KTH Computer Science
and Communication

ELQ: extensible element queries for modular responsive web components

Allowing modular responsive web components to respond to custom criteria instead of only the viewport size by implementing an element queries JavaScript library

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

June 2015

Abstract

Responsive web design is a popular approach to support devices with varying characteristics (viewport size, input mechanisms, media type, etc.) by conditionally style the content of a document by such criteria using CSS *media queries*. To reduce complexity it is also popular to develop web applications by creating reusable modules. Unfortunately, responsive modules require the user of a module to define the conditional styles since only the user knows the layout of the module. This implies that responsive modules cannot be encapsulated (i.e., that modules cannot perform their task by themselves), which is important for reusability and reduced complexity. This is due to the limitation of CSS media queries that elements can only be conditionally styled by the document root and device properties. In order to create encapsulated responsive modules, elements must be able to be conditionally styled by element property criteria, which is known as *element queries*.

Participants of the main international standards organization for the web, the W3C, are interested in solving the problem and possible solutions are being discussed. However, they are still at the initial planning stage so a solution will not be implemented natively in the near future. Additionally, implementing element queries imposes circularity and performance problems, which need to be resolved before writing a standard.

This thesis presents the issues that element queries impose to layout engines and shows some approaches to overcome the problems. In order to enable developers to create encapsulated responsive modules, while waiting for native support, a third-party element queries JavaScript library named ELQ has been developed. As presented in this thesis, the library provides both performance and usage advantages to other related libraries. An optimized subsystem for detecting resize events of elements has been developed using a leveled batch processor, which is significantly faster than similar systems. As part of the empirical evaluation of the developed library the Bootstrap framework has been altered to use element queries instead of media queries by altering ~50 out of ~8500 lines of style code, which displays one of the advantages of the library.

Referat

ELQ: utbyggbara element queries för modulära responsiva webbkomponenter

Responsiv webbutveckling är ett populärt sätt att stödja enheter med varierande egenskaper (storlek av visninsområdet, inmatningsmekanismer, mediumtyper, etc.) genom att ange olika stilar för ett dokument beroende på enhetens egenskaper med hjälp av CSS *media queries*. Det är också populärt att utveckla webbapplikationer genom att skapa återanvändbara moduler för minskad komplexitet. Tyvärr kräver responsiva moduler att användaren av en modul definierar de olika responsiva stilarna eftersom endast användaren vet i vilket kontext modulen används. Detta implicerar att responsiva moduler inte är enkapsulerade (alltså att de inte fungerar av sig själva), vilket är viktigt för återanvändning och reduktion av komplexitet. Det beror på CSS media queries begränsningar att det endast går att definiera olika stilar för element beroende på dokumentets rot och enhetens egenskaper. För att kunna skapa enkapsulerade responsiva moduler måste olika stilar kunna definieras för ett element beroende på ett elements egenskaper, vilket är känt som *element queries*.

Deltagare av det internationella industrikonsortiet för webbstandardisering, W3C, är intresserade av att lösa problemet och möjliga lösningar diskuteras. De är dock endast i det initiala planeringsstadiet, så det kommer dröja innan en lösning blir implementerad. Dessutom är det problematiskt att implementera element queries eftersom de medför problem gällande cirkularitet samt prestanda, vilket måste lösas innan en standard skapas.

Denna rapport presenterar de problem för webbläsares renderingsmoterer som element queries medför och visar sätt att övervinna vissa av problemen. För att möjliggöra skapandet av enkapsulerade responsiva moduler, i väntan på webbläsarstöd, har ett tredjepartsbibliotek för element queries namngett ELQ skapats i JavaScript. Biblioteket erbjuder både prestanda- och användningsfördelar jämfört med andra relaterade bibliotek. Ett optimerat delsystem för att detektera förändringar av elements storlekar har utvecklats som använder en nivåuppdelad *batch*-processerare vilket medför att delsystemet erbjuder signifikant bättre prestanda än relaterade system. Som del av den empiriska utvärderingen har det populära ramverket Bootstrap modifierats att använda element queries istället för media queries genom att ändra ~50 utav ~8500 rader stilkod, vilket visar en av fördelarna med det utvecklade biblioteket.

Acknowledgements

I would like to thank my supervisors Tomas Ekholm, Philipp Haller and Stefan Sennerö for their enthusiastic involvement in this thesis. They have provided me with great feedback and support throughout the project.

I would also like to thank Marcos Caceres of Mozilla for the countless mails that have helped me to achieve a deeper understanding of the W3C and the web. I am also very grateful for the invaluable feedback I have received from Marcos.

Last, I would like to specially thank Tomas Ekholm for his extraordinary guidance. Tomas also brought element queries to my attention.

Contents

Chapter 1

Introduction

1.1 Targeted audience

This thesis is targeted for *web* developers that wish to gain a deeper understanding of element queries and how they can be used without *native* support by using the *ELQ* library. 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 page 83 of this thesis. Unfamiliar readers might also want to notice the distinction between the words "user" and "end-user". For instance, an application is a user of modules while the human interacting with the application is an end-user.

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 modules media queries are no longer applicable. Responsive modules should be able to react and change style depending on the size that the module 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 by element property criteria.*

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 [29] 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 developers to implement this feature as a non-native solution. As presented in Chapter 6, efforts have been made to create a robust non-native solution, with

moderate success. It should be noted that no effort of implementing a native solution has been made. Since all current non-native solutions have shortcomings, there is still no de facto solution that developers use and the problem remains unsolved.

1.3 Objective

The main objective of this thesis is to design and develop a *third-party* non-native library that enables element queries in both modern and legacy *browsers*. The scientific question is if it is possible to construct such library that has high reliability, adequate performance, and enough features to support advanced compositions of responsive modules. The question is answered in the affirmative by the developed element queries library introduced in Chapter 4.

Since modules cannot make any assumptions about the usage context, both static and dynamic pages must be supported. Element queries operating in dynamic pages are more complicated, and therefore the focus of this thesis is to enable element queries in dynamic pages. To do so it is necessary to be aware of the premises, such as browser limitations and specifications that need to be conformed. The problems of implementing element queries natively are researched, in order to get a deeper understanding of the potential shape of a standardized Application Program Interface (API). Solving the main problem requires research and empirical studies of subproblems. Examples of such subproblems that are addressed in this thesis are the following:

- Should circularity be handled? If yes, should it be detected at runtime or parsetime, and what should happen on detection?
- How can element size changes be observed without any native support?
- How can a custom API be crafted that enables 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. 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?

1.4 Significance

Many libraries and techniques are being used in web development to keep the code from becoming an entangled mess. Modular development helps reducing complexity 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 criteria. Without element queries, responsive modules force the user to style them properly depending on the viewport size, 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 a rewrite of the media queries of the modules to take the new layout into account. Clearly, no sound option exists for having responsive modules so solving this problem would be a big advancement to web development.

The last couple of years a lot of articles have been written about the problem and how badly web developers need element queries [50, 53, 82, 66, 87, 69, 72, 56, 59, 76, 71, 62]. As already stated in Section 1.2, non-native implementation efforts have been made with moderate success. The W3C receives requests and questions about it, and the Responsive Issues Community Group (RICG) has started writing a draft about element queries use cases [30]. This thesis may be helpful for the W3C and others to get an overview of the problem and possibly get ideas how subproblems can be handled.

With the developed element queries library presented in this thesis, developers are enabled to create truly responsive modules while waiting for the W3C to standardize a solution. The novelty of the library is that it is performant, compatible, flexible, powerful, and extensible. No other related library simultaneously provides all of these desirable properties. The library provides a unique way to overcome some limitations of CSS without requiring currently-invalid syntax or parsing of an extended, incompatible version of CSS. Related approaches that overcome similar limitations require invalid syntax, which brings many drawbacks (e.g., reduced performance and development tools incompatibility). It has also been shown that the library has significantly better performance than related approaches in most configurations. Further, the API of the library makes it easy to integrate into existing projects, which has been shown by altering the *Bootstrap* framework to use element queries instead of media queries by only altering ~50 lines of style code. The library is able to provide such positive properties while still maintaining compatibility with legacy browsers (down to Internet Explorer version 8).

1.5 Delimitation

The focus of this thesis lies on developing a third-party non-native library that enables developers to use element queries. All theoretical studies and work have been performed to support the development of the library. The problems of implementing element queries natively are addressed, but no effort has been made to solve the problems. Also, no API or similar has been designed for a native solution. Graphical design of end-user interfaces are not addressed, other than necessary for understanding the problem.

1.6 Choice of methodology

The results and conclusions of this thesis are based on a scientific study using an iterative development process evaluated by empirical studies. The library design is motivated by identified requirements of element queries, which have been gathered partly by the literature study and studies of existing responsive applications. The decision to write the non-native library in *JavaScript* was made, because JavaScript is the common scripting language of browsers. For increased compatibility and ease of use, the use of other languages compiling to JavaScript was avoided.

The performance and browser compatibility of the library have been evaluated by empirically testing the library in different browsers. Since there are numerous related libraries only the performance of the element resizing detection subsystem of the libraries has been evaluated. There are only three different approaches to such a subsystem, and therefore the work needed to compare the libraries was reduced significantly. This restriction is motivated by the fact that the element resizing detection subsystem performs the heaviest task by far, and therefore has the largest performance impact. Another restriction is that only fully functioning automated related libraries have been evaluated, since others cannot be considered an equivalent alternative to the developed library.

The APIs and features of the library have been evaluated empirically by altering the popular Bootstrap framework to use element queries instead of media queries. It is hard to objectively evaluate features and API designs, but some objective data were gathered by the evaluation. The Bootstrap framework was chosen because it consists of a large open source codebase and is widely used as a foundation of many responsive applications. Only the developed library was evaluated, since it can be concluded by analyzing related libraries that none would be able to achieve equivalent results.

The premises of the library development method are considered valid since a JavaScript library is the natural choice when creating a browser extension that is to be used by a large audience of developers. The premises of the Bootstrap altering method are considered valid since Bootstrap has the properties that are required for such an evaluation to be sound (e.g., it is widely used in production, has a large code base, and has sufficiently advanced responsive features). The premises of the performance evaluation method are considered valid since it is possible to evaluate different systems in the same configuration in order to achieve comparable results.

1.7 Outline

Chapter 2, *Background*, presents the background of this thesis and aims to provide sufficient information about web development and *layout engines* to understand the problem and solution. In Section 2.1 the two web development concepts responsive web design and modularity are presented, and ends by describing the problem of responsive modules. Section 2.2 briefly describes how layout engines operate and

how avoiding *layout thrashing* can increase the performance of JavaScript code. The section ends with presenting the current research about parallelizing layout engines, which is important to understand since element queries affect the parallelization of layout engines. Expert readers may want to skip Chapter 2.

Chapter 3, *Element queries*, defines element queries and shows how they can be used to solve the problem of encapsulated responsive modules. The chapter also gives insight into the current research about element queries and presents some difficulties that the W3C is facing with implementing element queries in layout engines. Section 3.2 describes the problems that element queries imply to layout engines in detail and Section 3.3 presents some approaches discussed by the W3C to overcome some of the problems.

Chapter 4, *The element queries library*, presents the third-party non-native element queries library that enables developers to use element queries today. The chapter identifies possible technical requirements that different use cases may have, which is the main motivation to why the library should be plugin based. The library design and APIs are presented along with two plugins and their APIs. Last, some subsystems of the library are presented in detail including a highly optimized solution to observing element resize events using a leveled batch processor subsystem.

Chapter 5, *Empirical Evaluation*, presents the empirical evaluation of the library that includes altering the popular style framework Bootstrap, and measuring the performance of subsystems in different browsers. The Bootstrap framework provides responsive CSS classes to make user interfaces responsive by using media queries. By altering Bootstrap to use element queries instead for the responsive classes, some of the technical goals can be evaluated. Similarly, by measuring the performance of the library and different subsystems additional technical goals can be evaluated. The performance measurements are also used to compare the performance of the library to related approaches.

Chapter 6, *Related work*, describes related libraries and presents advantages as well as drawbacks to the different approaches. Since many of the libraries presented in this chapter share the same characteristics they are classified. This way, the different classes can be evaluated instead of evaluating all libraries in detail.

Chapter 7, *Discussion*, discusses advantages and drawbacks of non-native solutions, and related libraries are compared to the developed library. The fulfillment of the identified requirements of the developed library are also discussed.

Chapter 8, *Conclusions*, summarizes the conclusions of this thesis, and shows how the formulated problem is solved and how the scientific question is answered.

Chapter 9, *Future work*, presents possible future work, which includes many ideas of extensions and further improvements to the library.

Chapter 2

Background

This chapter aims to present sufficient background to understand why element queries are desired and why they are hard to implement natively in browsers. Expert readers may want to skip directly to Chapter 3 where element queries are described.

Section 2.1 presents a brief history of web development and motivates why responsive web design and modular development are both popular and needed concepts in modern web development. Ultimately, Section 2.1.4 describes the problem of having modular responsive modules, which element queries solves.

Section 2.2 describes briefly how layout engines operate, with focus on the layout process, to later present how they can be parallelized in Section 2.2.7. As will be shown later in Chapter 3, element queries as a concept hinders parallelization of layout engines. Section 2.2.6 defines layout thrashing and shows how it can be avoided in order to improve performance.

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.

Section 2.1.1 describes the transition from browsers rendering simple *documents* to being hosts for complex applications, and is aimed for readers not familiar with the rapid development of the web. Section 2.1.2 defines and motivates responsive web design, and is needed to understand the problem that element queries solve. Section 2.1.3 similarly defines and motivates modular development, which also is needed to understand the problem that element queries solve. Section 2.1.4 presents the problem that element queries solve, which is that it is not possible to construct encapsulated responsive modules. This section will be the key motivation to why element queries are needed.

2.1.1 From documents to 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 [85, 55, 54, 89, 65, 2].

As further described in Section A.2 in the appendix, browsers were initially applications that displayed *hypertext* documents with the ability to fetch linked documents in a 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 used only 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 was delivered to the client (browser) it was still static. There was no way for the server to change the content that the client had received, unless the client requested 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, giving feedback on user input without involving the server, and animating content. Plugins also started to exist during this time, which enabled browsers to execute embedded programs. *Java applet*[1] and *Flash*[2] are examples of browser embedded programs requiring browser plugins to execute. In concept, 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 ship with preinstalled plugins, while others do not support plugins at all.

[1] A Java applet is a small application that 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`

With the increase of smart devices (such as phones, televisions, cars, game consoles, etc.) that include browsers with limited third-party runtimes, plugins quickly decreased in popularity. Additionally, as the web platform was improved and users being discouraged 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 enabled 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 meeting held by the W3C in 2004, it was proposed to extend HTML to ease the creation of web applications that was rejected. The W3C was criticized for 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 that were later 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 are 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.

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 websites 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 [4, 77, 19].

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 for a minimum width.

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 mobiles and laptops in size, which would require another special version of the site. Further, when desktops support touch actions and smartphones support mouse actions, even more versions of the website need 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 declarations 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.

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. In the last couple of years, as client-side 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 on [3, 1, 61].

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 modular development. Many of these libraries also help with dividing the HTML up into modules, creating small packages of style, markup and code.

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) where each part 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 themselves 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 come 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.

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 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 old versions may be used in parallel to newer versions. Having a good semantic versioning convention is key to reducing the complexity of managing multiple versions of modules.

