KTH Computer Science
and Communication

Modular responsive web design

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

LUCAS WIENER
lwiener@kth.se

Master's Thesis at CSC

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

TRITA xxx yyyy-nn

Abstract

Abstract goes here.

Referat

Modulär responsiv webbutveckling

Sammanfattning ska vara här.

Contents

Todo list

Chapter 1

Introduction

Container or parent element? Using both now.

Rename framework to library?

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

Word says that all "their own" is wrong.

Change all wikipedia sources to the real sources

1.1 Targeted audience

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

Philipp wanted this to be more general.

1.2 Problem statement

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 media queries are no longer applicable.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

The main international standards organization for the web, World Wide Web

Is W3C okay with this?

Consortium (W3C), has unofficially stated that such feature would be infeasible to implement. Standards organizations 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. While awaiting a native solution it is up to the developers to implement this feature as a third-party solution. Efforts have been made to create a robust third-party solution, with moderate success. Since all current third-party solutions have shortcomings, there is still no de facto solution that developers use and the problem remains unsolved. Some problems with implementing element queries natively in CSS are:

- **Circularity:** If style rules can be applied by criteria of other elements, it is possible to create cyclic rules (infinite loops of styling). Some cyclic rules might be possible to detect during CSS parsing, but there are so many combinations of style properties that could result in cyclic rules that it will add a lot of complexity to the language, both for implementers and users. Also, it has been shown that some cyclic rules are impossible to detect at parsetime, due to being dependent on runtime factors.

- **Performance:** Layout engine vendors are interested in performing selector matching and layout computations in parallel to achieve better performance. The research front of browsers have shown successful ways of parallelising the layout engine. Element queries implies some undesired limitations to the layout engines. With element queries, layout engines need to first compute the layout of all elements in order to decide which selectors would conform to the element query conditions and then do a new layout computation with the new style rules activated, and so on until a stable state has been reached. Far worse, since selectors would depend on layout style, style and layout computations become very to parallelise.

1.3 Objective

The main objective of this thesis is to design and develop a third-party implementation of element queries. To do this, it is needed to research existing solution approaches in order to understand and analyze the advantages and shortcomings of the approaches. It is also necessary to be aware of the premises, such as browser limitations and specifications that need to be conformed. In addition, research will be done about the problems of implementing element queries natively, to get a deeper understanding of how an official Application Program Interface (API) could look like. Solving the main problem will require research and empirical studies of many subproblems. Examples of such subproblems that needs to be addressed are:

- How should circularity be handled? Should it be detected at runtime or parsetime, and what should happen on detection?

- How can one listen to element size changes without any native support?

- How can a custom API be crafted that will enable element queries and still conform to the CSS specification? Is it possible to create an API that feels natural to web developers and works in tandem with other tools and frameworks?
- If a custom API is developed, how would one make third-party modules (that uses media queries) work without demanding a rewrite of all third-party modules?
- Is it possible to solve the problem with adequate performance for heavy applications?
- How can adequate browser compatibility be achieved?

Remove me?

The scientific question to be answered is if it is possible to solve the problem without extending the current web standard. The hypothesis is that the problem can be solved with high reliability and adequate performance by developing a third-party implementation. A reliable implementation should also enable existing responsive components to react to a specified criterion (parent container size for example) with no modifications to the components. The goal of this thesis should be considered fulfilled if a solution was successfully implemented or described, or if the problems hindering a solution are thoroughly documented.

Remove me?

Is it really interesting for the reader to know when it was fulfilled or not? I know at this point of time that it has been fulfilled, so the other case is no more relevant.

1.4 Significance

Many frameworks and techniques are being used in web development to keep the code from becoming an entangled mess. Creating modules facilitates the development and increases the reusability. Unfortunately, it is currently impossible to create self-contained responsive modules since conditional styles cannot be applied to elements by element size criterion. Without element queries, responsive modules force the application to style them properly depending on the viewport sizes, which defeats the purpose of modules. Modules should be self-contained and may not require the user to perform some of the module logic. Another option would be to make modules context-aware so they can style themselves according to the viewport, but then they would not be reusable (since they assume a specific context). Also, changes to the application layout would then require to rewrite the media queries of the modules to take the new layout into account. Clearly, no sound option exists for having responsive modules.

The last couple of years a lot of articles have been written about the problem and how badly web developers need element queries. As already stated, third-party implementation efforts have been made with moderate success. W3C receives requests and questions about it, but the answer seems to be that no solution will be provided in the near future. The Responsive Issues Community Group (RICG) have started an initial planning regarding element queries. However, things are moving slow and a draft about element queries use cases is still being written.

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

1.5 Delimitations

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

1.5.1 What will be done

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

1.5.2 What will not be done

- No efforts will be made to solve the problems accompanied with a native solution.
- No API or similar will be designed for a native solution.
- User interface design will not be addressed, other than necessary for understanding the problem.

1.6 Outline

This thesis is divided into four parts. The first part *I Background* presents some history of the web platform and describes concepts needed to understand the problem. The background part starts with the history of browsers and information about market share between them followed by a brief history of the evolution of web development. After that, the responsive web design concept will be described along with why modularity is important in software development.

The second part *II Theory* presents theoretizations of the problem. It starts with describing a reference architecture of layout engines and the parts of layout engines that directly affects element queries will be described in detail. Here it

will also be presented why element queries impose problems to layout engines on a theoretical level. After that, a full description of element queries as a concept and a more in depth analysis of the problems will be given. Possible solutions to the given problems will also be presented followed by a fictional native API description based on probable design decisions.

The third part *III Third-party framework* presents the design and implementation of the solution as a third-party framework. Here possible approaches to solving the problem will be presented. Other implementations and solutions will be described and analyzed followed by a motivation of the chosen approach for the framework. Identified technical requirements will be given in order to motivate the framework architecture and API design. After that, interesting parts of the implementation will be presented.

Finish the Outline when the other parts have been written.

Maybe flesh this one out as it has been written.

Part I

Background

Chapter 2

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* [40, 11, 18, 7, 34].

Before addressing the birth of the web, it is necessary to define the meaning of the concepts *Internet* and *World Wide Web (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 made yet). Users are defined as humans having unrestricted access to the Internet. 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 billions.

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 a system for accessing interlinked hypertext documents and other media such as images and videos. 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 (images, videos, etc.). 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 HyperText Markup Language (HTML), and often includes CSS for styling and JavaScript for custom user interactions. As a complementary to HTML, the EXtensible Markup Language (XML) also exists with the purpose of describing data (as contrary to HTML which describes the *presentation* of data). To transfer the resources between computers the protocol HyperText Transfer Protocol (HTTP) is used. 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 2.3).

2.1 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* [25, 29, 28, 20].

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.

2.2 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* [14, 45, 19, 41, 20].

Recall from section 2.1 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 the 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 implemented. A browser and a web server had also been 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 that the web was to be free without any patents or royalties. At this time 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.

2.3 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* [45, 12, 24, 39, 33, 35, 26, 37, 44, 1].

The first web browser ever made was created in 1990 and was called WorldWideWeb (which was renamed to Nexus to avoid confusion). It was at the time the only way to view the web, and the browser only worked on NeXT computers. Built with the NeXT framework, it was quite sophisticated. It had a Graphical User Interface (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) were 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 National Center for Supercomputing Applications (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 NCSA for producing their own browser in 1994. Around the same time the leader of the team that developed Mosaic, Marc Andreessen, left 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 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.

