-based layout engines and Gecko will be considered in this section.

Engine	Browsers	Share
Blink	Chrome, Opera	44.38%
WebKit	Safari	17.64%
Trident	Internet Explorer	14.96%
Gecko	Firefox	12.83%

Engine	Browsers	Share
WebKit	Safari, Chrome, Opera	62.02%
Trident	Internet Explorer	14.96%
Gecko	Firefox	12.83%

Table 2.1. The major layout engines and browsers with market shares. The table to the right has grouped together Blink into WebKit since it is a recent for of WebKit. See section B.3 for more information how the market share data was gathered.

Remove decimals of percentages since they are estimated?

2.2.2 Constructing DOM trees

↪ *Here a brief explanation of DOM trees will be given, and how they are constructed. In order to construct a DOM tree, the content needs to be parsed which also will be covered briefly. This section is mainly based on* [13, 21, 4, 14].

The DOM defines a platform-neutral model for events and node trees, which is used for representing and interacting with documents. The DOM provides an interface for programs (JavaScript in the browser) to access and mutate the structure, style, and content of documents. Elements of the document are converted to DOM nodes, and therefore the whole document is represented as a tree structure of DOM nodes which is referred to as the DOM tree.

Layout engines construct DOM trees by parsing HTML with any included CSS and JavaScript. Parsing of HTML is not an easy task, partly because it is expected (although not required) of layout engines to be forgiving of errors, such as handling malformed HTML. As CSS and JavasScript are stricter, their grammar can be expressed in a formal syntax and can therefore be parsed with a context free grammar parser. Another big quirk to parsing HTML is that it is reentrant that means that the source may change during parsing, see listing 2.2. Script elements are to be executed synchronously when encountered by the parser. If such script mutates the DOM, then the parser will need to evaluate the changes made by the script and update the DOM tree. External scripts need to be fetched in order to be executed, which halts the parsing

unless the script element states otherwise. It should be noted that although DOM nodes do include a `style` property, the CSS cascade does not affect the nodes in the DOM tree, and the style properties do not represent the final style of the element. Instead, for scripts to obtain the final computed style for an element special functions needs to be called (e.g. `getComputedStyle` in JavaScript[3]). Since externally linked CSS documents do not directly affect the DOM tree they could conceptually be fetched and parsed after the parsing of the HTML. However, scripts can request the computed style of DOM nodes and therefore either the parsing of HTML needs to be halted in order to fetch and parse CSS when needed, or the scripts accessing style properties of DOM nodes need to be halted. A common optimization is to use a speculative parser that continues to parse the HTML when the main parser has halted (for executing scripts usually). The speculative parser does not change the DOM tree, instead it searches for external resources that can be fetched in parallel while waiting for the main parser to continue.

```
<html>
  <body>
    <div id="container">
      <p id="tip">When in doubt, mumble.</p>
    </div>
    <script>
      var container     = document.getElementById("container");
      var tip           = document.getElementById("tip");
      var intro         = document.createElement("h4");
      intro.innerHTML   = "A tip:";
      container.insertBefore(intro, tip);
    </script>
  </body>
</html>
```

Listing 2.2. Simple example of reentrant HTML. The layout engine needs to reconstruct the DOM tree after executing the JavaScript. The page will not be rendered until all JavaScript has been executed.

Philipp: A picture of the DOM corresponding to some HTML would be good.

Write about media type, and how the CSS is calculated if it is in this stage?

2.2.3 Render trees

↪ *When the DOM tree has been constructed and all external CSS has been fetched and parsed, it is time for the layout engine to create the render tree. Here an explanation of the creation process and the structure of render trees will be given. This section is mainly based on* [21, 4].

A render tree is a visual representation of the document. In contrast to the DOM tree, the render tree only contains elements that affect the rendered re-

[3]See `http://dev.w3.org/csswg/cssom/#dom-window-getcomputedstyleelt-pseudoelt` for details about the function.

sult in any way. The nodes of the render tree are called *renderers* (also known as *frames* or *render objects*), as the nodes are responsible for their own and all subnodes layout and rendering. In order to know how to render themselves, the final style of each renderer needs to be computed which is done by the layout engine while constructing the tree. Each renderer represents a rectangle (with a size and position) with a given style. There are different types of renderers, which affect how the renderer rectangle is computed. The type can be directly affected by the `display` style property.

Typically nodes of the DOM tree have a 1:1 relation to nodes of the render tree, but the relation can also be smaller or larger. Since the render tree only contains nodes that affect the rendered result, nodes that do not affect the layout flow of the document and that aren't visible will not be present in the render tree. For instance, a DOM node with the `display` property set to `none` will have the relation 1:0 (it will not be present in the render tree because it is not visible and will not affect the layout flow). It should be noted that nodes with the `visibility` style property set to `hidden` will be present in the render tree, although they are not visible, since they still affect the layout flow.

Even though all style properties have been resolved for each node in the render tree (through CSS cascading), the renderers still do not know about the size and position of their rectangles. This is because some properties depend on the flow of the document, which cannot be resolved by style cascading and this need to be computed through a layout. The layout process will resolve the final position and size of all renderers.

2.2.4 Style computation

↪ *Both the render tree and scripts need to be able to get the final style of elements. Here a brief explanation of selector matching, style cascading, inheritance and rule set weighting will be given. CSS property definitions will also be described. This section is mainly based on* [21, 4, 12].

To trace how the style of an element is computed can be a complex task, since there are many parameters to element style computations. First, styles for an element can be defined in several places:

1. In the default styles of the browser.
2. In the user defined browser style.
3. In external CSS documents.
4. In internal CSS `style` tags in the document head.

5. In inline CSS in the `style` element attribute.

6. In scripts modifying the element style through the DOM tree.

7. In special attributes of the element such as `bgcolor` (deprecated, but possible).

Clarify this.

This can be grouped into author, user and browser styles. The four first items have in common that they are *cascading* their style rules through *selector matching*, and may define rule sets. Selector matching conceptually finds all elements in the DOM tree that matches a CSS selector, to apply style declarations of the rule set. The rule sets are weighted so that if a property is assigned values by multiple rule sets the one with highest weight will be applied. The weighting process starts with the origin of the style:

- In case of normal rule declarations, the style weighting relation is: $browser < user < author$.

- Important rule declarations have the following relation: $author < user$.

The weighting process continues by calculating the *specificity* of all rule set selectors, the higher the specificity the higher the rule set will be weighted. This will make rule sets with more specific selectors override rule sets with more general selectors. Finally, if two rule declarations have the same weight, origin and specificity the rule set specified last will win. Inline CSS does not cascade since all rules given are automatically matched with the element of the style attribute. However, inline CSS is considered with highest possible specificity when performing the style cascade. It is important to note that the styling methods number five and six are conceptually the same, since they both alter the style property of the DOM representation of the element. The last styling method, using special style attributes, is deprecated and all styles applied this way are weighted as low as possible.

If the cascade results in no value for an element style property, then the property can *inherit* a value or have an initial value defined by the CSS *property definition*. Property definitions describe how the style properties should behave; see table 2.2 for the format of property definitions. For instance, the `width` property definition states that legal values are absolute lengths, percentages or `auto` (which will let the browser decide). The percentages are relative to the containing block. The initial value is `auto`, the properties applies to all non-replaced inline elements, table rows and row groups. The value may not be inherited, and the media group is `visual`. The computed value is either the absolute length, calculated percentage of the containing block or

what the browser decides (in case of `auto`). If a property may inherit values, it will inherit the value of the first ancestor that has a value that is not `inherited`.

Value	Defines the legal values and the syntax.
Initial	The initial value that the property will have.
Applies to	The elements that the property applies to.
Inherited	Determines if the value should be inherited or not.
Percentages	Defines if percentages are applicable, and how they should be interpreted.
Media	Defines which media group the property applies to.
Computed value	Describes how the value should be computed.

Table 2.2. The CSS property definition format that describes how all element style properties behave.

This process will resolve most style properties, but as stated in section 2.2.3 some properties require a layout in order to be resolved.

Explain the syntax of CSS?

2.2.5 The layout process

↪ *When the render tree has been constructed, and the style properties have been resolved for all nodes, it is time to perform the actual layout. The layout process will decide the final computed style of all elements, and needs to be done before rendering. This section will describe the layout process, discuss some performance aspects and show examples when layout happens. This section is mainly based on* [21].

Maybe remove the examples bit.

HTML is flow based, which means that a document layout (also known as a *reflow*) can generally be performed top to bottom and left to right in one pass. This is possible because the geometry of elements typically do not depend on the siblings or children. Layout is performed recursively by starting at the root of the tree, let the renderer render itself and all of its children which will render themselves and their children and so on. When a layout is performed on the whole render tree it is called a *global layout*. To avoid global layouts when a renderer has been changed, a dirty bit system is usually implemented. The system marks which renderers in the tree that need a layout, which avoids layout of unaffected elements. Layouts that only layout the dirty renderers are called *incremental layouts*. A layout engine that uses the dirty bit system usually keeps a queue of incremental layout commands. The scheduler system later triggers a batch execution of the incremental layout commands asynchronously. A global layout is triggered when the viewport is resized or when styles that affect the whole document is changed (such as `font-size`). Because the API of `getComputedStyle` promises resolved values for all style properties, calling the function forces a full layout (flushing the

Seems weird to say that flow is not dependent on children, since the height is propagated upwards.

incremental layout commands queue). When a renderer performs a layout the following usually happens:

1. The own width of the renderer is determined.
2. The renderer positions all children and requests them to layout themselves (with given position and width).
3. The heights, margins and paddings of the children are accumulated in order to decide the own height of the renderer.

Of course, only the children that need a layout will be affected (dirty, or global layout) in step 2. So, the widths and positions are sent down in the tree and the heights are sent up in the tree in order to construct the final layout.

The widths are calculated by the `width` style of the elements, relative to the container element width. Margins and paddings are also taken into account when calculating the widths. When the width of an element is calculated, it needs to be controlled against the `min-width` and `max-width` style properties and make sure that the width is inside the given range. If the content of an element does not fit with the calculated width (text usually needs to perform line break when the width is too small), the element needs to break up the content into multiple renderers in order to expand the height. The renderer that has decided that it needs to break the content up into multiple renderers propagates to the parent renderer that it needs to perform the breaking. When the parent renderer has created the renderers needed to fit the content with the given height, layout is performed on the new renderers and then the final height can be calculated and propagated upwards.

2.2.6 Parallelization

↪ *No longer can performance of an application increase over time without any code changes (as opposed to the times when the Central Processing Unit (CPU) clock speeds increased rapidly). Now, applications need to utilize the multiple cores of the CPU instead of relying on high clock speed. Parallelization is something layout engine vendors are interested in, and research is being done about utilizing multiple cores to increase the performance of the engines. In this section a small summary will be given about the current research front, and how the parallelization can be approached. This section is mainly based on* [9, 38, 50, 66, 74].

Check so that the section actually does what the meta says

As web applications grow bigger and more demanding, browsers continuously need to improve the performance on all levels. Fetching resources over the network is the only thing that is done in parallel today. The rest of the

browser system is designed and optimized to run sequentially. On computers, browsers usually achieve parallelism by running each page context in parallel (each tab and window of the browser runs in a separate process). This approach is appealing because it utilizes the cores of the machine by still having all subsystems run sequentially for each page, while improving the overall performance of the browser. However, this approach is not enough. The page performance is not improved if only one page is present. For the web to be a true competitor to heavy native applications, the page performance needs to be increased (not only the overall browser performance). It is then important to be able to dedicate multiple cores to one page instead of having all the pages present using their own core (perhaps all non-visible pages can share one core while the main visible page can have access to multiple cores if needed). Also, with the number of mobile devices browsing the web increasing rapidly it is important to be able to achieve good parallelism. Light devices such as small laptops, mobile phones and tablets share a common goal — they want to reduce power consumption while increasing the performance. This can be achieved with multicore processors that run at lower clock speeds. It has been shown that the performance of mobile browsers is CPU bound contrary to the common belief that they are network bandwidth bound [38]. This is why many researchers target light devices when trying to parallelize web browsers. Of course, once the methods have matured and been implemented, desktop layout engines will benefit from parallelization as well.

Reformulate to not use the word native.

Write that many cores is preferred over high clock speed? Why?

CSS selector matching is a good candidate for parallelization, due to matching nodes to selectors being independent from other selector matches. A successful parallelization of selector matching has been achieved with locality-aware task parallelism [38]. It has also been shown that selector matching and style resolving (through cascading) can be parallelized [9]. It is possible to resolve element styles in parallel as long as two requirements are fulfilled: the matching task must have finished for the element to resolve styles for, and the parent element must have resolved all styles (since the element might inherit some styles from the parent). As long as these requirements are fulfilled, selector matching and style resolving can run in parallel for different elements. The layout process can also be parallelized, since the layout process has been shown to be subtree independent for non-float elements [66, 74]. Siblings of the layout tree can be processed independently of each other (in the general case), and is suitable to parallelize with a work-stealing algorithm[4] [50].

Philipp: It would be very interesting to just shortly summarize typical performance gains due to parallelization.

[4]See `http://supertech.csail.mit.edu/papers/steal.pdf` for more information about work-stealing algorithms.

2.2.7 Layout thrashing and how to avoid it

↪ *This section will present what layout thrashing is and how it can be prevented. Avoiding layout thrashing is important when optimizing DOM manipulations, and will be a key element to constructing a performant third-party library.*

Recall from section 2.2.5 that layout engines store incremental layout commands in a queue in order to batch process layouts. The term layout thrashing refers to when the layout queue repeatedly is being flushed by scripts; forcing the layout engine to perform multiple independent layouts that could have in theory been batch processed. Layouts are *thrashed* since the layout engine usually needs to perform layout on the whole subtree of the affected element for each incremental layout command, hence thrashing most of the work it did the previous layout command. See listing 2.3 for an example of JavaScript code that results in layout thrashing.

```
/* Doubles the width of each element in the elements collection parameter. */
function doubleWidths(elements) {
  elements.forEach(function readWriteWidth(element) {
    var width = parseSize(getComputedStyle(element).width);
    element.style.width = width * 2 + "px";
  });
}

// 1000 div elements as only children of a div container.
var elements = [...];
doubleWidths(elements); // ~700 ms
```

Listing 2.3. Example of layout thrashing. The code reads and double the widths of 1000 elements in ~700 ms. The `parseSize` function is not important to understand the example.