2.1.4 The problem of responsive modules

Recall from Section 2.1.2 that responsive web design is the concept of having elements responding to size changes, so that a document may be usable on many screen sizes, resolutions, pixel-densities, and media. In Section 2.1.3 the desire for building web modules was presented. One positive outcome of building applications in a modular way is that general modules become reusable and complexity is reduced. Therefore, it is desired to extend the responsive web design concept to also let modules respond to size changes of the own working area. Since the only way of conditionally style elements 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 to write which somewhat defeats the purpose of writing modules. So developers today have two choices: writing modules that are not responsive and encapsulated, or writing modules that are responsive and not encapsulated (leaving the styling to the user).

Suppose that an encapsulated responsive module is to be developed. The module should be colored red when its width is narrow and blue when its width is wide. Further, suppose that media queries are used to conditionally style the module according to the defined behavior. As an example, a page is created that uses four module instances with different widths. See figure 2.1 for the desired behavior of the module instances on the example page. Unfortunately, media queries cannot be used to achieve this behavior (with the module still being encapsulated). See

figure 2.2 for the actual behavior of the example page. As evident in the figure, the page is broken since either all instances are colored blue or red. This is due to media queries only targeting the viewport, and not the actual width given to the instances.

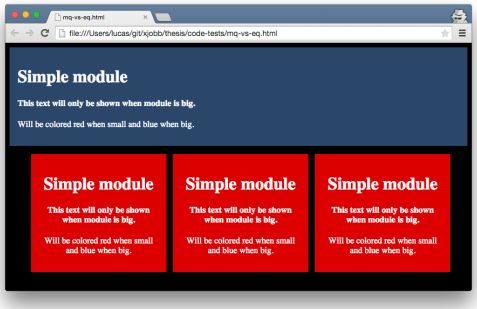

Figure 2.1. The desired behavior is that the small module instances should be colored red since their widths are narrow, and the big instance should be blue since its width is wide.

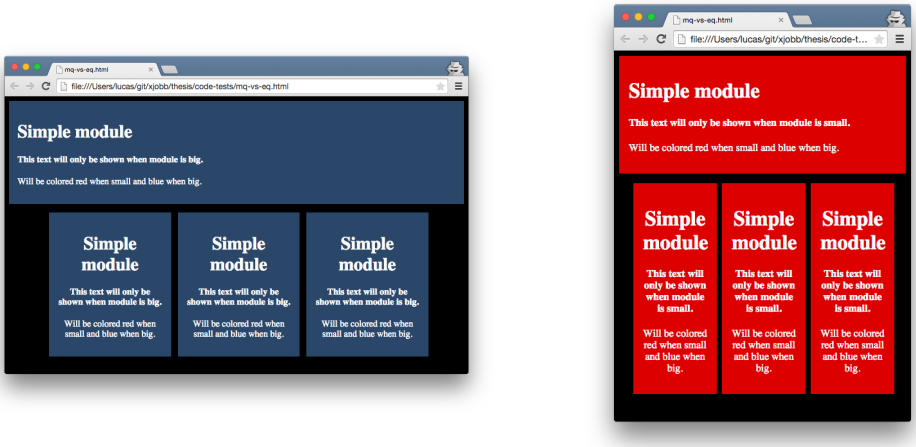

Figure 2.2. The module instances of the page do not behave as desired because they are conditionally styled by the viewport width, which does not reflect the individual width of the instances. All instances are always either colored blue or red, depending on the viewport width. In the left figure, the viewport is wide and therefore all instances are colored blue. In the right figure, the viewport is narrow and therefore all instances are colored red.

The solution Responsive modules should be able to react and change style by their own size, not the viewport size. An identified solution to this is element queries, which allows conditional styles to be applied by element criteria. The last

couple of years a lot of articles have been written about the problem and how badly web developers need element queries [50, 53, 82, 66, 87, 69, 72, 56, 59, 76, 71, 62]. As stated in Section 1.2, third-party non-native implementation efforts have been made with moderate success. The W3C receives requests and questions about it, and the RICG has started writing a draft [30] about element queries use cases.

2.2 Layout engines

In order to understand how element queries affect layout engines, it is important to understand how layout engines operate. 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. The render process will not be described, as it is not relevant for element queries.

Section 2.2.1 to Section 2.2.5 describes briefly how layout engines operate and also presents key layout engine concepts. The role of the layout engine and reference architecture for browsers is also presented. Section 2.2.6 defines layout thrashing will presents how it can be avoided. Section 2.2.7 gives insight into the current state of the research about parallelizing the layout process.

2.2.1 Reference architecture

Browsers are complex applications that consist of many subsystems. Reference architecture of browsers has been presented by [12], see figure 2.3. It is shown that

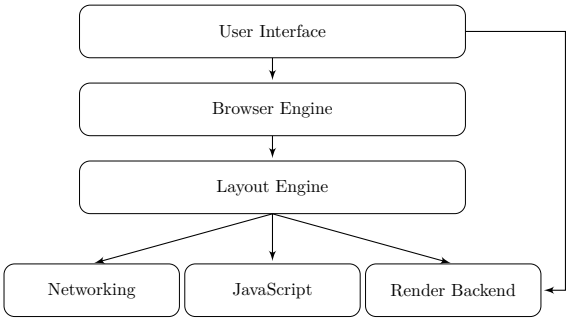

Figure 2.3. Reference browser architecture. The data persistence system, used by the browser engine and the user interface, has been omitted.

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 ren-

dering 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 [11]. The layout engine also performs the parsing of HTML. 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. In short, layout engines perform four distinct tasks:

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. It should be noted that although *Blink* is a recent *fork* of *WebKit*, they do differ quite significantly as shown in Chapter 5. Due to *Trident* being closed source, only the WebKit, Blink, and *Gecko* is 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%

Table 2.1. The major layout engines and browsers with market shares. See Section B.2 for more information how the market share data was gathered.

2.2.2 Constructing DOM trees

The DOM defines a platform-neutral model for events and node trees, which is used for representing and interacting with documents [22, 8]. 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 [11, 6]. 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 JavaScript 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 need 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.

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. This section is mainly based on [11, 6].

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 result in any way. The nodes of the render tree are called *renderers* (also known as *frames* or

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

render objects), as the nodes are responsible for their own and all subnodes layout and rendering. In order to know how to render them, 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 are not 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. In this section a brief explanation of selector matching, style cascading, inheritance and rule set weighting is given. CSS property definitions will also be described. This section is mainly based on [11, 6, 21].

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).

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 apply 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`.

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

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 [11]. The layout process will decide the final computed style of all elements, and needs to be done before rendering. HTML is flow based, which means that a document layout (also

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.

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 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 Layout thrashing and how to avoid it

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 doubles 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 2.4 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.

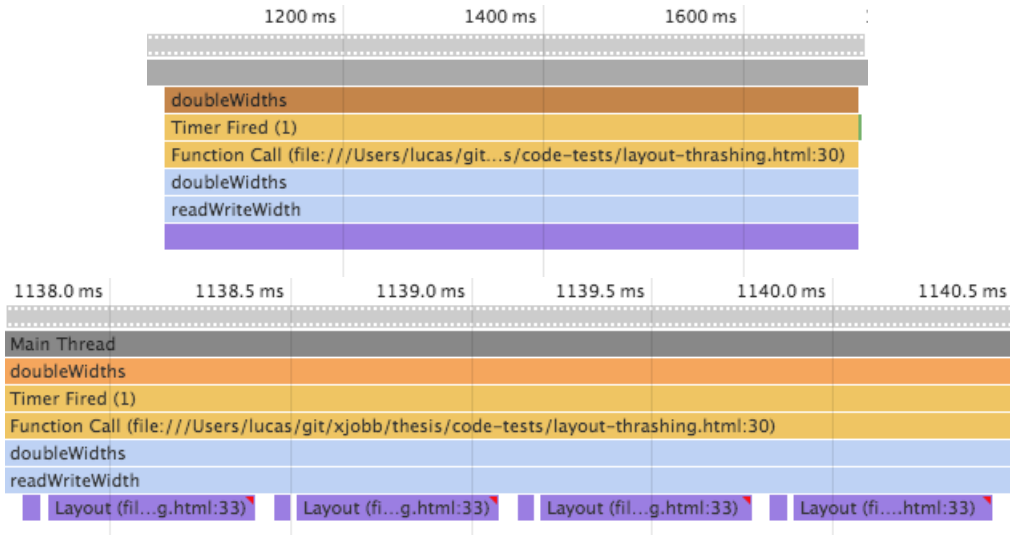

Figure 2.4. Two timeline figures of executing the example code given in listing 2.3. Both figures show the same timeline, where the top figure shows the whole execution and the bottom only shows a small section of the execution. Notice that they do not share the same time axis. Here it is clear that layout thrashing occurs, since layout is being done repeatedly throughout the whole execution (each iteration in the loop).

To avoid layout thrashing, the computed style requests and DOM mutations 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 read and write operations to the DOM. The code reads and doubles 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 2.5 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.

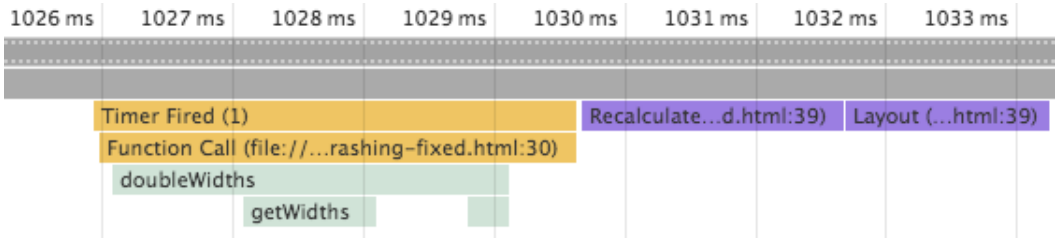

Figure 2.5. A timeline of executing the example code given in listing 2.4. It is clear in the figure that layout thrashing does not occur, since all style computations and layout is performed in a batch after that the example code has finished executing.

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 have 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.

2.2.7 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) [14]. 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 of layout engines can be approached.

As web applications grow bigger and more demanding, browsers continuously need to improve the performance on all levels [14]. 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 [7]. 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) [10, 7]. 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 as 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 [18, 14]. 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 [14, 18]. 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.

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 [14]. It has also been shown that selector matching and style resolving (through cascading) can be parallelized [7]. 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 [79, 84]. 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] [18].

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

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 as presented in Section 2.1.4.

Section 3.1 defines element queries, and shows how they can be used to solve the problem. It also presents the identified general use case for element queries. Since no syntax has been defined yet for element queries, pseudo-syntax is presented in this section to be used throughout this thesis. Some element queries terminology is also described, which is extensively used throughout this thesis. Section 3.2 presents some of the problems of implementing element queries natively in layout engines. Section 3.3 gives insight into some of the design approaches to element queries, as discussed by the W3C, to overcome some of the problems.

3.1 Definitions and Usages

Element queries are the solution to the problem of responsive modules as described in Section 2.1.4, since they allow modules to conditionally style themselves by element criteria.

Media queries and element queries are similar in the sense that they both enable developers to define conditional style rules that are applied by specified criteria [25]. The main difference is the type of criteria that can be used; in media queries device and media criteria are used, while element criteria are used in element queries. It can somewhat simplified be described as that media queries target the document root and up (i.e., the viewport, browser, OS, device, input mechanisms, etc.) while element queries target the document root and down (i.e., elements of the document).

Terminology The entity that is evaluated against a query is called the *target* of the query. For media queries, the target is usually the device or viewport. For element queries, the target is usually an element. It should be noted that the target is the entity that is evaluated against the query, and not the entity that

has conditionally styles applied. For instance, it is not possible to conditionally style the target of media queries, since CSS is incapable of altering the device. For element queries, the target entity is sometimes referred to as the target element. In the case of element queries, the target element may also be the element that has conditionally styles applied (but is generally not). CSS style declarations that are applied conditionally using either element queries or media queries are sometimes referred to as *conditional style declarations* or shortened *conditional styles*. When the layout engine evaluates a media or element query to determine whether the selectors match or not is called *query selector matching*. The expressions that define when different conditional style declarations should be applied are called *breakpoints*. For instance, the media query `@media (max-width: 500px)` selector defines one breakpoint; that the element should be styled differently depending on if it is wider than 500 pixels or not. There can be multiple breakpoints defined for both media and element queries. Element queries also define an element *style state*, which is in which state the target element of an element query is in (often relative to the breakpoints). For instance, if an element query targets an element `#foo` with a width breakpoint of 500 pixels (similar to the media query example), then the target element `#foo` would have two style states: narrower or wider than 500 pixels. The number of style states for an element of a given style property is $n + 1$ where n is the number of breakpoints for that style property.

Pseudo-syntax As already stated, element queries are not yet standardized and the exact behavior and syntax is still undefined. In this thesis, the pseudo-syntax of element queries is defined as a pseudo-class named `eq`. Recall from the CSS selectors level 3 specification [26] that pseudo-classes permit selection based on extended element information, and is often described as performing selection by the state of the element. The `eq` pseudo-class takes expressions as input, that is evaluated against the target element, as shown in the examples given in listing 3.1.

```
/* Matches all "a" elements that are wider than 500 pixels. */
a:eq(width > 500px) {
  color: yellow;
}

/* Matches any "a" element that is a child of a p element
   with the "foo" class that is narrower than 300 pixels. */
p.foo:eq(width < 300px) a {
  color: red;
}

/* Matches any p element with the "foo" class that is wider
   than or equal to 300 pixels that is a child of an "a" element. */
a p.foo:eq(width >= 300px) {
  color: blue;
}

/* Matches any "a" element that is in the hover pseudo-class state with a
   width between 300 and 500 pixels. */
a:hover:eq(300px <= width <= 500px) {
  color: purple;
}
```

Listing 3.1. Examples of the element queries pseudo-syntax.

General use case According to the draft written by the RICG the general use case is to conditionally style an element by an ancestor element's width to allow responsive web design for reusable modules [30]. Pages are usually designed to grow in height when content does not fit the viewport and therefore responsive web design usually targets the viewport width for layout breakpoints [4, 77, 19]. This can be verified by viewing the source code of popular responsive design frameworks (e.g., Bootstrap) where almost exclusively width breakpoints are present. Since element queries are to mainly allow responsive web design for modules, the same is assumed to apply for element queries (that only the width of elements are of interest). Further, it is assumed that element queries mainly target ancestor elements of the element to be conditionally styled, and therefore it is not crucial to support element queries for the own element width. This implies that element queries that target void elements (i.e., elements that cannot contain content) are also not crucial, and hence are not regarded as a general use case.

3.2 Problems

One issue 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 [32] by Zbarsky of Mozilla, Atkins of Google and Sprehn of Google that element queries are infeasible to implement without restricting them. In this section the two major problems are presented: performance and circularity.

3.2.1 Performance

Recall from Section 2.2.3 that the size and position of 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 resolving an element's size. If element queries that rely on the size of elements are present, 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 style declarations need to be evaluated against the elements for the element query selector matching.

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

So, element query selector match changes result in performing another layout that discards at least a subtree of the previous layout [32]. As already stated, this only occurs when the layout has changed in a way that changes the matches of element query selectors. Unfortunately, this means that if there is any matching element

query selector at page load, two layout passes always need to be performed. Also, it is common for internal APIs in layout engines to request updated element styles that do not require a layout to resolve (non-layout properties such as `color` and `font-family`) [32]. Since a layout of the element query selector target is required in order to resolve the correct element style, such internal APIs would need to force a layout in order to obtain the correct element style (even for non-layout related properties).

As shown in Section 2.2.7 layout engine vendors are interested in parallelizing their engines to increase the page performance. Element queries limit the parallelization, as layout engines would not be able to layout subtrees in parallel since an element in one subtree might affect an element in another subtree.