Timeline image and text perhaps

Chapter 3

Web development

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

Philipp: Here you should provide an overview of the different sections of the chapter. "In section 3.1 we outline... In section 3.2 we...

What to write here?

3.1 From documents to applications

↪ *This section will describe the transition from browsers rendering simple documents to being hosts for complex applications. Since web development trends are not easily pinned to exact dates, this section will only present dates as guidance and should not be regarded as exact dates for the events. This section is a summary of* [43, 6, 4, 46, 21, 23].

As described in section 2.2, browsers initially were applications that displayed hypertext documents with the ability to fetch linked documents in an user friendly way. Static content were written in 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 W3C officially recommended CSS, which came to be the preferred way of styling web content. Since HTML is only a markup language it is not possible to

generate dynamic content, which was sufficient at the time HTML was only used for annotating links in research documents.

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

Around 1996, client side scripting was born. The term Dynamic HTML (DHTML) was being used as an umbrella term for a collection of technologies used together to make pages interactive and animated, where client side scripting played a big role. Examples of things that were being done with DHTML are; refreshing the pages for the user so that new content is loaded, give feedback on user input without involving the server and animating content. Plugins also started to exist during this time that enabled browsers to handle and execute embedded programs. *Java applets* and *Flash* are examples of browser embedded programs requiring browser plugins to execute. In order for users to view the content, they first need to download the Java runtime or Flash runtime on their computers. When the runtimes have been installed, the browser then executes the included Java applets or Flash in the web pages by using the external runtimes. This enabled web developers to use "real" programming languages. The drawback of this is that users first have to download the runtimes in order to run the programs embedded in pages. As native client side scripting matured and standardized into *JavaScript*, plugins slowly decreased in popularity. With the increase of smart devices (such as phones, televisions, cars, game consoles, etc.) that included browsers but limited third-party runtimes, plugins quickly went extinct. Today, web sites rarely uses plugins.

As JavaScript and HTML supported more features, websites turned into small applications with user sessions and rich GUIs. Still, parts of the applications were defined as HTML pages, fetched from the server when navigating the site. When the *XMLHttpRequest* API was supported in the major browsers, pages no longer needed to reload in order to fetch new content as XMLHttpRequest enabled developers to perform asynchronous requests to servers with JavaScript. This opened up for the Asynchronous JavaScript and XML (AJAX) web development technique which became a popular way of communicating with servers "in the background" of the page. Developers pushed browsers and HTML to the limit when creating applications instead of documents that they were originally designed for. In a W3C meeting in 2004 it was proposed to extend HTML to ease the creation of web applications, which was rejected. W3C was criticized of not listening to the need of the industry, and some members of W3C left to create the Web Hypertext Application Technology Working Group (WHATWG). WHATWG started working on specifi-

cations to ease the development of web applications which came to be grouped together under the name HTML5. In 2006, W3C acknowledged that WHATWG were on the right track, and decided to start working on their own HTML5 specification based on the WHATWG version. HTML5 is an evolutionary improvement process of HTML, which means that browsers are adding support as parts of the specification is finished.

A new era of APIs and features came along with HTML5, which truly enabled developers to create rich client side applications. CSS3 was also recently released which also included many new features. Using HTML5 and CSS3 developers could utilize advanced graphics programming, geolocation, cache storage, file system access, offline mode, and much more.

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

3.2 Responsive web design

↪ *A few years ago, web developers could make assumptions about the screen size of user devices. Since typically only desktop computers with monitors accessed web sites they were designed for a minimum viewport size. If the size of the viewport was smaller than the supported one, the site would look broken. This was a valid approach in a time when tablets and smartphones were unheard of. Today, another approach is needed to ensure that sites function properly across a range of different devices and viewport sizes. This section is a summary of* [30, 36, 32].

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 *agent string* is usually sent with the request to identify what kind of browser the user is using. By reading the agent string on the server-side, a mobile-friendly version of the site could be served if the user was sending mobile agent strings and the desktop version could be served otherwise. The mobile version would be designed for a maximum width, and the desktop would be designed for a minimum width.

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

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

With the release of CSS3 media queries new possibilities opened up. Media queries enabled developers to write conditional style rules by media properties such as the viewport size. See listing 3.1 for an example of how media queries can be used to style elements differently by 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 3.1. The above CSS styles the body of the website blue if the viewport is less or equal to 600 pixels wide, and yellow otherwise.

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

3.3 Modularity

↪ *As the web was a platform for hypertext documents that quickly transitioned to serving complex applications, few techniques existed for writing modular code. The last couple of years as applications grew bigger, techniques and frameworks 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* [27, 16, 17].

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 frameworks that helps dividing up the client code into modules. The ever so popular frameworks *Angular*, *Backbone*, *Ember*, *Web Components*, *Requirejs*, *Browserify*, *Polymer*, *React* and many more all have in common that they embrace coding modular components. Many of these frameworks 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) that has a single responsibility and ideally performs a single task, complexity can be reduced. Of course, splitting software up into modules alone does not help reducing complexity. The modules also need to be self-contained, which means that they work by themselved and do not require the user of the module to write its logic. Modules should also be loosely coupled, and may not make any assumptions about the context they will be used in. It is then possible to reason about them and change the functionality locally inside the boundary of a module.

With modularity comes positive side effects. Loose coupling between modules facilitates integration testing of each module, since it is then possible to test parts of the software in isolation and independent of the system as a whole. Developers can also work on different parts of the software in parallel without being affected by each other, as modules are not allowed to affect each other (other than configurable options or injected strategies). 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.

Part II

Theory

Chapter 4

Layout engines

↪ *Before diving into the theory of element queries, it is important to understand how layout engines work. Only the layout engine of the browser will be of importance, since element queries do not affect any other browser subsystem. This chapter will give a brief explanation of the layout flow that the engines generally perform. It will also cover some optimizations that are done and which parts of the layout process that can be done in parallel. The render process will not be described, as it is not relevant for element queries. This chapter is mainly based on* [15, 13].

Have an image about the layout flow somewhere?

Browsers are complex applications that consist of many subsystems. A reference architecture of browsers has been presented by [15], see figure 4.1. It is shown that the layout engine is located between the *browser engine* system and the network, JavaScript, XML parsing, 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 Unique Resource Identifiers (URIs) of content that should be fetched and rendered. Additionally, browser engines usally provide the layout engine with layout and rendering options such as user preferred font size, zoom, etc. The main

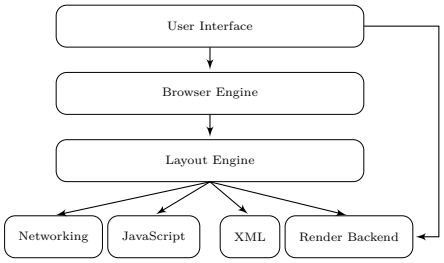

Figure 4.1. Reference browser architecture. The data persistence system, used by the browser engine and the user interface, has been omitted.

responsibility of the layout engine is to render the current state of the fetched hypertext document. Since documents may (and often do) change dynamically after parsetime it is important to keep in mind that the job of the layout engine is continuous, and is not a one time operation. The parsing of HTML is also done by the layout engine (and not in the XML subsystem) since HTML is less strict and reentrant. In short, layout engines perform four distinct tasks:

Explain DOM before this?

