Modular Responsive Web Design: An Experience Report

Lucas Wiener
EVRY AB
Sweden
lucas.wiener@evry.com

Tomas Ekholm
KTH Royal Institute of Technology
Sweden
tomase@kth.se

Philipp Haller
KTH Royal Institute of Technology
Sweden
phaller@kth.se

ABSTRACT

Responsive Web Design (RWD) enables web applications to adapt to the characteristics of different devices such as screen size which is important for mobile browsing. Today, the only W3C standard to support this adaptability is CSS media queries. However, using media queries it is impossible to create applications in a modular way, because responsive elements then always depend on the global context. Hence, responsive elements can only be reused if the global context is exactly the same. This makes it extremely challenging to develop large responsive applications, because the lack of true modularity makes certain requirement changes either impossible or expensive to realize.

In this paper we extend RWD to also include responsive modules, i.e., modules that adapt their design based on their local context, independently of the global context. We present the ELQ project that includes an approach to enabling modular responsivity, and a novel implementation of resize detection of DOM elements. ELQ conforms to the so-called *element queries* which generalize CSS media queries. Importantly, our design conforms to existing web specifications, enabling adoption on a large scale. ELQ is designed to be heavily extensible using plugins. Experimental results show speed-ups of the core algorithms of up to 37x compared to previous approaches.

CCS CONCEPTS

•**Software and its engineering** →**Markup languages;** *Domain specific languages;*

KEYWORDS

Responsive web design, Element queries, CSS, Modularity

ACM Reference format:
Lucas Wiener, Tomas Ekholm, and Philipp Haller. 2016. Modular Responsive Web Design: An Experience Report. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Responsive Web Design (RWD) is an approach to make an application respond to the viewport size and device characteristics. This is currently achieved by using CSS media queries that are designed to conditionally design content by the media, such as using serif fonts when printed and sans-serif when viewed on a screen [25]. In order to reduce complexity and enable reusability, applications are typically composed of modules, i.e., interchangeable and independent parts that have a single and well-defined responsibility [16]. In order for a module to be reusable it must not assume in which context it is being used.

In this paper we focus on the presentation layer of web applications. As it stands, using CSS media queries to make the presentation layer responsive precludes modularity. The problem is that there is no way to make a module responsive without making it context-aware, due to the fact that media queries can only target the viewport; this means that responsive modules can only respond to changes of the (global) viewport. Thus, a responsive module using media queries is layout dependent and has both reduced functionality and limited reusability [27]. As a result, media queries can only be used for RWD of non-modular static applications. In a world where no better solution than media queries exists for RWD, changing the layout of a responsive application becomes a cumbersome task since it may require many responsive modules to be updated. The limitations of CSS with regard to compositionality are well known (as shown in the tweet below). While we do not claim to solve the problem in its entirety, this paper provides a solution to compositionality issues in the context of RWD.

The Problem Exemplified. Imagine an application that displays the current weather of various cities as widgets, by using a weather widget module. The module should be responsive so that more information, such as a temperature graph over time, is displayed when the widget is big. When the widget is small it should only display the current temperature. Users should also be able to add, remove and resize widgets.

Such an application cannot be built using media queries, since the widgets can have varying sizes independent of the viewport (e.g., the width of one widget is 30% while another is 40%). To overcome this problem we must change the application, so that widgets always have the same sizes. This implies that the size of the module and the media query breakpoints are coupled/intertwined, i.e. they are proportional to each other. The problem now is that

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

we have removed the reusability of the weather module, since it requires the specific width that is correctly proportional to the media query breakpoints.

Imagine a company working on a big application that uses media queries for responsiveness (i.e., each responsive module assumes to have a specific percentage of the viewport size). The ability to change is desired by both developers and stakeholders, but is limited by this responsive approach. The requirement of changing a menu from being a horizontal menu at the top to being a vertical menu on the side implies that all responsive modules break, since the assumed proportionality of each module is changed. Even worse, if the menu is also supposed to hide on user input, the responsiveness of the module breaks, since the layout changes dynamically. The latter requirement is impossible to satisfy in a modular way without element queries.

Additionally, it is popular to define breakpoints relative to the font size so that conditional designs respect the size of the content [6]. Media queries can only target the font size of the document root, limiting their functionality drastically. With element queries breakpoints may be defined relative to the font size of the targeted element.

As we can see, even with the exemplified limited requirements there are still significant restrictions when using media queries for responsive modules.