3.2.2 Circularity

The most obvious occurrences of cyclic rules are when there are conflicting element queries with width criteria and width style declarations for an element. See listing 3.2 for the perhaps simplest example of cyclic rules.

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

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

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

Listing 3.2. Simple example of cyclic rules with directly conflicting width element queries criteria and declarations. Recall the element queries pseudo-syntax defined in Section 3.1.

The result of this is of course implementation dependent, but a probable outcome of such code is the following infinite process:

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

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

3. The width of the element is wider than 500 pixels, so the selector `#foo:eq(width > 500px)` matches. Note that the `#foo:eq(width < 300px)` does not match, since the width is not narrower than 300 pixels. Since the matched selector is more specific than the `#foo` selector, the new width of the element is 250 pixels. The next step is 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). 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 layout is enforced every frame, the performance impact would be significant. This example is somewhat similar to writing `while(true);` outside the scope of a generator function in JavaScript (i.e., it locks up the main thread), which obviously is a bad idea. However, cyclic rules may also occur in less obvious ways.

Indirect cycles Indirect cyclic rules are somewhat more complex to reason about than direct cyclic rules such as the example given in listing 3.2. For instance, if an 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 3.3.

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

/* CSS */
#foo {
  /* Matches 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 3.3. 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 example more complex than the previous example is that it is less obvious for developers 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 even 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 containing element `#foo` a cycle appears in another part of the application (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 need to be considered. Also, the variable assignment could happen in another part of the code.

Runtime factors 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 to detect the cycles by performing a static analysis of the code. Browsers could do such analysis during parsetime in order to warn about or handle the cycles. However, cycles can occur in a more complex way that cannot be detected by static analysis. Consider the code in listing 3.4.

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

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

Listing 3.4. Example of cyclic rules that cannot be detected by static analysis.

In this example, it is impossible to deduce if the rules are cyclic by static analysis. A static analysis could perhaps identify that the code could potentially 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 depends 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)` does not match and therefore the element has no longer font size `2em`. Since the element width is decreased below 300 pixels when the font size of `2em` no longer is applied, the selector matches again and therefore 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 and therefore the rules are not 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 especially hard to reason about during static analysis, 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 not known by a static analyzer. This implies that there is no way of knowing if the cycle appears or not without actually running the code. It also implies that the cycle may appear in different browsers and settings, which makes the cycle even harder to detect.

3.3 Approaches to overcome the problems

By limiting element queries to specially separated viewport container elements (i.e., a sub-viewport of the document) that can only be queried by child elements, many of the problems are resolved [32, 31, 28]. This can be achieved by either adding a new HTML element or attribute. For the sake of simplicity, a HTML element named `viewport` will be used to define such sub-viewports in all examples throughout this section. In order to avoid cyclic rules, the following limitations to the viewport element are defined:

1. The size of the viewport element may not be dependent on its children. This implies that all CSS that causes a viewport element to fit its 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 element ancestor of the right-most simple selector may be targeted by element queries in selectors.

See listing 3.5 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 3.5. Examples of valid and invalid selectors with the viewport element.

This approach has been discussed by the W3C and the initial draft of the RICG assumes that this is a prerequisite to a native implementation [32, 31]. The reason that the HTML declaration of element viewport behavior 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 resolve the viewport elements. With the viewport behavior declared in HTML, the layout engine knows after it has parsed the HTML which elements are to be treated as separate sub-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) [32]. Also, the internal APIs that request non-layout information for 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 element queries may only target the nearest viewport element ancestor, each viewport subtree can be laid out in parallel. However, it is still inconvenient that the layout engine needs to evaluate all element query selectors in the middle of a layout pass (after that the viewport elements has been laid out) in order to resolve the styles for the viewport children.

Obviously, this approach limits element queries a lot. The fact that the size of the viewport element cannot depend on its children (like normal block elements do [6]), limits the usability. Viewport elements 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 (they create a new document and script context) [6].

Recall that the idea is that a viewport element cannot be queried for properties that its 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 means that if no child query the height of the viewport for instance, then the height of the viewport may depend on its children. This is a powerful insight, since the general use case is to write element queries against the width of elements (including viewport elements) and have the height automatically adapted to fit the content as described in Section 3.1.

JavaScript library Another approach to enable element queries 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 element queries would be operated by the library, layout engines

can be parallelized unhindered and independently of how the library handles the queries. Similarly, cycle detection and handling can be performed on top of the layout engine by the library. However, without the layout engine being aware of the element queries it is hard to avoid multiple layouts. Although an implementation in JavaScript will most probably be less performant 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.

Chapter 4

The element queries library

Now that sufficient knowledge about element queries and layout engines has been acquired it is time to present the actual realization of the solution to responsive modules — the JavaScript element queries library. 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 this thesis from now on.

Section 4.1 states the identified technical goals of ELQ. This section is the main motivation to that the library should be plugin based. It should be noted that some of the goals presented in the section are to be fulfilled by future work. The goals of this section are frequently referred to in this and the following chapter. Section 4.2 presents the overall architecture of ELQ, and key subsystems are described. This section also motivates why it is important to develop a library that enables advanced users to alter or remove subsystems of the core. Section 4.3 defines the API of the core library and all of the existing plugins. This section presents the main outcome of this thesis, as it shows how element queries are exactly realized by the library plugins. It is also in this section where it is shown how users can alter subsystems of the core. In addition, it is described how limitations of CSS are overcome by library plugins. Section 4.4 presents details about key implementation design decisions and algorithms. This section will also describe optimizations that have been done.

4.1 Technical goals

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 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 when elements change size.
    b) The features should be flexible and not limited to only the general use case.
    c) It should be possible to handle element query breakpoints with JavaScript.
    d) It should be possible to specify conditional CSS styles to be applied on different element query breakpoints.
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 features 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. Plugin-based libraries solves this problem by providing plugins so

[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.

that developers can compose their own custom tailored solutions for their specific use cases. By providing a good library foundation and plugins it is up to developers to choose the right plugins for each project. 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. Additionally, features may have incompatible APIs that would result in a bloated library API (since it would need to expose both APIs) if they were not divided into plugins.

4.2 Architecture

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 extendibility. 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 is just an extra payload to be loaded. 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 general use case described in Section 3.1.

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 generally never be programming against the core directly, and it can therefore only be accessed by plugins through a hidden plugin API (with the exception of few core methods that are part of the public API). The role of the library is to provide plugins with subsystems to be used for performing element query tasks, which is done through the plugin API. The library provides a small public API that is mainly used to handle and invoke plugins. The plugins form the bigger part of the public API in the sense that they decide the features that exist and how they work. It should be noted that plugins might 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 rate, with few breaking changes to provide good backward compatibility. 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:

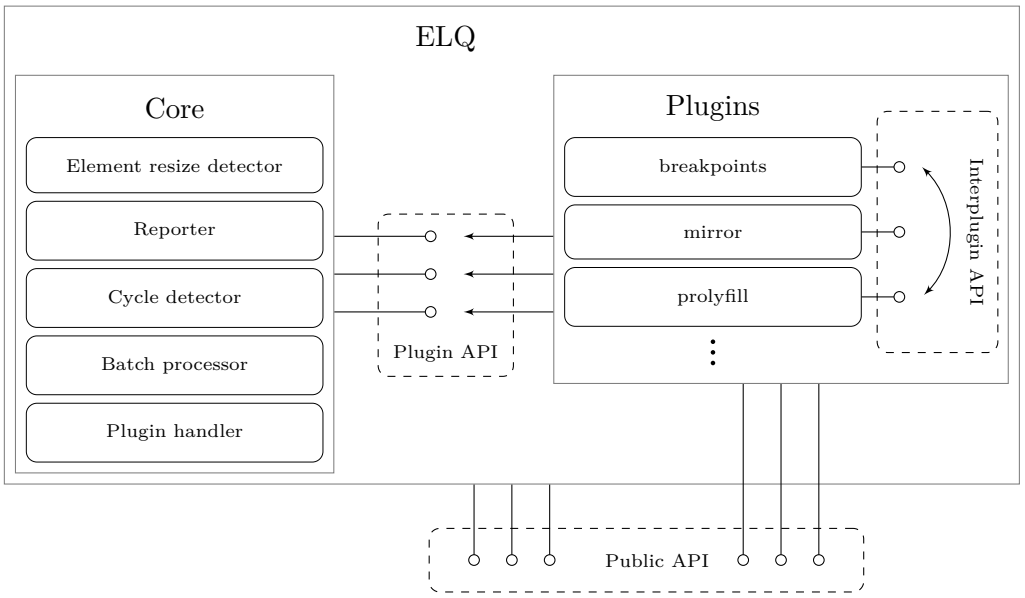

Figure 4.1. The overall library architecture, which shows how the library is divided into two parts; the core and plugins. The core is generally only accessible through the plugin API, and is not a part of the public API. Plugins may have interplugin APIs and dependencies. The bigger part of the public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

- **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 is also beneficial to have a centralized reporting system so plugins do not need to perform the same compatibility checks (such as checking if the browser actually supports console reporting).

- **Batch processor:** To avoid layout thrashing as described in Section 2.2.6, 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 batches. It supports executing batches asynchronously and also provides an interface for dividing batches up in levels that are executed in order. This subsystem is described further in Section 4.4.1.

- **Element resize detector:** Provides an interface for observing element resize events, which is fundamental to plugins fulfilling requirements 1a and 2a. It enables the library to observe elements resize events, instead of the applica-

tion keeping track of when elements are resized. This subsystem is described further in Section 4.4.2.

- **Cycle detector:** Responsible for detecting cycles and warning about them. As shown in Section 3.2.2 cyclic rules need to be detected at runtime. When a plugin wishes 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. This subsystem is described further in Section 4.4.3.
- **Plugin handler:** Of course, having plugins requires a subsystem for handling them. This system provides an interface for developers to register 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 some subsystems have been intentionally left out due to being too small or insignificant to understand the overall architecture of the library.

4.3 API design

4.3.1 Public API

As already stated, the bigger part of the public API is provided by plugins. In Section 4.1 it was determined that advanced users must be able to change the subsystems of the core. Therefore no global library instance is instantiated automatically on page load. Instead, a factory 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 = Elq();

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

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

The next step after creating an instance 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 instance. It is responsible for controlling that plugins do not conflict with each other and that plugins are initiated

correctly with the plugin API of the library instance. The method requires a *plugin definition object* (or shortened *plugin definition*) parameter, and also accepts an optional options object parameter. A plugin definition is responsible for describing a plugin and providing a factory function `make` that creates a plugin instance. The function requires an `elq` parameter that is the ELQ instance extended with the plugin API, and accepts an optional `options` object parameter. Plugin definitions also expose methods that are called by ELQ in order to retrieve the name, version and compatibility of a plugin. When `use` is called, it creates a plugin instance by invoking the `make` factory function (with the `use` options parameter as argument to `make`) and returns the created instance so that the user may store the reference to the plugin instance. The `using` method requires a *plugin identifier* parameter and tells if the given plugin is being used (i.e., has been registered) by the instance or not. The plugin identifier of a plugin is either the plugin definition or the name of the plugin. See listing 4.2 for an example plugin definition and how it is used with the two methods for handling plugins.

```
// An example plugin definition.
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) {
    // The elq parameter is the plugin API of the ELQ instance,
    // and is not the same object returned by the Elq factory function.

    // 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. Example plugin definition and examples of handling plugins.

When the desired plugins have been registered to the ELQ instance, it is time to initialize the library to the target elements. This is done with the `start` method that requires a collection of elements as a parameter. When called, it will invoke the `start` methods on all registered plugins that give 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 calling 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);

// Native API's are also supported.
var nativeElementsCollection = document.querySelectorAll("...");
elq.start(nativeElementsCollection);
```

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. Recall that access to the plugin API is given to a plugin when it is registered (i.e., when the ELQ instance invokes the `make` function of the plugin definition). Since the plugin handler and element resize detector subsystems are exposed through to public APIs they cannot be directly accessed through the plugin API. However, an additional method `getPlugin` of the plugin handler system is exposed that returns the registered plugin instance given a plugin identifier parameter. This method facilitates interplugin communications. Direct access is given to the following core subsystems:

- **Reporter**: via the `elq.reporter` property that refers to the reporter instance. The reporter has three methods: `log`, `warn` and `error` that are used to report information, warnings and errors respectively. The default behavior of the reporter is to report to the browser console[2] if possible.
- **Cycle detector**: via the `elq.cycleDetector` property that refers to the cycle detector instance. The cycle detector has a method `isUpdateCyclic` that tells if the desired state update of an element seems to be part of a cycle.

[2]See `https://developer.mozilla.org/en-US/docs/Web/API/Console` for more information.

To do so, it requires two parameters; the target element and the state update itself. A third update time parameter, which defines when the update was requested, is optional and will default to the time of the method call.

- **Batch processor**: via the `elq.BatchProcessor` property that refers to a factory function that creates batch processor instances. 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 processing batches. For example, some entities might want to force the batch to be processed at a point where it would be beneficial for other entities to delay the processing. The `createBatchProcessor` accepts an optional options object parameter and returns a batch processor instance. 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, the batch will automatically be processed as soon as possible asynchronously (this implies that `async` has to be enabled). The batch processor instance has two methods: `add` and `force`. The `add` method requires a function parameter that will be called when the batch is processed, and accepts an optional level parameter that defines at which level the given function should be processed. The `force` method commences the processing of the batch, which can happen synchronously or asynchronously defined by an optional parameter.

- **ID handler**: via the `elq.idHandler` property that refers to the ID handler instance. The ID handler has a 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.

4.3.3 Plugins

Now, it is time to describe the systems and APIs that actually realizes element queries as ELQ plugins. The plugins presented here are the suggested solution to element queries according to the research of this thesis. They are designed for adequate performance, good compatibility, and ease of use. For extreme requirements or corner cases, custom ELQ plugins might be preferred. It should be noted that the development of the prolyfill plugin is left for future work.

ELQ plugins may use custom element attributes as part of their APIs. The HTML standard supports custom attributes prefixed with `data-`. Modern browsers usually permit 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). Although not written in the examples the library and all plugins support both ways of defining custom element attributes, so it is able to conform to the HTML standard.

Breakpoints

This plugin is named `elq-breakpoints` and it observes element resize events in order to update the size states of elements according to defined breakpoints. 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 reflect the size states of the elements. Style states of an element are defined by breakpoints by custom element attributes. See listing 4.4 for an example of breakpoints definition attributes.