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

See section 4.1 and section 4.2 for a more in depth explanation of DOM and render trees. Modern browsers can display multiple pages at the same time (by using tabs, multiple windows or frames) where each page typically has an instance of the layout engine in a separate process. There are four layout engines that the major browsers use, as presented in table 4.1. Since Blink is a recent fork of WebKit they can be grouped as WebKit-based layout engines. This results in three different layout engines to consider; WebKit, Gecko and Trident. Due to Trident being closed source, only the WebKit based layout engines and Gecko will be considered from now on.

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

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

Table 4.1. The major layout engines and browsers with market shares. The table to the right has grouped together Blink into WebKit since it is a recent for of WebKit. See section A.3 for more information how the market share data was gathered.

4.1 Constructing DOM trees

↪ *Here a brief explanation of DOM trees will be given, and how they are constructed. In order to construct a DOM tree, the content need to be parsed which also will be covered briefly. This section is mainly based on* [9, 13, 2, 10].

The DOM is a convention for representing and interacting with objects in XML-based documents (such as HTML). The DOM provides an interface for programs (JavaScript in the browser) to access and mutate the structure, style, and content of documents. Each node of the document is a DOM object, and therefore the whole

document is converted to a tree structure of DOM nodes which is referred to as the DOM tree.

Layout engines construct DOM trees by parsing HTML with any included CSS and JavaScript. Parsing of HTML is not an easy task, partly because it is expected (alhough not required) of layout engines to be forgiving of errors (such as handling malformated HTML). As CSS and JavasScript are stricter, their grammar can be expressed in a formal syntax and can therefore be parsed with a context free grammar parser. Another big quirk to parsing HTML is that it is reentrant, which means that the source may change during parsing. Script elements are to be executed synchronously when it is occurred 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 nodes in the DOM tree are not affected by the CSS cascading, 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 a special function needs to be called (`getComputedStyle` in JavaScript). 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 tree nodes and therefore either the parsing of HTML needs to be halted in order to fetch and parse CSS when it occurs, 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 the 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.

Philipp: Maybe check the official HTML specs about this.

Philipp: Yes, a picture of the DOM corresponding to some HTML would be good.

code example of reentrant html?

Figure of how the DOM tree looks like? Maybe example HTML and DOM tree figure.

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

4.2 Render trees

↪ *When the DOM tree has been constructed and all external CSS has been fetched and parsed, it is time for the layout engine to create the render tree. Here an explanation of the creation process and the structure of render trees will be given. This section is mainly based on* [13, 2].

A render tree is a visual representation of the document. In contrast to the DOM tree, the render tree only contains of elements that will be rendered. The nodes of the render tree are called *renderers* (also known as *frames* or *render objects*), as the elements are responsible for their own and all subnodes layout and rendering. In order to know how to render themselves, the final style of each renderer needs to be computed which is done by the layout engine while constructing the tree. Each renderer represents a rectangle (given by style 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 in the DOM tree map in a 1:1 relation to nodes in the render tree, but the relation can also be smaller or larger. The render tree only contains nodes that will affect the rendered result, so nodes that do not affect the layout flow of the document and that aren't visible will not be present in the render tree. For instance, a DOM node element 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 elements with the `visibility` 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 the rectangle. This is because some properties cannot be computed by just computing the styles for an element. Some properties depend on the flow of the document, and needs to be resolved through a layout. The layout process will resolve the final position and size for all renderers.

4.3 Style computation

↪ *Both the render tree and scripts need to be able to get the final style of an element. In order to know the final style of an element, the style for each element must be computed. Here a brief explanation of selector matching, style cascading, inheritance and rule set specificity will be given. CSS property definitions will also be described. This section is mainly based on* [13, 2, 8].

To trace how the style of an element was 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 can be regarded as CSS because they are *cascading* their style rules through *selector matching*. Selector matching conceptually finds all elements in the DOM tree that matches a CSS selector, to apply 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 element. The last item is deprecated, and all styles applied this way will be 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 the initial value defined by the CSS *property definition*. Property definitions describe how the style properties should behave, see table 4.2. For instance, the width property definition states that legal values are absolute lengths, percentages or `auto` (which will let the browser decide). The percentages are relative to the containing block. The initial value is `auto`, the properties applies to all elements 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 will resolve most style properties, but as stated in section 4.2 some properties require a layout in order to be resolved.

How does selector matching work? Parallel?

Cascading, parallel?

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

4.4 The layout process

↪ *When the render tree has been constructed and the style properties has been resolved for all nodes, it is time to perform the actual layout. The layout process will decide the final computed style of all elements, and needs to be done before rendering. This section will describe the layout process, discuss some performance aspects and show examples when layout happens. This section is mainly based on* [13].

The layout (also called *reflow*) of HTML is flow based, which means that document layout 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*. Layout engines 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 are changed (such as `font-size`). Because the API of `getComputedStyle` promises resolved values for all style properties, a call will force a full layout (flushing the incremental layout commands queue). When a renderer performs a layout the following usually happens:

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

1. The own width is determined.

2. Positions all children and request 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 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 height is 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 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 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.

Is incremental layout commands really batch processed?

4.5 Parallelization

↪ *No longer can performance of an application increase over time without any code changes (as opposed to the times when Central Processing Unit (CPU) clock speeds increased rapidly). Now, applications need to utilize the multiple cores of the CPU instead of relying on high clock speed. Parallelization is something layout engine vendors are very interested in, and much research is being done about utilizing multiple cores to increase the speed of the engine. In this section a small summary will be given about the current research front, and how the parallelization can be approached. This section is mainly based on* [5, 22, 31, 38, 42].

As web applications grow bigger and more demanding, browsers continuously need to improve the performance on all levels. Fetching resources over the network is the only thing that is done in parallel today. The rest of the browser system is designed and optimized to run sequentially. On computers, browsers usually achieve parallelism by running each page context in parallel (each tab of the browser window is separate process). This approach is appealing because it utilizes the cores of the machine by still having all subsystems run sequentially for each page, while improving the overall performance of the browser. However, this approach is not enough. The page performance is not improved if only one page is present. For the web to be a true competitor to heavy native applications, the page performance needs to be increased (not only the overall browser performance). It is then important to be able to dedicate multiple cores to one page instead of having all the pages present using their own core (perhaps the non-visible pages in different tabs will share one core while the main visible tab will get access to multiple cores if needed). Also, with the number of mobile devices browsing the web increasing rapidly it is re-

Reformulate to not use the word native.

ally important to be able to achieve good parallelism. Light devices such as small laptops, mobile devices 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 [22]. This is why most researchers target light devices when trying to parallelize web browsers. Of course, once the methods has matured and been implemented, desktops will benefit from this as well.

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

CSS selector matching is a good candidate for parallelization, due to matching nodes to selectors being independent from other selector matches. A successful parallelization of selector matching has been achieved with locality-aware task parallelism [22]. It has also been shown that selector matching and style resolving (through cascading) can be parallelized [5]. 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 sub-tree independent for non-float elements [38, 42]. 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 system.