Requirements. The desired behavior of a responsive module is having its inner design respond to the size of its *container* instead of the viewport. Only then is a responsive module independent of its layout context. Realizing responsive modules requires CSS rules that are conditional upon *elements*, instead of the global viewport. We have identified the following requirements of a solution:

- It must provide the possibility for an element to automatically respond to changes of its parent's properties.
- It must conform to the syntax of HTML, CSS, and JavaScript to retain the compatibility of tools, libraries and existing projects.
- It must have adequate performance for large applications that make heavy use of responsive modules.
- It must enable developers to write encapsulated style rules, so that responsive modules may be arbitrarily composed without any conflicting style rules.

Approach. In this paper we extend the concept of RWD to also include responsive modules. The W3C has discussed such a feature under the name of *element queries* given its analogy to media queries [26]. This paper presents a novel implementation of element queries in JavaScript named ELQ that enables new possibilities of RWD. Our approach satisfies all requirements given above. We have released ELQ as an open-source library under the MIT license.[1] The implementation supports all major browsers, including Internet Explorer version 8, Chrome version 42 (the last version compatible with Android version 4), Safari version 5, and Opera version 12.

One could argue that a solution does not need to be executed on the client side, but instead generate media queries on the server side for all modules with respect to the current application layout. However, this approach is insufficient, since it limits modules to applications with static layouts [27]. Also, the generated media queries would not be able to respond to the user changing properties of elements such as layout and font size.

Contributions. This paper makes the following contributions:

- A library that enables responsive modules while conforming to the syntax of HTML, CSS, and JavaScript. We also provide an integration component[2] for the React user interface library.
- Our approach is the first to enable nested elements that are responsive in a modular way, i.e., modules fully encapsulate any styling required for RWD. As a side effect, responsive modules may also be arbitrarily styled with CSS independent of their context.
- An extensible architecture that enables plugins to significantly extend the behavior of ELQ, our library implementation. This makes it possible to create plugins in order to enable new features and to ease integration of ELQ into existing projects.
- A novel implementation of element resize detection[3] that offers substantially higher performance than previous approaches. The implementation batch-processes DOM operations in order to avoid layout thrashing (i.e., forcing the layout engine to perform multiple independent layouts).
- A run-time cycle detection system that detects and breaks cycles stemming from cyclic rules due to unrestricted usage of element queries [27].

The rest of the paper is organized as follows. Section 2 introduces ELQ and its API from a user's perspective. Section 3 provides an overview of the implementation of ELQ's element resize detection system. Section 4 evaluates the performance of ELQ. Section 5 reports on case studies of using ELQ. Section 6 relates ELQ to prior work. Section 7 discusses limitations of ELQ and related libraries, as well as the current state of standardization of element queries. Section 8 concludes.

2 OVERVIEW OF ELQ

An *element breakpoint* is defined as a point of an element property range which can be used to define conditional behavior, similar to breakpoints of media queries. For example, if an element that is 300 pixels wide has two width breakpoints of 200 and 400 pixels the *element breakpoint states* are "wider than 200 pixels" and "narrower than 400 pixels".

The main idea is to define element breakpoints of interest so that children can be adapted to the different breakpoint states. As a library, ELQ provides a JavaScript API to registering element breakpoints, and detecting breakpoint state changes. ELQ then observes the elements, in order to automatically let the system know when a breakpoint has changed state. The JavaScript API is extensible through plugins. Mainly, plugins provide alternative behaviors and API's for breakpoint registration and action on breakpoint state changes. In our companion technical report [28] we show an example plugin that provides a grid API similar to the CSS Bootstrap framework.

[1] https://github.com/elqteam/elq

[2] https://github.com/elqteam/react-responsive-block

[3] https://github.com/wnr/element-resize-detector

Default plugins. The default plugins of ELQ let users define element breakpoints by HTML attributes in addition to the JavaScript API:

```
<div class="foo" data-elq-breakpoints-widths="300 500">
  <p>When in doubt, mumble.</p>
</div>
```

The plugins also update element classes to reflect the current breakpoint states, which may be targeted in CSS selectors. For instance, if the element is 400 pixels wide, the element has the two classes `elq-min-width-300px` and `elq-max-width-500px`. For each breakpoint only the min/max part changes, to mimic CSS media queries. This is how the classes may be used in CSS to conditionally style the children:

```
.foo.elq-max-width-300px {
    background-color: blue;
}
.foo.elq-min-width-300px.elq-max-width-500px {
    background-color: green;
}
.foo.elq-min-width-500px p {
    color: white;
}
```

This is however not sufficient for nestable modules since there is no way to limit the CSS matching search of the selectors. The last style rule specifies that all paragraph elements should have white text if *any* `.foo` ancestor breakpoints element is wider than 500 pixels. Since the ancestor selector may match elements outside of the module, such selectors are dangerous to use in the context of responsive modules. The problem may be somewhat reduced by more specific selectors and such, but it cannot be fully solved for arbitrary styling [27].