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

Listing 4.4. Example of an element that has two width breakpoints (300 and 500 pixels) defined by using 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-breakpoints-width="300 500"` defines that the element should have 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):

- $width < 300$
- $300 \leq width < 500$
- $500 \leq width$

The plugin appends classes to the element that reflects the size state of the element. For each breakpoint, one class is present that tells if the size is above or below the breakpoint. Recall from Section 3.1 that 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. Since the style state can be either above or below each breakpoint, 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 is: `elq-[width|height]-[above|below]-[breakpoint][postfix]`. It is possible to define a postfix to be applied to all breakpoint classes by the `postfix` option either at element level (i.e., by adding `postfix` as the attribute value of `elq-breakpoints`) or as a plugin instance option when registering the plugin with the `use` method of ELQ. For example, if the width of the element described in listing 4.4 is narrower than 300 pixels, it will have the following two classes present:

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

The breakpoint classes are used for applying conditional styles for elements based on the size state of the target element. See listing 4.5 for an example of conditional styling the the `#target` element and its children as presented in listing 4.4.

```
#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 an element can be conditionally styled by their own size. Additionally, children can be conditionally styled by adding the child simple selector to the right of the simple selector containing the element breakpoint classes. For instance, in the example all paragraph elements `p` will be styled conditionally by the target element size state.

Mirror

This plugin complements the `elq-breakpoints` plugin and is needed in some cases due to limitations of CSS. 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, which may be generated dynamically at runtime. Therefore, the structure of the application is not known when writing the CSS, so no assumptions can be made about the markup structure. Consider the example markup given in listing 4.6 for an example HTML structure. It should be noted that the `container` elements might be styled in any way (e.g., the widths of the inner container element 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 HTML structure where all paragraphs are desired to be conditionally styled by the nearest ancestor container.

A naive approach to conditionally style the paragraph elements is 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 conditionally style the paragraph elements by the size of the nearest ancestor container element.

The problem with this is that the selectors state 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 because they have an ancestor container that is wider than 500 pixels (i.e., the outer container). The desired behavior is that the first paragraph is colored green and the two inner paragraphs colored red, since the widths of the inner containers are narrower than 300 pixels. As of the CSS selectors level 3 specification, there it is not possible to select the nearest ancestor of an element [26].

The `elq-mirror` plugin overcome these limitations by mirroring the breakpoint classes of an `elq-breakpoints` element and thus the correct behavior can be achieved. Elements are initialized to mirror such elements by adding attributes in the same way like with the `elq-breakpoints` plugin. The mirror elements will then match the breakpoint 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 are now used in conjunction with the paragraph simple selector instead of the container in the CSS selectors. This way, the conditional styles behave as expected since all paragraphs are by the nearest (instead of any) container element.

By mirroring element breakpoint classes, elements can be styled by criteria 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.

4.4 Details of subsystems

In this Section the algorithms and approaches of some subsystems are presented. It should be noted that only the non-trivial subsystems are described.

4.4.1 Batch processor

Both Section 2.2.5 and 2.2.6 are recommended to read in order to understand the optimizations described in this section. The batch processor is the foundation for

good performance of the library, and is therefore used by several subsystems. It serves two purposes:

1. Automatically process a batch asynchronously.

2. Process a batch in different levels.

Automated processing The automated asynchronous processing of batches is important because some subsystems do not know how many operations are to be performed when invoked. For instance, the public ELQ API method `start` is called with elements to be initiated by the library. As part of the initialization, the element resizing detection subsystem may be invoked so it can prepare elements to be detectable. This needs to be batch processed, which is possible to do synchronously. However, if the `start` method is called multiple times synchronously layout thrashing occurs since one batch for each method call is created. See listing 4.9 for example code that invokes the `start` method multiple times synchronously.

```
elq.start(document.getElementById("..."));
elq.start(document.getElementById("..."));
elq.start(document.getElementById("..."));

var elements = [...];
elements.forEach(elq.start);
```

Listing 4.9. Example of multiple synchronous calls to the ELQ `start` method.

Of course, such usage could be warned against in the API documentation but to keep the API as simple as possible another approach has been taken. By delaying the batch to execute asynchronously all synchronous calls of the method is grouped into the pending batch. Since a single shared batch is used for all method calls layout thrashing is avoided.

This behavior can be controlled by the `async` and `auto` options of the batch processor factory function. If the `async` option is enabled, the batch is processed asynchronously as soon as possible after that the `force` method has been invoked. If the `auto` option is enabled, the batch is automatically processed as soon as possible asynchronously (this implies that `async` has to be enabled).

Leveled processing Being able to process a batch in levels is important when different types of operations, that are to be processed in a specific order (usually to avoid layout thrashing), needs to be grouped together in a batch. For instance, a function that doubles an element's width and reads the new calculated height benefits by being batch processed in three levels: reading the width, mutating the width, and reading the new height. Such function could also benefit from automatically process the batch so that the user may call the function multiple times synchronously. See listing 4.10 for an example implementation of such function that uses the leveled batch processor. The function achieves a 45-fold speedup when applied to 1000 elements by processing the batch in levels compared to not doing so in Chrome version 43.

```
var batchProcessor = BatchProcessor({
    auto: true,
    async: true
  });

function doubleWidth(element, callback) {
  var width = element.offsetWidth;
  var newWidth = (width * 2) + "px";

  // Implicit level 0 of the batch. Will be processed first.
  batchProcessor.add(function mutateWidth() {
    element.style.width = newWidth;
  });

  // Level 1 of the batch. Will be processed after level 0.
  // Changing the level number from "1" to "0" results in layout thrashing.
  batchProcessor.add(1, function readHeight() {
    var height = element.offsetHeight;
    callback(height);
  });
}

var elements = [...];
elements.forEach(function (element) {
  doubleWidth(element, function (height) {
    ...
  });
});
```

Listing 4.10. Example function that doubles an element's width and returns the new height with a callback. The function uses the leveled batch processor to avoid layout thrashing and thus gains a 45-fold speedup.

4.4.2 Element resizing detection

As described in Section 4.2, ELQ needs a subsystem that is able to detect resize events of elements. 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 resize events.

Unfortunately, there is no standardized resize event for arbitrary elements [24]. Only documents emit resize events in modern browsers and therefore such events can only be observed for frame elements (since a frame element has its own document). Legacy versions of Internet Explorer (version 8 and down, but also some later versions depending on the quirks and document mode[3]) do support the resize event for arbitrary elements. According to the general use case described in Section 3.1, a valid limitation is to only support element resize detection for non-void elements (i.e., elements that may contain content). This limitation is important since most approaches depend on injecting elements into the target element. It is a reasonable limitation since void elements can easily be wrapped with non-void elements without affecting the page visually. Possible solutions to detect element resize events in modern browsers are:

1. **Polling-based solution**: To have a script running asynchronously checking elements if they have resized. This can be achieved by using the JavaScript

[3]See `http://en.wikipedia.org/wiki/Quirks_mode` for more information about quirks mode.

`setInterval`[4] function. A short polling interval (high frequency) leads to being able to detect resize events quicker but having worse performance. A long polling interval (low frequency) leads to not being able to detect resize events as quickly but having better performance. The longest delay between an actual resize event and the detection is given by the polling interval time. This approach is the most robust, as it supports arbitrary elements (including void elements) and provides excellent browser compatibility.

2. **Object-based solution**: Since frame elements are the only elements that support resize events natively, the idea is to inject a frame element as a child to the target element that is observed instead. It has been shown by [52] that `object` is the most suitable frame element to be used for this purpose as they have good browser compatibility and adequate performance. The `object` is styled so that it always matches the size of the target element and so that it does not affect the page visually. Then a resize event handler is attached to the document of the `object` that emits a `resize` event[5] for every target element resize.

3. **Scroll-based solution**: This solution injects an element to the target element that contains multiple overflowing elements that listen to `scroll` events[6]. The overflowing elements are styled so that `scroll` events are emitted when the target element is resized. For detecting when the target element shrinks, two elements are needed; one for handling the scrollbars and one for causing them to scroll. Similarly, for detecting when the target element expands, two elements are needed in the same way. A container element that contains the four elements is injected and styled so that it matches the size of the target element and does not affect the page visually.

4. **Flow-based solution**: This solution is similar to the scroll-based solution in the sense that it also injects elements that have overflowing content. Instead of listening to `scroll` events, this solution uses flow events. The idea is to detect when scrollbars disappear (i.e., underflow of content since the target container has increased in size) and when scrollbars appear (i.e., overflow of content since the target container has decreased in size). Unfortunately such events are not standardized, but Gecko, WebKit and Blink support them (although by different APIs)[7]. Blink still supports the events but has deprecated them [75] since version 34.

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

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

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

[7]See `https://developer.mozilla.org/en-US/docs/Web/Events/overflow` for more information about the Gecko flow event API.

The polling-based solution is appealing because it does not mutate the DOM, supports arbitrary elements, and it provides excellent browser compatibility since it does not rely on special element behavior or similar. However, in order to prevent the responsive elements lagging behind the size changes of the user interface, polling needs to be performed quite frequently. Recall from Section 2.2.5 that layout engines typically have a layout queue in order 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. Since the polling is performed all the time the overall page performance is decreased even if the page is idle, which is undesired especially for devices running on battery.

Injection solutions The object-based, scroll-based and flow-based solutions have similar characteristics and have all been originally presented by [52]. The flow-based solution was presented first, but was rejected in favor of the scroll-based solution as it was discovered that in turn was rejected in favor of the object-based solution. Fortunately, somewhat reworked versions of the flow-based and scroll-based solutions are still provided by [44] and [46] respectively. While not stated officially by [52], the flow-based solution was probably rejected due to the lack of browser compatibility. The flow-based solution is not presented in detail, due to the limited browser compatibility and similarity with the scroll-based solution. The scroll-based solution was probably rejected due to issues with handling target elements with sides that are zero in length and also with the target element getting removed from the render tree (e.g., by setting the `display` style property to `none`).

All three solutions mutate the DOM and rely 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 injecting the elements 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 widths and heights set to `100%` and the `visibility` set to `hidden`, the injected elements do not affect the visual representation of the document. However, injecting elements into 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 *` also matches the injected elements, which may result in them being styled in an undesired way. Especially the scroll-based solution 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` may not behave as expected.

- **JavaScript may break**: Similar to with 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 as the target element has an extra child. Also, code that alter the content of a target element (such as `element.InnerHTML = ...`) may undesirably remove the injected element.

- **The target element must be positioned**: Absolute positioned elements cannot be styled relative to `static` positioned elements and are therefore moved up to the first non-static positioned ancestor in the render tree [27]. 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` (i.e., the default positioning for many elements). Fortunately, elements positioned `relative` behave exactly like `static`, given they 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 is positioned `static`, the properties can be removed or regarded as developer errors. This way target elements can be positioned `relative` with the special style properties removed, to obtain the same visual representation as being positioned `static`.

The object-based solution The `object` element provides excellent performance for detecting element resize events, but injecting many `object` elements is quite a heavy task for browsers and it consumes a significant amount of memory as later presented in Section 5.2. The main algorithm that is performed when an element e is to be observed for resize events is the following:

1. Get the computed style of e.
2. If the element is positioned (i.e., `position` is not `static`) the next step is 4.
3. Set the position of e to be `relative`. Here additional checks can be performed to warn the developer about unwanted side effects of doing this.
4. Create an `object` element and attach an event handler for the `load` event[8]. When the element has been styled and configured properly, it is injected into e.
5. The algorithm waits for the `load` event handler to be called by the layout engine. When the handler is called, a `resize` event handler is attached to the document of the `object` element.

In the rewriting of the code provided by [52] to better suite ELQ, efforts were made to optimize the solution as the original code suffers from layout thrashing. Step 1 and 3 could theoretically be executed in different batches to avoid layout thrashing. Unfortunately, each `object` element creation in step 4 forces a full layout, which makes the batch processing optimization negligible. Since the creation of `object` elements forms the significant performance penalty, no further optimization attempts were made.

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

The scroll-based solution As this solution only injects `div` elements, it offers greater opportunities for optimizations. The algorithm is conceptually similar to the algorithm of the object-based solution. The main algorithm that is performed when an element e is to be observed for resize events is the following:

1. Get the computed style of e.
2. If the element is positioned (i.e., `position` is not `static`) the next step is 4.
3. Set the position of e to be `relative`. Here additional checks can be performed to warn the developer about unwanted side effects of doing this.
4. Create the four elements needed (two for detecting when e shrinks, and two for detecting when e expands) and attach event handlers for the `scroll` event of the elements. When the elements have been styled and configured properly, they are added as children to an additional container element that is injected into e.
5. The current size of e is stored and the scrollbars of the injected elements are positioned correctly.
6. The algorithm waits for the `scroll` event handlers to be called asynchronously by the layout engine (they are called since the previous step repositioned the scrollbars). When the handlers have been called, the injection is finished and observers can be notified on resize events of e when `scroll` events occur.

Due to problems such as layout thrashing, bugs and an undesired API, the implementation provided by [46] was completely rewritten. Layout thrashing can be avoided by using the leveled batch processor described in Section 4.4.1. The rewritten version (from now on referred to as the ELQ scroll-based solution) performs the algorithm steps in the following levels:

1. **The read level:** Step 1 is performed to obtain all necessary information about e. The information is stored in a shared state so that all other steps can obtain the information without reading the DOM.
2. **The mutation level:** Steps 2, 3 and 4 are performed, which mutates the DOM. All mutations performed in this level can be queued by the layout engine.
3. **The forced layout level:** Step 5 is performed, which forces the layout engine to perform a layout.

Since repositioning a scrollbar forces a layout, such operations need to be performed after that all other queueable operations have been executed. Therefore, step 5 is performed in level 3 as the last step. Layout engines can theoretically queue the scrollbar repositioning operations too as they do not affect the layout of each other. The Blink layout engine is able to do so, which results in exceptional performance

as shown in Section 5.2. Gecko and WebKit are not able to do so and therefore each iteration in step 5 forces a layout. However, it is still beneficial to batch process the algorithm for such layout engines since only pure layouts need to be performed (instead of having to recompute styles and synchronize the DOM and render trees before each layout). As step 6 is performed by the layout engine asynchronously and does not interact with the DOM, it does not need to be batch processed.

MutationObservers The `MutationObserver` API defined in the W3C working draft of the DOM level 4 specification [23] may seem like a good candidate to use for detecting element resize events at first glance. Although not yet standardized, the API is implemented in modern browsers according to the working draft. By using mutation observers it is possible to observe attribute or subtree changes of an element. By observing the `style` attribute of an element and the content of it (if the element size depends on its children), it is possible to detect direct style changes of an element (e.g., changes to the `style` attribute by JavaScript). The main limitation is that it is not possible to observe the computed style state of an element. For instance, detecting that the width of an element has been set to 50 % with JavaScript is possible (it would not have been possible to detect the style change if the width style was calculated by a CSS cascade). However, when the parent container changes size the observer would not detect that the width of the element has changed, as the style attribute width still is 50 %. This could theoretically be solved by also observing the parent of the element, and so on up to the root of the document. Such solution would probably end up observing most elements of a page, which might be a performance penalty, and it would still not be able to detect all element resize events. Element resize events may occur in numerous ways that do not alter the DOM by CSS (e.g., conditional styles defined by media queries, pseudo-classes, etc.). Using the `MutationObserver` API is interesting, but it is only useful if it supports observing computed style changes of an element for detecting element resize events. Due to these limitations, the API was not considered to be used in ELQ.

4.4.3 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 handle 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 tells if the desired update seems to be part of a style cycle or not. Further, recall from Section 3.2.2 that there may be multiple factors of a cycle (e.g., content, browser behavior, CSS, JavaScript). Due to cycle graphs being non-trivial a simplistic approach to detecting cycles has been taken. The idea is to keep track of all style state changes of elements in order to decide if new state changes

are cyclic by examining the state history. This approach is simple to implement and quite sufficient for detecting 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 in time, 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, a chronologically ordered list of states L_{states} is kept. For each call to the function, the following cycle detection algorithm is performed:

1. Construct L_{states} if needed.
2. Construct 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 than 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 then also be too old. Therefore, all tuples from (and including) u_i are removed from L_{states}. The next step is 5. Of course, removal is not performed unless u_i is too old.
   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. The 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 5

Empirical Evaluation

This chapter presents the empirical evaluation that has been done of the ELQ library. The main focus of the evaluation has been to evaluate objective concepts of the library in order to achieve comparable results.

Section 5.1 evaluates the APIs by a case study that alters the popular Bootstrap framework to use element queries. Bootstrap uses media queries to provide responsive CSS classes that often are desired to use in web applications. By using element queries instead, the classes become encapsulated and can be used in responsive modules. Section 5.2 evaluates the performance of ELQ in different browsers. Key subsystems are evaluated independently so that they can be compared to similar subsystems of related libraries.

5.1 ELQ Bootstrap

The responsive parts of Bootstrap version 3.3.2 (the grid, containers, responsive utility classes, forms, etc.) are built with media queries, and therefore only behave as desired when the Bootstrap elements can use the full width of the viewport. It should be noted that no height breakpoints are being used by Bootstrap (as also identified by the general use case in Section 3.1). This implies that modules that use any responsive Bootstrap classes inherently only behave as desired when the module can use the full width of the viewport, which means that such modules are not encapsulated and therefore many of the benefits of modules disappear. Such modules force applications to use them in a specific layout. As an example, the Bootstrap CSS documentation page was altered to instead offer a two-column documentation. The two-column page presents two instances of the original CSS documentation. It is shown in figure 5.1 that the original version of Bootstrap (that uses media queries) cannot adapt to such layout changes.

In order to make Bootstrap behave as desired in any layouts the responsive parts of the framework were modified to use element queries instead of media queries by using ELQ. Two Bootstrap classes have been chosen to be treated as sub-viewports that are `elq-breakpoints` elements: `container` and `container-fluid`. Both

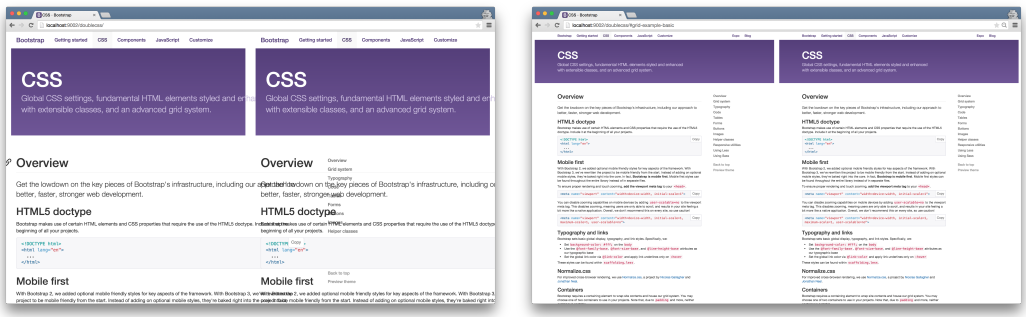

Figure 5.1. The responsive classes of Bootstrap cannot adapt to layout changes as shown with the two-column CSS documentation page example. Both documentation instances use media queries, which undesirably style both columns as if they each fill the whole viewport width. Since the columns only fill half of the viewport width and are styled as they both fill the whole viewport width, the page appears broken as shown in the left figure. The page only behaves as desired if the viewport width is large enough so that the width of each column is bigger than the largest media query breakpoint, as shown in the right figure. See figure B.1 and B.2 in the appendix for the left and right figure in full scale, respectively.

classes are used in Bootstrap to define new parts of a page (e.g., a grid is required to have a container ancestor). They are also nestable, which makes them suitable to be used as sub-viewports. The `container` class centers content with fixed widths by different viewport sizes, and the `container-fluid` uses the full available width. The idea is to have all responsive classes conditionally styled with element queries by the size of the nearest ancestor container element. This implies that all elements with responsive classes are converted to `elq-mirror` elements since they need to mirror the breakpoints of the nearest ancestor `elq-breakpoints` element (i.e., a container element). Since container elements may be nested, `container` elements are both `elq-mirror` and `elq-breakpoints` elements (because `container` elements also conditionally style themselves, as opposed to `container-fluid` elements). Recall from section 4.3.3 that it is beneficial for responsive elements to be `elq-mirror` elements in order to not limit the element usage to a specific HTML structure.

Altering the style code Bootstrap mainly uses three width numbers for responsive breakpoints: 768, 970 and 1170 pixels. Since the Bootstrap CSS is generated by the *LESS* preprocessor, they are defined as constants as presented in listing 5.1.