Write about the fork-join model of the parallelization as described in servo?

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

Chapter 5

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 — the solution to modular responsive web design. This chapter will define element queries, present how they solve the problem, and address some of the issues hindering element queries from being implemented natively. Further, some possible restrictions will be presented to resolve some of the issues.*

Instead write that some solutions will be presented?

Recall from section 3.2 that responsive web design is the concept of having elements respond to the given size. Responsive web design implements this by using media queries, applying conditional rules by the viewport size. In section 3.3 the desire for building modular web components was presented. One implication of building applications in a modular way is that general components become reusable. Since the only way of applying conditional rules to elements today is by using media queries, all responsive styles must be defined by the application and not by the modules (because only the application knows how much space each module will be given in the layout). The implication of this is that the HTML and JavaScript can be written in a modular way, but the CSS is left for the user of the module (application) to write which somewhat defeats the purpose of writing modules. So developers today have two choices: writing modules that are not responsive but self-contained, or writing modules that are responsive but not self-contained (leaving the styling to the user).

The solution to this is element queries, which allows conditional rules to be applied based on any criteria for any element. Figure 5.1 shows the problem with media queries and the element query solution. The typical use case is to write conditional styles to be applied based on the container element size. Although the size of the container element is typically the most interesting element query with responsive web design, element queries as a concept does not need to be limited to that. It is also possible to apply conditional rules by any element display property, color, text alignment, and so on. Theoretically it should be possible to query any style property of an element, but the focus will be the `width` and `height` properties since they are the most interesting in responsive web design.

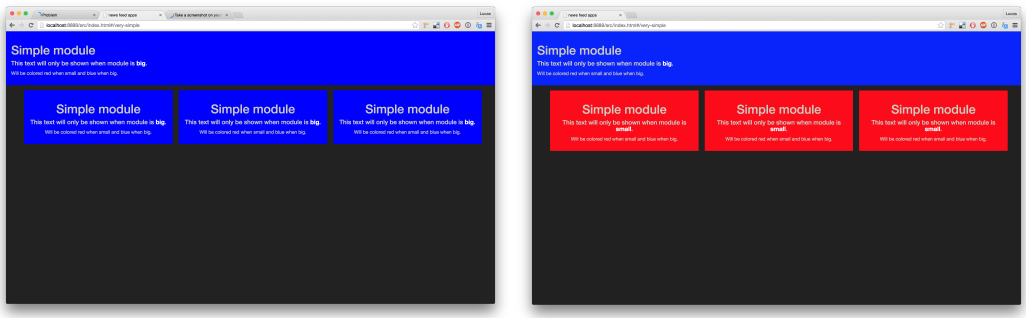

Figure 5.1. The problem of media queries and the element query solution. Both images shows four instances of a module in different sizes. The module is responsive, so when it is small it should change color to red and change the inner text a bit. The left version is built with media queries; hence all module instances stay blue even though some are small (only a smaller viewport will trigger the modules to change style). The right version is built with element queries, hence the modules display differently depending on the given size that each module has, which is the desired behavior.

5.1 Problems

↪ *The main reason why element queries have not been implemented natively in browsers is because they bring problems and limitations to the browsers. It is important to understand the problems in order to speculate about a potential native API design. In this section the two major problems will be presented; performance and circularity.*

What to write here?

5.1.1 Performance

↪ *This subsection will present the performance impacts that element queries have on layout engines. As shown in section 4.5 browser vendors are very interested in parallelizing their browsers and especially the layout engines to increase the page performance. Element queries limits the parallelization badly, which will be the focus of this subsection.*

Referera till w3c mail-inglista?

Recall from section 4.4 that the box size and position for each element is calculated at in the layout process, and cannot be determined before an actual layout has happened. This imposes that a layout pass needs to be performed before knowing the element sizes. If element queries are present that rely on the size of elements, which is the identified common use case, the following process needs to happen:

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

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

3. If the element query selector matching has resulted in a different matching set than in step 1, another layout pass (with the new rules applied) needs to be performed.

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

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

Totally forgot to write about parallel problems

5.1.2 Circularity

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

The most obvious occurrences of cyclic rules are when there are specified conflicting width constraints and width updates for an element. See listing 5.1 for the perhaps most simple example of cyclic rules. The syntax of element queries will be presented in section 5.3, and is not important to understand the problem.

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

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

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

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

Listing 5.1. Very simple example of cyclic rules.

The following happens to the element:

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

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

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

Clearly, the browser will be stuck in an infinite layout cycle pending back and forth between step 2 and 3 (250 pixels and 550 pixels). The result of this is of course implementation dependent. One reasonable outcome of such infinite layout loop is that the layout engine executes one layout pass and then evaluate the next set of matched selectors and so on, which leads to a functioning page but since a relayout is enforced every update, the performance impact is huge. This example is somewhat similar to writing `while(true);` in JavaScript, which obviously is a bad idea. However, cyclic rules can also occur in less obvious ways.

Indirect cyclic rules are somewhat more complex to reason about, than the example given above (which is an example of direct cyclic rules). For instance, if the `#foo` element matches the width of a child element, and the child changes width depending on the parent width, a cycle might occur. Consider the code in listing 5.2.

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

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

#child {
  width: 250px;
}

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

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

Listing 5.2. Example of indirect cyclic rules.

What makes this situation more complex than the previous example is that it is less obvious for numerous reasons:

- The rules of the child element and the rules of the `#foo` element might be separated into different parts of the stylesheet (or even different stylesheets). The problem cannot be found without considering both of the rule sets.
- The child element might be another module, and by adding one line to the parent module (the `#foo` element) a cycle has appeared.
- The developer has to be well aware of how the display property affects the element and what implications the `inline-block` has to the element.

An JavaScript quivalent of this example would perhaps be `var bar = true; while(bar);` with the motivation that it is still very obvious that it is an infinite loop but both the while loop and the variable needs to be reviewed. Also, the variable assignment could happen in another part of the program.

The examples given so far have been simple and easily detectable. However, cycles can occur in a much more complex way. The two given examples could both be identified as cyclic by simply reviewing the CSS code. It is also possible for a program to analyze the examples to deduce that they are cyclic. Unfortunately, not all cycles can be detected by static analyze. Consider the code in listing 5.3.

