**KTH Computer Science
and Communication**

Modular responsive web design

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

LUCAS WIENER
lwiener@kth.se

Abstract

Abstract goes here.

Referat

Modulär responsiv webbutveckling

Sammanfattning ska vara här.

Contents

Todo list

Chapter 1

Introduction

1.1 Targeted audience

The 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.

1.2 Problem statement

By using 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 media queries are no longer applicable.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: *Elements can not specify conditional style rules depending on their own size, or the size of any other 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 World Wide Web (World Wide Web Consortium, abbreviated W3C) has unofficially stated that such feature would be infeasible to implement. Some problems with implementing element queries in CSS are:

- **Circularity:** If style rules can be applied by criterias of other elements, it will be 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.

Margin notes:
- Write somewhere that modular and responsive web development right now is two disjunct terms?
- This or the thesis? Now both is being used.
- runtime or run time? same for parse time. Both is being used right now.
- Container or parent element? Using both now.
- Renamed framework to library or plugin?
- Back this up or maybe reformulate it a bit.

1

Also, it has been shown that some cyclic rules are impossible to detect at parse time, due to being dependent on run time factors. See section 6.1.2 for examples and a deeper analysis of cyclic rules.

Rewrite this. Some of this is totally wrong.

- **Performance:** layout engines typically perform selector matching and layout computations in parallel to achieve good performance. If element queries would be implemented, the layout engines would 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, and so on until a stable state has been reached. Far worse, since selectors now depend on layout style, this cannot be done in parallel which impacts performance heavily.

Back this up or rewrite.

Because of the problems, it is stated that such feature will not be implemented in the near future. So it is now up to the developers to implement this feature as a third-party solution. Efforts have been made to create a robust implementation, with moderate success. Since all implementations have shortcomings, there is still no de facto solution that developers use and the problem remains unsolved.

1.3 Objective

The main objective of this thesis is to develop a third-party implementation of element queries (or equivalent to solve the problem of modular responsive components). To do this, it is needed to research existing implementation attempts in order to understand and analyze the advantages and shortcomings of the approaches. It is also neccesary 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 API would look like. There are many challenges along the way that will need to be researched and worked around. Examples of such subproblems that would need to be investigated 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?

- 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?

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

This might be a bit to much.

solved in a reliable and performant way by crafting a third-party implementation. A reliable implementation should also enable existing responsive components to react to a specified criteria (parent container size for example) with no modifications to the components. The goal of the thesis should be considered fulfilled if a solution was successfully implemented or described, or if the problems hindering a solution are thoroughly documented.

1.4 Significance

Many frameworks and techniques are being used in web development to keep the code from becoming an entangled mess. Creating modules helps to ease development and increase reusability. One of the biggest issues that keeps web modules from being truly modular is that they cannot adapt to given sizes. This makes the modules either force the client to style them properly depending on viewport size, or not being responsive. Both options are undesirable for developing larger applications. A third option would be to make the modules context aware and style themselves according to the viewport, which defeats the purpose of modules (making them not reusable).

The last couple of years a lot of articles have been written about the problem and how badly we need element queries. As already stated, third-party implementation efforts have been made by small and big players, with moderate success. W3C keep getting requests and questions about it, but the answer seems to lean towards no. An organization called Responsive Issues Community Group (abbreviated RICG) have started an initial planning regarding element queries. However, things are moving slow and a draft about element queries use cases are still being made.

Solving this problem would be a big advancement to web development, enabling developers to create truly modular components. By studying the problem, identifying approaches and providing a third-party solution the community can take a step closer to solve the problem. 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 Methodology

What should go here?

1.6 Delimitations

The focus of the 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.

1.6.1 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, CSS, HTML and responsive web design will be given to fully understand the problem.

1.6.2 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.

- UI and UX design will not be addressed, other than necessary for understanding the problem.

- No complete history of browsers, the Internet or responsive web design will be given other than neccesary.

1.7 Outline

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 the rise of the web. This section is a summary of* [36][8][16][38][31].