```
/* File "less/variables.less" of Bootstrap. */
@screen-sm-min: 480px;
@screen-md-min: 992px;
@screen-lg-min: 1200px;
```

Listing 5.1. The main breakpoints used by Bootstrap defined as LESS constants.

Notice that the numbers also include the `px` postfix. Recall from Section 4.3.3 that the breakpoint classes added by the `elq-breakpoints` plugin do not include the `px` postfix by default (i.e., if an element is above 480 pixels the class would

be `elq-width-above-480` and not `elq-width-above-480px`). Fortunately, the `elq-breakpoints` plugin may be configured to append a postfix to the breakpoint classes by the `postfix` option. By configuring the plugin to use `px` as postfix, the constants can be used seamlessly in element queries. As shown in listing 5.2, media queries are easily replaced by element queries.

```
/* File "less/grid.less" of Bootstrap. */

// Original Boostrap using media queries.
.container {
  .container-fixed();

  @media (min-width: @screen-sm-min) {
    width: @container-sm;
  }
  @media (min-width: @screen-md-min) {
    width: @container-md;
  }
  @media (min-width: @screen-lg-min) {
    width: @container-lg;
  }
}

// ELQ Bootstrap using element queries.
.container {
  .container-fixed();

  &.elq-width-above-@{screen-sm-min} {
    width: @container-sm;
  }
  &.elq-width-above-@{screen-md-min} {
    width: @container-md;
  }
  &.elq-width-above-@{screen-lg-min} {
    width: @container-lg;
  }
}
```

Listing 5.2. Media queries can easily be replaced with element queries. By using the `elq-breakpoints` postfix option; the breakpoint constants can be used directly in the selectors. Notice that only three lines have been altered.

By using the power of preprocessors, ELQ element queries become as pleasant to work with as media queries. In fact, only ~50 lines out of ~8500 lines of Bootstrap LESS code needed to be altered. Most changes were of the character shown in listing 5.2, which basically replaces the media query syntax with the ELQ element queries syntax. This is especially advantageous when keeping a forked project up to date with the original project, as fewer diverged lines implies a lowered risk of merge conflicts.

Adding the ELQ library The altered Bootstrap version depends on ELQ including two plugins (`elq-breakpoints` and `elq-mirror`) and must therefore be included. ELQ and the plugins could be bundled with the JavaScript of Bootstrap, but it was decided to keep them separated. Bootstrap's other dependency, jQuery, is also separated from the Bootstrap JavaScript. Since it is beneficial to not require changes to existing Bootstrap empowered pages, all ELQ element attributes are added with JavaScript. The attributes could of course be written directly in the HTML instead; the choice is up to the user. Listing 5.3 presents an example

of such JavaScript code that dynamically adds the required element attributes and initializes all responsive elements.

```
// Creating the ELQ instance needs to be done once.
var elq = Elq();
elq.use(elqBreakpoints, {
  postfix: "px"
});
elq.use(elqMirror);

// This function initiates all responsive elements of the document.
function init(elq) {
  // Find all elements that have responsive classes.
  var breakpointsElements = [...]; // body, .container-fluid, .navbar, ...
  var containerElements   = [...]; // .container
  var mirrorElements      = [...]; // .visible-xs, .col-md-1, .col-md-12, ...

  // Add all attributes for the elq-breakpoints elements.
  breakpointsElements.concat(containerElements).forEach(function (element) {
    element.setAttribute("elq", "");
    element.setAttribute("elq-breakpoints", "");
    element.setAttribute("elq-breakpoints-width", "480 768 992 1200");
  });

  // Add all attributes for the elq-mirror elements.
  mirrorElements.concat(containerElements).forEach(function (element) {
    element.setAttribute("elq", "");
    element.setAttribute("elq-mirror", "");
  });

  // .container elements are both elq-mirror and elq-breakpoints,
  // and therefore the noclasses option is added so that the plugins
  // do not interfere with each other.
  containerElements.forEach(function (element) {
    element.setAttribute("elq-breakpoints", "noclasses");
  });

  var elements = breakpointsElements.concat(containerElements, mirrorElements);
  elq.start(elements);
}