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

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

#child {
  font-size: 1.5em;
}

#foo:eq(max-width: 300px) #child {
  /* This rule applies only when the width of #foo is < 300px. */
  font-size: 3em;
}

#foo:eq(min-width: 500px) #child {
  /* This rule applies only when the width of #foo is > 500px. */
  font-size: 1em;
}
```

Listing 5.3. Example of indirect cyclic rules.

In this example, it is virtually impossible to tell if the rules are cyclic. So what happens is that the `#foo` element get the size of the child element, which width depend on the text inside it. The width of the text depends on the font size, which is initially set to `1.5em`. The `em` unit is relative to the inherited font size of the element. 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 child is increased to `3em` and when above 500 pixels the font size is decreased to `1em`. So if `3em` results in a font size big enough to make the child element bigger than

500 pixels, and if `1em` results in a font size small enough to make the child element smaller than 300 pixels the rules are cyclic. The factors that creates the cycle are the following:

- The display type of the `#foo` element
- The element queries of the child element
- The font size value of the child element
- **The inherited font size of the `#foo` element**
- **The text of the child element**

The bolded factors are of a far worse type than the ones presented until now. They are impossible to determine at parsetime. In this example the text is static but it could have been added dynamically. Also, the inherited font size depends on the closest ancestor with a font size property defined. Since ancestors also can have their font sizes defined in relative units, the dependency tree can go up to the root of the document. Further, if no ancestor defines an absolute value for the font size it is up to the browser to default to a size, which is impossible to reason about at parsetime. This implies that there is no way of telling if the cycle appears or not without actually running the code. Also, the cycle may appear in different browsers and settings, which makes the cycle even harder to detect. Far worse is that the previous examples have been of the `while(true);` character, which gives the false hope that even though it is possible to create cycles developers would never encounter it since "real" code would not be close to the examples given here. With the third example, this is no longer true. Increasing the font size when the layout area is smaller and decreasing the font size when the layout area is bigger is something that is frequently done in responsive web design. Typically web site authors want the font size to increase on handheld devices, to increase the readability. Of course, the numbers given in the example might not be reasonable, but in concept it is valid use case that may result in cyclic rules.

5.2 Possible solutions

↪ *Now that the problems of element queries have been presented, it is time to show some possible solutions to the problems. The different solution approaches either limits element queries in some way or solves only a subproblem, so no perfect solution that keeps element queries unlimited with decent performance has been found.*

Write somewhere that iframes are not a solution? But they are close to a solution...

5.2.1 Viewport elements

↪ *This solution limits element queries a lot, but avoids many of the problems this way. This approach has been discussed at W3C and RICG assumes that this is a prerequisite to a native implementation.*

cite where this has been gathered from?

By limiting element queries to special viewport container elements that can only be queried by child elements, much of the problems are resolved. This means that a new HTML tag will be crafted (perhaps named `viewport`) that defines separated viewports in the document. The size of viewports is not dependent on their children, and children may only target the closest ancestor viewport element in the element queries. This way, cyclic rules can no longer occur since the children may not alter the viewport size. The reason that a new HTML tag is proposed instead of a new CSS property that defines the behavior is because the layout engine in the latter case needs to resolve the styles for all elements in order to know which are viewport elements. With a new HTML tag, the layout engine can detect that that the element is a viewport before any resolving styles.

Write why it is important to know viewport elements before resolving styles.

This approach also solves many of the performance problems, but not all. The internal APIs that request non-layout information for the elements using element queries only need to make sure that the containing viewport element has been laid out before resolving the styles (which is a much better situation that the unrestricted version of element queries). Further, the layout can be done in one pass as long as the viewport elements are laid out before the children. It is still a bit inconvenient that the layout engine would need to evaluate all element query selectors in the middle of a layout pass (after the that the viewport elements has been laid out) in order to resolve the styles for the viewport children. Parallelization is also easier, since the queries may only be targeting the closest viewport element. This means that each viewport subtree can be laid out in parallel.

This solution would behave much like the `iframe` element layout-wise. It should be noted that `iframe` elements are not suitable as an alternative to the proposed `viewport` element, since `iframe` elements are much more limited by nature (creates a new document and script context).

Obviously, this limits element queries a lot. The fact that the size of the viewport element cannot depend on the children (like normal block elements do), limits the usability. The idea is that the viewport cannot be queried for properties that the children may affect (such as the width and height style properties). In order to allow the children to query the properties, they cannot be affected by the children. In theory it should mean that if no children query for instance the height of the viewport, then the viewport may depend on the children for it's height. This is a powerful insight, since the general use case would be to write element queries against the width and have the elements adapt their heights accordingly (including the viewport element).

Back this up. Maybe check with RICG if this is possible?

5.2.2 Element queries with runtime checks

↪ *This solution does not limit element queries, but does also not solve all problems. Instead of making cyclic rules impossible, a mechanism to handle the cyclic rules is desired. Unfortunately this does not improve the performance at all, which is a major drawback.*

Should this be incorporated with some other section or be written?

Maybe write about having a third-party framework as a solution?

5.3 Native API design

↪ *In this section a possible native element query API will be presented. This is based on the current information and approaches of W3C and RICG, and should therefore be regarded as a guideline to how an API might look like.*

As presented in subsection 5.2.1, viewport elements solve many of the element query problems and are therefore assumed to be a part of a native API. Since it would be favorable to be able to specify multiple conditional rule sets for the element queries (like media queries) the CSS syntax may be of the form as presented in listing 5.4.