For each element given as parameter to the `doubleWidths` function, the `readWriteWidth` function is called with an element as argument. In that function, the computed style of the element is requested, which will force a style computation for that element. If any DOM node affecting the element is marked as dirty, a layout will also need to be performed before returning the computed style of the element. When the computed style has been acquired, the width of the element is set to a new value, which is a DOM mutation that will need to be synchronized with the render tree. Therefore, the element and its ancestors will be marked as dirty. The next time `getComputedStyle` is called, which happens when processing the next element in the collection, a layout will be forced since DOM nodes that affects the elements has been marked as dirty. See figure 2.2 for a visualization of the execution of the `doubleWidths` function. It is shown in the figure that style computations and layout is performed in each iteration. The total time spent on style computations is ~40 ms and layout is ~600 ms, which leaves ~60 ms for the actual script execution.

To avoid layout thrashing, the computed style requests and DOM muta-

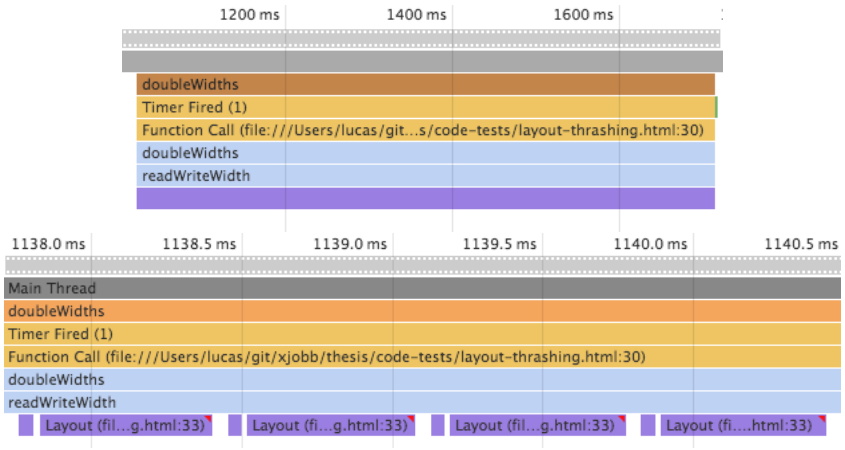

Figure 2.2. Two timeline figures of executing the example code given in listing 2.3. Both figures show the same timeline, where the top figure shows the whole execution and the bottom only shows a small section of the execution. Notice that they do not share the same time axis. Here it is clear that layout thrashing occurs, since layout is being done repeatedly throughout the whole execution (each iteration in the loop).

tions need to be performed in batches, as presented in listing 2.4. While it is algorithmically inefficient to iterate over the same collection twice — the batched approach executes ~100 times faster than the original version.

```
/* Doubles the width of each element in the elements collection parameter. */
function doubleWidths(elements) {
  var widths = [];

  // First retrieve all element widths.
  elements.forEach(function getWidths(element) {
    var width = parseSize(getComputedStyle(element).width);
    widths.push(width);
  });

  // Then mutate the DOM with the new widths.
  elements.forEach(function writeWidths(element, index) {
    element.style.width = widths[index] * 2 + "px";
  });
}

// 1000 div elements as only children of a div container.
var elements = [...];
doubleWidths(elements); // ~7 ms
```

Listing 2.4. Example of avoiding layout thrashing by batch processing reads and writes to the DOM. The code reads and double the widths of 1000 elements in ~7 ms.

Now, the `doubleWidths` function performs two batches that both iterates over the elements collection. In the first batch, all widths of all elements are acquired and stored in a collection. Since no DOM mutation occurs in the first batch, the layout engine does not need to perform any layout while retrieving the styles. In the second batch, all elements are assigned the new widths based on the widths retrieved in the first batch. Since the script does not read the DOM in any way during in the second batch, all DOM mutations

can be queued by the layout engine to be batch processed. See figure 2.3 for a visualization of the execution of the batched example version. In the figure, it is clear that the layout engine is able to first execute the JavaScript, and later perform all style and layout computations in a batch.

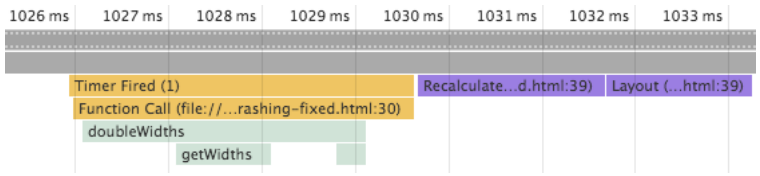

Figure 2.3. A timeline of executing the example code given in listing 2.4. It is clear in the figure that layout thrashing does not occur, since all style computations and layout is performed in a batch after that the example code has finished executing.

The total time spent on style computations has been reduced to ~2 ms, layout to ~1.5 ms, and the actual script execution to ~3.5 ms. It should be noted that both versions recalculate styles for the same number of elements, from now on denoted by n. Pausing the JavaScript context and switching the layout engine to style computation mode causes an overhead cost, from now on denoted by the constant O_s. Let C_s denote the time it takes for the layout engine to calculate the style of one element, then the total time needed for style computations in the first version is given by $T(n) = n(O_s+C_s)$. Since the second version enabled the layout engine to calculate the style of all elements in a batch, the total time needed was reduced to $T(n) = nC_s + O_s$. The style computation was reduced from ~40 ms to ~2 ms because the second version avoids $n - 1$ overhead costs. However, it should be noted that both versions have time complexity $O(n)$.

The major part of the time reduction is due to the layout time being significantly reduced. The reason why the layout time was significantly reduced is because the two versions do not perform layout on the same amount of elements. When an element and its ancestors has been marked as dirty, a layout will need to be performed on the subtree which contains at least all n elements and possibly some ancestor elements. So the minimum number of nodes that need to be laid out is given by $n + n_a$, where n_a is the number of ancestor elements. Since the first version forces the layout engine to perform a layout in each iteration, the minimum number of elements laid out is given by $n(n+n_a)$. Similar to style computation, performing a layout also has an overhead cost O_l. Let C_l denote the time it takes for the layout engine to layout one element, then the total time needed for layout in the first version is given by $T(n) = n(O_l+C_l(n+n_a))$. The second version enables the layout engine to batch process the DOM manipulations, and therefore only one layout of the subtree will need to be performed, which results in the minimum number of

elements laid out being $n+n_a$. The total time needed for layout in the second version is then given by $T(n) = O_l + C_l(n + n_a)$. The first version has time complexity $O(n^2)$ while the second version has time complexity $O(n)$, which is a significant optimization.

Explain that the script execution time was probably reduced due to avoiding n-1 context switches between the JS engine and the layout engine?

Analyze the numbers to give some rough numbers on overhead and cost times?

Explain that n_a is not a parameter to the T function?

Refer to some section in the appendix for how to read the timeline images? Philipp said that I should try to explain it very briefly in this section.

Chapter 3

Element queries

↪ *Now that a good understanding of browsers (especially the layout engine), responsive design, and modular development has been acquired it is time to address element queries — a solution to the modular responsive web design problem. This chapter will define element queries, present why they are needed. It will also address some of the issues of implementing element queries being natively in section 3.1. Further, some possible approaches to overcome the problems will be presented in section 3.2.*

Instead write that some solutions will be presented?

Describe a common use case? The thesis talks about "the common/general use case" a lot, but never really defines what that is.

Make additional sections: "common/general use cases" and "definition" to include the text below?

Recall from section 2.1.2 that responsive web design is the concept of having elements respond to size changes. Responsive web design implements this by using media queries, applying conditional rules with relation to the viewport size. In section 2.1.3 the desire for building modular web components was presented. One positive outcome of building applications in a modular way is that general components become reusable. Since the only way of applying conditional rules to elements today is by using media queries, all responsive styles must be defined by the application and not by the modules (because only the application knows how much space each module will be given in the application layout). The implication of this is that the HTML and JavaScript can be written in a modular way, but the CSS is left for the user of the module (application) to write which somewhat defeats the purpose of writing modules. So developers today have two choices: writing modules that are not responsive but encapsulated, or writing modules that are responsive but not encapsulated (leaving the styling to the user).

Tomas: Or that modules can programatically work in many different sizes

Marcos: Maybe embellish that bit... "...respond to size change, so that a document may be usable on many screen sizes, resolutions, pixel-densities, and media." or something like that.

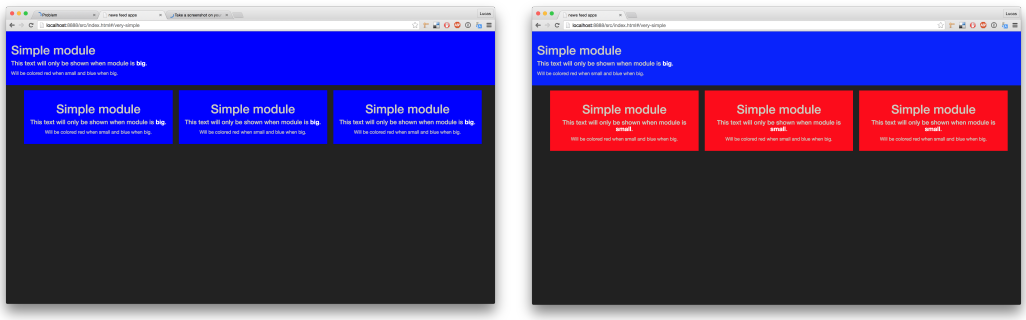

Figure 3.1. The problem of media queries and the element query solution. Both images shows four instances of a module in different sizes. The module is responsive, so when it is small it should change color to red and change the inner text a bit. The left version is built with media queries; hence all module instances stay blue even though some are small (only a smaller viewport will trigger the modules to change style). The right version is built with element queries, hence the modules display differently depending on the given size that each module has, which is the desired behavior.

An identified solution to this is element queries, which allows conditional rules to be applied based on any criteria for any element. Figure 3.1 shows the problem with media queries and the element query solution. The typical use case is to write conditional styles to be applied based on the container element size. Although the size of the container element is typically the most interesting element query with responsive web design, element queries as a concept does not need to be limited to that. It is also possible to apply conditional rules by any element display property, color, text alignment, and so on. Theoretically it should be possible to query any style property of an element, but the focus will be the `width` and `height` properties since they are the most interesting in responsive web design.

Write about the pseudo-syntax that will be used as example from now on?

3.1 Problems

↪ *One reason that hinders element queries from being implemented natively in browsers is that they bring problems and limitations to layout engines. It is stated on the W3C's www-style mailing list [73] by Zbarsky of Mozilla, Atkins of Google and Sprehn of Google that unrestricted element queries is infeasible to implement. It is important to understand the problems in order to speculate about a potential native API design. In this section the two major problems will be presented: performance and circularity.*

Rewrite a bit if native API is dropped.

3.1.1 Performance

↪ *This section will present the performance impacts that element queries have on layout engines. As shown in section 2.2.6 layout engine vendors are interested in parallelizing their engines to increase the page performance. Element queries limits the parallelization, which will be the focus of this section. This section is partly based on* [73].

Recall from section 2.2.5 that the rectangle size and position for each element is calculated in the layout process, and cannot be determined before an actual layout has happened. This imposes that a layout pass needs to be performed before knowing the element's sizes. If element queries are present that rely on the size of elements, which is the identified common use case, the following process needs to happen:

1. A layout pass needs to be performed in order to calculate the size of the elements targeted by element queries.

2. The element query conditional rules need to be evaluated against the elements to know which rules should be applied (element query selector matching).

3. If the element query selector matching has resulted in a different matching set than in step 1, the process is repeated (with the new rules applied).

So, element query selector matching changes result in performing another layout, which discards at least a subtree of the previous layout. As already stated, this only occurs when the layout has changed in a way that changes the element query selectors. Unfortunately, this means that if there is any matching element query selector at page load, two layout passes will always be performed.

Further, it is common for internal APIs in layout engines to request updated element styles that do not require layout to resolve (non-layout properties such as `color` and `font-family`). Since a layout of the element query selector target is required in order to resolve the correct element style, such internal APIs requires a layout in order to obtain the correct element style (even for non-layout related properties). Also, layout engines would not be able to layout subtrees in parallel since an element in one subtree might affect an element in another subtree.

Totally forgot to write about parallel problems

3.1.2 Circularity

↪ *This section will show how cyclic rules may occur and why it is a problem. It will also be presented why cyclic rules must be detected at runtime. Some examples will be given to better understand the complexity of the problem.*

The most obvious occurrences of cyclic rules are when there are specified conflicting width element queries and width updates for an element. See listing 3.1 for the perhaps simplest example of cyclic rules. The syntax of element queries will be presented in section 3.3, and is not important to understand the problem.

Have the explanation before this section? On the other hand this section is needed for the motivation of the syntax of the native API

```
#foo {
  width: 250px;
}

#foo:eq(width < 300px) {
  /* This rule will be applied only when the width of #foo is < 300px. */
  width: 550px;
}

#foo:eq(width > 500px) {
  /* This rule will be applied only when the width of #foo is > 500px. */
  width: 250px;
}
```

Listing 3.1. Simple example of cyclic rules with directly conflicting width element queries and updates.

The following happens to the element:

1. The initial width of the `#foo` element is set to 250 pixels. After a layout, the `#foo:eq(width < 300px)` matches and therefore next step will be 2.

2. The width of the element is under 300 pixels, so the selector `#foo:eq(width < 300px)` matches. Note that the `#foo:eq(width > 500px)` does not match, since the width is under 500 pixels. Since the matched selector is more specific than the `#foo` selector, the new width of the element is 550 pixels. Next step will be 3.

3. The width of the element is above 500 pixels, so the selector `#foo:eq(width > 500px)` matches. Note that the `#foo:eq(width < 300px)` does not match, since the width is above 300 pixels. Since the matched selector is more specific than the `#foo` selector, the new width of the element is 250 pixels. Next step will be 2.

Clearly, the browser will be stuck in an infinite layout cycle pending back and forth between step 2 and 3 (250 pixels and 550 pixels). The result of this is of course implementation dependent. One reasonable outcome of such infinite layout loop is that the layout engine executes one layout pass and then evaluate

the next set of matched selectors and so on, which leads to a functioning page but since a relayout is enforced every update, the performance impact is significant. This example is somewhat similar to writing `while(true);` outside the scope of a generator function in JavaScript (i.e., it just locks up the main thread), which obviously is a bad idea. However, cyclic rules can also occur in less obvious ways.

Indirect cyclic rules are somewhat more complex to reason about, than the example given above (which is an example of direct cyclic rules). For instance, if the `#foo` element matches the width of a child element, and the child changes width depending on the parent width, a cycle might occur. Consider the code in listing 3.2.

```
/* HTML */
<div id="foo">
  <div id="module">
    <div id="child"></div>
  </div>
</div>

/* CSS */
#foo {
  /* Will match the width of the child element #module. */
  display: inline-block;
}

#module {
  /* Matches the width of the parent element #foo. */
  width: 100%;
}

#child {
  width: 250px;
}

#module:eq(max-width: 300px) #child {
  /* This rule applies only when the width of #module is <= 300px. */
  width: 550px;
}

#module:eq(min-width: 500px) #child {
  /* This rule applies only when the width of #module is >= 500px. */
  width: 250px;
}
```

Listing 3.2. Example of indirect cyclic rules. Here the user (`#foo`) of the module (`#module`) creates cyclic rules indirectly by specifying that it should match the width of the module.