// Init all responsive elements. This needs to be executed when
// new responsive elements are added.
init(elq);
```

Listing 5.3. Example of JavaScript code that adds the required ELQ attributes dynamically to all responsive elements and initializes them.

The result Altering the LESS code, including the ELQ library, and the initiating JavaScript is all that is required to make the double column documentation page behave as desired. Figure 5.2, 5.3 and 5.4 shows different sections of the double column documentation page empowered by ELQ Bootstrap. For reference, the same sections using the original Bootstrap is also included that shows how they behave without element queries. As shown in the figures, the double column documentation page using the original Bootstrap styles the two columns as if they both had the whole viewport width available. Therefore the two columns intersect because some content elements are styled wider than a column. The double column documentation page using ELQ Bootstrap on the other hand enables all responsive elements to style themselves according a parent container element. Therefore, the two half-page columns detect that they only have half the viewport width and style themselves accordingly. The visual result is the same as having two documentation sites of the

original Bootstrap in two `iframe` elements as two columns (since `iframe` elements creates a separate viewport).

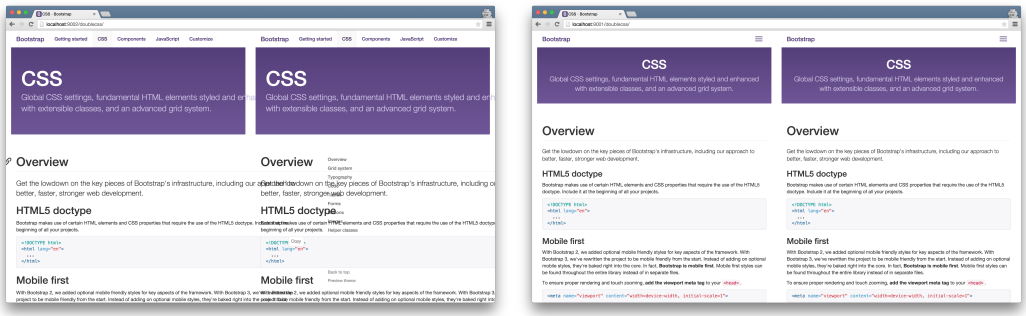

Figure 5.2. The left image shows the original Bootstrap header, which is broken since the content of both columns are styled as if they had the full viewport width. The right image shows same section of the same page, using ELQ Bootstrap. With element queries, both columns know that they only have half of the viewport width and therefore style themselves accordingly. See figure B.1 and B.3 in the appendix for the left and right figure in full scale, respectively.

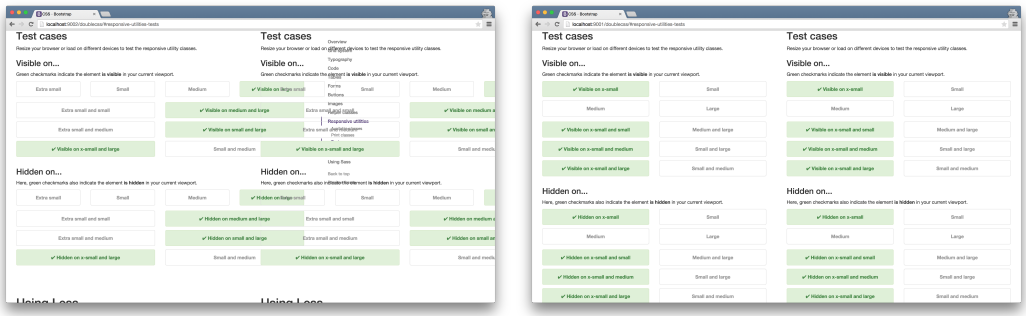

Figure 5.3. The left image shows the original Bootstrap responsive utilities classes matrix. Here it is clear that both columns are styled as if they each have the full viewport width, as they display the large utility classes in green. The right image shows same section of the same page, using ELQ Bootstrap. Only the small utility classes are displayed in green as desired, since both columns only have half of the viewport width available. See figure B.4 and B.5 in the appendix for the left and right figure in full scale, respectively.

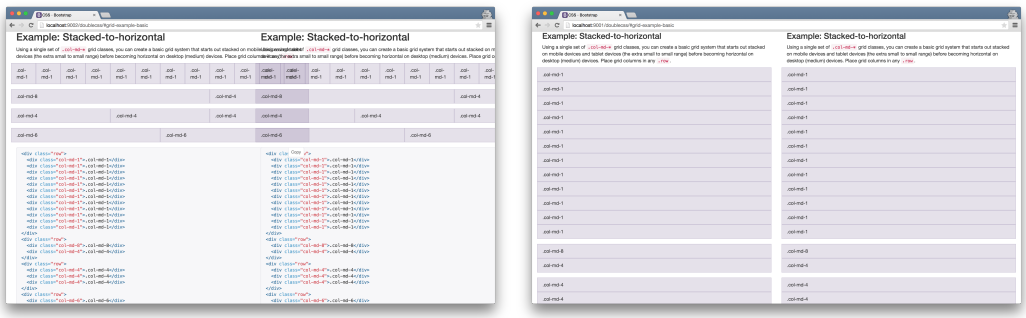

Figure 5.4. The left image shows the original Bootstrap responsive grid. It is styled as if both columns have the full viewport width available, and therefore the grids intersect. The right image shows same section of the same page, using ELQ Bootstrap. The grids detect that only the half viewport width is available, and styles themselves accordingly. See figure B.6 and B.7 in the appendix for the left and right figure in full scale, respectively.

5.2 Performance

The following tests were performed on a computer with a 2.5 GHz processor and 16 GB of memory[1]. The library has been tested in the following browsers: Chrome 42.0.2311.152, FireFox 37.0.1 and Safari 8.0.6. Measurements and graphs show evaluations performed in Chrome unless stated otherwise.

This section evaluates the performance of the object-based and scroll-based solutions to detecting element resize events described in Section 4.4.2. The optimized ELQ version of the scroll-based solution is also be evaluated. Since the element resizing detection subsystem performs the heaviest tasks, only the performance of that subsystem is evaluated in detail. The other subsystems entail no significant performance penalties.

The object-based solution performs well when detecting resize events. However, injecting `object` elements is quite a heavy task. See figure 5.5 for graphs that show the performance of the object-based solution. As shown by the graphs, the injection can be performed with adequate performance as long as the number of elements is low. The solution does not scale well as the number of element increases.

The scroll-based solution also performs well when detecting resize events. As no `object` elements are injected the memory footprint is reduced significantly, which improves the injection performance. See figure 5.6 for graphs that show how the both solutions perform compared to each other. It is clear that the scroll-based solution both performs and scales better than the object-based solution during injection. 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 noticeably.

[1]The serial number of the computer is C02N4G9TG3QD and the vendor is Apple Inc. CPU: 2.5 GHz Intel Core i7. Memory: 16 GB 1600 MHz DDR3. GPU: Intel Iris Pro 1536 MB.

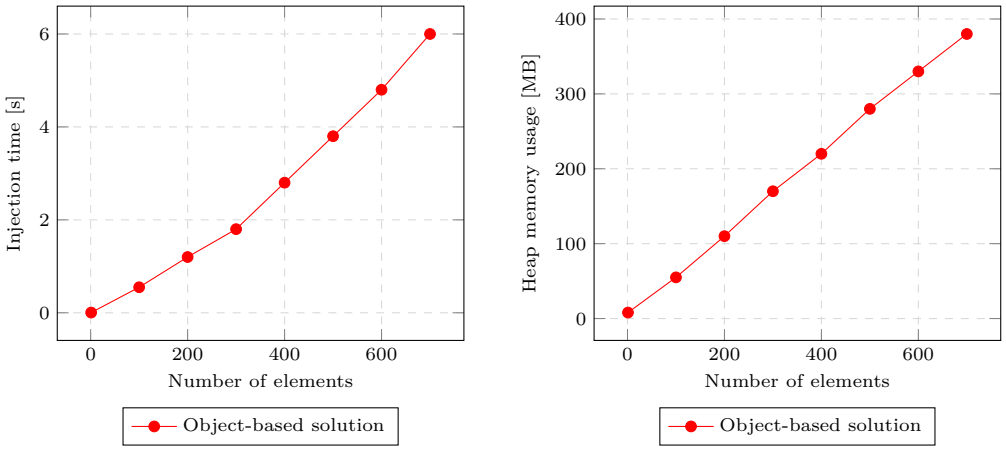

Figure 5.5. The injection performance of the object-based solution. The left graph shows the injection time. The right graph shows the heap memory used when all `object` elements have been injected.

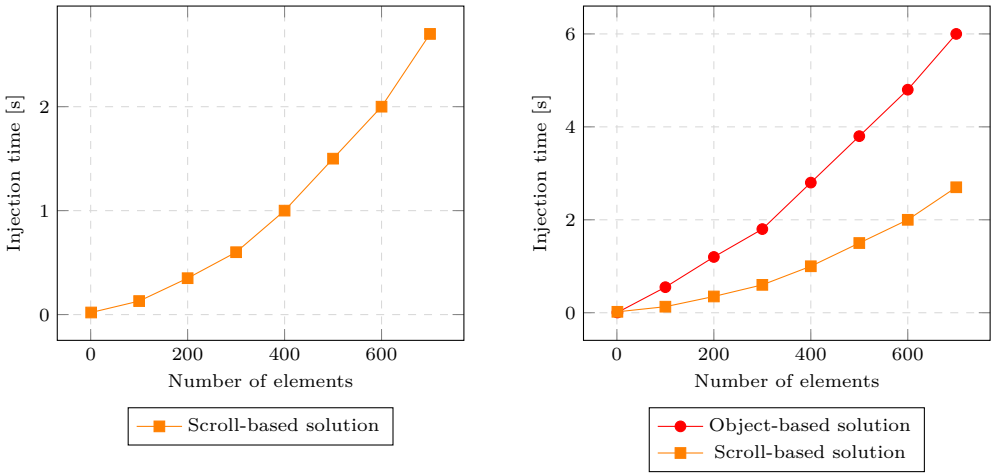

Figure 5.6. The injection performance of the scroll-based element resizing detection solution provided by [46] and originally created by [52]. The left graph shows the injection time of the scroll-based solution. The right graph also includes the object-based solution for reference. The heap memory usage graph has been omitted as the memory usage for the scroll-based solution is near constant.

Recall that the scroll-based solution was rewritten and optimized; which is referred to as the ELQ version of the scroll-based solution. By avoiding layout thrashing, the injection performance was improved significantly. See figure 5.7 for graphs that show how it performs compared to the other solutions. As evident in the figure, the optimized ELQ solution has significantly reduced injection times compared to the other two. It achieves a 32-fold speedup compared to the object-based solution

and a 13-fold speedup compared to the scroll-based solution when preparing 500 elements for resize detection. The ELQ solution also scales better, as more clearly shown in figure 5.8 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).

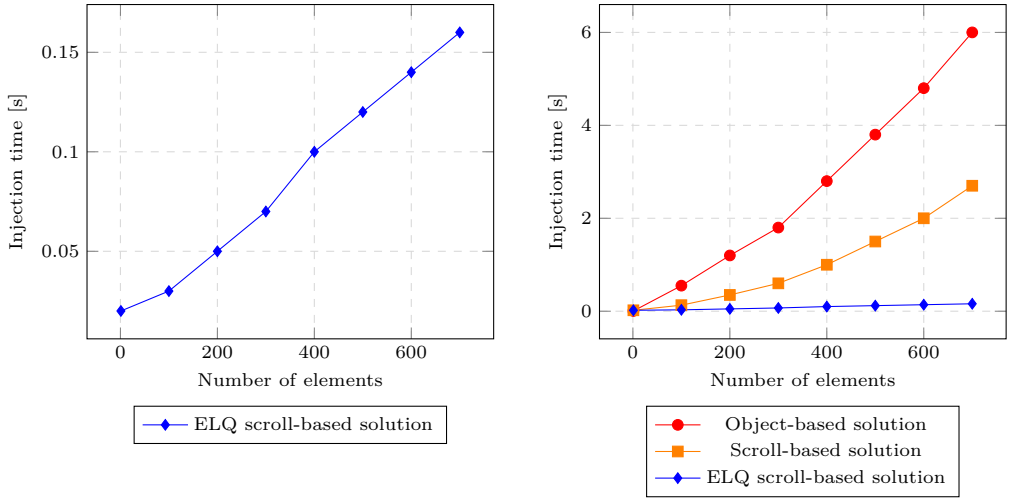

Figure 5.7. The injection performance of the optimized ELQ scroll-based element resizing detection solution. The left graph shows the injection time of the ELQ scroll-based solution. The right graph shows all three solutions for comparison. The heap memory usage graph has been omitted as the memory usage for the scroll-based solutions are near constant.

FireFox and Safari As shown, great performance can be achieved with the optimized ELQ scroll-based solution in Chrome. Unfortunately, there is no silver bullet to observing element resize events; as the other browsers behave differently. See table 5.1 for the performance of the object-based and scroll-based solutions operating on 100 elements in different browsers. The ELQ scroll-based solution is preferred for Chrome, as the injection is 32-fold faster (when operating on 500 elements) than the object-based solution while the resize detection performance is the same for both solutions. In FireFox, the object-based solution detects resize events 2-fold faster than the ELQ scroll-based solution when operating on 100 elements (still, 100 ms for detecting resize events is acceptable). However, the injection time needed for the object-based solution is 5.5-fold of the time needed for the ELQ scroll-based solution. The ELQ scroll-based solution is therefore probably desired in FireFox for the general use case (as described in Section 3.1). In Safari, the ELQ scroll-based solution detects resize events in 800 ms while the object-based solution detects them in 25 ms, which of course is unacceptable. Unfortunately, the injection time needed for the object-based solution is 3-fold slower than the ELQ scroll-based solution. Since a delay of 800 ms when detecting resize events is undesired in most use cases,

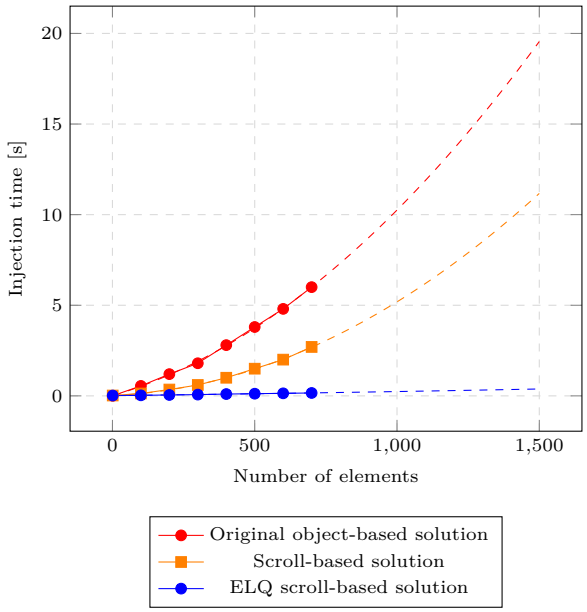

Figure 5.8. The injection performance of all three solutions including graph predictions by polynomial regression.

the object-based solution is preferred is Safari. Recall from Section 4.4.2 that this is due to WebKit and Gecko not being able to queue the scroll mutation operations as Blink does.

Layout engine	Injection		Resize detection	
	scroll	object	scroll	object
Blink	**30 ms**	600 ms	30 ms	30 ms
Gecko	**200 ms**	1100 ms	100 ms	**50 ms**
WebKit	**100 ms**	300 ms	800 ms	**25 ms**

Table 5.1. Performance of the object-based and ELQ scroll-based solutions in different layout engines when operating on 100 elements.

Chapter 6

Related work

There are numerous third-party element queries JavaScript libraries, which this chapter aims to list and analyze. Since many of them share the same characteristics, they are classified and analyzed in groups. A constraint-based approach is presented more in depth as it differs from the other libraries.

Identified libraries that directly or indirectly enables element queries are presented in table 6.1. The table also includes the classification of each library and some additional comments. It should be noted that most of the current approaches are combinations of different classes. The characteristics that they are classified after are the following:

- **Syntax**: Libraries can either require custom syntax or valid syntax. Custom syntax is considered to be invalid syntax since it does not conform to web language standards (custom CSS is the most common case of custom syntax). Libraries that do no require custom syntax are considered to have valid syntax.
- **Page type**: All libraries support static pages, and some also support dynamic pages. Static pages do not change layout at runtime, and therefore no element resize detection or element queries library runtime is needed. Dynamic pages may change layout at runtime, and therefore needs an element queries library runtime with element resize detection.
- **Resize detection**: Libraries need to detect element resize events for dynamic layouts, and there are three levels of detection support for elements: viewport only, non-void elements, and arbitrary elements. It should be noted that this only reflects the theoretical support of elements; in practice many libraries have issues and bugs that make them unable to detect all element resize events.

Invalid syntax The libraries [36, 38, 44, 33, 35] have in common that they require developers to write custom (invalid) CSS. Since they no longer conform to the CSS standard, new features can be supported through the custom CSS. As shown by [38, 35] quite advanced features can be implemented this way. Additionally, adding new

Implementation		Syntax	Resize detection	Page type	Comments
[36]	MagicHTML	Invalid	-	Static	Compiles invalid CSS to valid CSS at server side.
[38]	EQCSS	Invalid	Viewport only	Dynamic	
[44]	Element Media Queries	Invalid	Non-void elements	Dynamic	Flow-based element resizing detection.
[33]	Localised CSS	Invalid	Arbitrary elements	Dynamic	Polling-based element resizing detection.
[35]	Grid Style Sheets 2.0	Invalid	Arbitrary elements	Dynamic	Using the Cassowary constraints solver.
[48]	Class Query	Valid	-	Static	Writes media queries to style on load.
[47]	breakpoints.js	Valid	Viewport only	Dynamic	
[42]	MediaClass	Valid	Viewport only	Dynamic	Inline JS syntax.
[41]	ElementQuery	Valid	Viewport only	Dynamic	
[40]	Responsive Elements	Valid	Viewport only	Dynamic	
[43]	SickleS	Valid	Viewport only	Dynamic	Parses CSS.
[49]	Responsive Elements	Valid	Viewport only	Dynamic	
[37]	breaks2000	Valid	Viewport only	Dynamic	
[45]	eq.js	Valid	Viewport only	Dynamic	
[34]	Element Queries	Valid	Non-void elements	Dynamic	Object-based element resizing detection.
[46]	CSS Element Queries	Valid	Non-void elements	Dynamic	Scroll-based element resizing detection.
[39]	Selector queries and responsive containers	Valid	Arbitrary elements	Dynamic	Polling.

Table 6.1. Classification of related element queries libraries.

CSS features implies that it is possible to implement a solution to element queries that does not require any changes to the HTML, which may be preferable since all styling then can be written in CSS (which is the purpose of CSS). However, there are numerous drawbacks with libraries that require 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 understands the CSS directly when it has been retrieved. Server-side compilation implies that the layout of the page cannot be changed at runtime and is therefore only useful for static pages. By instead having the compilation step at runtime, dynamic layouts can be used since element queries may be re-evaluated on layout changes. However, 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 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 its 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 needs to parse the CSS added to the DOM, which means that the styles of the page is parsed twice.

- The custom parser engine needs to be included in the page, which implies a larger library script size.

Second, by not conforming to CSS standards many tools such as preprocessors, validators and linters are no longer compatible. Also, editors and other code-displaying tools are not able to understand the syntax and are therefore not able to highlight the code properly. It should be noted that it is in some cases possible to create plugins to such tools to extend their capabilities.

Third, The custom API and parser needs 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.

Resize detection As shown in table 6.1, most libraries simply observe the viewport resize event, which may be enough for static pages. However, observing element resize events is desired for pages that change layout during runtime. The libraries [33, 39, 44, 35, 34, 46] have in common that they observe elements for resize events. The libraries [33, 39] use polling to detect element resize events, and therefore support arbitrary element resizing detection. However, as presented in Section 4.4.2, polling is undesired.

The three remaining libraries uses three different injection solutions, as described in Section 4.4.2, to detect element resize events; [34] uses the object-based solution, [44] uses the flow-based solution, and [46] uses the scroll-based solution. According to the evaluation in Section 5.2, all three libraries are less performant than the element resizing detection system used in ELQ. Also, recall from Section 4.4.2 that the flow-based and scroll-based solutions both have limitations and bugs.

Constraint-based approach Shortly after the introduction of CSS by the W3C, a proposal for Constraint Cascading Style Sheets (CCSS) was submitted as a more general and flexible alternative to CSS [5]. As the name suggests, the idea of CCSS is to layout documents by constraints that would imply an unifying implementation mechanism (i.e., using a constraint solver). The constraint-based approach provides extended features and reduced complexity compared to CSS according to the authors. The Cassowary[1] constraint solver was used (among other tools) to solve the constraints for the demonstration implementation of CCSS. In 2011, Apple released their Auto Layout[2] technology that uses a similar constraint-based approach. Apple's implementation also uses the Cassowary algorithms to solve the