Before addressing the birth of the web, lets define the meaning of the concepets of the *Internet*, *Web* and *World Wide Web*. 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 TCP/IP. Since the Internet is decentralized, there is no owner. Or in other words, the owners are all the network end-points which means all users of the Internet. One can argue that the owners of the Internet are the ISP's, providing the services and infrastructure making the Internet possible. On the other hand, the backbones of the Internet are usually co-founded by countries and companies. Or is it the ICANN[1] organization which has the responsibility for managing the IP addresses in the Internet namespace? 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 World Wide Web, file sharing and telephony. In 2014 the number of Internet users was measured to just below 3 billions, and estimations shows that we have surpassed 3 billions users today (no report for 2015 has been made yet). Users are here defined as humans having unrestricted acccess 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 Word Wide Web (abbreviated WWW or W3) is a system that operates through the Internet. The World Wide Web is usually shortened to simply *the web*. The web is a system for accessing interlinked hypertext documents

---

[1]The Internet Corporation for Assigned Names and Numbers

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 which 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 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 compose a *web site* or simply a *site* and are usually hosted from the same domain. To transfer the resources between computers the protocol HTTP is used. Typically the way of retrieving resources on the web is by using a *web browser* or simply a *browser*. Browsers handles the fetching, parsing and rendering of the hypertext (more about this in section 2.2).

2.1 The origin of the web

↪ *Since the web is a system operating on top of the Internet, it is needed to first investigate the origin 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 of a technically interesting history. This section is a summary of [19][23][22][14].*

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 ARPANET (Advanced Research Projects Agency Network) was developed, which was the first network to implement the TCP/IP protocol suite. The TCP/IP protocol suite together with packet switching are fundamental technologies of the Internet. ARPANET was funded by the United States Department of Defense (DoD) in order to interconnect their research sites in the United States. The first nodes of ARPANET was installed at four major universities in the western United States 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 sattelite 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 United States.

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 stack was proposed in order to have a more robust and scalable system for end-to-end network communication. The TCP/IP stack is a key technology for the decentralization of the ARPANET, to allow the massive expandation 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 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 insitutions 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 serverd 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 communcations instead of funding and running the Internet backbone. That, along with an increasing preassure 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.

2.1.1 The World Wide Web

↪ *Now that the history of the Internet has been described, it is time to talk about the birth of the World Wide 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* [10][43][13][37][14].

Recall 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 communica-

Should have another subsection or not have this in a subsection. Only one subsection is not nice.

tion system that allowed interlinked documents was submitted to the management at CERN. The idea was to allow links embedded in text documents, to enable users to view the linked document by clicking it. 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, 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, the Hypertext Transfer Protocol (HTTP), the Hypertext Markup Language (HTML), a browser and a web server had 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 the Wolrd Wide Web Consortium (W3C) was founded by Berners-Lee with support from the Defense Advanced Research Projects Agency 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 World Wide 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 simpled 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 lincensing 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.

2.2 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"
the users needed a client to handle the fetching and displaying of the hypertext docu-
ments, 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* [43][44][25][24][28][30][29][32][20][34][42][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 GUI and a WYSIWYG[2]
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 compatability 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. The
code that the two browsers shared was in 1993 bundled as a library called *libwww.*
The library was licensed as *public domain* to encourage the development of web
browsers. Many browsers were develop 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 Supercom-
puting 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 the intuitive GUI, reliability, simple
installation and Windows compatability. The company *Spyglass, Inc.* licensed the
browser from NCSA for producing their own browser 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*
which 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 appli-

---

[2]What You See Is What You Get is a classification that ensures that text and graphics during
editing appears close to the result.

cations, 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 KDE. The browser was released in 1998 and was bundled in the KDE Software Compilation. Konqueror used the KTHML 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 form was made fully open source in 2005. In 2008, *Google Inc.* also released a browser based on WebKit, named *Google Chrome*, or shortened *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 of 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), parsing and 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.*

> 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 [41][6][4][45][15][18].*

As described in section 2.1.1, 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 the Cascading Style Sheets (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 only was 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 langauges. To generate dynamic content on the server is sometimes referred to as *serverside scriping.* This enabled developers to generate dynamic websites, with different content for different users for instance. However, when the content had been delivered to the client (browser) it was still static content. 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 existed during this time, which teaches the browsers how to handle and execute programs. *Java applets* and *Flash* are examples of this. The user then had to download the Java runtime or Flash runtime on the computer. When the runtimes are present, the browser then executed 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 had to download the runtimes in order to run the programs embedded in pages. As 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.) which included browsers but limited third-party runtimes, plugins quickly went extinct. Today, web sites rarely use plugins.