What makes this situation more complex than the previous example is that it is less obvious for the developer to identify that there are cyclic rules. First, the problem cannot be found without considering the rule sets of both the module and the `#foo` element. Second, the rules of the module elements and the rules of the `#foo` element might be separated into different parts of the stylesheet or different stylesheets. Third, the user of the module must be aware of how it styles itself in order to understand the limitations it imposes. By adding one line to the containg element `#foo` a cycle appears in another part of the program (in the module). A JavaScript equivalent of this example would perhaps be `var bar = true; while(bar);` with the motivation that

it is still obvious that it results in an infinite loop but both the loop and the variable needs to be considered. Also, the variable assignment could happen in another part of the program.

The examples given so far have been simple, and can easily be identified as cyclic by reviewing the CSS code. It would also be possible for the browser (or any other program) to detect the cycles at parsetime. However, cycles can occur in a complexer way that cannot be detected at parsetime. Consider the code in listing 3.3.

```
/* HTML */
<span id="foo">
  When in doubt, mumble.
</span>

#foo:eq(width < 300px) {
  /* This rule applies only when the width of #foo is < 300px. */
  font-size: 2em;
}
```

Listing 3.3. Example of cyclic rules that cannot be detected at parsetime.

In this example, it is impossible to deduce if the rules are cyclic at parsetime. It could perhaps guess that it could result in a cycle in some cases but that is also the point of the example — it is only cyclic in some scenarios. The size of the `#foo` element will depend on the content of it (i.e, the text). The width of the text depends on the font size, which is inherited. So the width of the `#foo` element is dependent on the inherited font size. When the `#foo` element is below 300 pixels wide, the font size of the element is increased to `2em` (which is a unit that is relative to the inherited value). If a `2em` font size results in a computed font size big enough to make the element wider than 300 pixels, the `#foo:eq(width < 300px)` selector will not match and therefore the element will no longer have font size `2em`. Since the element width is decreased below 300 pixels when the font size of `2em` no longer is applied, the selector will match again — the rules are cyclic. However, in another scenario the inherited font size might not be big enough to make the `#foo` element wider than 300 pixels which would not result in a cycle. The factors that creates the cycle are the following:

- The display type of the element
- The element queries of the element
- The font size value of the element
- **The inherited font size of the element**
- **The content of the element**

The factors in bold are worst than the other ones presented, since they may depend on runtime actions and values. In this example the text is static but it

could have been added dynamically. Also, the inherited font size value depends on the closest ancestor with a font size property defined. Since ancestors can have their font sizes defined in relative units, the dependency tree can go up to the root of the document. Further, if no ancestor defines an absolute value for the font size it is up to the browser to default to a size, which is impossible to reason about at parsetime. This implies that there is no way of telling if the cycle appears or not without actually running the code. Also, the cycle may appear in different browsers and settings, which makes the cycle even harder to detect.

Try to descibe a scenario where a cycle appears, that is actually a valid use case?

3.2 Possible solutions

↪ *Now that the problems of element queries have been presented, it is time to show two approaches to solve the problems. The first approach limits element queries a lot, but avoids many of the problems this way while still enabling element queries for the common use case. This approach has been discussed at the W3C and the initial draft of the RICG assumes that this is a prerequisite to a native implementation. The second approach is to provide element queries as a third-party library which operates element queries on top of the layout engine (in JavaScript). Although not a solution to the native problems, it still provides functioning unrestricted element queries while not hindering parallelization of layout engines. However, this approach does not solve the problem of cyclic rules. The different solution approaches either limits element queries in some way or solves only a subproblem, so no perfect solution that hinders element queries unlimited with decent performance and low complexity has been found. The section is mainly based on* [73, 62, 16, 4].

Marcos: Should rewrite this to not be exclusive to a viewport element. Could also be an attribute.

By limiting element queries to special viewport container elements that can only be queried by child elements, many of the problems are resolved. This means that a new HTML tag would be crafted (perhaps named `viewport`) that defines separated viewports in the document. In order to avoid cyclic rules, the following limitations to the `viewport` would be defined:

1. The size of the element may not be dependent on its children. This implies that all CSS that causes the element to be shrunken to fit the content are invalid.

2. The selector may only include the `viewport` element targeted by element queries as a part of the expression of a selector (not the right-most simple selector).

3. Only the nearest `viewport` ancestor of the right-most simple selector may be targeted for element queries in selectors.

See listing 3.4 for examples of valid and invalid CSS selectors according to the rules defined above.

```
/* HTML */
<viewport id="outer">
  <viewport id="inner">
    <p id="foo">Imaginary viewport elements</p>
  </viewport>
</viewport>

/* CSS */

/* valid */
viewport:eq(width > 500px) p { ... }

/* valid */
viewport viewport:eq(width > 500px) p { ... }

/* invalid, violates rule 1 */
viewport { display: inline; }

/* invalid, violates rule 2 */
p viewport:eq(width > 500px) { ... }

/* invalid, violates rule 3 */
viewport:eq(width > 1000) viewport:eq(width > 500px) p { ... }

/* invalid, violates rule 3 */
#outer:eq(width > 500px) #foo { ... }
```

Listing 3.4. Examples of valid and invalid selectors with the imaginary `viewport` element.

This approach also solves many of the performance problems, but not all. The reason that a new HTML tag is proposed instead of a new CSS property that defines the behavior is because the layout engine in the latter case needs to resolve the styles for all elements in order to know which are viewport elements. With a new HTML tag, the layout engine can know after it has parsed the HTML which elements are supposed to be treated at separate viewports. This way, parallelizing the selector matching and style computation is possible (as opposed to if the style for each element needs to be resolved in order to know the `viewport` elements). Also, the internal APIs that request non-layout information for the elements using element queries only need to make sure that the containing viewport element has been laid out before resolving the styles. The layout can be done in one pass as long as the viewport elements are laid out before their children. Since the queries may only be targeting the nearest viewport element ancestor, each viewport subtree can be laid out in parallel. However, it is still inconvenient that the layout engine would need to evaluate all element query selectors in the middle of a layout pass (after the

that the viewport elements has been laid out) in order to resolve the styles for the viewport children.

Obviously, this limits element queries a lot. The fact that the size of the viewport element cannot depend on the children (like normal block elements do), limits the usability. The `viewport` element would behave much like the `iframe` element layout-wise. It should be noted that `iframe` elements are not suitable as an alternative to the proposed `viewport` element, since `iframe` elements are much more limited by nature (creates a new document and script context). The idea is that the viewport cannot be queried for properties that the children may affect (such as the width and height style properties). In order to allow the children to query the properties, they cannot be affected by the children. In theory it should mean that if no child query for instance the height of the viewport, then the viewport may depend on the children for it's height. This is a powerful insight, since the general use case would be to write element queries against the width and have the elements adapt their heights accordingly (including the viewport element).

Maybe note seamless iframe in theory works in the same way. But would it be relevant?

Another approach to enable element queries for developers is to provide a third-party JavaScript library. Since a JavaScript library does not depend on vendors of layout engines (other than features in the current language specifications) it can therefore be designed in any desired way. It would be more feasible to have unrestricted element queries in a JavaScript library, than implementing it natively. Since the element queries feature would be operated by the library, layout engines can be parallelized unhindered and independently of how the library handles the queries. Similar, cycle detection and handling can be handled on top of the layout engine by the library. Although an implementation in JavaScript will be slower than a native implementation, it can still be beneficial to have such feature implemented as a third-party library to avoid complex and restricting code in layout engines. Also, without the layout engine aware of the element queries it could be hard to avoid multiple relayouts.

3.3 Native API design

Out of scope?

Chapter 4

Library design

↪ *Before starting the actual implementation, it is important to have an overall design to follow. In this chapter, the library design will be described and a motivation why the library should be plugin based will be given. The subsystems of the library and their interactions will also be presented.*

Write somewhere that it might be an option to not have a resize detection system, but instead let the user manually check the elements when they may have changed.

First of all, a working name of the library needs to be established — the library will from now on have the working name ELQ. This name serves as a prefix for many of the library APIs, and will be frequently used in the report from now on. The technical goals of ELQ will be stated in section 4.1. In section 4.2 the architecture will be described, and the subsystems of ELQ will be presented. Last, the API will be defined in section 4.3.

4.1 Technical goals

↪ *Before presenting the overall design of the library, the technical goals will be stated. It will be shown that the goals and use cases can vary from case to case, which will be the main argument to why the library should be plugin based.*

Expectations and requirements of element queries vary greatly by use cases. As usual in software development, tradeoffs need to be made. Some projects value simplicity and ease of use, while other projects demand advanced features and high performance. The requirements can be grouped into four categories; features, ease of use, performance and compatibility. Identified possible

requirements of the library are the following:

1. **Features**
    a) The library needs to augment native element queries in the sense that styles are applied automatically on element resizes.
    b) The features should be flexible and not limited to only the common use cases.
2. **Ease of use**
    a) Developers should not need to perform big rewrites of their modules or applications in order to use the library.
    b) The APIs need to feel natural and should not seem alien to web developers.
    c) The library should help identifying cyclic rules and prevent them from causing infinite layouts.
3. **Performance**
    a) The library needs to have adequate performance to empower large applications running on light devices.
    b) The library load time needs to be kept low.
4. **Compatibility**
    a) Older browsers must be supported.
    b) The library may not require invalid CSS, HTML or JavaScript.
    c) Dependencies and assumptions about the host application (including the development environment) should be kept low.
    d) The library must act as a prolyfill[1] for native element queries.

It would be possible to create a library that conforms to all of the requirements but added features, automation and compatibility most probably would decrease the performance and load time. In the same way, too many advanced features may decrease the ease of use and simplicity of the API. Some feature may also restrict the compatibility. Trying to conform to all of the requirements and trying to find a balance would result in a worse solution than a library tailored for specific requirements. By providing a good library foundation and plugins for different use cases it is up to developers to choose the

[1] A polyfill is something that provides a functionality that is expected to be provided natively by browsers. Polyfills usually fixes some standard functionality for browsers that do not yet support it. A prolyfill is a polyfill for something that will probably become a standard.

right plugins for each project. This way, developers are composing their own custom tailored solution for their specific use cases. In addition, by letting the plugins satisfy the requirements it is easy to extend the library with new plugins when new requirements arise. For instance, the requirement 4d is beneficial to satisfy with a plugin since the API for native element queries can only be guessed at this point of time, which will probably lead to frequent changes to the plugin. By separating it from the core library (and the other plugins), only developers that really desire the prolyfill behavior must handle the rapid API changes.

4.2 Architecture

↪ *As it has been shown that a plugin based library is suitable, it is time to present the overall architecture of the library. The roles of each subsystem of the core library will also be explained.*

Here the style state of the element is talked about. Maybe this should have been mentioned before? Perhaps in the chapter text?

To decide if a subsystem of the solution should be in the library core or a plugin is a difficult task. A balance needs to be found between ease of use, performance and extendability. All subsystems of the core need to provide a fundamental functionality to the library and also need to be general enough to be used by plugins in different ways. If a plugin needs to write a custom version of a subsystem in the core in order to work, the subsystem should perhaps not be part of the core. Each subsystem added to the core will impact the performance negatively (by slower load time in the best case) for all users, so they must impose a real value to existing and future plugins. The subsystems of the core always have the risk of being redundant, unnecessary and in other ways unwanted for some use cases. Therefore it is important that it is possible for developers to either remove them or change them. This way the more advanced users can choose which subsystems to alter, keeping the library easy to use for the common use cases.

Figure 4.1 shows the overall architecture of the library. The core consists of fundamental and general subsystems that are utilized by plugins. Developers should never be programming against the core directly, and it can therefore only be accessed by plugins through the hidden plugin API. The library provides a small public API which is mainly used to invoke and handle plugins. The plugins form the bigger part of the public API in the sense that they decide which features exist and how they work. It should be noted that plugins

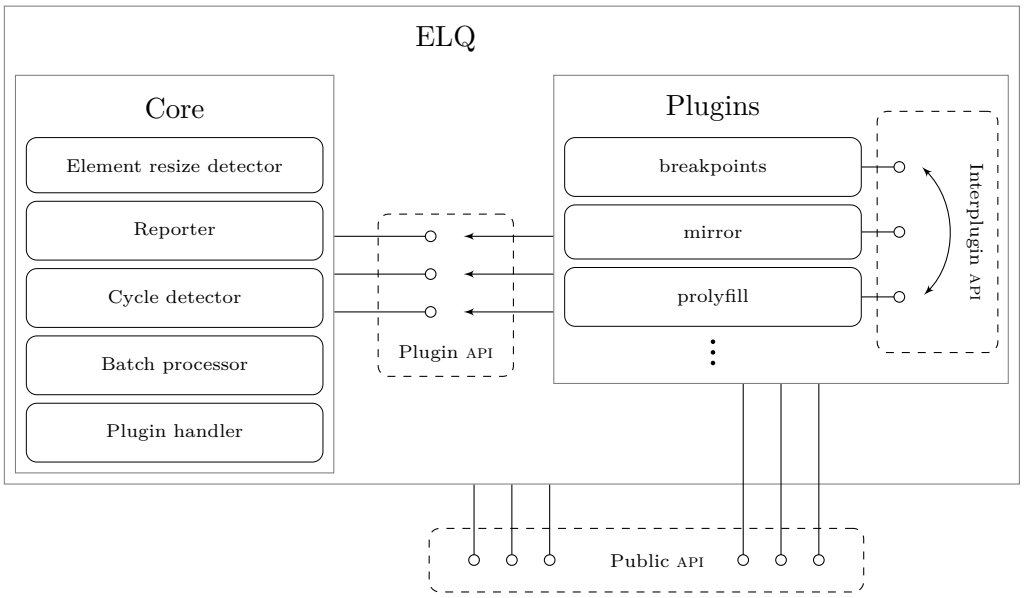

Figure 4.1. The overall library architecture, which shows how the library is divided into two parts; the core and plugins. The core is only accesible through the plugin API, and is not a part of the public API. Plugins may have interplugin API's and dependencies. The bigger part of the public API's is defined by the plugins.

may provide APIs through the library by registering methods, or by specifying custom syntax (such as markup or CSS) which is beyond the control of the core. Plugins may also have interplugin APIs and dependencies, which also is beyond the control of the core. The core is supposed to have a slow development phase, with few changes and good backwards compatability. Plugins may be developed in a high rate, with frequent changes to support new features. Subsystems that are part of the core are the following:

- **Reporter:** Responsible for reporting information, warnings and errors to the developer. By having a centralized reporter, it is possible to decide at a global library level how much plugins are allowed to report. Other options such as throwing exception on errors or warnings could also be set at a global library level. Also, the library can make sure that all reports are standardized and can provide extra information such as the name and version of the plugin along with its report. To avoid code duplication, it also beneficial to have a centralized report system so plugins do not need to perform the same compatibility checks (such as checking if the browser actually supports console reporting).
- **Element resize detector:** Provides an interface for observing element resize events. This system is fundamental to plugins fulfilling require-

ments 1a and 2a since it enables the library to be aware of when elements change sizes, instead of the host application keeping track of when elements have changed sizes.