[1]Cassowary is an incremental constraint solving toolkit that solves systems of linear equalities and inequalities. See `http://sourceforge.net/projects/cassowary/` for more information.

[2]For more information about Apple's Auto Layout, see `https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG/Introduction/Introduction.html`.

constraints. The Grid Style Sheets library [35] builds upon the ideas of CCSS and uses a JavaScript port[3] of Cassowary to solve the constraints at runtime. The library also adapted Apple's Visual Format Language (VFL) to specify layout constraints. While not directly offering element queries, the library enables the possibility to conditionally style elements by element criteria and thus makes it a good candidate to solve the problem of responsive modules. However, the library has two major issues: performance and browser compatibility [57]. One approach to resolve both issues is to precompute the layout in a compilation step at the server. However, as earlier stated, precompiling styles implies static layouts. There are other approaches discussed that would increase the performance, but they also limit the dynamism of the page layout. Additionally, the library evaluates all elements on DOM mutations using the `MutationObserver` API, which does not detect all element resize events as discussed in Section 4.4.2.

[3]See `https://github.com/slightlyoff/cassowary.js` for the JavaScript port of Cassowary that is used by the Grid Style Sheets library.

Chapter 7

Discussion

It has been identified that requirements and usages of element queries vary from case to case, and it is therefore beneficial to provide a plugin-based library so that developers can tailor their custom solutions by choosing which plugins to use. Different parts of the library have also been shown to have different development rates, which also is an argument in favor to having the library plugin-based. Two plugins were developed that enables adequately advanced features for the general use case (as described in Section 3.1) that satisfies goals 1d and 1c. The `elq-breakpoints` plugin is the actual entity that directly enables element queries and defines the API to be used. A key restriction was made to the API in order to satisfy goal 4b, 2b, 2a and 4c: it must conform to the current web standard, and must therefore not require invalid syntax. This way preprocessors, validators, tools and editors can be embraced and used in tandem with the library instead of shutting them out. This design decision has proved to be very valuable when performing the altering of Bootstrap as described in Section 5.1. If the plugin would require invalid syntax it would not be as easy to alter the Boostrap LESS files, since the LESS preprocessor would not understand the syntax. Further, a lot of performance and compatibility issues have been avoided by not having to parse any CSS at runtime. Since Bootstrap was successfully altered to use element queries in ~50 changed LESS lines and a few lines added JavaScript, the technical goals 2a and 4c are considered to be fulfilled. In order to satisfy goal 2c a simple runtime cycle detection subsystem was implemented. It is guaranteed to detect cycles, as long as plugins registers element style state changes to the subsystem.

Element resizing detection The perhaps most difficult goals to achieve was 1a and 3a that requires the library to be both automatic (i.e., it should evaluate element queries on element resize events) and performant. Detecting element resize events in a performant way has been shown to be problematic since there is no standardized way of doing so. No related library conforms to both goals, as related libraries either do not observe element resize events or do so in a non-performant way. A heavily optimized solution to detecting element resize events has been developed in

order to enable ELQ to support both goals. It is of course ambiguous if the solution is performant enough to conform to goal 3a, as the optimized solution still impacts the performance in some browsers (e.g., FireFox and Safari). At least, the solution performs significantly better than the existing solutions as used by related libraries. Such good performance was achieved by batch processing all DOM interactions, by using the leveled batch processor subsystem that has been developed to be used by ELQ. Both the element resize event detection system and the leveled batch processor was decided to be released as two independent open source projects[1], since they are general enough to be used in other projects than ELQ.

Drawbacks Unfortunately, all JavaScript element queries libraries have inherent drawbacks. The perhaps most imminent drawback is that element queries are not evaluated when layout engines render the page for the first time, as layout engines usually renders a speculative state of the page before all content has been downloaded and parsed (e.g., external scripts). Browsers behave a bit different regarding the speculative rendering, but at least Blink, Gecko and WebKit generally renders the page before all external scripts have been executed. A naive solution to this could be to embed all external scripts inline with the HTML, which would keep the layout engine to perform the speculative rendering since external scripts are not fetched. However, layout engines may halt the parsing and evaluation of script tags in order to perform the speculative rendering. A way to guarantee that the element queries are evaluated before the first page render is to place the scripts in the document head. However, evaluating element queries synchronously in the head of a document is impossible, since the layout engine have yet to parse and construct the document body DOM. Further, element queries requires the layout engine to perform a layout in order to be able to get the final size of elements. When performing a layout, render engines usually take the opportunity to render the new state of the page. So, a flash of invalid style at page load is inevitable (and is also the case with all related libraries that evaluate element queries at runtime). For instance, the ELQ Bootstrap example documentation page would on load be rendered as it should be on small screens, since the small layout is the default that it falls back to if no queries match. Such flash could be avoided by rendering a white rectangle over the viewport and remove it when the page has been rendered correctly.

Another drawback that all element queries libraries that do not parse CSS have in common is that the breakpoints of elements are defined in either JavaScript or HTML. It is desired to define the breakpoints in CSS (since the purpose of CSS is to define the style of a document), as with case with media queries. Due to both performance and compatibility problems, parsing of CSS at runtime is not done by ELQ. It was decided that the negative outcomes of doing that outweighs the positive. One compatibility issue is that browsers generally do not support scripts to access external style sheets served from another domain unless specified by both

[1] See `https://github.com/wnr/element-resize-detector` for the element resize event detector and `https://github.com/wnr/batch-processor` for the leveled batch processor.

the server and client. Since ELQ is plugin-based a plugin could easily be crafted that adds such functionality if desired.

ELQ and related work ELQ and its plugins provide a low level API for building responsive design frameworks and applications by enabling the user to conditionally style elements with valid CSS. By not requiring invalid CSS, the library can be easily incorporated into existing projects. Additionally, CSS tools can advantageously be used in tandem with the ELQ syntax — something that is not possible with related libraries that require invalid CSS. The mirror plugin of ELQ enables developers to overcome some CSS limitations, which has been proven to be important when writing responsive modules that should not be limited to a predefined HTML structure (as shown in Section 5.1). Related libraries only support such feature by using invalid CSS. By using related libraries that require invalid CSS, the Bootstrap framework would not be possible to alter with as few altered lines of code as achieved by ELQ since the LESS preprocessor would not understand the invalid CSS syntax. Also, such related libraries impose a significant performance impact since the responsive elements need to be evaluated at runtime in Bootstrap.

Since ELQ is plugin based, different plugins may be developed for different use cases without bloating the library API or decreasing the overall performance. Related libraries have different advantages and limitations. For instance, static pages may prefer a solution that does not observe element resize events (as it is not needed for static pages). Related libraries that are limited and not able to detect element resize events are then suitable as the performance penalty of doing so is avoided. Fortunately, since ELQ is plugin based the same behavior as such related libraries can be achieved with plugins (or options) and may therefore still be the preferred library to use. An advantage of using the same library for multiple use cases is that developers only need to learn one library and that it can easily be extended when application requirements change.

A major difference to related libraries is that ELQ is powered by a highly optimized subsystem for detecting element resize events, which provides a significant speedup to related libraries that also detect element resize events. The subsystem used by ELQ has been showed to have the same or better performance as the other subsystems in FireFox and Safari. It has been shown that the optimized subsystem performs 32-fold (for the object-based solution) or 13-fold (for the scroll-based solution) faster than the subsystems used by related libraries in Chrome when preparing 500 elements to be detectable. Additionally, some solutions used by related libraries suffer from issues that ELQ does not as described in Section 4.4.2.

Chapter 8

Conclusions

As the number of combinations of screen sizes, input mechanisms, etc., increases it is important to develop responsive applications so that they can adapt to different end-user devices and usages. Modular development is also desired for various positive effects. The problem is that modules cannot be responsive, due to CSS media queries not being able to conditionally style elements by element criteria. It has been shown that element queries solve the problem of responsive modules, and is therefore an important missing feature of the web as a platform.

A plugin-based element queries JavaScript library, named ELQ, has been developed that contains subsystems to be used by element queries plugins. Examples of such subsystems are a leveled batch processor, an element resize detector, and a cycle detector. Two plugins have been developed: `elq-breakpoints` that is used to conditionally style elements by element criteria, and `elq-mirror` that is used to overcome some limitations of CSS. The library and the plugins have been designed to conform to the standards and specifications of the web languages, which has been valuable when integrating the solution into existing projects.

Recall that the main objective of this thesis is to design and develop a third-party non-native library that enables element queries in both modern and legacy browsers. The scientific question is if it is possible to construct such library that has high reliability, adequate performance, and enough features to support advanced compositions of responsive modules.

The empirical evaluation of ELQ shows that it is both reliable and provides enough features for advanced web applications such as the Bootstrap framework and documentation page. Adequate browser compatibility is also provided, since ELQ supports the same set of browsers as the Bootstrap framework does. ELQ has been shown to have overall significantly better performance than related libraries. Some related libraries achieve the same performance as ELQ in Safari, but not in other browsers. In Chrome, the speedup is 32-fold when operating on 500 elements compared to related libraries that use the object-based element resize detection subsystem. Also, the evaluation shows that the library is easy to integrate into existing projects since only ~50 lines out of ~8500 lines of style code was needed

to be altered for Bootstrap to be fully based on element queries. Therefore, the scientific question is answered in the affirmative.

Since ELQ is released as open source, developers can now use element queries to create responsive modules as ELQ is performant, compatible, flexible, powerful, and extensible. No other related library simultaneously provides all of these desirable properties.

Chapter 9

Future work

It would be beneficial to further investigate possible use cases in order to evaluate if the current plugins are sufficient. Perhaps new plugins would need to be developed, or existing APIs extended. This is needed in order to be sure that goal 1b is satisfied.

Extending the mirror plugin In order to allow more advanced element query constellations, `elq-mirror` could be extended to enable mirror elements to target arbitrary elements (instead of just the nearest `elq-breakpoints` ancestor element). This way, it would be possible to write element queries against children, siblings, etc. This would also be a key feature in order to further satisfy goal 1b.

Prolyfill To satisfy goal 4d an element queries prolyfill plugin could be created that would parse custom at runtime CSS to simulate native element queries. It would then use an imaginary syntax of element queries, which could only be guessed at this point of time. Before any advancement has been made by the RICG regarding the syntax, the pseudo-syntax presented in Section 3.1 is a good start.

Static layout More research can be done regarding static pages that wish to use element queries, since the main focus of this thesis has been dynamic pages. It is possible that the best approach to static pages is to parse custom CSS at server side in order to generate static media queries. Media queries are at the moment the most performant way of applying conditional styles by viewport size, which possibly is the only factor to static page layout. However, it is uncertain if the viewport size is the only layout factor to static pages as browser and user styles might affect pages in a way that would imply that a layout change is desired without affecting the viewport size.

Performance As of now, different subsystems of ELQ are forced to create their own batch processor, which is favorable in some cases. However, it can also be favorable to have a shared instance of a batch processor in order to avoid unnecessary