As JavaScript and HTML supported more features, websites turned into small applications with user sessions (using cookies) and user interfaces. Still, parts of the applications were defined as HTML pages, fetched from the server when navigating the site. When *XMLHttpRequest* API was supported in the major browsers, pages no longer needed to reload the page to fetch new content as XMLHttpRequest enabled developers to perform asynchronous requests to the server without page reload. This opened up for *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 which it was originally designed for. In a W3C meeting in 2004 it was proposed to extend HTML to allow 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 specifications to ease the development of web applications which came to be named HTML5. In 2006, W3C realized 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 API's and features came along with the HTML5 standard, which truly enabled developers to create rich client side applications. CSS3 was also recently released which also included many new features. Using HTML5 together with CSS3 developers could utilize advanced graphics programming, geolocation, cache storage, file system acccess, offline mode, and much more.

3.2 Responsive web design

↪ *Back in the days, web developers could make assumptions about the screen size of the users. 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* [26][33]*.*

According to *StatCounter*[1], 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 visisted by touch devices or mouse-based devices (such as laptops or desktops). 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 — assumptions about the user device, but this time the other way around. When a browser request 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 at 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, the developer 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. Third, the desktop site assumes that desktop users have big screens (which in most of the times 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

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.

---

[1]Statistics are based on aggregate data collected by StatCounter on a sample exceeding 15 billion pageviews per month collected from across the StatCounter network of more than 3 million websites.

the same time on the screen, which would break the assumptions about the layout size avaiable 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 viewport size. By using media queries, elements can be styled in the following manner:

```
@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.

This can be used to tailor a site for a specific medium or viewport size at runtime. The term *Responsive Web Design* (RWD) refers to the approach of having a single site *responding* to different media properties (mainly 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.

3.3   Modularity

↪   *This section is a summary of* [21][12].

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.

Flesh this one out, with meta text.

Chapter 4

Modular development

Remove this chapter?

4.1   Web Components

Part II

Theory

Chapter 5

Layout engines

$\hookrightarrow$ *Before diving into the theory of element queries, it is important to understand how layout engines work. Only the layout engines of the browser will be of importance, since element queries does not affect any other browser subsystem. This chapter will give a brief explanation of the layouting flow that the engines generally perform, and how content changes affects the engine. 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 [11][9].*

Browsers are complex applications that consists of many subsystems. A reference architecture of browsers has been presented by [11], see figure 5.1. 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 URI's that should be rendered with layout options. The main responsibility of the layout engine is to render the current state of the fetched hypertext document. Since documents may (and often do) change dynamically after parse

