Modular Responsive Web Design using Element Queries

Lucas Wiener[1], Tomas Ekholm[1,2], and Philipp Haller[2]

[1] EVRY AB, Sweden
[2] KTH Royal Institute of Technology, Sweden
lucas.wiener@evry.com,tomase@kth.se,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 which implements our approach. ELQ is a novel implementation of so-called *element queries* which generalize 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.

Keywords: Responsive web design, Element queries, CSS, Modularity, Web

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 [30]. 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 [20]. 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 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 [33]. 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 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 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 [10]. 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 [32]. 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.[3] 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 [33]. 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 new design for element queries that enables responsive modules while conforming to the syntax of HTML, CSS, and JavaScript.
- 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 new implementation 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).

[3] `https://github.com/elqteam/elq`

– A run-time cycle detection system that detects and breaks cycles stemming from cyclic rules due to unrestricted usage of element queries [33].

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

2 Overview of ELQ

Media queries and element queries are similar in the sense that they both enable developers to define conditional designs that are applied by specified criteria. The main difference is the type of criteria that can be used. With media queries critera of the device, document, and media are used, while element criteria are used with element queries. It can somewhat simplified be described as that media queries target the document root and up such as viewport, browser, device, and input mechanisms. Element queries target the document root and down, i.e., elements of the document.

ELQ is designed to be plugin-based for increased flexibility and extensibility. By providing a good library foundation and plugins it is up to developers to choose the right plugins for each project. In addition, by letting the plugins satisfy the requirements it is easy to extend the library with new plugins when new requirements arise.

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, an element breakpoint of 500 pixels in width enables conditional styling depending on if the element is narrower or wider than 500 pixels. An element may have multiple breakpoints. An *element breakpoint state* is defined as the state of the element breakpoint relative to the current element property value. 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".

When the breakpoint states of an element changes, ELQ performs cycle detection in order to detect and handle possible style cycles. If a cycle is detected, the new element breakpoint states are not applied in order to avoid an infinite loop of layouts. The cycle detection system is implemented as an conservative algorithm, and may in some cases detect false positives.

The default breakpoints API. In this section, we use the `elq-breakpoints` plugin API (that is bundled with ELQ as default) that let us define element breakpoints. The main idea is to define element breakpoints of interest so that children can be conditionally styled in CSS by targeting the different element breakpoint states. As CSS3 does not support custom at-rules/selectors [31], responsive elements are

annotated in HTML by element attributes. ELQ then observes the annotated elements in order to automatically update breakpoint state classes. Although not written in the examples, the API also supports attributes defined with the `data-` prefix to conform to the HTML standard [29].

The following example shows the HTML of an element that has two annotated width breakpoints at 300 and 500 pixels:

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

When ELQ has processed the element it has two classes that reflect each breakpoint state. For instance, if the element is 400 pixels wide, the element has the two classes `elq-min-width-300px` and `elq-max-width-500px`. Similarly, if the element is 200 pixels wide the classes are instead `elq-max-width-300px` and `elq-max-width-500px`. So for each breakpoint only the min/max part changes. It may seem alien that the classes describe that the width of the element is both maximum 300 and 500 pixels. This is because we have taken a user-centric approach so that the CSS usage of the classes is similar to the API of media queries. However, developers may use other plugins if this API is undesired.

Now that we have defined the breakpoints of the element, we can conditionally style it in CSS by using the classes as shown in listing 1.1.

```
.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; }
```

Listing 1.1. Example usage of the breakpoint state classes in CSS.

In order for the conditional styles to be applied, the elements that have breakpoints must be activated by the ELQ JavaScript runtime. ELQ can either be required as a module (by the CommonJS or AMD syntax), or it can be included in a HTML `script` tag which then exposes a global constructor `Elq`. The following is an example of how to create an ELQ instance and activating elements:

```
var elq = Elq();
elq.use(myPlugin);
var elements = document.querySelectorAll("[elq]");
elq.activate(elements);
```

In this example we create an ELQ instance and register a plugin with it. Then we query the document for all elements with an `elq` attribute (as annotated in the previous example) and then pass them as an argument to the activation method of ELQ. It should be noted that it is up to developers how to activate elements; annotating elements with `elq` is used for simplicity in the example. The only requirement is that conditionally-styled elements are processed by the `activate` method at some point. This can, for example, also be achieved with a plugin that listens to DOM mutations to perform the activation automatically, or a plugin that parses CSS and activates all elements that have conditional styles defined.