```
/* Media query syntax */
@media ... {
  p {
    color: red;
  }
}

/* Element query syntax */
... {
  p {
    color: red;
  }
}
```

Listing 5.4. Element queries are assumed to have nested rule sets, like media queries

Continue this if possible?

Part III

Third-party framework

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

Write about the simple cycle detector.

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

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

Chapter 6

Analysis of approaches

6.1 Current implementations

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

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

Chapter 7

Framework design

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

First of all, a working name of the framework needs to be established — the framework will from now on have the working name ELQ. This name serves as a prefix for many of the framework APIs, and will be frequently used in the report from now on. The technical goals of ELQ will be stated in section 7.1. In section 7.2 the architecture will be described, and the subsystems of ELQ will be presented. Last, the API will be defined in section 7.3.

7.1 Technical goals

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

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

1. **Features**

    a) The framework needs to augment native element queries in the sense that styles are applied automatically on element resizes.

b) Existing modules written with media queries should be automatically converted to element queries so that existing third-party modules can be used in element query empowered applications.

Remove this?

c) The features should be flexible and not limited to only the common use cases.

2. **Ease of use**

   a) Developers should not need to perform big rewrites of their modules or applications in order to use the framework.

   b) The APIs need to feel natural and should not seem alien to web developers.

   c) The framework should help identifying cyclic rules and prevent them from causing infinite layouts.

3. **Performance**

   a) The framework needs to have adequate performance to empower heavy applications running on light devices.

   b) The framework load time needs to be kept low.

4. **Compatibility**

   a) Older browsers must be supported.

   b) The framework 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 framework must act as a prolyfill[1] for native element queries.

It would be possible to create a framework that conforms to all of the requirements but added features, automation and compatibility most probably would decrease the performance. In the same way, too many advanced features may decrease the ease of use and simplicity of the APIs. Some feature may also restrict the compatibility. Trying to conform to all of the requirements and trying to find a balance would result in a worse solution than a framework tailored for specific requirements. By providing a good framework foundation and plugins for different use cases it is up to developers too choose the right plugins for each project. This way, developers are composing their own custom tailored solution for their specific use cases. In addition, by letting the plugins satisfy the requirements it is easy to extend the framework 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

[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.

at this point of time, which will probably lead to frequent changes to the plugin. By separating it from the core framework (and the other plugins), only developers that really desire the prolyfill behavior must handle the rapid API changes.

7.2 Architecture

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

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

To decide if a subsystem of the solution should be in the framework core or a plugin is a difficult task. A balance needs to be found between ease of use, performance and extendability. All subsystems of the core needs to provide a fundamental functionality to the framework and also needs to be general enough to be used by plugins in different ways. If a plugin needs to write a custom version of a subsystem in the core in order to work, the subsystem should perhaps not be part of the core. Each subsystem added to the core will impact the performance negatively for all users, so they must impose a real value to the 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 framework easy to use for the common use cases.

Figure 7.1 shows the overall architecture of the framework. The core consists of fundamental and general subsystems that are utilized by plugins. Developers should never be programming against the core directly, and it can therefore only be accessed by plugins through the hidden plugin API. The framework provides a small public API which is mainly used to invoke and handle the plugins. The plugins form the bigger part of the public API in the sense that they decide which features exist and how they work. It should be noted that plugins may provide APIs through the framework by registering methods, or by specifying custom syntax (such as markup or CSS) which is beyond the control of the core. Plugins may also have interplugin APIs and dependencies, which also is beyond the control of the core. The core is supposed to have a slow development phase, with few changes and good backwards compatability. Plugins may be developed in a high rate, with frequent changes to support new features. Subsystems that are part of the core are the following:

- **Reporter:** Responsible for reporting information, warnings and errors to the developer. By having a centralized reporter, it is possible to decide at a global framework-level how much plugins are allowed to report. Other options such

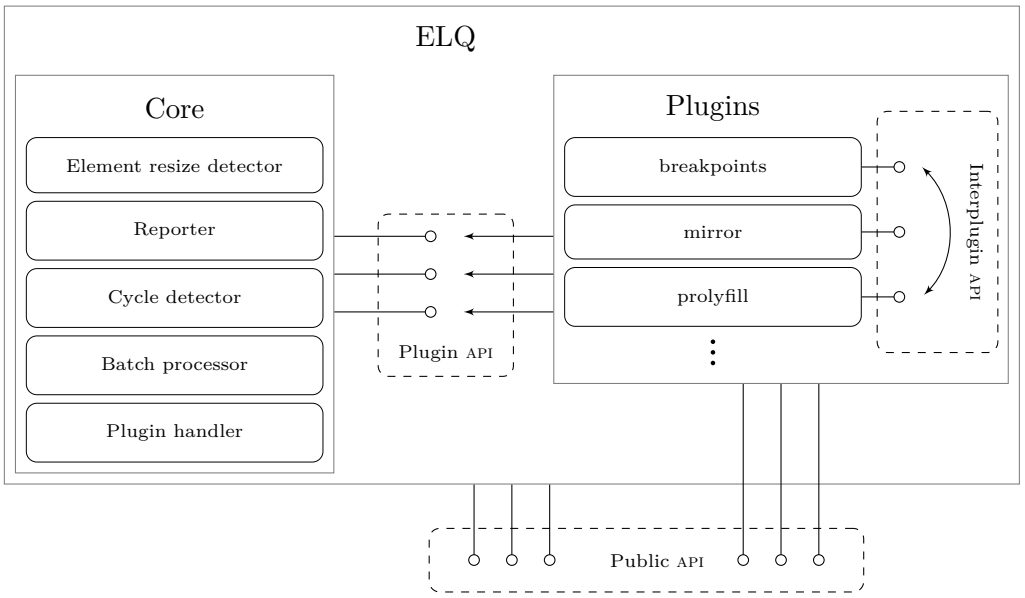

Figure 7.1. The overall framework architecture, which shows how the framework is divided into two parts; the core and plugins. The core is only accesible through the plugin API, and is not a part of the public API. Plugins may have interplugin API's and dependencies. The bigger part of the public API's is defined by the plugins.

as throwing exception on errors or warnings could also be set at a global framework-level. Also, the framework can make sure that all reports are standardized and can provide extra information such as the name and version of the plugin along with its report. To avoid code duplication, it also beneficial to have a centralized report system so the plugins do not need to perform the same compatibility checks (such as checking if the browser actually supports console reporting).

- **Element resize detector:** Provides an interface for listening to element resize events. This system is fundamental to plugins fulfilling requirements 1a and 2a since it enables the framework to be aware of when elements change sizes, instead of the host application keeping track of when elements have changed sizes.

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

- **Cycle detector:** As shown in section 5.1.2 cyclic rules need to be detected at runtime. When a plugin wish to update the size state of an element (which in turn applies the conditional styles to the element) it can ask this system if

the update seems to be part of a cycle. If the update seems to be part of a cycle, it is up to the plugin how that should be handled.

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

- **Batch processor:** To avoid layout thrashing, it is important to read and mutate the DOM in separate batches. This subsystem provides an interface for plugins to store functions to later be executed in a batch. To avoid layout thrashing by other scripts it can beneficial to asynchronously postpone the batch until the next layout cycle of the browser, which is also handled by the subsystem.
- **Plugin handler:** Of course, having plugins requires a subsystem for handling them. This system provides the interface for developers to add plugins to the framework. The system is also responsible for retrieving the plugins, and invoking them on different framework events.

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

7.3 API design

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

Recall from section 7.2 that the bigger part of the public API is provided by plugins. The role of the framework is to provide plugins with systems to be used for performing element query tasks, which is done through the plugin API. Three plugins will be developed;

7.3.1 Public API

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

In section 7.1 it was determined that advanced users must be able to change the subsystems used in the core. Therefore no global framework instance will be instantiated automatically on load. Instead, a function `Elq` that creates ELQ instances is provided in order to be able to inject dependencies. The function accepts an optional options object parameter, where it is possible to set options and subsystems to be used for the created instance. If no options are given, default options will be used. See listing 7.1 of example usages of the `Elq` function. The function returns an object containing the core public API methods of the ELQ instance.

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

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

Listing 7.1. Example usages of the `Elq` function that creates ELQ instances.

The next step after that an instance has been created is to register the plugins to be used by the instance. There are three methods for handling plugins; `use`, `using` and `getPlugin`. The `use` method registers a plugin to be used by the framework. It is responsible for controlling that plugins do not conflict with each other and that plugins are initiated correctly with the framework instance. The method requires a *plugin object* and accepts an optional options object parameter. A plugin object acts as a plugin definition and is responsible for describing a plugin and providing a function to create a plugin instance. The `using` method requires an *plugin identifier* as parameter and tells if the given plugin is being used (has been reigstered) by the instance or not. A plugin identifier can either be the plugin object or the name of the plugin. The `getPlugin` method requires a plugin identifier and returns the plugin instance being used by the ELQ instance that matches the plugin identifier. See listing 7.2 for details about the plugin object and how it is used with the three methods for handling plugins.