```
┌─────────────────────────┐
│      User Interface      │──────────┐
└─────────────────────────┘          │
             │                        │
             ▼                        │
┌─────────────────────────┐          │
│      Browser Engine      │          │
└─────────────────────────┘          │
             │                        │
             ▼                        │
┌─────────────────────────┐          │
│      Layout Engine       │          │
└─────────────────────────┘          │
   │      │       │      │            │
   ▼      ▼       ▼      ▼            │
┌───────┐┌─────────┐┌────┐┌──────────────┐
│Network││JavaScript││XML ││Render Backend│◄┘
│ing    │└─────────┘└────┘└──────────────┘
└───────┘
```
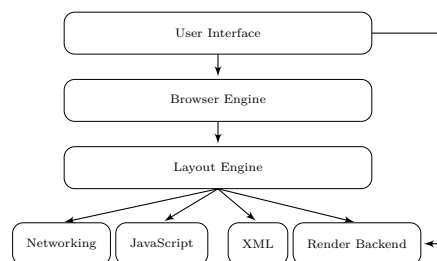
Figure 5.1. Reference browser architecture. The data persistance system used by the browser engine and the user interface has been omitted.

21

time, 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:

1. Fetch of content (typically HTML and CSS) and parse it in order to construct a 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 5.1 and section 5.2 for a more in depth explanation of DOM and render trees. Modern browsers can display multiple pages at the same time (by using tabs or multiple windows) 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 5.1. 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	Engine	Browsers	Share
Blink	Chrome, Opera	44.38%	WebKit	Safari, Chrome, Opera	62.02%
WebKit	Safari	17.64%	Trident	Internet Explorer	14.96%
Trident	Internet Explorer	14.96%	Gecko	Firefox	12.83%
Gecko	Firefox	12.83%			

Table 5.1. The major layout engines and their 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.

5.1 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. Since parsing of content is not very relevant for element queries, it will be adressed only briefly. This section is mainly based on* [39][9][2].

DOM is an abbreviation of Document Object Model which is a convention for representing and interacting with object in XML-based documens (such as HTML). The DOM provides an interface for programs (JavaScript scripts in the browser) to dynamically access and mutate the content, structure and style 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 refered 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 because the specification of HTML states that it must be forgiving (i.e. accepting malformated markup). As CSS and JavasScript are more strict, their grammar can be expressed in a formal syntax and can therefore be parsed with a context free grammar parser (which can be autogenerated). 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 occured by the parser. If such script mutate 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 specalutive 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.

> Is it true that the sepc says that malformated html must be accepted. Servo experience report seems to say otherwise.

5.2 Render trees

> code example of reentrant html?

↪ *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* [9][2].

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

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 laying out and rendering themselves and all of its subnodes. 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 differnt types of renderers, which affects how the renderer rectangle is computed. The type can be directly affected by the display style property.

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

Typically nodes in the DOM tree map in a 1:1 relation to nodes in the render tree, but the relation can also be smallar 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, an 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 renderes.

5.3   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, inheritence and rule set specificity will be given.* CSS *property definitions will also be described. This section is mainly based on* [9][2][7].

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 which means that 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 weightened

24

so that if a property is assigned values by multiple rule sets the one with highest weight will be applied. The weightening process starts with the origin of the style:

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

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

The weightening process continues by calculating the *specificity* of all rule set selectors, the higher the specificity the higher the rule set will be weightened. 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 weightened 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 5.2. 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 groups 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 `inherit`.

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

This process will resolve most style properties, but as stated in section 5.2 some properties require a layout in order to be resolved.

5.4   The layout process

↪   *When the render tree has been constructed and the style properties has been*

How does selector matching work? Parallel?

Cascading, parallel?

resolved for all nodes, it is time to perform the actual layouting. 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 layouting happens. This section is mainly based on [9].

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 an renderer has been changed, a dirty bit system is usually implemented. The system will mark which renderers in the tree that need a layout, which avoids layouting 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 dependant 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 needs 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 takin into account when calculating the widths. When the width of an element is calculated, it needs to 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 renderes 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 propageted upwards.

5.5 Parallelization

$\hookrightarrow$ *No longer can performance of an application increase over time without any code changes (as opposed to the times when* 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][17][27][35][40].

As web applications grow bigger and more demanding, browsers continously 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 achieves parallelism by running each page context in parallel (each tab of the browser window is an own 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 really 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 are CPU bound contrary to the common belief that they are network bandwidth bound [17]. 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.

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 [17]. It has also been shown that selector matching and style resolving

27

(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 [35][40]. 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?

Chapter 6

Element queries

↪ *Now that a good understanding of browsers (especially the layout engine), responsive design and modular development has been aqcuired 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.*

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 solutions to this is elemenet queries, which allows conditional rules to be applied based on any criteria for any element. Figure 6.1 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 typically the size of the container element is the most interesting element query with reponsive 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 lay on the `width` and `height` properties since they are the most interesting in responsive web design.

> Instead write that some solutions will be presented?

Figure 6.1. 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.

6.1 Problems

↪ *The main reason why element queries has 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?

6.1.1 Performance

↪ *This subsection will present the performance impacts that element queries have on layout engines. As shown in section 5.5 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 5.4 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 impose 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 usecase, 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 API's 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 API's will require 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

6.1.2 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 run time. Some examples of will be given to better understand the complexity of the problem.*

The most straight forward occurances of cyclic rules are when there are specified conflicting width constraints and width updates for an element. See listing 6.1 for the perhaps most simple example of cyclic rules. The syntax of element queries will be presented in section **??**, and is not important to understand the problem.

```
#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 6.1. Very simple example of cyclic rules.

The following will happen 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)` will match 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)` will match. 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 will be 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)` will match. 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 will be 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 dependant. One reasonable outcome of such infinite layout loop is that the layout engine will execute one layout pass and then evaluate the next set of matched selectors and so on, which will lead to a functioning site but since a relayout is enforced every update, the performance impact will be 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 6.2.

```
/* 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 will be applied only when the width of #foo is < 300px. */
    width: 550px;
}

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

Listing 6.2. 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 6.3.

```
/* 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 will be applied only when the width of #foo is < 300px. */
  font-size: 3em;
}

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

Listing 6.3. 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 will 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 parse time. In this example the text is static but it could have beeen 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 parse time. This implies that there is no way of telling if the cycle will appear 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 encouter 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 deisgn. 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 which may result in cyclic rules.

6.2 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...

6.2.1 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.*

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 are 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 viewport size may not be altered by the children. the reason that a new HTML tag is proposed instead of a new CSS property that defines the behavoir 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.