Nested modules. The `elq-breakpoints` API is sufficient for applications that do not need nested breakpoint elements, and similar features are provided by related

libraries such as [22, 34]. However, using such an API in responsive modules still limits composability, since modules then may not exist in an outer responsive context. The reason this API is not sufficient for nested modules is that there is no way to limit the CSS matching search of the selectors. The last style rule of the example given in listing 1.1 specifies that all paragraph elements should have white text if *any* 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 [33].

To solve this problem, we provide a plugin that 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. Then, the conditional style of the mirror element may be written as a combinatory selector that is relative to the nearest ancestor breakpoints element:

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

In this example, the paragraph elements always have the same element breakpoint classes as the parent `elq-breakpoints` elements. This enables us to write CSS that does not traverse the ancestor tree:

```
.foo { width: 50%; } /* So that the nested modules have different size */
.foo p.elq-min-width-500px { color: white; }
```

In the examples we have given so far we have annotated element breakpoints manually; however, it is possible to combine JavaScript and CSS in order to create more flexible APIs. In Appendix A.4 we show an example plugin that provides a grid API similar to the CSS Bootstrap framework.

3 Extensions via plugins

For example, if annotating HTML is undesired it is possible to create a plugin that instead generates element breakpoints by parsing CSS. Likewise, if adding breakpoint state classes to elements is undesired it is possible to create a plugin that does something else when an element breakpoint state has changed. A plugin is defined by a *plugin definition object* and has the structure shown in listing 1.2.

```
var myPluginDefinition = {
  getName: function () {
    return "my-plugin";
  },
  getVersion: function () {
    return "0.0.0";
  },
  isCompatible: function (elq) {
```

```
    return true;
  },
  make: function (elq, options) {
    return {
      // Implement plugin instance methods.
      ...
    };
  }
};
```

Listing 1.2. The structure of plugin definition objects.

All of the methods are invoked when registered to an ELQ instance. The `getName` and `getVersion` methods tells the name and version of the plugin. The `isCompatible` tells if the plugin is compatible with the ELQ instance that it is registered to. In the `make` method the plugin may initialize itself to the ELQ instance and return an object that defines the plugin API accessible by ELQ and other plugins. ELQ invokes certain methods of the plugin API, if implemented, to let plugins decide the behavior of the system. Those methods are the following:

- `activate(element)` Called when an element is requested to be activated, in order for plugins to initialize listeners and element properties.
- `getElements(element)` Called in order to let plugins reveal extra elements to be activated in addition to the given element.
- `getBreakpoints(element)` Called to retrieve the current breakpoints of an element.
- `applyBreakpointStates(element, breakpointStates)` Called to apply the given element breakpoint states of an element.

In addition, plugins may also listen to the following ELQ events:

- `resize(element)` Emitted when an ELQ element has changed size.
- `breakpointStatesChanged(element, breakpointStates)` Emitted when an element has changed element breakpoint states (e.g., when the width of an element changed from being narrower to being wider than a breakpoint).

There are two main flows of the ELQ system; activating an element and updating an element. When ELQ is requested to activate an element, the following flow occurs:

1. Initialize the element by installing properties and a system that handles listeners.
2. Call the `getElements` method of all plugins to retrieve any additional elements to activate. Perform an activation flow for all additonal elements.
3. Call the `activate` method of all plugins, so that plugin-specific initialization may occur.
4. If any plugin has requested ELQ to detect resize events of the element, install a resize detector to the element.
5. Pass the element through the update flow.

The update flow is as follows:

1. Call the `getBreakpoints` method of all plugins to retrieve the breakpoints of the element.
2. Calculate the breakpoint states of the element.
3. If any state has changed since the previous update:
   (a) Perform cycle detection. If a cycle is detected, then abort the flow and emit a warning.
   (b) Call the `applyBreakpointStates` method of all plugins in order for plugins to apply the new element breakpoint states.
   (c) Emit an `breakpointStatesChanged` event.

Of course, there are options to disable some of the steps such as cycle detection and applying breakpoint states. The update flow is also triggered by element resize events.

Plugins may also use an extended API of ELQ that offers access to subsystems such as the plugin handler, cycle detector, batch processor, etc. The extended API is exposed to plugins as an argument to the `make` method of the plugin definition object. In addition, plugins may set behavior properties of an element by the `element.elq` property. It is also possible for plugins to define own behavior properties for inter-plugin collaboration, or for storing plugin-specifc element state. Examples of behavior properties of the ELQ core are:

- `resizeDetection` Indicates if resize detection should be performed.
- `cycleDetection` Indicates if cycle detection should be performed.
- `updateBreakpoints` Indicates if the element should be passed through the update flow.
- `applyBreakpointStates` Indicates that plugins may apply breakpoint states of the element (for some elements it is only necessary to emit element breakpoint state changes, without applying them to the actual element [33]).