```
var myPlugin = {
  getName: function() {
    // Returns the name of the plugin, which has to be
    // unique in an elq instance.
    return "my-plugin";
  },
  getVersion: function() {
    // Returns the version of the plugin.
    return "1.0.0";
  },
  isCompatible: function(elq) {
    // Returns a boolean that indicates if this plugin
    // is compatible with the given elq instance.
    return true;
  },
  make: function(elq, options) {
    // Returns a plugin instance. It is optional to use
    // the elq instance or options object in the initiation process.
    return {...};
  }
};

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

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

// 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

// It is possible to get the plugin instance.
var myPluginInstance = elq.getPlugin(myPlugin);

// Also by name.
elq.getPlugin("my-plugin") === myPluginInstance; // true
```

Listing 7.2. Plugin object definition and examples of handling plugins.

When the desired plugins have been registered to the ELQ instance, it is time to apply the framework to the target elements (i.e. the elements that will be queried for conditional styles). This is done with the `start` method which requires a collection of elements as a parameter. When called, it will invoke all `start` methods on all registered plugins which gives them the opportunity to initiate the elements according to their own mechanisms. To satisfy the requirements 2b and 4c the function is agnostic about the elements collection — all objects that are iterable and contains elements are accepted. It is also possible to provide a single element without a collection. The effect of invoking the `start` method on an element multiple times is defined by the plugins. See listing 7.3 for example usages of the `start` method.

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

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


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

Listing 7.3. Example usages of the `start` method. The method only requires an iterable collection, so it is library agnostic.

7.3.2 Plugin API

The plugin API is a superset of the public API. Plugins have additional direct access to most core subsystems presented in section 7.2. The plugin API access is given to a plugin when it is registered (i.e. when the ELQ instance invokes the `make` function of the plugin definition). Recall from subsection 7.3.1 that the `make` function of the plugin definition is given two parameters at invokation; `elq` and `options`. The first parameter is a reference to the ELQ instance (to which the plugin was registered) extended with the plugin API. Direct access is given to the following core subsystems:

- **Reporter**: via the `elq.reporter` property that refers to the reporter instance.
- **Cycle detector**: via the `elq.cycleDetector` property that refers to the cycle detector instance.

- **Batch processor**: via the `elq.createBatchProcessor` property that refers to a function that creates a batch processor intance.
- **ID handler**: via the `elq.idHandler` property that refers to the ID handler instance.

Include id handler in section 7.2?

Direct access is not given to the plugin handler and element resize detector subsystems since their APIs are already exposed in the public API.

The reporter has three methods; `log`, `warn` and `error`. They are used to report information, warnings and errors respectively. The default reporter is a wrapper around the native `console` object, and has therefore the same API as it.

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

Instead of having a single batch processor instance shared among all framework entities (i.e. core subsystems and plugins), each entity has to create an own instance. This to avoid entities interfering with each other while performing batch processing. For example, some entities might want to force the batch to process at a point where it would be beneficial for other entities to delay the batch. The `createBatchProcessor` accepts an optional options object parameter and returns a batch processor instance that has two methods; `add` and `force`. The `add` method requires a function parameter that will be called when the batch is processed. Since the batch processor is leveled, as motivated in section **??**, the method also accepts an optional parameter that defines at which level the given function should be processed. The `force` method commence the processing of the batch, which can happen synchronously or asynchronously a the optional parameter. Two important options to the `createBatchProcessor` are the `async` and `auto` options. If the `async` option is enabled, the batch will be processed asynchronously as soon as possible after that the `force` method has been invoked. If the `auto` option is enabled, `force` will be invoked when the first function of the batch has been added (this implies that `async` has to be enabled).

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

Code examples?

7.3.3 Plugins

↪ *So far, only prerequisites for third-party element queries has been presented. Now, it is time to describe the systems and APIs that will actually realize element queries as ELQ plugins. It should be noted that the plugins presented here is the suggested solution to element queries according to the research of this thesis. They*

are designed for good performance, good compatibility, and ease of use. For extreme requirements or corner cases, custom ELQ plugins might be preferred.

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

Breakpoints

This is the plugin that listens to element resizes and updates their size states according to predefined breakpoints. The name of the plugin is `elq-breakpoints` and it does not have any dependencies (other than the element resize detector of the ELQ core). The main idea of the plugin is to apply classes to target elements that reflects in which size state they are. Each element defines breakpoints to which the plugin will update the size state. The breakpoints of an element are defined by custom element attributes. See listing 7.4 for an example of breakpoints definitions.

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

Listing 7.4. Example an HTML element that uses the `elq-breakpoints` plugin.

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

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

The plugin will append classes to the element that reflects the size state of the element. For each breakpoint, one class will be present that tells if the size is above or below the breakpoint. The number of size states of an element in a dimension is given by $n_{ss}(n_b) = n_b + 1$, where n_b is the number of breakpoints in that dimension. The number of breakpoint classes of an element in a dimension is given by $n_{bc}(n_b) = 2n_b$. The number of breakpoint classes active at the same

time for an element in a dimension is given by $n_{abc}(n_b) = n_b$. The format of the classes are `elq-[dimension]-[above|below]-[breakpoint]`. For example, if the example element is below 300 pixels wide, it will have the following two classes present:

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

Breakpoint classes are utilized for applying conditional styles for elements based on the size state of the target element. See listing 7.5 for conditional styles of the target element and it's children.

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

#target.elq-width-below-300 {
  font-size: 10px;
}

#target.elq-width-above-300 {
  color: red;
}

#target.elq-width-above-500 {
  font-size: 20px;
}

#target.elq-width-above-500 p {
  padding: 10px;
}

#target.elq-width-below-500 p {
  padding: 5px;
}
```

Listing 7.5. Example of conditional styles using the `elq-breakpoints` plugin classes

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

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

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

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

Mirror

Prolyfill

Chapter 8

Implementation

8.1 Element resize detection

↪ *In order to be able to perform any element queries, one first needs to be able to detect when elements change size. The element resize detection system will therefore be the backbone of the framework. This section is mainly based on* [3].

Unfortunately, there is no standard resize event for arbitrary elements. Only frames (elements that render it's content independently of the container element, such as `iframe` and `object`) support the resize event by standard since they have their own viewport. Legacy versions of Internet Explorer (version 8 and down, but also some later versions depending on the document mode and if the browser is in quirks mode) do support the resize event on arbitrary elements. Since the framework needs to support all commonly used HTML elements a solution to listen to any element resize needs to be found. Possible solutions are:

- **Polling**: To have a script running asynchronously to check all elements if they have resized. This can be achieved by using the `setInterval` function. One can set the interval time to shorter or longer, where shorter would lead to being able to detect resize changes quicker but having worse performance. A longer interval time would not be able to detect changes as quickly but increase the performance.
- **Scroll events by overflowing elements**: By injecting multiple elements, into the target element, that have overflowing content scroll events occurs when the target element is resized. Five elements needs to be injected; one container element that is styled to have the same size as the target element that contains all the other elements

  Describe target element
- **Injecting frames**: Since frame elements are the only ones that support resize events natively, the idea is to inject frame elements as children to the elements that one want to listen for resize changes (also called *target elements*). By styling the frame so that it depends on the target element and set the size to

be 100 percent, the frame will always be the same size as the target element. Then a resize event handler can be attached to the frame that emits a resize event for every target element resize. So in order to listen for resize events for an element, a frame element is injected and listened to instead.

The first solution is appealing because it does not mutate the DOM and it works on all browsers since it does not rely on special element behaviors or similar. However, in order to prevent the responsive elements lagging behind the size changes of the user interface, the polls need to be performed quite frequently. Recall from section 4.4 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. The performance impact of polling the DOM frequently is not desirable. Since the polling is be performed all the time, the overall page performance will be decreased even if the page is idle, which is undesirable especially for devices running on battery. This is not a great solution since reading the DOM constantly can be affect the performance negatively.