Maybe this shouldn't be a core subsystem? Since this system imposes the biggest performance impact. It might make more sense to have this as a plugin.

- **Cycle detector:** As shown in section 3.1.2 cyclic rules need to be detected at runtime. When a plugin wish to update the size state of an element (which in turn applies the conditional styles) it can ask this system if the update seems to be part of a cycle. If the update seems to be part of a cycle, it is up to the plugin how that should be handled.

  Write that this system is more general, and could potentially be used for detecting other cyclic behaviors?

- **Batch processor:** To avoid layout thrashing, it is important to read and mutate the DOM in separate batches. This subsystem provides an interface for plugins to store functions to later be executed in a batch. To avoid layout thrashing by other scripts it can beneficial to asynchronously postpone the batch until the next layout cycle of the browser, which is also handled by the subsystem.

- **Plugin handler:** Of course, having plugins requires a subsystem for handling them. This system provides the interface for developers to add plugins to the library. The system is also responsible for retrieving the plugins, and invoking them on different library events.

Some of the subsystems may be partially accessible through the public API such as the plugin handler and the element resize detector. It should be noted that the presented core subsystems are not the only things that the library consists of, but the presented systems are the major ones.

4.3 API design

↪ *This section will describe the API of the library. First, the public API of the core will be presented, followed by the plugin API. After that, all plugins will be described in depth addressing their features, dependencies and APIs.*

Recall from section 4.2 that the bigger part of the public API is provided by plugins. The role of the library is to provide plugins with systems to be used for performing element query tasks, which is done through the plugin

API. Three plugins will be developed; `breakpoints`, `mirror` and `prolyfill`. The `breakpoints` plugin provides

4.3.1 Public API

Write how to install it. AMD, script, commonjs, etc.

In section 4.1 it was determined that advanced users must be able to change the subsystems used in the core. Therefore no global library instance will be instantiated automatically on load. Instead, a function `Elq` that creates ELQ instances is provided in order to be able to inject dependencies. The function accepts an optional options object parameter, where it is possible to set options and subsystems to be used for the created instance. If no options are given, default options will be used. See listing 4.1 of example usages of the `Elq` function. The function returns an object containing the core public API methods of the ELQ instance.

```
//Default options being used.
var elq = new Elq();

//A custom reporter and cycle detector being used.
var customCycleDetector = ...;
var customReporter = ...;
var elq2 = new Elq({
  reporter: customReporter,
  cycleDetector: customCycleDetector
});
```

Listing 4.1. Example usages of the `Elq` function that creates ELQ instances.

The next step after that an instance has been created is to register the plugins to be used by the instance. There are two methods for handling plugins: `use` and `using`. The `use` method registers a plugin to be used by the library. It is responsible for controlling that plugins do not conflict with each other and that plugins are initiated correctly with the library instance. The method requires a *plugin object* and accepts an optional options object parameter. When called, it will create a plugin instance and return it. A plugin object acts as a plugin definition and is responsible for describing a plugin and providing a function to create a plugin instance. The `using` method requires an *plugin identifier* as parameter and tells if the given plugin is being used (has been reigstered) by the instance or not. A plugin identifier can either be the plugin object or the name of the plugin. See listing 4.2 for details about the plugin object and how it is used with the three methods for handling plugins.

```
var myPlugin = {
  getName: function() {
    // Returns the name of the plugin, which has to be
    // unique in an elq instance.
    return "my-plugin";
  },
  getVersion: function() {
    // Returns the version of the plugin.
```

```
    return "1.0.0";
  },
  isCompatible: function(elq) {
    // Returns a boolean that indicates if this plugin
    // is compatible with the given elq instance.
    return true;
  },
  make: function(elq, options) {
    // Returns a plugin instance. It is optional to use
    // the elq instance or options object in the initiation process.
    return {...};
  }
};

// Tell the elq instance to use myPlugin with default options.
var myPluginInstance1 = elq.use(myPlugin);

// Tell the elq2 instance to use myPlugin with custom options.
var options = {...};
var myPluginInstance2 = elq2.use(myPlugin, options);

// The plugin instances are not equal since they have been initated to different elq instances.
myPluginInstance1 !== myPluginInstance2 // -> true

// Check if the plugin is being used.
elq.using(myPlugin); // true

// Also possible to check by the plugin name.
elq.using("my-plugin"); // true

// Plugins not being used by the instance returns false.
elq.using("other-plugin"); // false
```

Listing 4.2. Plugin object definition and examples of handling plugins.

When the desired plugins have been registered to the ELQ instance, it is time to apply the library to the target elements (i.e., the elements that will be queried for conditional styles). This is done with the `start` method which requires a collection of elements as a parameter. When called, it will invoke all `start` methods on all registered plugins which gives them the opportunity to initiate the elements according to their own mechanisms. To satisfy the requirements 2b and 4c the function is agnostic about the elements collection — all objects that are iterable and contains elements are accepted. It is also possible to provide a single element without a collection. The effect of invoking the `start` method on an element multiple times is defined by the plugins. See listing 4.3 for example usages of the `start` method.

```
// Initiating the library to a single element.
var singleElement = document.getElementById("...");
elq.start(singleElement);

// Initiating the library to multiple elements.
var singleElement2 = document.getElementById("...");
elq.start([singleElement, singleElement2]);

// Initiating the library to multiple elements using
// third-party libraries (in this case jQuery).
var jqueryElementsCollection = $("...");
elq.start(jqueryElementsCollection);
```

Listing 4.3. Example usages of the `start` method. The method only requires an iterable collection, so it is library agnostic.

4.3.2 Plugin API

The plugin API is a superset of the public API. Plugins have additional direct access to most core subsystems presented in section 4.2. The plugin API access is given to a plugin when it is registered (i.e., when the ELQ instance invokes the `make` function of the plugin definition). Recall from subsection 4.3.1 that the `make` function of the plugin definition is given two parameters at invokation; `elq` and `options`. The first parameter is a reference to the ELQ instance (to which the plugin was registered) extended with the plugin API. Direct access is given to the following core subsystems:

- **Reporter**: via the `elq.reporter` property that refers to the reporter instance.
- **Cycle detector**: via the `elq.cycleDetector` property that refers to the cycle detector instance.
- **Batch processor**: via the `elq.BatchProcessor` property that refers to a factory function that creates batch processor instances.
- **ID handler**: via the `elq.idHandler` property that refers to the ID handler instance.

Include id handler in section 4.2?

Direct access is not given to the plugin handler and element resize detector subsystems since their APIs are already exposed in the public API. However, an additional method of the plugin handler system is exposed; the `getPlugin` method that returns the registered plugin instance given a plugin identifier parameter. This method is needed to enable interplugin communications.

The reporter has three methods; `log`, `warn` and `error`. They are used to report information, warnings and errors respectively. The default behavior of the reporter is to report to the browser console if present.

The cycle detector has a method `isUpdateCyclic` that tells if the desired state update of an element is part of a cycle. To do so, it requires two parameters; the element that the update should be applied to, and the state update itself. A third time parameter, that tells when the update was requested, is optional and will default to the time of the method call.

Instead of having a single batch processor instance shared among all library entities (i.e., core subsystems and plugins), each entity has to create an own instance. This to avoid entities interfering with each other while performing *batch processing*. For example, some entities might want to force the batch to process at a point where it would be beneficial for other entities to delay the batch. The `createBatchProcessor` accepts an optional options object parameter and returns a batch processor instance that has two methods; `add` and `force`. The `add` method requires a function parameter

that will be called when the batch is processed. Since the batch processor is leveled, as motivated in section **??**, the method also accepts an optional parameter that defines at which level the given function should be processed. The `force` method commence the processing of the batch, which can happen synchronously or asynchronously a the optional parameter. Two important options to the `createBatchProcessor` are the `async` and `auto` options. If the `async` option is enabled, the batch will be processed asynchronously as soon as possible after that the `force` method has been invoked. If the `auto` option is enabled, `force` will be invoked when the first function of the batch has been added (this implies that `async` has to be enabled).

The ID handler has one method `get` that returns the ID of the required element parameter. If the element does not have any ID one will be assigned to the element. It is possible to override the assignment with an option parameter.

Code examples?

4.3.3 Plugins

↪ *So far, only prerequisites for third-party element queries has been presented. Now, it is time to describe the systems and APIs that will actually realize element queries as ELQ plugins. It should be noted that the plugins presented here is the suggested solution to element queries according to the research of this thesis. They are designed for good performance, good compatibility, and ease of use. For extreme requirements or corner cases, custom ELQ plugins might be preferred.*

ELQ plugins may use custom element attributes to define rules and plugin options. The HTML standard supports custom attributes prefixed with `data-`. Modern browsers usually permits to discard the `data-` prefix for custom attributes. For visual reasons, custom attributes will be written in the shortest possible way in this thesis (without the `data-` prefix). It should be noted that although not written in the examples, the library and all plugins support all combinations of custom attribute declarations so if desired, the library is able to conform to the HTML standard.

Breakpoints

This is the plugin that observes element resizes and updates their size states according to predefined breakpoints. The name of the plugin is `elq-breakpoints` and it does not have any dependencies (other than the element resize detector of the ELQ core). The main idea of the plugin is to apply classes to target elements that reflects in which size state they are. Each element defines

breakpoints to which the plugin will update the size state. The breakpoints of an element are defined by custom element attributes. See listing 4.4 for an example of breakpoints definitions.

```
<div id="target" elq elq-breakpoints elq-breakpoints-width="300 500">
...
</div>
```

Listing 4.4. Example an HTML element that uses the `elq-breakpoints` plugin.

The `elq` attribute states that it is an ELQ element, and the `elq-breakpoints` attribute states that the element should be handled by the plugin. The `elq-breakpointswidth="300 500"` tells the plugin that the element should have two width breakpoints at 300 and 500 pixels. Height breakpoints are defined in the same way with the `elq-breakpoints-height` attribute. The possible size states of the element would then be (in pixels):

- $width < 300$
- $300 <= width < 500$
- $500 <= width$

The plugin will append classes to the element that reflects the size state of the element. For each breakpoint, one class will be present that tells if the size is above or below the breakpoint. The number of size states of an element in a dimension is given by $n_{ss}(n_b) = n_b + 1$, where n_b is the number of breakpoints in that dimension. The number of breakpoint classes of an element in a dimension is given by $n_{bc}(n_b) = 2n_b$. The number of breakpoint classes active at the same time for an element in a dimension is given by $n_{abc}(n_b) = n_b$. The format of the classes are `elq-[dimension]-[above|below]-[breakpoint]`. For example, if the example element is below 300 pixels wide, it will have the following two classes present:

- `elq-width-below-300`
- `elq-width-below-500`

Breakpoint classes are utilized for applying conditional styles for elements based on the size state of the target element. See listing 4.5 for conditional styles of the target element and it's children.

```
#target {
  color: black;
  font-size: 15px;
}

#target.elq-width-below-300   { font-size: 10px;  }
#target.elq-width-above-300   { color: red;       }
#target.elq-width-above-500   { font-size: 20px;  }
#target.elq-width-above-500 p { padding: 10px;    }
#target.elq-width-below-500 p { padding: 5px;     }
```

Listing 4.5. Example of conditional styles using the `elq-breakpoints` plugin classes

In this example it is shown that elements can be conditionally styled depending on their own sizes. Additionally, children can be styled by stating the target element size state as the left-most selector. In the example, all paragraph elements `p` will be styled conditionally depending on the target element size state.

Write about HTML compatability with data- prefix and empty attributes.

Example of how to register the plugin with an elq instance + options?

Write that above is inclusive and below is exclusive? Why? Option?

Mirror

This plugin is needed as an addition to the `elq-breakpoints` plugin in some cases due to limitations of CSS. As of the CSS3 specification, there is no way to select ancestor elements of an element. For instance, `p img` is a selector that matches all image elements that are contained by a paragraph. However, it is not possible to construct a selector that matches all paragraphs that contains images. The `elq-mirror` plugin overcome these limitations by mirroring the breakpoint classes of an `elq-breakpoints` element. Suppose that it is desired for an application to style all paragraph elements differently by the size of the nearest element with a `container` class. Paragraphs may appear nested in other elements, that may be generated dynamically at runtime. Therefore, the structure of the application is not known when writing the style code, so no assumptions can be done about the markup structure. Consider the example markup given in listing 4.6 for an example markup structure. It should be noted that the `container` elements may be styled in any way (e.g., the widths of the inner container elements could be relative to the outer container element).

```
<div class="container" elq elq-breakpoints elq-breakpoints-width="300 500">
  <div>
    <p>Paragraph 1</p>
  </div>

  <div class="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <p>Paragraph 2</p>
  </div>
  <div class="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <p>Paragraph 3</p>
  </div>
</div>
```

Listing 4.6. Example markup structure where all paragraphs are desired to be conditionally styled by the nearest ancestor container.

A naive approach to applying conditional styles for the paragraph elements would be using the selector structure given in listing 4.7.

```
.container p                          { background-color: red;    }
.container.elq-width-above-300 p      { background-color: yellow; }
```

```
.container.elq-width-above-500 p  { background-color: green;  }
```

Listing 4.7. A naive approach to styling the paragraphs conditionally.

The problem with this is that the selectors states that all paragraphs that are children of *any* container should be styled in some way. So if the outer container element is 600 pixels wide and the inner containers are 200 pixels wide, all paragraphs would be colored green. In this particular case, the desired behavior is that the first paragraph is colored green and the two inner paragraphs colored red. By using the `elq-mirror` plugin, the correct behavior can be achieved. Elements are initiated to mirror elements by adding attributes in the same way like with the `elq-breakpoints` plugin. The mirror elements will then match the `elq-breakpoints` classes of the nearest ancestor that has the `elq-breakpoints` attribute. See listing 4.8 for how the plugin can be used to achieve the desired behavior of the conditionally styled paragraphs.

```
/* HTML */
<div class="container" elq elq-breakpoints elq-breakpoints-width="300 500">
  <div>
    <p elq elq-mirror>Paragraph 1</p>
  </div>

  <div class="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <p elq elq-mirror>Paragraph 2</p>
  </div>
  <div class="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <p elq elq-mirror>Paragraph 3</p>
  </div>
</div>

/* CSS */
.container p                        { background-color: red;    }
.container p.elq-width-above-300    { background-color: yellow; }
.container p.elq-width-above-500    { background-color: green;  }
```

Listing 4.8. By using the `elq-mirror` plugin to overcome the limitations of CSS, the correct behavior can be achieved of the conditionally styled paragraphs.