To enable nestable modules, the default plugins let us define elements to "mirror" the breakpoints classes of the nearest ancestor breakpoints element (the target of the mirror element). This means that the mirror element always reflects the element breakpoint states of the target. The following is an example of using mirroring to have a `.foo` module contain another `.foo` module:

```
<div class="foo" data-elq-breakpoints-widths="300 500">
  <div class="foo" data-elq-breakpoints-widths="300 500">
    <p data-elq-mirror>...</p>
  </div>
  <p data-elq-mirror>...</p>
</div>
```

The paragraph elements are told to mirror the nearest breakpoints element by the `data-elq-mirror` annotation. Then, the conditional style of pragraph elements may be written as a combinatory selector:

```
.foo p.elq-min-width-500px { color: white; }
```

Since the breakpoint state class is now combined with the paragraph, the conditional style will only be applied in relation to the actual desired breakpoints element parent.

2.1 Advanced breakpoint logic

Morphing shared markup into structurally different layouts is complex by only using CSS. A better way is to produce the different markup by using the expressiveness of JavaScript.

For instance, one interface design approach is to have buttons sorted in importance priority from left to right (more important to the right) for wide views. For narrow views, it might be desired to have the buttons stacked vertically sorted in priority top to bottom. Since the natural flow of HTML is to render top left to bottom right, we want to structurally change the markup order of the buttons. Another example is when table columns should disappear for narrow views, and the data instead should be presented elsewhere.

The CSS solution to this can be very complex, compared to a simple JavaScript condition rendering different markup for the two cases. The maintainability of a local solution in JavaScript excells over a solution in the global space of CSS. ELQ provides a JavaScript API that is suitable to build higher-level abstractions upon. We have for instance created a React component[4] on top of ELQ that sends the current breakpoint state as an input property to the view component, for seamless integration into React-based code. This component is heavily used in our responsive view modules.

3 ELEMENT RESIZE DETECTION

Unfortunately, there is no standardized resize event for arbitrary elements [24]. It is possible to resort to polling the element sizes in order to detect changes, but there are also two event-based approaches to detecting element resize events as originally presented by [3]. One is to use `object` elements, since frame elements emit resize events [27].

It is also possible to use multiple overflowing elements that listen to scroll events in order ot detect size changes, which is the approach of ELQ. The overflowing elements are styled so that scroll events are emitted when the target element is resized. For detecting when the target element shrinks, two elements are needed; one for handling the scrollbars and one for causing them to scroll. Similarly, for detecting when the target element expands, two elements are needed in the same way. As this approach only injects `div` elements, it offers greater opportunities for optimizations. The main algorithm that is performed when an element e is to be observed for resize events is the following:

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

Layout thrashing can be avoided by batching DOM operations, which results in a significant performance improvement as shown in

[4] https://github.com/elqteam/react-responsive-block

Section 4. The algorithm steps are batch processed in the following levels:

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

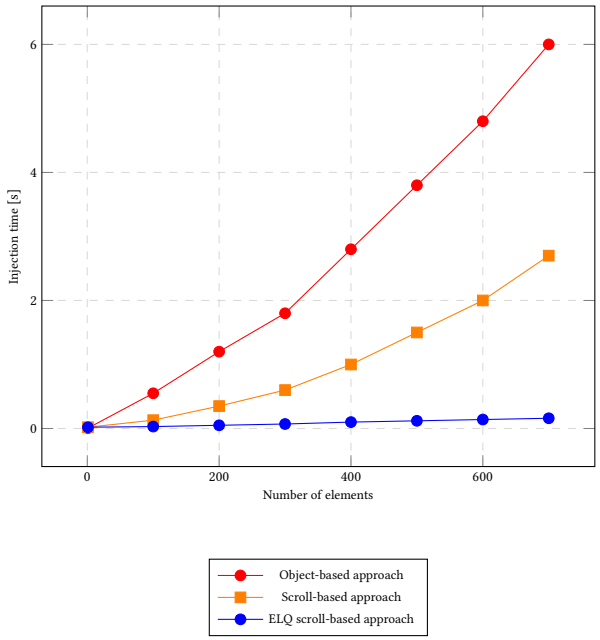
Since repositioning a scrollbar in some layout engines forces a layout, such operations need to be performed after all other queueable operations have been executed. Therefore, step 5 is performed in level 3 as the last step. Even though some layout engines are unable to queue the repositioning of scrollbars, it is still beneficial to batch process the algorithm, since only pure layouts need to be performed (instead of having to recompute styles, and synchronize the DOM and render trees before each layout). As step 6 is performed by the layout engine asynchronously and does not interact with the DOM, it does not need to be batch processed.

4 EXPERIMENTAL EVALUATION

Only the performance of the element resize detection system has been evaluated. This is due to the fact that detecting element resize events entails all the significant performance penalties of ELQ. Fortunately, element resize detection is the common denominator of all automatic libraries and the results of this system can be compared faithfully. Measurements and graphs show evaluations performed in Chrome version 42 unless stated otherwise. Previous implementations use one of two approaches [3]: (a) *object-based* resize detection, which uses `object` elements, and (b) *scroll-based* resize detection, which uses overflowing elements. The approach of ELQ extends the scroll-based approach with batch processing to increase performance [28].

The following plot compares the start-up performance of ELQ's scroll-based approach with the other two approaches. ELQ achieves a 37-fold speedup compared to the object-based approach and a 17-fold speedup compared to the scroll-based approach when preparing 700 elements for resize detection. The memory footprint of the object approach grows roughly by 0.55 MB per element, in contrast to the scroll approach whose memory consumption is insignificant.

Both approaches perform well when detecting resize events. For few elements, they both detect changes with a delay of roughly 25 ms. The object approach scales a bit better, as shown in table 1. However, the installation time penalty is significant at scale for the object approach. ELQ uses both approaches in order to target legacy browsers. The default strategy is to use the scroll approach, but it is possible to manually choose which strategy to use. See table 1 for the performance of ELQ's two resize detection strategies in different browsers.



Browsers	Injection		Resize detection	
	scroll	object	scroll	object
Chrome v. 42	30 ms	550 ms	25 ms	20 ms
Firefox v. 40	150 ms	1000 ms	70 ms	30 ms
Safari v. 9	100 ms	400 ms	30 ms	20 ms
Internet Explorer v. 11	350 ms	6700 ms	100 ms	80 ms
iOS Safari v. 9	350 ms	1600 ms	150 ms	60 ms
Android v. 5 Chrome v. 39	40 ms	1000 ms	20 ms	10 ms

Table 1: Performance of the two resize detection strategies, operating on 100 elements.

5 CASE STUDIES

In order to evaluate ease of integration with existing projects, we have adapted the popular Bootstrap framework (version 3) to use element queries instead of media queries. According to its website, "Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web." [15]

To modularize Bootstrap, we redefine the behavior of its responsive elements so that they no longer respond to the viewport but to enclosing container elements. The following observation guides our modularization: all responsive elements should respond to their closest enclosing `container` or `container-fluid` element. Both classes are used in Bootstrap to define new parts of a page (e.g., a grid is required to have a container ancestor). We also enable them to be nestable, which is important to satisfy the requirement of composable modules.

The breakpoints of the container elements are defined using the `elq-breakpoints` API. Since the Bootstrap API uses a predefined set of breakpoints, they are all added to the container elements dynamically with JavaScript. According to this design, we convert all responsive elements of Bootstrap to `elq-mirror` elements,

since they need to mirror the breakpoints of the nearest ancestor `elq-breakpoints` element. Since container elements may be nested, they have both the `elq-breakpoints` and `elq-mirror` behavior.