4 Element resize detection

Unfortunately, there is no standardized resize event for arbitrary elements [28]. 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 [6]. One is to use `object` elements, since frame elements emit resize events [33]. See Appendix A.2 for a more in-depth description of the polling and object-based approaches.

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 than the other approaches. 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 using a leveled batch processor (as described in Appendix A.3), which results in a significant performance improvement as shown in Section 5.1. 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 the 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 that 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 repositing 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.

5 Empirical evaluation

5.1 Performance

Only the performance of the element resize detection system has been performed. This due to the fact that detecting element resize events entails the significant performance penalties of ELQ. Also, it is hard to compare performance results of related libraries since the functionality is different. 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.

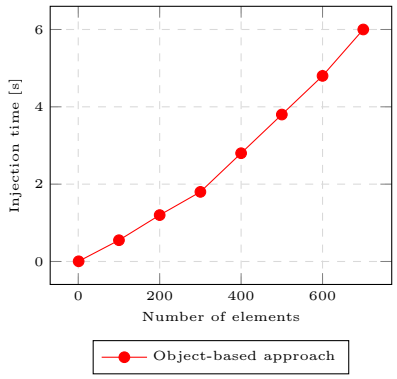

Fig. 1. The injection performance of the object-based approach.

The object-based approach (as presented by [6]) performs well when detecting resize events, which it does with a delay of 30 ms for 100 elements. However, the injection performance is not great as presented in figure 1. As shown by the graph, the injection can be performed with adequate performance as long as the number of elements is low. The approach does not scale well as the number of elements increases. This is probably due to the fact that the heap memory usage grows roughly by 0.55 MB per element.
As the scroll-based approach (as presented by [6]) does not inject `object` elements the memory footprint is reduced significantly, which improves the injection performance. The amount of used memory is too low for reliable measurements. See figure 2 for graphs that show how the ELQ scroll-based approach performs compared to the other two approaches. As evident in the figure, the optimized ELQ approach has significantly reduced injection times. It 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. It also performs well when detecting resize events, which it does with a delay of 25 ms for 100 elements.

ELQ uses the object-based approach as a fallback for legacy browsers. Therefore the performance of the ELQ resize detection system is at minimum as performant as related approaches. See table 1 for the performance of ELQ's two resize detection strategies in different browsers.

5.2 Case studies

In this section we aim to provide answers to the following questions:

- How can ELQ be used to modularize existing responsive code bases?
- How much effort is this modularization?

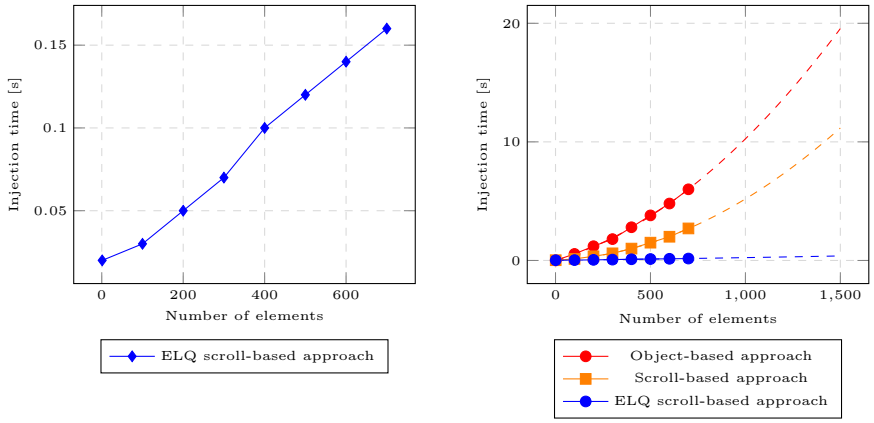

Fig. 2. The left graph shows the injection time of the ELQ scroll-based approach. The right graph shows all three approaches, including graph predictions by polynomial regression.

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.