Notice that the breakpoint classes now are used in conjunction with the paragraphs instead of the containers in the CSS selectors. Now, the conditional styles will be applied to all paragraphs that are children of containers; with the size criterion being relative to the nearest ancestor container.

By mirroring an element's breakpoint classes, elements can be styled by criterion of any element (ancestor, sibling, child, etc.). The plugin is at the moment restricted to the nearest ancestor `elq-breakpoint` element, but could easily be extended to support more advanced mirror constellations.

Maybe it should be presented how such API could look like?

How about elements that want to write element queries against a child? CSS cant go upwards, but might be supported in the future. See `http://dev.w3.org/csswg/selectors-4/#relational`. This could be supported with an extended version of mirror that doesn't only go upwards.

Prolyfill

Have not been done. Should it? Perhaps remove this as it could be out of scope. Low priority.

Chapter 5

Implementation

Perhaps merge this chapter with Library design (and rename it to just The library or something like that?

Write about the simple cycle detector.

Write about the N layouts, and how it was fixed with the batch processing. Investigate how this behaves for nested elements.

Write about the drawbacks (invalid layout at page load, injecting objects is heavy, DOM is mutated.)

5.1 Element resize detection

↪ *As described in section 4.2, ELQ needs a subsystem that is able to detect when elements change size. As the element resize detection system is a core subsystem of ELQ, and extensively used by the* `elq-breakpoints` *plugin, it is important to find a both stable and performant approach to detect element size changes. This section is mainly based on* [5, 65].

Explain target element

Unfortunately, there is no standardized resize event for arbitrary elements. Only frames (elements that render it's content independently of the container element, such as `iframe` and `object`) support the resize event by standard since they have their own viewport. Legacy versions of Internet Explorer (version 8 and down, but also some later versions depending on the document mode and if the browser is in quirks mode) do support the resize event on arbitrary elements. Since the library needs to support commonly used HTML elements, a solution to observe resize events of any element is desired. According to the common use case, a valid limitation is to only support element resize detection for non-void elements (i.e., elements that can contain content).

Write more about void elements, that they can easily be wrapped with other elements to achieve the desired element query effect?

Possible solutions are:

1. **Polling**: To have a script running asynchronously to check all elements if they have resized. This can be achieved by using the JavaScript `setInterval`[1] function. Short polling intervals (high frequency) would lead to being able to detect resize changes quicker but having worse performance. Longer polling intervals (low frequency) would not be able to detect changes as quickly but increases the performance. The longest possible delay between an actual resize event and the polling detecting is given by the polling interval time. This approach is the most robust, as it supports all elements (including void elements) and provides excellent browser compatibility.

2. **Injecting frames**: Since frame elements are the only ones that support resize events natively, the idea is to inject frame elements as children to the target elements.. By styling the frame so that it depends on the target element and set the size to be 100 percent, the frame will always be the same size as the target element. Then a resize event handler can be attached to the frame that emits a resize event for every target element resize. So in order to observe resize events of an element, a frame element is injected and observed instead.

3. **Injecting overflowing elements**: Similar to the frames approach, an element is injected to the target element that emit events, which can be observed in JavaScript, when the target element is resized. Multiple overflowing elements are configured, and injcected as children, so that they emit scroll events[2] when the target element resize. For detecting when the target element shrinks, two elements are needed; one for handling the scrollbars and one for triggering them. Similarly, for detecting when the target element expands, two element are needed in the same way. A container element is injected, to contain the four elements and make sure that they match the size of the target element.

Solution 1 is appealing because it does not mutate the DOM, supports all elements (including void elements), and it provides excellent browser compatibility since it does not rely on special element behaviors or similar. However, in order to prevent the responsive elements lagging behind the size changes of the user interface, the polls need to be performed quite frequently. Recall from section 2.2.5 that layout engines typically have a layout queue in order

[1] See `http://www.w3.org/html/wg/drafts/html/master/webappapis.html#timers` for more information about JavaScript timer functions.

[2] See `http://www.w3.org/TR/DOM-Level-3-Events/#event-type-scroll` for more information about the scroll event.

to perform layout in batches for increased performance. Each poll would force the layout queue to be flushed since the computed style of elements needs to be retrieved in order to know if elements have resized or not. Also, since the polling is performed all the time, the overall page performance will be decreased even if the page is idle which is undesired especially for devices running on battery.

Should make quick test to see how much CPU it uses when polling frequently

Solution 2 and 3 have similar characteristics. Both solutions mutate the DOM and relies on special element behavior, but they offer many advantages over polling. Since they are event based, layout engines only need to perform extra work when installing the solution to the target elements and when the target elements actually resize. Event based resize detection also implies minimal delay between the actual resize event and the detection. By positioning the injected elements `absolute` with width and height set to `100%`, the injected elements do not affect the visual representation of the document. However, injecting elements into the target elements has some implications:

- **CSS selectors may break**: Since target elements get an extra child, CSS selectors may behave differently. For instance, the selector `#target *` selector also matches the injected elements which may result in the injected element being styled in a way that it will be visible or in other ways interfere with the original user interface. Especially solution 3 is sensitive to unintentional styles being applied, as the injected elements are finely styled and tuned in order to behave as desired. Additionally, selectors such as `:first-child` and `:last-child` will behave differently since the target element have an additional child injected.

- **JavaScript may break**: Similar to with the CSS selectors, JavaScript DOM selectors may break. The first node (or last) of the target element may not be what developers expect it to be, since an element has been injected. Also, code that alter the content of elements (such as `element.InnerHTML = ...`) may undesirably remove the injected element of the target element.

- **The target element must be positioned**: Recall that absolute positioned elements are moved up to the first non-static positioned ancestor. Since the injected element must be a child of the target element (and not moved upwards in the layout tree), the target element cannot be positioned `static` that is the default positioning for many elements. Fortunately, elements positioned `relative` behave exactly like `static`, given that the elements do not have any styles applicable to relative elements (such as `top` or `bottom`). Since the style properties that depend on the element being `relative` positioned do not affect the element if it

Recall from where? This have not been described.

is `static` positioned, the properties can be removed or regarded as developer errors. This way the target element can be changed to `relative` positioning with the special style properties removed, to obtain the same visual representation as being positioned `static`.

For solution 2, it has been shown by [5] that `object` is the most suitable frame element to be used for this purpose as they have good browser compatibility and adequate performance. One disadvantage with this solution is that installing `object` elements is quite a heavy task. See figure 5.1 for graphs that show the performance of the object-based element resize detector provided by [5]. As shown by the graphs, the object solution can be installed with adequate performance as long as the number of elements is low. The solution does not scale well as the number of element increases.

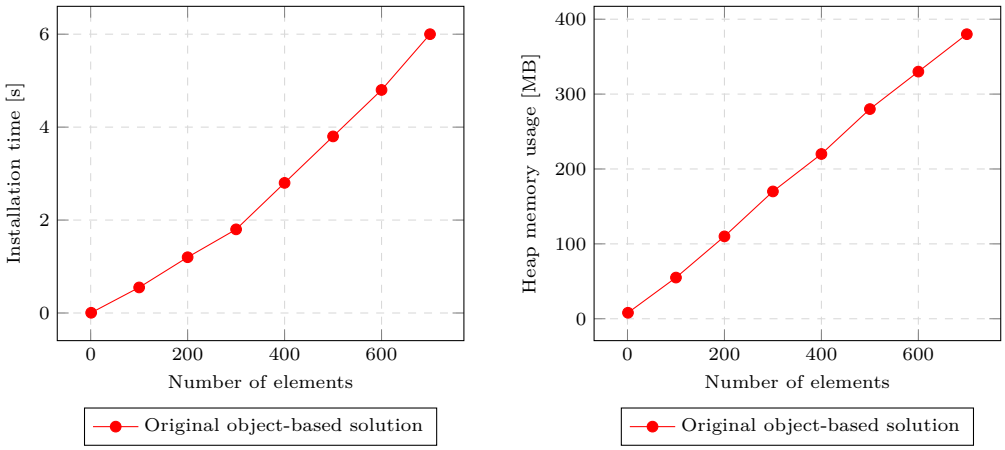

Figure 5.1. The installation performance of the original object-based element resize detection solution provided by [5]. The left graph shows the installation time. The right graph shows the heap memory used when all `object` elements have been installed.

Before the newer object solution was presented, the scroll-based solution 3 was described by [5]. However as the object solution was discovered, solution 3 was rejected in favor of the new solution. Fortunately, a somewhat reworked version of the scroll-based solution is still available [65]. See figure 5.2 for graphs that show how the scroll-based solution performs compared to the object-based solution. It is clear that the scroll-based solution both performs and scales better than the object-based solution. The memory footprint is reduced significantly, which improves the performance further. The amount of memory used by the scroll-based solution is so low that reliable measurements could not be gathered, as the number of elements was not high enough to affect the memory usage.

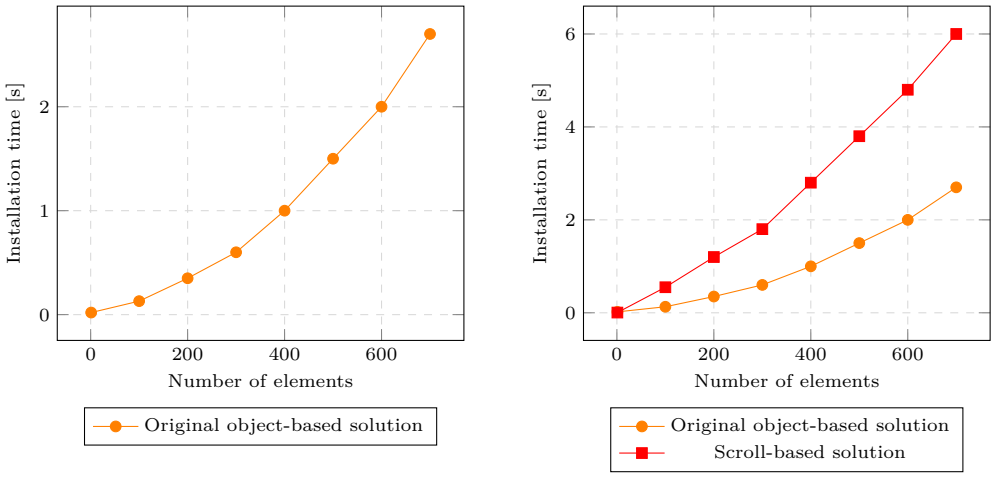

Figure 5.2. The installation performance of the original scroll-based element resize detection solution provided by [65] and originally created by [5]. The left graph shows the installation time of the scroll-based solution. The right graph also includes the original object-based solution for reference. The heap memory usage graph has been omitted as the memory usage for the scroll-based solutions are near constant.

Due to problems such as layout thrashing, bugs and an undesired API; the scroll-based solution provided [65] was completely rewritten to better fit ELQ. Layout thrashing was avoided by using the leveled batch processor described in section 4.3.2. See figure 5.3 for graphs that show how the ELQ scroll-based solution performs compared to the other solutions. As evident in the graph, the optimized ELQ solution has significantly reduced installation times compared to the other two. The ELQ solution also scales better, as more clearly shown in figure 5.4 that includes polynomial regression graphs for all three solutions. Both scroll-based solutions have the same memory footprint (i.e., too low for reliable measurements).

Describe the leveled batch processor in detail somewhere?

Describe how stuff was batch processed in detail?

Write that object was probably preferred over scroll since scroll have some shortcomings? Which have been fixed by ELQ.

Mention that both the object and scroll solutions were completely rewritten to fit in the ELQ framework?

Mention that the scroll solution had severe bugs that were fixed? the display:none or 0 length for instance.

Write that measurements have been done in Chrome?

Tests have shown that my own and backalleys object approaches perform the same. I thought the batch processor would boost it.

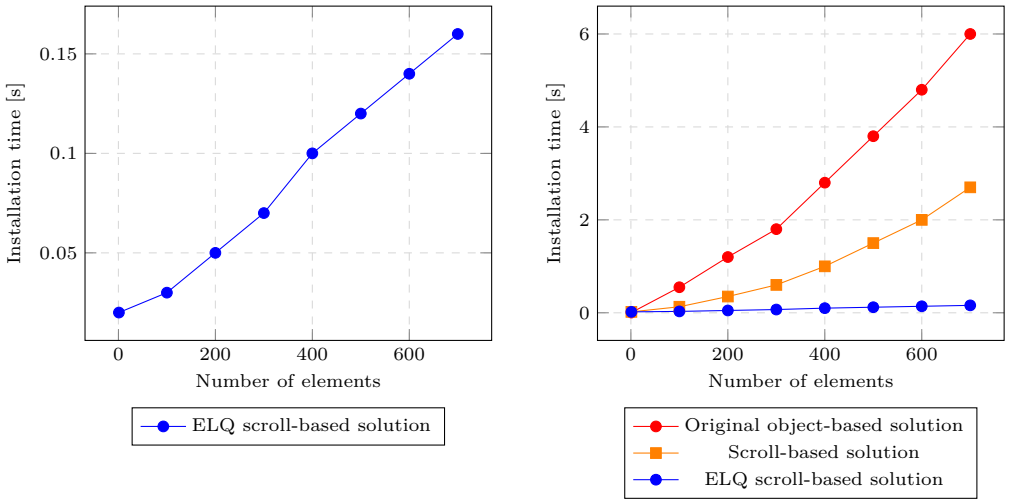

Figure 5.3. The installation performance of the optimized ELQ scroll-based element resize detection. The left graph shows the installation time with the original scroll-based solution graph included as reference. The right graph shows all three solutions for comparison. The heap memory usage graph has been omitted as the memory usage for the scroll-based solutions are near constant.

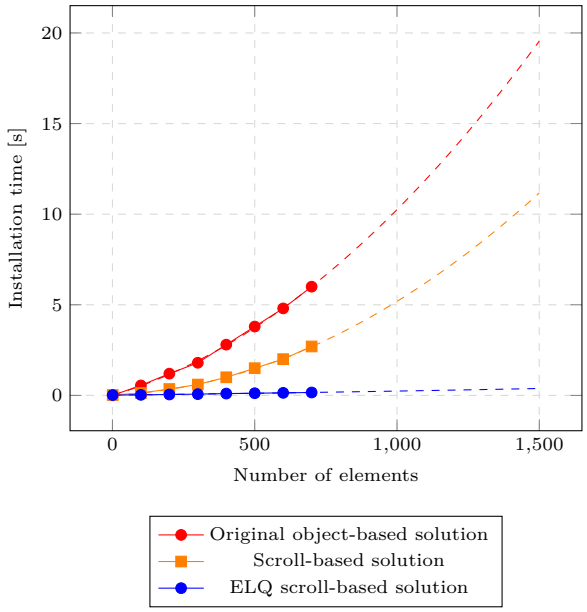

Figure 5.4. The installation performance of all three solutions including graph predicitions by polynomial regression.