The breakpoints of Bootstrap are defined as the following constants:[5]

```
@screen-sm-min: 480px;
@screen-md-min: 992px;
@screen-lg-min: 1200px;
```

The following example shows how Bootstrap's style definitions are changed from using media queries to using ELQ's element queries:

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

// Original Bootstrap using media queries.
.container {
  @media (min-width: @screen-sm-min) {
    width: @container-sm;
  }
  ...
}

// ELQ Bootstrap using element queries.
.container {
  &.elq-min-width-@{screen-sm-min} {
    width: @container-sm;
  }
  ...
}
```

By using the power of preprocessors, ELQ element queries become as pleasant to work with as media queries. In fact, only about 0.6% of the style code (LESS syntax) need to be altered. Most changes are similar to the one shown above, which replaces the media query syntax with the ELQ element queries syntax. This is especially advantageous when keeping a forked project up to date with the original project, as fewer diverged lines implies a lowered risk of merge conflicts.

In summary we have shown that it is easy to adapt existing responsive code to use ELQ's element queries instead of media queries. With only a small number of changes, the widely used Bootstrap framework can be modularized.

Industrial use of ELQ. We have also been gathering experience with the application of ELQ in large financial applications developed at EVRY. Our practical experience shows that complex applications require a variety of features to be supported by element queries. Such features can be provided effectively by ELQ plugins. We have noticed that in most of our responsive modules, it has been beneficial for us to use the JavaScript API to conditionally render whole chunks of HTML instead of only changing the style using CSS. Two teams at EVRY have independently come to this same conclusion, and have developed plugins to ease the usage with the different frameworks that the teams are using (Angular and React).

6 RELATED WORK

The libraries [1, 5, 9, 17, 23] have in common that they require developers to write custom CSS, unlike ELQ. Since they do not conform to the CSS standard, new features are supported through custom CSS parsed using JavaScript. As shown by [9, 23] quite advanced features can be implemented this way. Additionally, adding new CSS features implies that it is possible to implement a solution to element queries that does not require any changes to the HTML, which may be preferable since all styling then can be written in CSS. However, there are numerous drawbacks with libraries that require custom CSS. Extending the CSS syntax violates the requirement of compatability and also introduces a compilation step which decreases the performance [27].

Resize detection. The libraries [8, 9, 11–14, 18, 21, 29] simply observe the viewport resize event, which may be enough for static pages, but not enough to satisfy the requirements of reusable responsive modules [27]. Approach [22] does not detect resize events at all. Like ELQ, [1, 4, 10, 17, 19, 23] observe *elements* for resize events. The libraries [1, 10] use polling while ELQ and [4, 17, 19, 23] use different injection approaches. As shown in Section 4, the injection approaches used by related libraries have significantly less performance than ELQ's element resizing detection system.

Constraint-based CSS. CCSS [2] proposes a more general and flexible alternative to CSS. The idea of CCSS is to layout documents based on constraints. The Grid Style Sheets library [23] builds upon the ideas of CCSS. While not directly offering element queries, the library enables the possibility to conditionally style elements by element criteria and thus makes it a good candidate to solve the problem of responsive modules. However, the library has two major issues: performance and browser compatibility [7]. In contrast, ELQ only considers element queries, but without browser compatibility limitations and with higher performance.

7 DISCUSSION

Inherent to all current implementations of element queries is that the conditional style is applied "one layout behind". Since a layout pass needs to have been performed in order for an element to change size, the conditional styles defined by the element queries cannot be applied until the next layout. Therefore, the element displays an invalid style until another layout has been performed. The flash of invalid design is usually so short that users do not notice it, but in some cases developers need to work around this issue to avoid more apparent results (especially when combined with animations). Another caveat is presented by the element resize detection approaches, as they mutate the DOM. Developers need to be aware of this as CSS selectors and JavaScript may also match the injected elements. This is easily avoided by good practices.