layouts between subsystems. For instance, the element resizing detection subsystem and the `elq-breakpoints` plugin both use a batch processor to avoid layout thrashing. However, since they process their batches independent of each other, the element resizing detection subsystem forces a layout before the plugin batch processor is invoked. This means that two layouts are performed, when in theory only one is needed. This could be solved by sharing a batch processor between them so that read, write, and force operations are executed synchronized with each other.

Flash of invalid layout It would be possible to prevent the flash of invalid layout during page reloads, by saving all element styles states to a persistent local browser storage. When the page reloads, a script can be executed that reads the element styles states of the local storage and applies the breakpoint classes to all elements before the first layout. This way, all elements would have the right classes before a layout is performed by the layout engine and thus resulting in an instant valid page layout. However, this still would not avoid the invalid layout when the page is loaded for the first time. Also, it is unclear if such approach would work as it might be hard to control script execution to happen before the first layout.

Parsing CSS Event though there are several issues with runtime parsing of CSS, it is probably beneficial in some cases. Recall from Section 4.4.2 that CSS selectors may unintentionally style the injected elements of the element resizing detection subsystem. By parsing CSS it would be possible to warn developers of such selectors. The feature could be valuable in the development phase of applications, but should probably be disabled in production due to performance reasons. Users that value true separation of style and HTML could also benefit from parsing CSS, as the breakpoints of all elements then could be deduced by considering the element queries in the CSS.

Element resizing detection As shown in Section 5.2, different solutions of detecting element resize events are preferred for some browsers. The ELQ element resize detection subsystem could easily be extended to detect the layout engine context at runtime and then choose a specific solution for that engine. This way, the best solution could be used for each layout engine.

Bibliography

Books

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

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

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

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

Articles

[5] Greg J Badros et al. "Constraint cascading style sheets for the web". In: *Proceedings of the 12th annual ACM symposium on User interface software and technology*. ACM. 1999, pp. 73–82.

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

[7] 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.

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

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

[10] Lingjun Fan et al. "Optimizing web browser on many-core architectures". In: *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on*. IEEE. 2011, pp. 173–178.

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

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

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

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

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

[16] 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).

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

[18] 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.

Master's theses

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

Material of the W3C

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

[21] 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).

[22] World Wide Web Consortium et al. *Document Object Model (DOM) Level 3 Core Specification*. Jan. 2004. URL: `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/` (visited on 03/05/2015).

[23] World Wide Web Consortium et al. *Document Object Model (DOM) Level 4*. Apr. 2015. URL: `http://www.w3.org/TR/DOM/` (visited on 06/04/2015).

[24] World Wide Web Consortium et al. *Document Object Model Events*. 2000. URL: `http://www.w3.org/TR/DOM-Level-2/events.html` (visited on 03/14/2015).

[25] World Wide Web Consortium et al. *Media Queries*. June 2012. URL: `http://www.w3.org/TR/css3-mediaqueries/` (visited on 05/19/2015).

[26] World Wide Web Consortium et al. *Selectors Level 3*. Sept. 2011. URL: `http://www.w3.org/TR/css3-selectors/` (visited on 05/19/2015).

[27] World Wide Web Consortium et al. *Visual formatting model*. June 2011. URL: http://www.w3.org/TR/CSS21/visuren.html (visited on 06/10/2015).

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

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

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

[31] *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).

[32] *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).

Related Libraries

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Online resources

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

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

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

[53] 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).

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

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

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

[57] *Elemenet queries with precompilation*. An issue that discusses the problems of dynamic layout when using Grid Style Sheets. URL: https://github.com/gss/engine/issues/178 (visited on 06/08/2015).

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

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

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

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

[62] 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).

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

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

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

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

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

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

[69] 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).

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

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

[72] 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).

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

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

[75] *overflowchanged event (deprecated)*. URL: https://www.chromestatus.com/feature/5242458724106240 (visited on 06/04/2015).

[76] 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).

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

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

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

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

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

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

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

[84] 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).

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

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

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

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

[89] *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. 5, 18, 20, 21, 22, 38, 42, 45, 46, 47, 48, 50, 51, 69, 73, 75, 84

Blink The open source layout engine used by Chrome and Opera. The engine is a recent fork of WebKit. 15, 48, 51, 62, 70, 85, 99

Bootstrap The most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web. See `http://getbootstrap.com/about/` for more information. 3, 4, 5, 26, 55, 56, 57, 58, 59, 69, 70, 71, 73, 95

browser Client application for navigating the World Wide Web (WWW) and displaying web pages. More formally known as a user agent. 2, 3, 5, 7, 8, 9, 10, 14, 15, 17, 18, 23, 25, 27, 28, 30, 36, 37, 41, 42, 47, 48, 49, 50, 52, 60, 62, 66, 69, 70, 73, 75, 76, 84, 85, 89, 92, 93, 94, 99

CSS3 The third revision of the CSS standard. 9, 10

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, 12, 14, 15, 16, 17, 18, 25, 30, 31, 32, 47, 48, 49, 50, 52, 66, 67, 70, 83, 84, 85, 89, 91, 92, 93

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. 1, 2, 10, 12, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26, 27, 28, 29, 30, 31, 32, 36, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 55, 57, 58, 60, 62, 60, 62, 65, 67, 70, 71, 73, 75, 76, 84, 85

ELQ The name of the third-party JavaScript element queries library that is the main result of this thesis. See Chapter 4. 1, 35, 39, 40, 41, 42, 43, 46, 47, 50, 51, 52, 55, 57, 58, 59, 60, 62, 67, 69, 70, 71, 73, 74, 75, 76, 95

encapsulated An encapsulated module handles its task without any help from the user of the module. 2, 5, 7, 11, 12, 55

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, 57, 83, 85, 94

Gecko The open source layout engine used by FireFox. 15, 48, 51, 62, 70, 99

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. 8, 11, 14, 89, 92, 93

JavaScript Native browser script language. Web documents often include JavaScript to make the documents more dynamic and interactable. 3, 4, 9, 11, 12, 14, 15, 16, 20, 21, 22, 28, 29, 32, 35, 36, 47, 49, 52, 57, 58, 65, 67, 69, 70, 73, 83, 84, 85, 87, 89

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. 4, 7, 14, 20, 21, 38, 46, 47, 50, 51, 60, 75

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, 5, 7, 14, 15, 16, 18, 20, 21, 22, 23, 25, 27, 28, 32, 35, 48, 49, 50, 51, 62, 66, 70, 76, 83, 84, 85, 93, 94, 99

LESS A popular CSS preprocessor. See `http://lesscss.org/` for more information. 56, 57, 58, 69, 71

media queries The CSS feature of specifying conditional style rules for elements by conditions such as the viewport size. 1, 2, 3, 5, 10, 12, 25, 52, 55, 56, 57, 65, 70, 75

native Refers to APIs and systems implemented and provided by browsers. Such APIs and systems are often described by standards specifications. 1, 2, 3, 5, 7, 13, 23, 25, 27, 32, 36, 48, 66, 73, 75, 84

render tree Used by layout engines as the model for the visual representation of the document. 15, 16, 17, 18, 20, 49, 51

responsive Elements and content of the document can detect size changes and act accordingly. Usually a restructure of content is performed at certain breakpoints. 1, 2, 3, 4, 5, 7, 10, 12, 13, 25, 26, 35, 48, 55, 56, 57, 58, 59, 67, 71, 73, 83, 95

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. See `http://gs.statcounter.com/` for more information. 9, 99

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, 5, 8, 13, 32, 65, 73, 83

Trident The closed source layout engine used by Internet Explorer. 15, 99

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. 1, 2, 9, 10, 12, 13, 18, 25, 26, 31, 32, 55, 58, 59, 65, 67, 70, 75, 84, 95

web Short for the world wide web. 1, 2, 3, 4, 7, 8, 9, 11, 12, 13, 23, 25, 26, 36, 55, 65, 69, 73, 83, 84, 89, 90, 91, 92, 93, 99

WebKit The open source layout engine used by Safari. Google's Blink layout engine is a recent fork of WebKit. 15, 48, 51, 62, 70, 94, 99

WYSIWYG A classification that ensures that text and graphics during editing appears close to the result. 88, 93

Acronyms

AJAX Asynchronous JavaScript and XML. 9

API Application Program Interface. 2, 3, 4, 5, 9, 12, 18, 27, 32, 35, 36, 37, 39, 41, 42, 46, 48, 51, 52, 55, 67, 69, 71, 75, 84, 85

ARPANET Advanced Research Projects Agency Network. 90, 91

CCSS Constraint Cascading Style Sheets. 67

CERN Conseil Europeen pour la Recherche Nucleaire. 91, 92

CGI Common Gateway Interface. 8

CPU Central Processing Unit. 23

CSNET Computer Science Network. 91

CSS Cascading Style Sheets. 1, 2, 3, 5, 8, 10, 12, 15, 16, 17, 18, 23, 25, 26, 30, 31, 32, 35, 36, 37, 44, 49, 52, 55, 56, 65, 66, 67, 69, 70, 71, 73, 75, 83, 84, 85, 89

DARPA Defense Advanced Research Projects Agency. 92

DHTML Dynamic HTML. 8

DOD Department Of Defense. 90

DOM Document Object Model. 15, 16, 17, 18, 20, 21, 22, 38, 48, 49, 51, 52, 66, 67, 69, 70, 83

FTP File Transfer Protocol. 90, 91, 92

GUI Graphical User Interface. 9, 93

HTML HyperText Markup Language. 8, 9, 11, 12, 14, 15, 18, 31, 32, 36, 42, 44, 55, 57, 65, 70, 71, 76, 83, 84, 87, 89, 92

HTTP HyperText Transfer Protocol. 83, 89, 92, 93

ICANN Internet Corporation for Assigned Names and Numbers. 89

ICCC International Computer Communication Conference. 90

IP Internet Protocol. 89, 90, 91

ISP Internet Service Provider. 89

KDE K Desktop Environment. 94

MILNET Military Network. 91

NCSA National Center for Supercomputing Applications. 8, 93

NSF National Science Foundation. 91

NSFNET National Science Foundation Network. 91

OS Operating System. 8

RICG Responsive Issues Community Group. 3, 13, 26, 32

RWD Responsive Web Design. 10

TCP Transmission Control Protocol. 89, 90, 91

URL Uniform Resource Locator. 14, 83

US United States. 90, 91

VFL Visual Format Language. 67

W3C World Wide Web Consortium. 1, 3, 8, 9, 13, 32, 67, 92

WHATWG Web Hypertext Application Technology Working Group. 9

WWW World Wide Web. 83, 89

WYSIWYG What You See Is What You Get. *Glossary:* WYSIWYG, 93

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 [81, 9, 13, 20, 73].

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 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).

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 [16, 17, 68, 64].

In the early 1960's *packet switching* was being researched, which is a prerequisite 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 [60, 88, 63, 83, 64].

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.

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 [88, 58, 15, 80, 70, 74, 67, 78, 86, 51].

The first web browser ever made was created in 1990 and was called World-WideWeb (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 Full-scale Bootstrap figures

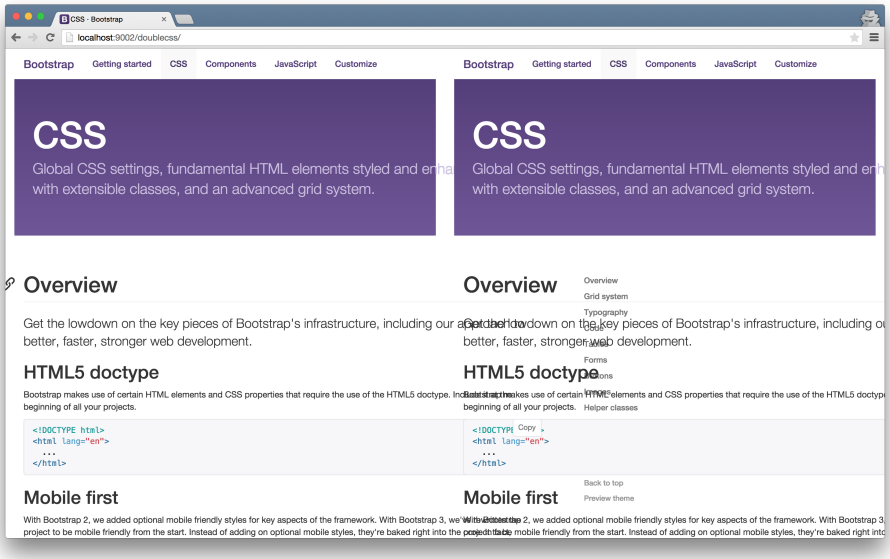

Figure B.1. Double column documentation page powered by Bootstrap showing the header section of the page.

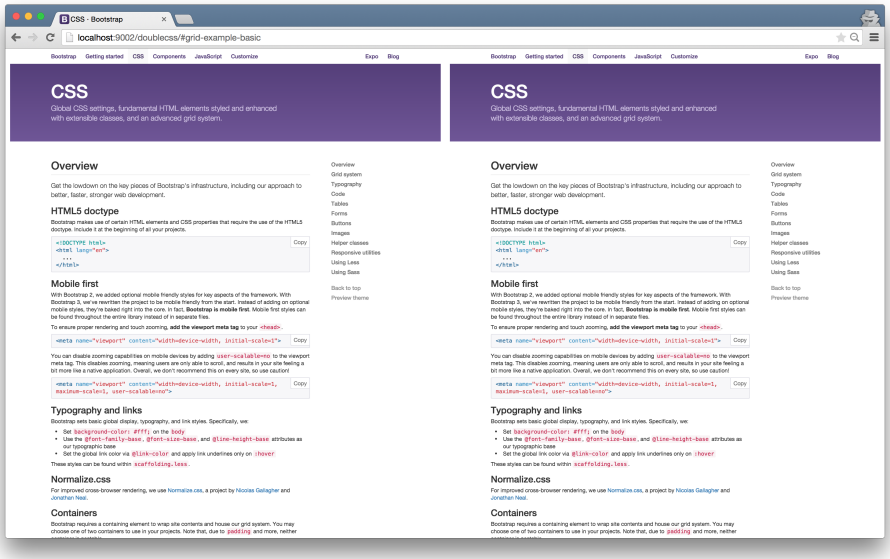

Figure B.2. Double column documentation page powered by Bootstrap with showing the header section of the page. The viewport is big enough to give each column sufficient space.

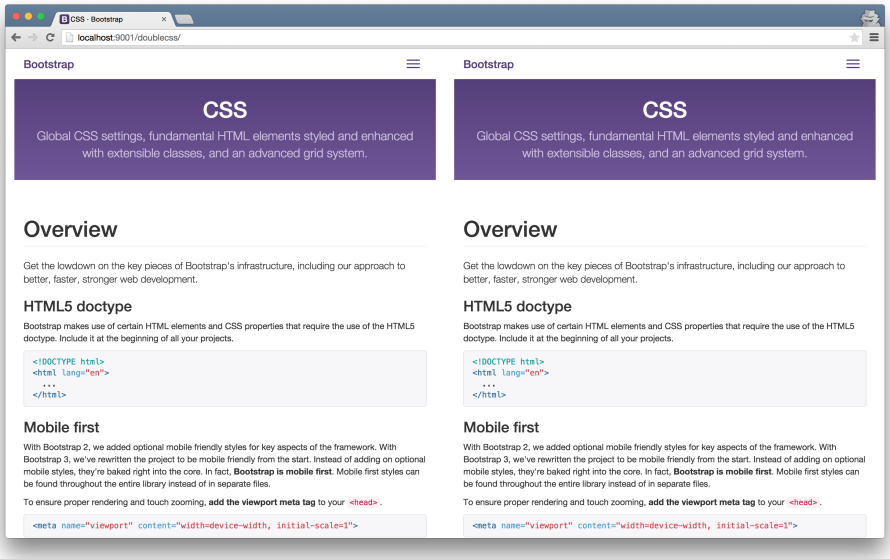

Figure B.3. Double column documentation page powered by ELQ Bootstrap showing the header section of the page.

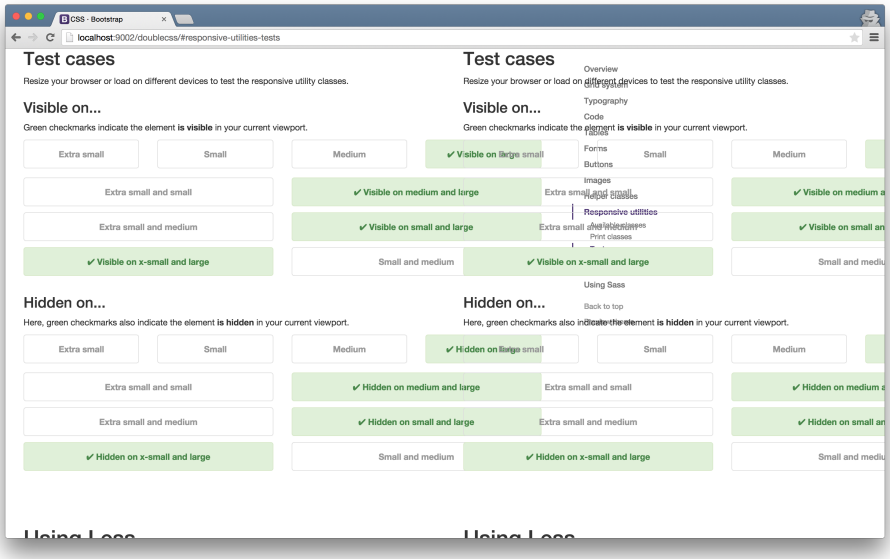

Figure B.4. Double column documentation page powered by Bootstrap showing the responsive classes matrix section of the page.

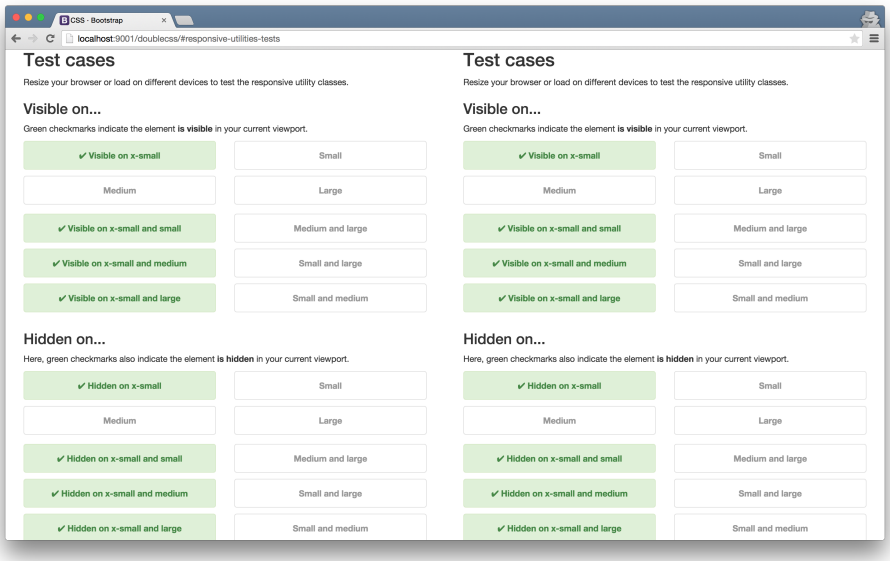

Figure B.5. Double column documentation page powered by ELQ Bootstrap showing the responsive classes matrix section of the page.

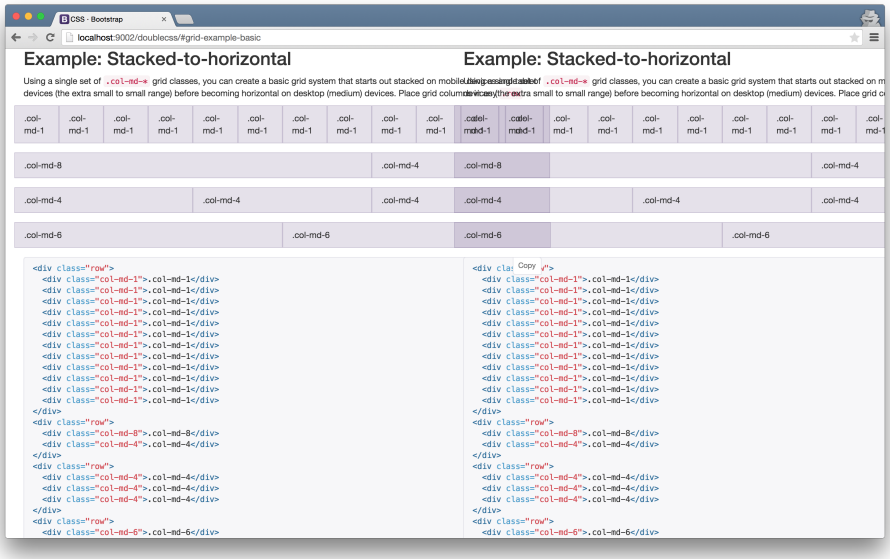

Figure B.6. Double column documentation page powered by Bootstrap showing the grid section of the page.

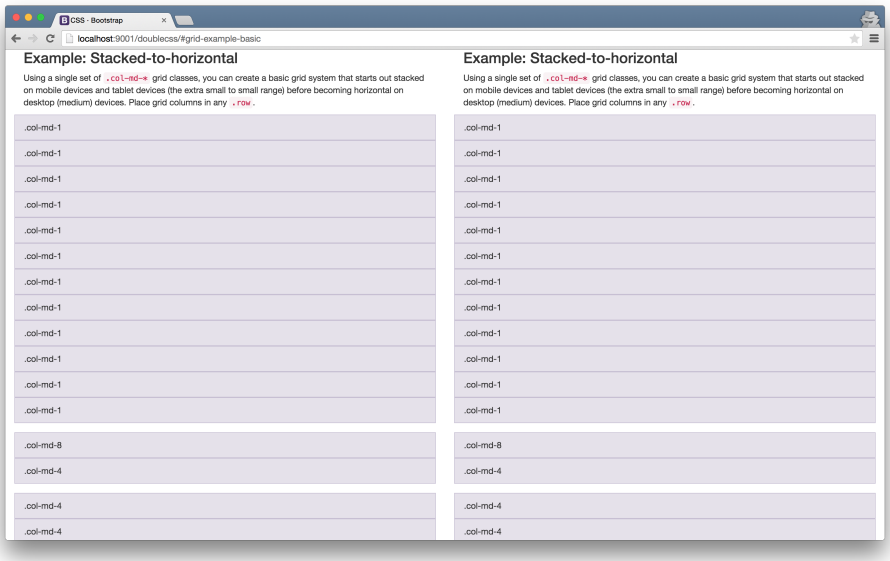

Figure B.7. Double column documentation page powered by ELQ Bootstrap showing the grid section of the page.

B.2 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 [51]. 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 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%.

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