Write that it might be up to the user to decide if solution 1 or 3 is desired? Or is it always better to use 3?

Write that object is faster than scroll when it comes to the actual resize events?

5.2 Detecting runtime cycles

↪ *As described in section 4.2, ELQ needs a subsystem that is able to detect cyclic element style state updates, in order to warn about or prevent cyclic rules. As the cycle detection system is a core subsystem of ELQ, and extensively used by the* `elq-breakpoints` *plugin, it is important to find a both stable and performant approach to detect cyclic updates.*

Recall from Section 4.3.2 that the cycle detection subsystem has a function `isUpdateCyclic` that is to tell if the desired update is part of a cycle or not. Further, recall from Section 3.1.2 that cycles may depend on multiple factors (e.g., content, browser behavior, CSS, JavaScript). Since a cycle graph can be both spread over different factors and chained; a simplistic approach to detecting cycles has been taken. The idea is to keep track of all style state changes to elements in order to tell if new state changes are cyclic by examining the state history. This approach is simple to implement and quite sufficient for detecting and breaking cycles. It should be noted that all state cycles are not undesired, as some applications might have features that result in elements transitioning between multiple style states (e.g., a menu might be hidden and revealed by user input, which to the cycle detection system seems like the menu element suffers from cyclic rules). This is perhaps the biggest drawback with such simple approach; a more intelligent cycle detection algorithm might be able to distinguish between desired and undesired cycles. To mitigate the false positive detections, two parameters can be tuned by the user of the system: the time parameter T that defines how long the history should be, and the parameter C that defined how many cycles should be allowed per element before flagging it as a cycle to the user.

Recall that the `isUpdateCyclic` function requires an element e parameter and state s parameter. For each element e, an chronologically ordered list of states L_{states} is kept. For each call of `isUpdateCyclic`, the following cycle detection algorithm is performed:

1. Construct L_{states} if needed.
2. Constuct an update tuple u that consists of the current time t and the new state s.
3. Set the cycle counter c to zero.

4. Iterate L_{states} from beginning to end (starting with the most recent update tuple).

   a) If the time t_i (of the current update tuple u_i of L_{states}) and t differs more the time parameter T, then u_i is regarded as too old to consider. Since L_{states} is chronologically ordered all update tuples after u_i must also be too old. Therefore, all tuples from (and including) u_i are removed from L_{states}. Next step is 5.

   b) If the state s_i (of the current update tuple u_i of L_{states}) and s are equal, then increment the cycle counter c.

   c) If c is greater than the parameter C, then a cycle has been detected. The function should then **return true**.

   d) The next update tuple in the list is considered. Next step is 4a.

5. Prepend u to L_{states} (i.e., u will be the most recent element in the list).

6. No cycle has been detected, and therefore the function should **return false**.

Chapter 6

State of the Art

Write meta.

Write about media query CSS approach as an approach?

There are numerous implementations of third-party element query libraries. This chapter will list and analyze them. Since many of them share the same characteristics, they will be classified and each class will be analyzed as a group. Identified element query third-party libraries that directly or indirectly enables element queries are the ones presented in table 6.1. They can be divided into the following classes:

- **Valid syntax**: Approaches that conforms to the CSS, HTML, and JavaScript standards.
- **Invalid syntax**: Approaches that requires custom (invalid) syntax or in any other way do not conform to the standards described above.
- **Static**: Approaches that in a compilation step calculates desired element queries and transforms them into valid CSS with an element query runtime or media queries. These approaches do not support layout changes at runtime.
- **Dynamic**: Approaches that handles element queries at runtime (and therefore support pages that changes dynamically).
- **Automatic**: Approaches that updates affected element queries on viewport and element resize events automatically.
- **Semi-automatic**: Approaches that partly updates affected element queries. Usually only viewport resize events are handled automatically.

It should be noted that most of the current approaches are combinations of different classes. See table 6.1 for a classification of current approaches.

Not really happy with the dynamic vs static. Need to define this exactly. Is it dynamic because CSS is parsed at runtime or does it need a

Implementation		Syntax	Automation	Behavior	Comments
[19]	MagicHTML	Invalid	Semi-automatic	Static	Compiles invalid CSS to valid CSS at server side.
[34]	EQCSS	Invalid	Semi-automatic	Dynamic	
[60]	Element Media Queries	Invalid	Automatic	Dynamic	
[1]	Localised CSS	Invalid	Automatic	Dynamic	Unfinished. Polling.
[17]	Grid Style Sheets 2.0	Invalid	Automatic	Dynamic	Limited browser support.
[70]	Class Query	Valid	-	Static	Writes media queries to style on load.
[69]	breakpoints.js	Valid	Semi-Automatic	Dynamic	JS-centric approach.
[54]	MediaClass	Valid	Semi-automatic	Dynamic	Custom inline JS syntax.
[48]	ElementQuery	Valid	Semi-automatic	Dynamic	
[37]	Responsive Elements	Valid	Semi-Automatic	Dynamic	
[56]	SickleS	Valid	Semi-automatic	Dynamic	Part of bootstrap-like library. No JS API, instead parses CSS.
[79]	Responsive Elements	Valid	Semi-automatic	Dynamic	
[27]	breaks2000	Valid	Semi-automatic	Dynamic	
[36]	Selector queries and responsive containers	Valid	Semi-automatic	Dynamic	
[7]	Element Queries	Valid	Automatic	Dynamic	
[63]	eq.js	Valid	Automatic	Dynamic	
[65]	CSS Element Queries	Valid	Automatic	Dynamic	Parses CSS.

Table 6.1. Classification of current approaches.

The approaches [19, 34, 60, 1, 17] have in common that they require developers to write cunstom (invalid) CSS. Since they do no longer conform the CSS specification, there are no longer any limitations to what can be done with CSS. As shown by [34, 17] quite advanced features can be implemented this way. Additionally, teaching CSS new tricks implies that it is possible to implement an solution to element queries that does not require any changes to the HTML, which can be preferable since all styling then can be written in CSS (which is the purpose of CSS). However, there are numerous flaws to the approaches that requires invalid CSS. First, it requires a compilation step in order to produce valid CSS that layout engines understand. This can either be done at server side or in the browser at runtime. The advantage of having the compilation step at server side is increased performance since the browser will understand the CSS directly. However, having compiled the CSS at server side the the layout of the page cannot be changed at runtime. By instead having the compilations step at runtime, dynamic layouts are possible. The performance impact of having the compilation step at runtime can be significant for the following reasons:

- The received CSS cannot be understood by the browser, and therefore all parsing and reasoning about it has to be postponed until the library script executes. Note that the layout engines could in theory parse the CSS and perform speculative selector matching while waiting for other parts of the page to finish (such as executing scripts), which cannot be done with invalid CSS.
- When the library script executes, it has to parse the custom CSS, a

process which is most likely slower than native parsing.

- When parsed, the library needs to apply it's logic to the parsed CSS and produce valid CSS. The produced CSS needs to be applied to the document by mutating the DOM.
- The layout engine will need to parse the CSS added to the DOM, which means that the styles of the page has been parsed twice.
- The custom parser engine will need to be included in the page, which implies a bigger library script size.

Write about reentrant HTML and layout thrashing etc?

Additional problems with custom CSS, that also applies to custom CSS compiled at server side, are:

- By not conforming to CSS standards many tools such as preprocessors, validators and linters will no longer be compatible. Also, editors and other code-displaying tools will not be able to understand the syntax and will therefore not be able to highlight the code properly. It should be noted that it is in some cases possible to create plugins or similar to such tools to extend their capabilities.
- The API and parser will need to be kept up to date with CSS standards in order to make sure new features of CSS are supported and that they are not conflicting with the custom API.
- Write that scripts are not allowed to access content of CSS server from another domain than the script?
- Problems with documentation, resources and tutorials since the syntax is custom?

6.1 GSS

6.2 CSS Element Queries

6.3 eq.js

Insight: Custom CSS: If a page has normal CSS present and performs getComputedStyle(), the script will be blocked until the CSS has been resolved so that the valid styles will be returned. If invalid CSS is present (a la Tommy's approach), then does the script wait or does it result in the wrong styles being returned? Probably.

Insight: Is it true that if a approach does not listen to element resizes and does not support layout changes, a static approach is always best?

6.4 Current implementations

Write about GSS, Tommy's implementation and the different approaches

Write somewhere that an extra layout on elements resize is inevitable since it is required in theory.

Chapter 7

Evaluation

7.1 Bootstrap

7.2 Performance

Chapter 8

Discussion and Conclusions

8.1 Future work

Should this be here?

Prolyfill?

Save all elq element class states in localStorage so that on page reload they can be read in order to apply the right classes to all elq elements to avoid flash of invalid layout.

Evaluate CSS selectors on elq elements before injecting elements so that warnings or errors can be produced on selectors that conflict with the injected elements.

For static layouts it would be beneficial to produce media queries for the breakpoints (at server side if possible or at client side)

Bibliography

[1] Chris Ashton. *Implementation: "Localised CSS"*. URL: https://github.com/ChrisBAshton/localised-css (visited on 04/29/2015).

[2] Chris Ashton. *Localised CSS*. URL: http://ashton.codes/blog/localised-css/ (visited on 05/01/2015).

[3] *Blink (layout engine)*. Feb. 2015. URL: http://en.wikipedia.org/wiki/Blink_(layout_engine) (visited on 03/02/2015).

[4] Bert Bos et al. "Cascading Style Sheets, level 2 revision 1 CSS 2.1 Specification". In: *W3C working draft, W3C, June* (2005).

[5] Daniel Buchner. *Backalleycoder*. URL: http://www.backalleycoder.com/ (visited on 03/23/2015).

[6] Daniel Buchner. *Everybody's looking for Element Queries*. Apr. 2014. URL: http://www.backalleycoder.com/2014/04/18/element-queries-from-the-feet-up/#more-139 (visited on 05/01/2015).

[7] Daniel Buchner. *Implementation: "Element Queries"*. URL: https://github.com/csuwildcat/element-queries (visited on 04/29/2015).

[8] *Cascading Style Sheets*. Feb. 2015. URL: http://en.wikipedia.org/wiki/Cascading_Style_Sheets (visited on 03/03/2015).

[9] Calin Cascaval et al. "ZOOMM: a parallel web browser engine for multicore mobile devices". In: *ACM SIGPLAN Notices*. Vol. 48. 8. ACM. 2013, pp. 271–280.

[10] *Common Gateway Interface*. Feb. 2015. URL: http://en.wikipedia.org/wiki/Common_Gateway_Interface (visited on 03/03/2015).

[11] World Wide Web Consortium. *About The World Wide Web*. Jan. 2001. URL: http://www.w3.org/WWW/ (visited on 02/12/2015).

[12] World Wide Web Consortium et al. *CSS Style Attributes*. Nov. 2013. URL: http://www.w3.org/TR/css-style-attr/ (visited on 04/23/2015).

[13] World Wide Web Consortium et al. *Document Object Model (DOM)*. Jan. 2005. URL: http://www.w3.org/TR/DOM/ (visited on 03/05/2015).

[14] World Wide Web Consortium et al. "HTML5 specification". In: *Technical Specification, Jun* 24 (2010), p. 2010.

[15] Chris Coyier. *Thoughts on Media Queries for Elements*. Mar. 2014. URL: https://css-tricks.com/thoughts-media-queries-elements/ (visited on 05/01/2015).

[16] *CSS Containment Draft*. An issue that discusses why a special element viewport element is needed. URL: https://github.com/ResponsiveImagesCG/eq-usecases/issues/7 (visited on 04/29/2015).

[17] et al. Dan Tocchini. *Implementation: "Grid Style Sheets 2.0"*. URL: http://gridstylesheets.org/ (visited on 04/29/2015).

[18] Dave Evans. "The internet of things: How the next evolution of the internet is changing everything". In: *CISCO white paper* 1 (2011).

[19] Gabriel Felipe. *Implementation: "MagicHTML"*. URL: https://github.com/gabriel-felipe/MagicHTML (visited on 04/29/2015).

[20] *Frequently asked questions*. URL: http://www.w3.org/People/Berners-Lee/FAQ.html (visited on 04/23/2015).

[21] Tali Garsiel and Paul Irish. "How Browsers Work: Behind the scenes of modern web browsers". In: *Google Project, August* (2011).

[22] Hugo Giraudel. *WHY ELEMENT QUERIES MATTER*. Apr. 2014. URL: http://hugogiraudel.com/2014/04/22/why-element-queries-matter/ (visited on 05/01/2015).

[23] *Gopher (protocol)*. Feb. 2015. URL: http://en.wikipedia.org/wiki/Gopher_(protocol) (visited on 02/24/2015).

[24] Alan Grosskurth and Michael W Godfrey. "Architecture and evolution of the modern web browser". In: *Preprint submitted to Elsevier Science* (2006).

[25] Responsive Issues Community Group. *Responsive Issues Community Group*. URL: http://ricg.io/ (visited on 04/30/2015).

[26] Responsive Issues Community Group. *Use Cases and Requirements for Element Queries*. Jan. 2015. URL: https://responsiveimagescg.github.io/eq-usecases/ (visited on 04/28/2015).

[27] Daniel Hägglund. *Implementation: "breaks2000"*. URL: https://github.com/judas-christ/breaks2000 (visited on 04/29/2015).

[28] Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press, 2014. ISBN: 978-1593275846.

[29] Rich Hickey. *Simple Made Easy*. Oct. 2011. URL: http://www.infoq.com/presentations/Simple-Made-Easy (visited on 04/21/2015).

[30] David L Hicks et al. "A hypermedia version control framework". In: *ACM Transactions on Information Systems (TOIS)* 16.2 (1998), pp. 127–160.

[31] Matt Hinchliffe. *A proposal for context aware CSS selectors*. Apr. 2013. URL: http://maketea.co.uk/2013/04/11/proposal-context-aware-css-selectors.html (visited on 05/01/2015).

[32] *History of the Internet*. URL: http://www.webdevelopersnotes.com/basics/history_of_the_internet.php3 (visited on 02/24/2015).

[33] *History of the Internet*. Mar. 2009. URL: http://www.historyofthings.com/history-of-the-internet (visited on 02/24/2015).

[34] Tommy Hodgins and Maxime Euzière. *Implementation: "EQCSS"*. URL: http://elementqueries.com/ (visited on 04/29/2015).