In order to answer the questions, 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." [19]

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:[4]

```
@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. In addition to the Bootstrap case study, we have been gathering experience with the application of ELQ in large financial applications 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.

6 Discussion

Limitations. 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 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

[4] The Bootstrap CSS is generated using the LESS preprocessor [25].

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. It should be noted that all limitations described only affects the elements that uses the element queries functionality. ELQ does not impose potential problems to other parts of the DOM other than where applied explicitly. Currently ELQ only supports breakpoints for the width and height element properties, as it has been identified as the general use case [33]. In the future, we aim to support plugins to define custom breakpoint properties.

Standardization. It is stated on the W3C's www-style mailing list [32] by Zbarsky (Mozilla), Atkins, and Sprehn (both Google) that element queries are infeasible to implement without restricting them. By limiting element queries to specially separated container elements that can only be queried by child elements, many of the problems are resolved [2, 1]. Therefore, the Responsive Issues Community Group (RICG) is currently investigating the possibility of standardizing *container* queries. Unfortunately, even such limited container queries require significant effort to implement due to the complex changes to browsers required [1]. Atkins argues that a full implementation that avoids the double layout issue is unlikely to be implemented, and therefore it might be wiser to pursue sub-standards that aids third-party solutions instead. In the future, we hope that ELQ may use the aiding sub-standards pushed by RICG, to achieve greater flexibility and performance. A standardized resize event would enable us to avoid injecting elements, and to reduce the code base of ELQ significantly. Support for custom at-rules/selectors would also enable us to define a more natural API in CSS. Finally, being able to tell elements to ignore children while computing their size would decrease the need for cycle detection.

7 Related Work

Table 2 attempts to classify all existing approaches, of modular RWD, known to us. We discuss these approaches according to two different aspects: (a) syntax extensions and (b) resize detection.

Syntax extensions. The libraries [9, 13, 21, 3, 8] 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 [13, 8] 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 [33].

Implementation	Syntax	Resize detection	Page dynamism	Composability	Cycle detection
MagicHTML [9]	Custom CSS	-	Static	-	-
EQCSS [13]	Custom CSS	Viewport only	Dynamic	Full support	-
Element Media Queries [21]	Custom CSS	Non-void elements	Dynamic	-	-
Localised CSS [3]	Custom CSS	Arbitrary elements	Dynamic	-	-
Grid Style Sheets 2.0 [8]	Custom CSS	Arbitrary elements	Dynamic	Partial support	-
Class Query [27]	-	-	Static	-	-
breakpoints.js [26]	-	Viewport only	Dynamic	-	-
MediaClass [17]	-	Viewport only	Dynamic	-	-
ElementQuery [16]	-	Viewport only	Dynamic	-	-
Responsive Elements [15]	-	Viewport only	Dynamic	-	-
SickleS [18]	-	Viewport only	Dynamic	-	-
Responsive Elements [34]	-	Viewport only	Dynamic	-	-
breaks2000 [12]	-	Viewport only	Dynamic	-	-
eq.js [22]	-	Viewport only	Dynamic	-	-
Element Queries [7]	-	Non-void elements	Dynamic	-	-
CSS Element Queries [24]	-	Non-void elements	Dynamic	-	-
Selector queries and responsive containers [14]	-	Arbitrary elements	Dynamic	-	-
ELQ	-	Non-void elements	Dynamic	Full support	Yes

Table 2. Classification of related approaches to modular RWD.

Resize detection. The libraries [13, 26, 17, 16, 15, 18, 34, 12, 22] simply observe the viewport resize event, which may be enough for static pages, but not enough to satisfy the requirements of reusable responsive modules [33]. Approach [27] does not detect resize events at all. Like ELQ, [3, 14, 21, 8, 7, 24] observe *elements* for resize events. The libraries [3, 14] use polling while ELQ and [21, 8, 7, 24] use different injection approaches. As shown in Section 5.1, the injection approaches used by related libraries have significanly less performance than the element resizing detection system used in ELQ.

Constraint-based CSS. CCSS [4] proposes a more general and flexible alternative to CSS. As the name suggests, the idea of CCSS is to layout documents based on constraints. According to its authors, the constraint-based approach provides extended features and reduced complexity compared to CSS. To solve the constraints CCSS uses the Cassowary constraint solving algorithm [5].