This approach also solves many of the performance problems, but not all. The internal API's that request non-layout information for the elements using element queries only need to make sure that the containing viewport element has been layed 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 layed 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 layed out) in orde 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 layed 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` elements, since `iframe`'s 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 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).

[margin note] cite where this has been gathered from?

[margin note] Write why it is important to know viewport elements before resolving styles.

[margin note] Back this up. Maybe check with RICG if this is possible?

6.2.2 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.*

6.3 Native API design

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

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

↪   *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 6.2.1, 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 presnted in listing 6.4.

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

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

Listing 6.4. Element queries are assumed to have nested rule sets, like media queries

Continue this if possible?

Part III

Third-party framework

Chapter 7

Analysis of approaches

7.1 Current implementations

Chapter 8

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 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 will serve as a prefix for many of the framework API's, and will be frequently used in the report from now on.

> Write more here or move the above text somewhere else?

8.1   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.*

1. The framework needs to be performant enough for heavy applications running on light devices.

2. The framework may not require invalid CSS, HTML or JavaScript.

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

4. 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 power applications.

5. The API's need to be natural for web developers, and may not seem alien.

8.2 Framework plugins

$\hookrightarrow$

8.3 Subsystems

$\hookrightarrow$

Chapter 9

Implementation

9.1   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.

- **Overflow hacks**: ⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯   [Write me, or is this no longer an alternative?]

- **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 which will emit 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 will work 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. The performance impact of polling the DOM frequently is not desirable. Since the polling will be performed all the time, the overall page performance will be decreased even if the page is idle, which is undesireable especially for battery devices. 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 the best performance. Browsers will only need to perform extra work when preparing elements for being resize-listenable and when the elements actually change size. By positioning the injected frames `absolute` with width and height set to `100%`, the frames does not affect the visual representation of the document. 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 will also match 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 the developer expect it to be, since the frame element has been injected. Also, code that replaced the content of elements (such as `element.InnerHTML = ...`) will undesireably remove 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` which 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.

44

It has been shown by [3] that `object` are the most suitable frame element to use for this purpose, as they have good browser compatability, adequate performance and are easy to work with.

9.1.1 Native resize event

9.1.2 Object injection

9.2 Runtime cyclic rules detection

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?

remove me?

remove me?

Part IV

Outro

Chapter 10

Related work

Chapter 11

Result

Chapter 12

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 et al. "CSS style attributes". In: (2013).

[8] Dave Evans. "The Internet of Things". In: (Apr. 2011). URL: `http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf` (visited on 02/12/2015).

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

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

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

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

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

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

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

[16]   *Hypermedia*. Aug. 2014. URL: http://en.wikipedia.org/wiki/Hypermedia (visited on 02/12/2015).

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

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

[19]   Gary C. Kessler. "An Overview of TCP/IP Protocols and the Internet". In: (Nov. 2010). URL: http://www.sci.brooklyn.cuny.edu/~parsons/courses/3120-fall-2012/notes/tcp-ip-notes.pdf (visited on 02/17/2015).

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

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

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

[23]   Barry Leiner et al. "Brief History of the Internet". In: (Oct. 2012). URL: http://www.internetsociety.org/sites/default/files/Brief_History_of_the_Internet.pdf (visited on 02/11/2015).

[24]   *libwww*. Jan. 2015. URL: http://en.wikipedia.org/wiki/Libwww (visited on 02/25/2015).

[25]   *Line Mode Browser*. Dec. 2014. URL: http://en.wikipedia.org/wiki/Line_Mode_Browser (visited on 02/25/2015).

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

[27]   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.

[28]   *Mosaic (web browser)*. Feb. 2015. URL: http://en.wikipedia.org/wiki/Mosaic_(web_browser) (visited on 02/25/2015).

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

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

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

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

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

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

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

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

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

[38] W3C. "About The World Wide Web". In: (Jan. 2001). URL: http://www.w3.
      org/WWW/ (visited on 02/12/2015).

[39] W3C. "Document Object Model (DOM)". In: (Jan. 2005). URL: http://www.
      w3.org/DOM/ (visited on 03/05/2015).

[40] 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).

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

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

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

[44] *WorldWideWeb.* Feb. 2015. URL: http://en.wikipedia.org/wiki/WorldWideWeb
      (visited on 02/25/2015).

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

Appendix A

Resources

A.1 Practical problem formulation document

TODO: Should this be in the real document instead of appendix?

A.2 CSS terminology

TODO: 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 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

---

[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`

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%.

---

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