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

The third solution does mutate the DOM and relies on special element behavior, but it offers many advantages. Browsers only need to perform extra work when preparing elements for being resize-detectable and when the elements actually resize. Further, since the resize detection instead is event-based there is no delay between the actual resize and the detection. By positioning the injected frames `absolute` with width and height set to `100%`, the frames do not affect the visual representation of the document. It has been shown by [3] that `object` is the most suitable frame element to use for this purpose, as they have good browser compatibility, adequate performance and are easy to work with. However, injecting frames into the elements has some implications:

- **CSS selectors may break**: Since target elements get an extra child (the frame element), CSS selectors may behave differently. For instance, the selector `#target *` selector also matches the injected frame element which may result in the frame being styled in a way that it will be visible or in other ways interfere with the original user interface. Also, selectors such as `:first-child` and `:last-child` since the object element is prepended (or appended) to the target element.

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

- **The target element must be positioned**: Recall that absolute positioned elements are moved up to the first non-static positioned ancestor. Since it is

desired to have the injected frame being a direct child of the target element, the target element cannot be positioned `static` that is the default positioning for many elements. Fortunately, elements positioned `relative` behave exactly like `static`, given that the elements do not have any styles applicable to relative (such as `top` or `bottom`). Since the style properties that depend on the element being `relative` positioned does not affect the element if it is `static` positioned, the properties can be removed or regarded as user errors. This way the target element can be changed to `relative` positioning with the special style properties removed, to obtain the same visual representation as the `static` version.

Write that in theory the detection of resize is delayed by the frequency of the polling?

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

8.1.1 Native resize event

remove me?

8.1.2 Object injection

remove me?

8.2 Detecting runtime cycles

8.3 Avoiding layout thrashing

Part IV

Outro

Chapter 9

Related work

Chapter 10

Result

Chapter 11

Discussion

Bibliography

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

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

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

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

[5] 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.

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

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

[8] 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[25] 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).

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

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

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

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

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

[31] 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.

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

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

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

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

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

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

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

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

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

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

[42] 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).

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

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

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

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

Glossary

browser Client application for navigating the WWW and displaying web pages. 2, 3, 4, 9, 12, 13, 14, 15, 16, 17, 23, 24, 26, 27, 29, 31, 32, 34, 36, 46, 47, 49, 55, 56, 71, 72, 77, 78

CSS3 The third revision of the CSS standard. 17, 18

document Strictly defined as something that has an URI and can return representations of the identified resource in response to HTTP requests. If otherwise states, document will refer to HTML web pages in this thesis. In JavaScript, document refers to the DOM root. 9, 11, 12, 15, 16, 18, 23, 24, 25, 26, 27, 36, 37, 55, 56, 71, 72

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, 3, 18, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 45, 48, 49, 51, 55, 56

Flash Flash is a multimedia and software platform for producing cross platform interactive animations. See `www.flash.com`. 16

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. 14, 24, 72

HTML5 The fifth revision of the HTML standard. 16, 17

hypertext Documents with text and media with links (hyperlinks) to other documents immediately accessible for the user. Often used as a synonym for hypermedia. 9, 12, 15, 18, 23

Java applet A Java applet is a small application which is written in Java and delivered to users in the form of bytecode. See `www.java.com`. 16

JavaScript Native browser script language. Web documents often include JavaScript to make the documents more dynamic and interactable. 9, 16, 18, 23, 24, 31, 34, 35, 46, 56, 71, 72, 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. 2, 4, 13, 14, 23, 24, 25, 27, 29, 31, 32, 33, 34, 36, 37, 56, 72, 77

media queries The CSS feature of specifying conditional style rules for elements by conditions such as the viewport size. 1, 3, 18, 31, 38, 45

native Refers to APIs and systems implemented and provided by browsers. Such APIs and systems are often described by standards specifications. 1, 2, 3, 4, 31, 36, 38, 45, 46, 55, 71

render tree Used by layout engines as the model for the visual representation of the document. 24, 25, 26, 27

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, 3, 4, 15, 18, 31, 36, 56

self-contained A self-contained module handles its task without any help from the user of the module. 3, 31

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. 17

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. 1, 2, 3, 4, 5, 16

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, 3, 17, 18, 27, 31, 36, 37, 38, 55

web Short for the world wide web. 1, 2, 3, 4, 9, 10, 11, 12, 15, 16, 17, 18, 29, 31, 36, 46, 71, 72, 77

WebKit The open source layout engine used by Safari. Google's Blink layout engine is a recent fork of WebKit. 14, 24, 77

WYSIWYG A classification that ensures that text and graphics during editing appears close to the result. 12, 76

XHTML Well-formed XML-based markup language that extends or mirros HTML. May be parsed by XML parsers since it conforms to the stricter XML syntax. 52

Acronyms

AJAX Asynchronous JavaScript and XML. 16

API Application Program Interface. 2, 3, 4, 5, 16, 17, 27, 31, 33, 37, 38, 45, 46, 47, 49, 51, 52, 72

ARPANET Advanced Research Projects Agency Network. 10, 11

CERN Conseil Europeen pour la Recherche Nucleaire. 11

CGI Common Gateway Interface. 16

CPU Central Processing Unit. 29

CSNET Computer Science Network. 11

CSS Cascading Style Sheets. 1, 2, 9, 15, 18, 24, 25, 26, 27, 30, 31, 35, 36, 38, 46, 47, 56, 71

DARPA Defense Advanced Research Projects Agency. 12

DHTML Dynamic HTML. 16

DOD Department Of Defense. 10

DOM Document Object Model. 24, 25, 26, 27, 49, 56, 71

FTP File Transfer Protocol. 10, 11, 12

GUI Graphical User Interface. 12, 13, 16

HTML HyperText Markup Language. 9, 12, 15, 16, 18, 23, 24, 27, 31, 36, 46, 52, 53, 55, 71, 72

HTTP HyperText Transfer Protocol. 9, 12

ICANN Internet Corporation for Assigned Names and Numbers. 9

ICCC International Computer Communication Conference. 10

IP Internet Protocol. 9, 10, 11

ISP Internet Service Provider. 9

KDE K Desktop Environment. 14

MILNET Military Network. 11

NCSA National Center for Supercomputing Applications. 13, 16

NSF National Science Foundation. 11

NSFNET National Science Foundation Network. 11

RICG Responsive Issues Community Group. 3, 36, 38

RWD Responsive Web Design. 18

TCP Transmission Control Protocol. 9, 10, 11

URI Unique Resource Identifier. 23, 71

US United States. 10, 11

W3C World Wide Web Consortium. 1, 3, 12, 15, 16, 36, 38

WHATWG Web Hypertext Application Technology Working Group. 16

WWW World Wide Web. 9, 71

WYSIWYG What You See Is What You Get. *Glossary:* WYSIWYG, 12

XML EXtensible Markup Language. 9, 16, 23, 24, 72, 75

Appendix A

Resources

A.1 Practical problem formulation document

Should this be in the real document instead of appendix?

A.2 CSS terminology

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

A.3 Layout engine market share statistics

Browser market share was retrieved by *StatCounter*[1]. Since the graph only display browser market share and not layout engine, it is needed to further divide the browsers into layout engine percentages. The Blink engine was introduced with Chrome version 28 and Android version 4.4 [1]. 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

[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`

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%.

A.4 Usage share of browser versions

Put this somewhere else?

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

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

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