8 CONCLUSION

This paper extends Responsive Web Design (RWD) with *responsive modules* through element queries. Our approach is the first to enable nested elements that are responsive in a modular way, i.e., modules fully encapsulate any styling required for RWD. Our implementation, ELQ, is fully compatible with existing web standards and technologies. The element resize detection of ELQ performs up to 37x better than previous algorithms. We present a case study which shows that changing only about 0.6% of the LOC is sufficient to enable the use of the popular Bootstrap framework in responsive modules. We also report on first commercial usage of ELQ.

[5] The Bootstrap CSS is generated using the LESS preprocessor [20].

REFERENCES

[1] Chris Ashton. 2015. Localised CSS. (2015). Retrieved April 29, 2015 from https://github.com/ChrisBAshton/localised-css

[2] Greg J Badros, Alan Borning, Kim Marriott, and Peter Stuckey. 1999. Constraint cascading style sheets for the web. In *Proceedings of the 12th annual ACM symposium on User interface software and technology*. ACM, 73–82.

[3] Daniel Buchner. 2013. Cross-Browser, Event-based, Element Resize Detection. (2013). Retrieved March 23, 2015 from http://www.backalleycoder.com/2013/03/18/cross-browser-event-based-element-resize-detection/

[4] Daniel Buchner. 2015. Element Queries. (2015). Retrieved April 29, 2015 from https://github.com/csuwildcat/element-queries

[5] Gabriel Felipe. 2015. MagicHTML. (2015). Retrieved April 29, 2015 from https://github.com/gabriel-felipe/MagicHTML

[6] Lyza Gardner. 2012. The EMs have it: Proportional Media Queries FTW! (2012). Retrieved March 2, 2015 from http://blog.cloudfour.com/the-ems-have-it-proportional-media-queries-ftw/

[7] Grid Style Sheets. 2015. Element queries with precompilation. (2015). Retrieved June, 8 2015 from https://github.com/gss/engine/issues/178

[8] Daniel Hägglund. 2015. breaks2000. (2015). Retrieved April 29, 2015 from https://github.com/judas-christ/breaks2000

[9] Tommy Hodgins and Maxime Euzière. 2015. EQCSS. (2015). Retrieved April 29, 2015 from http://elementqueries.com/

[10] Andy Hume. 2015. Selector queries and responsive containers. (2015). Retrieved April 29, 2015 from https://github.com/ahume/selector-queries/

[11] Kumail Hunaid. 2015. Responsive Elements. (2015). Retrieved April 29, 2015 from https://github.com/kumailht/responsive-elements

[12] Tyson Matanich. 2015. ElementQuery. (2015). Retrieved April 29, 2015 from https://github.com/tysonmatanich/elementQuery

[13] Jonathan Neal. 2015. MediaClass. (2015). Retrieved April 29, 2015 from https://github.com/jonathantneal/MediaClass

[14] Truong Nguyen. 2015. SickleS. (2015). Retrieved April 29, 2015 from http://singggum3b.github.io/SickleS/

[15] Mark Otto and Jacob Thornton. 2016. Bootstrap. (2016). Retrieved January 15, 2016 from http://getbootstrap.com/

[16] David Lorge Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058.

[17] François Remy. 2015. prollyfill-min-width. (2015). Retrieved April 29, 2015 from https://github.com/FremyCompany/prollyfill-min-width/

[18] Sam Richard. 2015. eq.js. (2015). Retrieved April 29, 2015 from github.com/Snugug/eq.js

[19] Marc J. Schmidt. 2015. CSS Element Queries. (2015). Retrieved April 29, 2015 from https://github.com/marcj/css-element-queries

[20] Alexis Sellier. 2015. LESS. (2015). Retrieved March 9, 2015 from http://lesscss.org/

[21] Joshua Stoutenburg. 2015. breakpoints.js. (2015). Retrieved April 29, 2015 from https://github.com/reusables/breakpoints.js

[22] Matt Stow. 2015. Class Query. (2015). Retrieved April 29, 2015 from https://github.com/stowball/Class-Query

[23] Dan Tocchini. 2015. Grid Style Sheets 2.0. (2015). Retrieved April 29, 2015 from http://gridstylesheets.org/

[24] W3C. 2000. Document Object Model Events. (2000). Retrieved March 14, 2015 from http://www.w3.org/TR/DOM-Level-2/events.html

[25] W3C. 2012. Media Queries. (2012). Retrieved May 19, 2015 from http://www.w3.org/TR/css3-mediaqueries/

[26] W3C. 2013. W3C public mail archive: The :min-width/:max-width pseudo-classes. (2013). Retrieved April 28, 2015 from https://lists.w3.org/Archives/Public/www-style/2013Mar/0368.html

[27] Lucas Wiener. 2015. *ELQ: Extensible Element Queries for Modular Responsive Web Components*. Master's thesis. KTH Royal Institute of Technology, Sweden.

[28] Lucas Wiener, Tomas Ekholm, and Philipp Haller. 2015. Modular Responsive Web Design using Element Queries. *CoRR* abs/1511.01223 (2015). http://arxiv.org/abs/1511.01223

[29] Corey Worrell. 2015. Responsive Elements. (2015). Retrieved April 29, 2015 from https://github.com/coreyworrell/responsive-elements