[35] *HTML5*. Mar. 2015. URL: http://en.wikipedia.org/wiki/HTML5 (visited on 03/03/2015).

[36] Andy Hume. *Implementation: "Selector queries and responsive containers"*. URL: https://github.com/ahume/selector-queries/ (visited on 04/29/2015).

[37] Kumail Hunaid. *Implementation: "Responsive Elements"*. URL: https://github.com/kumailht/responsive-elements (visited on 04/29/2015).

[38] Christopher Grant Jones et al. "Parallelizing the web browser". In: *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*. 2009.

[39] Tab Atkins Jr. *Element Queries*. Apr. 2013. URL: http://www.xanthir.com/b4PR0 (visited on 05/01/2015).

[40] Jeremy Keith. *HTML5 FOR WEB DESIGNERS*. A Book Apart, 2010. ISBN: 978-0-9844425-0-8.

[41] Jay P Kesan and Rajiv C Shah. "Deconstructing Code". In: *Yale JL & Tech.* 6 (2003), p. 277.

[42] Gary C Kessler. "An overview of TCP/IP protocols and the internet". In: *URL: http://www. hill. com/library/tcpip. html. Last accessed* 17 (1997).

[43] *Konqueror*. Jan. 2015. URL: http://en.wikipedia.org/wiki/Konqueror (visited on 03/02/2015).

[44] Philip A. Laplante. *What Every Engineer Should Know about Software Engineering*. CRC Press, 2007. ISBN: 978-0849372285.

[45] Timothy B. Lee. *40 maps that explain the internet*. June 2014. URL: http://www.vox.com/a/internet-maps (visited on 02/24/2015).

[46] Barry M Leiner et al. "A brief history of the Internet". In: *ACM SIGCOMM Computer Communication Review* 39.5 (2009), pp. 22–31.

[47] Ethan Marcotte. *RESPONSIVE WEB DESIGN*. A Book Apart, 2011. ISBN: 978-0984442577.

[48] Tyson Matanich. *Implementation: "ElementQuery"*. URL: https://github.com/tysonmatanich/elementQuery (visited on 04/29/2015).

[49] Tyson Matanich. *Media Queries Are Not The Answer: Element Query Polyfill*. June 2013. URL: http://www.smashingmagazine.com/2013/06/25/media-queries-are-not-the-answer-element-query-polyfill/ (visited on 05/01/2015).

[50] Leo A Meyerovich and Rastislav Bodik. “Fast and parallel webpage layout”. In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 711–720.

[51] Eivind Hanssen Mjelde. “Performance as design-Techniques for making websites more responsive”. MA thesis. The University of Bergen, 2014.

[52] *Mozilla*. Feb. 2015. URL: `http://en.wikipedia.org/wiki/Mozilla` (visited on 03/02/2015).

[53] Jonathan Neal. *Element Queries*. May 2014. URL: `http://discourse.specifiction.org/t/element-queries/26` (visited on 05/01/2015).

[54] Jonathan Neal. *Implementation: "MediaClass"*. URL: `https://github.com/jonathantneal/MediaClass` (visited on 04/29/2015).

[55] Jonathan T. Neal. *Thoughts on Media Queries for Elements*. Feb. 2013. URL: `http://www.jonathantneal.com/blog/thoughts-on-media-queries-for-elements/` (visited on 05/01/2015).

[56] Truong Nguyen. *Implementation: "SickleS"*. URL: `http://singggum3b.github.io/SickleS/` (visited on 04/29/2015).

[57] *OED Online*. Dec. 2014. URL: `http://www.oed.com/` (visited on 02/12/2015).

[58] *Opera (web browser)*. Feb. 2015. URL: `http://en.wikipedia.org/wiki/Opera_(web_browser)` (visited on 03/02/2015).

[59] François Remy. *Element Media Queries (:min-width)*. Apr. 2013. URL: `http://fremycompany.com/BG/2013/Element-Media-Queries-min-width-883/` (visited on 05/01/2015).

[60] François Remy. *Implementation: "Element Media Queries"*. URL: `https://github.com/FremyCompany/prollyfill-min-width/` (visited on 04/29/2015).

[61] *Responsive web design*. Feb. 2015. URL: `http://en.wikipedia.org/wiki/Responsive_web_design` (visited on 03/03/2015).

[62] *RICG IRC log*. The log where the element query limitations were discussed. URL: `http://ircbot.responsiveimages.org/bot/log/respimg/2015-03-05#T117108` (visited on 04/29/2015).

[63] Sam Richard. *Implementation: "eq.js"*. URL: `https://github.com/Snugug/eq.js` (visited on 04/29/2015).

[64] *Safari (web browser)*. Feb. 2015. URL: `http://en.wikipedia.org/wiki/Safari_(web_browser)` (visited on 03/02/2015).

[65] Marc J. Schmidt. *Implementation: "CSS Element Queries"*. URL: `https://github.com/marcj/css-element-queries` (visited on 04/29/2015).

[66] *Servo repository wiki*. Feb. 2015. URL: `https://github.com/servo/servo/wiki` (visited on 03/17/2015).

[67] Eric Sink. *Memoirs from the browser wars*. 2003. URL: `http://ericsink.com/Browser_Wars.html` (visited on 04/23/2015).

[68] internet live stats. *Internet Users*. Feb. 2015. URL: http://www.internetlivestats.com/internet-users/ (visited on 02/12/2015).

[69] Joshua Stoutenburg. *Implementation: "breakpoints.js"*. URL: https://github.com/reusables/breakpoints.js (visited on 04/29/2015).

[70] Matt Stow. *Implementation: "Class Query"*. URL: https://github.com/stowball/Class-Query (visited on 04/29/2015).

[71] Ian Storm Taylor. *Media Queries are a Hack*. Apr. 2013. URL: http://ianstormtaylor.com/media-queries-are-a-hack/ (visited on 05/01/2015).

[72] *The World Wide Web (WWW) basics and fundamentals*. URL: http://www.webdevelopersnotes.com/basics/the_world_wide_web.php3 (visited on 02/24/2015).

[73] *W3C public mail archive*. Mail thread subject: The :min-width/:max-width pseudo-classes. URL: https://lists.w3.org/Archives/Public/www-style/2013Mar/0368.html (visited on 04/28/2015).

[74] Patrick Walton. *Revamped Parallel Layout in Servo*. Feb. 2014. URL: http://pcwalton.github.io/blog/2014/02/25/revamped-parallel-layout-in-servo/ (visited on 03/17/2015).

[75] *Web development*. Feb. 2015. URL: http://en.wikipedia.org/wiki/Web_development (visited on 03/03/2015).

[76] *WebKit*. Feb. 2015. URL: http://en.wikipedia.org/wiki/WebKit (visited on 03/02/2015).

[77] *Working around a lack of element queries*. June 2013. URL: http://www.filamentgroup.com/lab/element-query-workarounds.html (visited on 05/01/2015).

[78] *World Wide Web*. Feb. 2015. URL: http://en.wikipedia.org/wiki/World_Wide_Web (visited on 02/24/2015).

[79] Corey Worrell. *Implementation: "Responsive Elements"*. URL: https://github.com/coreyworrell/responsive-elements (visited on 04/29/2015).

[80] *XMLHttpRequest*. Dec. 2014. URL: http://en.wikipedia.org/wiki/XMLHttpRequest (visited on 03/03/2015).

Glossary

batch processing Refers to when multiple instructions are processed together. Batch processing is desired when there is an overhead associated with preparing the system prior to execution. With batch processing, the system can be prepared once for all instructions instead. 19, 21, 41, 44, 52, 74

browser Client application for navigating the World Wide Web (WWW) and displaying web pages. More formally known as a user agent. 2, 3, 7, 8, 9, 10, 13, 14, 15, 17, 18, 20, 27, 30, 32, 38, 40, 41, 51, 52, 54, 74, 75, 79, 80, 82, 83, 84, 85, 87, 88

CSS3 The third revision of the CSS standard. 9, 11, 47

document Strictly defined as something that has a URL and can return representations of the identified resource in response to HyperText Transfer Protocol (HTTP) requests. If otherwise states, document will refer to HTML web pages in this thesis. In JavaScript, document refers to the DOM root. 7, 8, 9, 11, 13, 15, 16, 17, 19, 32, 33, 35, 51, 53, 73, 74, 75, 79, 80, 82, 83, 84

element HTML documents consists of elements, which can be regarded as the building blocks of web pages. A web page describes a tree structure of elements and text. 2, 3, 11, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 27, 29, 30, 31, 32, 33, 34, 35, 38, 40, 41, 43, 51, 52, 53, 54

encapsulated An encapsulated module handles its task without any help from the user of the module. 3, 12, 27

fork When developers copy the source code of a project to start independent development on it creating a separate piece of software. The new code is often rebranded to avoid confusion. Common in the open source community. 15, 75, 85

HTML5 The fifth revision of the HTML standard. 9

hypertext Documents with text and media with links (hyperlinks) to other documents immediately accessible for the user. Often used as a synonym for hypermedia. 7, 11, 13, 80, 82, 83, 84

JavaScript Native browser script language. Web documents often include JavaScript to make the documents more dynamic and interactable. 9, 11, 13, 14, 15, 21, 27, 31, 35, 38, 52, 53, 57, 59, 73, 74, 77, 80

layout thrashing When the layout engine is forced to flush the incremental layout command queue repeatedly due to JavaScript, forcing each incremental layout to be processed separately when they could in theory be processed in a batch. 21, 41

layout engine The part of the browser that handles parsing, laying out and rendering web content. Layout engines are often open source and browsers usually acts as a shell on top of a layout engine. Sometimes also called rendering engines. 4, 13, 14, 15, 16, 19, 20, 21, 27, 29, 30, 34, 35, 52, 74, 75, 84, 85, 87

media queries The CSS feature of specifying conditional style rules for elements by conditions such as the viewport size. 2, 3, 11, 27, 28, 27

native Refers to APIs and systems implemented and provided by browsers. Such APIs and systems are often described by standards specifications. 2, 3, 4, 27, 28, 33, 35, 38, 52, 74

render tree Used by layout engines as the model for the visual representation of the document. 14, 16, 17, 19

responsive Elements and content of the document can detect size changes and act accordingly. Usually a restructure of content is performed at certain breakpoints. 2, 3, 4, 7, 11, 27, 28, 27, 52

specificity Is the means by which layout engines decides which CSS rule property values are the most relevant to an element, which is based only on the form of the selector. See `http://www.w3.org/TR/CSS2/cascade.html#specificity` for more information. 18

StatCounter Collects and aggregates on a sample exceeding 15 billion page views per month collected from across the StatCounter network of more than 3 million websites. 10

third-party Refers to APIs and systems implemented on top of the browser, usually in JavaScript. Such APIs and systems must be included by developers into the document, and are usually not described by standards specifications. 2, 3, 4, 9, 35

viewport The outer frame that defines the visible area of the document. Usually defined by the browser window, but may be restricted by other factors such as frames. 2, 3, 10, 11, 19, 27, 28, 33, 34, 35, 51

web Short for the world wide web. 1, 2, 3, 4, 7, 8, 9, 10, 11, 20, 27, 38, 73, 74, 79, 80, 82, 83, 84, 87

WebKit The open source layout engine used by Safari. Google's Blink layout engine is a recent fork of WebKit. 15, 85, 87

WYSIWYG A classification that ensures that text and graphics during editing appears close to the result. 78, 84

Acronyms

AJAX Asynchronous JavaScript and XML. 9

API Application Program Interface. 2, 3, 4, 9, 13, 19, 28, 29, 34, 37, 38, 39, 41, 42, 43, 44, 45, 54, 61, 74

ARPANET Advanced Research Projects Agency Network. 80, 81

CERN Conseil Europeen pour la Recherche Nucleaire. 82, 83

CGI Common Gateway Interface. 8

CPU Central Processing Unit. 20

CSNET Computer Science Network. 81

CSS Cascading Style Sheets. 2, 3, 7, 11, 14, 15, 16, 17, 18, 21, 27, 32, 33, 34, 38, 39, 53, 57, 59, 60, 61, 73, 74, 80

DARPA Defense Advanced Research Projects Agency. 83

DHTML Dynamic HTML. 8

DOD Department Of Defense. 80

DOM Document Object Model. 14, 15, 16, 17, 18, 21, 22, 23, 24, 41, 52, 53, 61, 73

FTP File Transfer Protocol. 80, 82, 83

GUI Graphical User Interface. 9, 84

HTML HyperText Markup Language. 7, 8, 9, 11, 13, 14, 15, 19, 27, 33, 34, 38, 45, 46, 51, 59, 73, 77, 80, 83

HTTP HyperText Transfer Protocol. 73, 80, 83, 84

ICANN Internet Corporation for Assigned Names and Numbers. 79

ICCC International Computer Communication Conference. 80

IP Internet Protocol. 79, 80, 81

ISP Internet Service Provider. 79

KDE K Desktop Environment. 85

MILNET Military Network. 81

NCSA National Center for Supercomputing Applications. 8, 84

NSF National Science Foundation. 81

NSFNET National Science Foundation Network. 81, 82

OS Operating System. 8

RICG Responsive Issues Community Group. 4, 33

RWD Responsive Web Design. 11

TCP Transmission Control Protocol. 79, 80, 81

URL Uniform Resource Locator. 13, 73

US United States. 80, 81

W3C World Wide Web Consortium. 2, 4, 7, 9, 33, 83

WHATWG Web Hypertext Application Technology Working Group. 9

WWW World Wide Web. 73, 79, 80

WYSIWYG What You See Is What You Get. *Glossary:* WYSIWYG, 84

Appendix A

History of the Internet and browsers

↪ *Browsers and the Internet is something that many people today take for granted. It is not longer the case that only computer scientists are browsing the web. Today the web is becoming increasingly important in both our personal and professional lives. This chapter will give a brief history of browsers and how the web transitioned from handling science documents to commercial applications. This section is a summary of* [68, 18, 30, 11, 57].

Change all wikipedia sources to the "real" sources?

Before addressing the birth of the web, it is necessary to define the meaning of the *Internet* and the *WWW*. The word "internet" can be translated to *something between networks*. When referring to the Internet (capitalized) it is usually the global decentralized internet used for communication between millions of networks using the Transmission Control Protocol (TCP) and Internet Protocol (IP) suite. Since the Internet is decentralized, there is no single owner of the network. In other words, the owners are all the network end-points (all users of the Internet). One can argue that the owners of the Internet are the Internet Service Providers (ISPs), providing the services and infrastructure making the Internet possible. On the other hand, the backbones of the Internet are usually co-founded nationally. Also, it is the Internet Corporation for Assigned Names and Numbers (ICANN) organization that has the responsibility for managing the IP addresses in the Internet namespace, which reduces the ownership of the ISPs further. Clearly, the Internet wouldn't be what it is today without all the actors. The Internet lays the ground for many systems and applications, including the WWW, file sharing and telephony. In 2014 the number of Internet users was measured to just below 3 billions, and estimations show that we have surpassed 3 billions users today (no report for 2015 has been published yet). Users are defined as humans having access to the

Marcos: s/co-founded/controlled ?

Internet at home. If one instead measures the number of connected entities (electronic devices that communicates through the Internet) the numbers are much higher. An estimation for 2015 of 25 billions connected entities has been made, and the estimation for 2020 is 50 billion.

As already stated, the WWW is a system that operates on top of the Internet. The WWW is usually shortened to simply *the web*. The web is an information space that interoperates through standardized protocols and standards, which affords users with the ability to access various types of resources. This can include interlinked hypertext documents, which themselves can contain other media such as images and videos, and/or data services. Since not only hypertext is interlinked on the web, the term *hypermedia* can be used as an extension to hypertext that also includes other nonlinear medium of information. Although the term hypermedia has been around for a long time, the term hypertext is still being used as a synonym for hypermedia. Further, the web can also be referred to as the universe of information accessible through the web system. Therefore, the web is both the system enabling sharing of hypermedia and also all of the accessible hypermedia itself. Hypertext documents are today more known by the name *web pages* or simply *pages*. Multiple related pages (that are often served from the same domain) compose a *web site* or simply a *site*. Hypertext documents are written in HTML, and often includes CSS for styling and JavaScript for custom user interactions. To transfer the resources between computers the protocol HTTP is used. Typically the way of retrieving resources on the web is by using a *user agent*, known colloquially as a web *browser*. Typically the way of retrieving resources on the web is by using a *web browser* or simply a *browser*. Browsers handle the fetching, parsing and rendering of the hypertext (more about this in section A.3).

Marcos: Maybe drop the term pages since it is a bit outdated?

A.1 The history of the Internet

↪ *Since the web is a system operating on top of the Internet, it is needed to first investigate the history of the Internet. This can be viewed from many angles and different aspects need to be taken into consideration. With that in mind, the origin of the Internet is not something easily pinned down and what will be presented here will be more technically interesting than the exact history. This section is a summary of* [42, 46, 45, 33].

Marcos: This doesn't relate to your thesis at all, TBH - which is about layout problems. I would remove this unless it somehow relates to layout of information.

In the early 1960's *packet switching* was being researched, which is a pre-

requisite of internetworking. With packet switching in place, the very important ancestor of the Internet Advanced Research Projects Agency Network (ARPANET) was developed, which was the first network to implement the TCP/IP suite. The TCP/IP suite together with packet switching are fundamental technologies of the Internet. ARPANET was funded by the United States (US) Department Of Defense (DOD) in order to interconnect their research sites in the US. The first nodes of ARPANET was installed at four major universities in the western US in 1969 and two years later the network spanned the whole country. The first public demonstration of ARPANET was held at the International Computer Communication Conference (ICCC) in 1972. It was also at this time the email system was introduced, which became the largest network application for over a decade. In 1973 the network had international connections to Norway and London via a satellite link. At this time information was exchanged with the File Transfer Protocol (FTP), which is a protocol to transfer files between hosts. This can be viewed as the first generation of the Internet. With around 40 nodes, operating with raw file transfers between the hosts it was mostly used by the academic community of the US.

The number of nodes and hosts of ARPANET increased slowly, mainly due to the fact that it was a centralized network owned and operated by the US military. In 1974 the TCP/IP suite was proposed in order to have a more robust and scalable system for end-to-end network communication. The TCP/IP suite is a key technology for the decentralization of the ARPANET, which allowed the massive expansion of the network that later happened. In 1983 ARPANET switched to the TCP/IP protocols, and the network was split in two. One network was still called ARPANET and was to be used for research and development sites. The other network was called Military Network (MILNET) and was used for military purposes. The decentralization event was a key point and perhaps the birth of the Internet. The Computer Science Network (CSNET) was funded by the National Science Foundation (NSF) in 1981 to allow networking benefits to academic intsitutions that could not directly connect to ARPANET. After the event of decentralizing ARPANET, the two networks were connected among many other networks. In 1985 NSF started the National Science Foundation Network (NSFNET) program to promote advanced research and education networking in the US. To link the supercomputing centers funded by NSF the NSFNET served as a high speed and long distance backbone network. As more networks and sites were linked by the NSFNET network, it became the first backbone of the Internet. In 1992, around 6000 networks were connected to the NSFNET backbone with many international networks. To this point, the Internet was still a network for scientists, academic institutions and technology enthusiasts. Mainly because

NSF had stated that NSFNET was a network for non-commercial traffic only. In 1993 NSF decided to go back to funding research in supercomputing and high-speed communications instead of funding and running the Internet backbone. That, along with an increasing pressure of commercializing the Internet let to another key event in the history of the Internet - the privatization of the NSFNET backbone.

In 1994, the NSFNET was systematically privatized while making sure that no actor owned too much of the backbone in order to create constructive market competition. With the Internet decentralized and privatized regular people started using it as well as companies. Backbones were built across the globe, more international actors and organizations appeared and eventually the Internet as we know it today came to exist.

A.2 The birth of the World Wide Web

↪ *Now that the history of the Internet has been described, it is time to talk about the birth of the web. Here the initial ideas of the web will be described, the alternatives and how it became a global standard. This subsection is a summary of* [23, 78, 32, 72, 33].

Marcos: Same with this. This has been described many times, I would recommend dropping this as it adds little value and doesn't relate to layout problems.

Recall from section A.1 that the way of exchanging information was to upload and download files between clients and hosts with FTP. If a document downloaded was referring to another document, the user had to manually find the server that hosted the other document and download it manually. This was a poor way of digesting information and documents that linked to other resources. In 1989 a proposal for a communication system that allowed interlinked documents was submitted to the management at Conseil Europeen pour la Recherche Nucleaire (CERN). The idea was to allow links to other documents embedded in text documents, directly accesible for users. A quote from the draft:

> Imagine, then, the references in this document all being associated with the network address of the thing to which they referred, so that while reading this document you could skip to them with a click of the mouse.

This catches the whole essence of the web in a sentence — to interlink resources in an user friendly way. The proposal describes that such text embedded links

would be hypertext. It continues to explain that interlinked resources does not need to be limited to text documents since multimedia such as images and videos can also be interlinked which would similarly be hypermedia. The concept of browsers is described, with a client-server model the browser would fetch the hypertext documents, parse them and handle the fetching of all media linked in the hypertext.

In 1990, HTTP and HTML were implemented by Tim Bernes Lee at CERN. A browser and a web server were also created and the web was born. One year later the web was introduced to the public and in 1993 over five hundred international web servers existed. It was stated in 1994 by CERN that the web was to be free without any patents or royalties. At this time the W3C was founded with support from the Defense Advanced Research Projects Agency (DARPA) and the European Commission. The organization comprised of companies and individuals that wanted to standardize and improve the web.

Marcos: Citation needed.

Marcos: citation needed.

As a side note, the Gopher protocol was developed in parallel to the web by the University of Minnesota. It was released in 1991 and quickly gained traction as the web still was in very early stages. The goal of the system, just like the web, was to overcome the shortcomings of browsing documents with FTP. Gopher enabled servers to list the documents present, and also to link to documents on other servers. This created a strong hierarchy between the documents. The listed documents of a server could then be presented as hypertext menus to the client (much like a web browser). As the protocol was simpler than HTTP it was often preferred since it used less network resources. The structure provided by Gopher provided a platform for large electronic library connections. A big difference between the web and the Gopher platform is that the Gopher platform provided hypertext menus presented as a file system while the web hypertext links inside hypertext documents, which provided greater flexibility. When the University of Minnesota announced that it would charge licensing fees for the implementation, users were somewhat scared away. As the web matured, being a more flexible system with more features as well as being totally free it quickly became dominant.

A.3 The history of browsers

↪ *In the mid 1990's the usage of the Internet transitioned from downloading files with FTP to instead access resources with the HTTP protocol. To fulfill the vision that users would be able to skip to the linked documents "with a click of the mouse" users needed a client to handle the fetching and displaying of the hypertext documents, hence the need for browsers were apparent. Here the evolution of the browser clients will be given, while emphasizing the timeline*

of the popular browsers we use today. This section is a summary of [78, 20, 41, 67, 52, 58, 43, 64, 76, 3].

Marcos: This only gets interesting (with relation to the thesis) when you start talking about layout. The rest of the history is not relevant to the thesis.

Marcos: Confusion with what?

Marcos: Developed by who?

The first web browser ever made was created in 1990 and was called WorldWideWeb (which was renamed to Nexus to avoid confusion). It was at the time the only way to view the web, and the browser only worked on NeXT computers. Built with the NeXT framework, it was quite sophisticated. It had a GUI and a What You See Is What You Get (WYSIWYG) hypertext document editor. Unfortunately it couldn't be ported to other platforms, so a new browser called *Line Mode Browser* (LMB) was quickly developed. To ensure compatibility with the earliest computer terminals the browser displayed text, and was operated with text input. Since the browser was operated in the terminal, users could log in to a remote server and use the browser via telnet. In 1993, the core browser code was extracted and rewritten in C to be bundled as a library called *libwww*. The library was licensed as *public domain* to encourage the development of web browsers. Many browsers were developed at this time. The *Arena* browser served as a testbed browser and authoring tool for Unix. The *ViolaWWW* browser was the first to support embedded scriptable objects, stylesheets and tables. *Lynx* is a text-based browser that supports many protocols (including Gopher and HTTP), and is the oldest browser still being used and developed. The list of browsers of this time can be made long.

In 1993, the *Mosaic* browser was released by the NCSA which came to be the ancestor of many of the popular browsers in use today. As Lynx, Mosaic also supported many different protocols. Mosaic quickly became popular, mainly due to its intuitive GUI, reliability, simple installation and Windows compatibility. The company *Spyglass, Inc.* licensed the browser from the NCSA for producing their own browser in 1994. Around the same time the leader of the team that developed Mosaic, Marc Andreessen, left the NCSA to start *Mosaic Communications Corporation*. The company released their own browser named *Mosaic Netscape* in 1994, which later was to be called *Netscape Navigator* that was internally codenamed *Mozilla*. Microsoft licensed the Spyglass Mosaic browser in 1995, modified and renamed it to *Internet Explorer*. In 1997 Microsoft started using their own *Trident* layout engine for Internet Explorer. The Norwegian telecommunications company *Telenor* developed their own browser called *Opera* in 1994, which was released 1996. Internet Explorer and Netscape Navigator were the two main browsers for many years, competing for market dominance. Netscape couldn't keep up with

Microsoft, and was slowly losing market share. In 1998 Netscape started the open source Mozilla project, which made available the source code for their browser. Mozilla was to originally develop a suite of Internet applications, but later switched focus to the *Firefox* browser that had been created in 2002. Firefox uses the *Gecko* layout engine developed by Mozilla.

Another historically important browser is the *Konqueror* browser developed by the free software community K Desktop Environment (KDE). The browser was released in 1998 and was bundled in the KDE Software Compilation. Konqueror used the KHTML layout engine, also developed by KDE. In 2001, when *Apple Inc.* decided to build their own browser to ship with OS X, a fork called WebKit was made of the KHTML project. Apple's browser called *Safari* was released in 2003. The WebKit layout engine was made fully open source in 2005. In 2008, *Google Inc.* also released a browser based on WebKit, named *Chrome*. The majority of the source code for Chrome was open sourced as the *Chromium* project. Google decided in 2013 to create a fork of WebKit called *Blink* for their browser. Opera Software decided in 2013 to base their new version of Opera on the Chromium project, using the Blink fork.

Appendix B

Miscellaneous

B.1 Practical problem formulation document

Should this be in the real document instead of appendix?

B.2 CSS terminology

Write custom or refer to this `http://www.impressivewebs.com/css-terms-definitions/`?

B.3 Layout engine market share statistics

Browser market share was retrieved by *StatCounter*[1]. Since the graph only display browser market share and not layout engine, it is needed to further divide the browsers into layout engine percentages. The Blink engine was introduced with Chrome version 28 and Android version 4.4 [3]. Since Chrome has very good adoption rate[2] of new versions the Chrome market share percentage of 39.72% is considered to be Blink based. However, Android has not as good adoption rate as Chrome with only 44.2% using Android version 4.4 and up[3]. Android has a browser market share of 7.21%. 44.2% of the 7.21% Android browsers is assumed to be Blink based and 55.8% to be WebKit based (since the Android browser was WebKit based before Blink). Of course, the assumption that users with old versions of Android browse the web as much as users with new versions are probably invalid, but the data source itself is uncertain enough to make such assumptions and the percentages should only be regarded as guidelines. Opera with the lowest market

[1] StatCounter graph `http://gs.statcounter.com/#all-browser-ww-monthly-201402-201502-bar`
[2] According to `http://clicky.com/marketshare/global/web-browsers/google-chrome/`
[3] According to `https://developer.android.com/about/dashboards/index.html`

share at 3.97% started using the Blink engine in late 2013 as of version 15. StatCounter shows that 37% of the Opera users are using Opera Mini (their mobile browser), which does not use the Blink engine (it uses Opera's own Presto layout engine which will be ignored). All desktop users of Opera are assumed to be using version 15 or above and hence using the Blink engine. The total market share percentage of the Blink engine is then calculated to $39.72 + 0.442 \cdot 7.21 + 0.37 \cdot 3.97 = 44.38\%$. Safari, with the market share percentage of 7.46%, has always been WebKit based. iOS also uses WebKit and has the market share percentage of 6.16%. The WebKit market share percentage is calculated to $7.46 + 0.558 \cdot 7.21 + 6.16 = 17.64\%$. FireFox, with the market share percentage of 12.83%, has always been Gecko based and is the only major browser that uses the Gecko engine. The market share percentage of Gecko is therefore 12.83%. Internet Explorer, with the market share percentage of 14.96%, has been Trident based since version 4. Since Internet Explorer 4 is no longer in use[4], the market share percentage of the Trident engine is 14.96%.

B.4 Usage share of browser versions

Put this somewhere else?

Have a figure or not? If yes, then reference it here.

For compatability, the library needs to support older browsers. Most of the end users are using a self updating browser (those browsers are usually referred to as the *evergreen* browsers), which improves the adoption rate of new versions greatly. See figure for the adoption rate of new versions of Chrome for an example of how evergreen browsers keep their users updated. Unfortunately, there are some laggards that still use very outdated versions of some browsers. Each version of Internet Explorer 8 up to 11 need to be supported since they all still have a significant user share. Opera version 12 does also have

[4] According to `http://www.w3schools.com/browsers/browsers_explorer.asp`