The Grid Style Sheets library [8] builds upon the ideas of CCSS and uses a JavaScript port [23] of Cassowary to solve the constraints at runtime. 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 [11]. One approach to resolve both issues is to precompute the layout in a compilation step at the server. However, precompiling styles implies static layouts. The authors discuss other approaches [11] that would increase the performance while limiting the dynamism of page layout. In contrast, ELQ only considers element queries, but without these limitations and with higher performance.

8 Conclusion

Responsive Web Design (RWD) enables web applications to adapt to the characteristics of different devices, which is achieved using CSS media queries. However, using media queries it is impossible to create responsive applications in

a modular way, because responsive elements then always depend on the global context.

This paper extends RWD to also include responsive modules through element queries. We present ELQ, an open-source implementation of our approach, that conforms to the current standards of HTML, CSS and JavaScript. It enables developers to create responsive modules that are independent of their context, and a way to encapsulate their conditional style rules. The element resize detection of ELQ, used to automatically evaluate element queries on changes of responsive elements, performs up to 37x better than previous algorithms.

Using a case study based on the popular Bootstrap framework we show that large code bases using media queries can be converted to using ELQ's element queries with little effort. Changing only about 0.6% of the LOC of style related code was sufficient to enable the use of Bootstrap in responsive modules. We also report on first commercial usage of ELQ.

We believe ELQ is an important contribution to realizing a modular form of element queries, in particular since standardization bodies like the RICG do not intend to standardize a complete solution. In the future we intend to improve ELQ by using forthcoming standards developed by the RICG to avoid some current limitations.

References

1. CSS containment draft. Retrieved April 29, 2015 from `https://github.com/ResponsiveImagesCG/cq-usecases/issues/7`.
2. RICG IRC log. Retrieved April 29, 2015 from `http://ircbot.responsiveimages.org/bot/log/respimg/2015-03-05#T117108`.
3. C. Ashton. Localised CSS. Retrieved April 29, 2015 from `https://github.com/ChrisBAshton/localised-css`.
4. G. J. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 73–82. ACM, 1999.
5. G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact*, 8(4):267–306, 2001.
6. D. Buchner. Backalleycoder. Retrieved March 23, 2015 from `http://www.backalleycoder.com/`.
7. D. Buchner. Element Queries. Retrieved April 29, 2015 from `https://github.com/csuwildcat/element-queries`.
8. e. a. Dan Tocchini. Grid Style Sheets 2.0. Retrieved April 29, 2015 from `http://gridstylesheets.org/`.
9. G. Felipe. MagicHTML. Retrieved April 29, 2015 from `https://github.com/gabriel-felipe/MagicHTML`.
10. L. Gardner. The EMs have it: Proportional media queries FTW! Retrieved April 28, 2015 from `http://blog.cloudfour.com/the-ems-have-it-proportional-media-queries-ftw/`.
11. Grid Style Sheets. Element queries with precompilation. Retrieved June 8, 2015 from `https://github.com/gss/engine/issues/178`.

12. D. Hägglund. breaks2000. Retrieved April 29, 2015 from https://github.com/judas-christ/breaks2000.
13. T. Hodgins and M. Euzière. EQCSS. Retrieved April 29, 2015 from http://elementqueries.com/.
14. A. Hume. Selector queries and responsive containers. Retrieved April 29, 2015 from https://github.com/ahume/selector-queries/.
15. K. Hunaid. Responsive Elements. Retrieved April 29, 2015 from https://github.com/kumailht/responsive-elements.
16. T. Matanich. ElementQuery. Retrieved April 29, 2015 from https://github.com/tysonmatanich/elementQuery.
17. J. Neal. MediaClass. Retrieved April 29, 2015 from https://github.com/jonathantneal/MediaClass.
18. T. Nguyen. SickleS. Retrieved April 29, 2015 from http://singggum3b.github.io/SickleS/.
19. M. Otto and J. Thornton. Bootstrap. Retrieved October 15, 2015 from http://getbootstrap.com/.
20. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
21. F. Remy. Element Media Queries. Retrieved April 29, 2015 from https://github.com/FremyCompany/prollyfill-min-width/.
22. S. Richard. eq.js. Retrieved April 29, 2015 from https://github.com/Snugug/eq.js.
23. A. Russell. Cassowary/JS. Retrieved April 28, 2015 from https://github.com/slightlyoff/cassowary.js.
24. M. J. Schmidt. CSS Element Queries. Retrieved April 29, 2015 from https://github.com/marcj/css-element-queries.
25. A. Sellier. LESS. Retrieved October 10, 2015 from http://lesscss.org/.
26. J. Stoutenburg. breakpoints.js. Retrieved April 29, 2015 from https://github.com/reusables/breakpoints.js.
27. M. Stow. Class Query. Retrieved April 29, 2015 from https://github.com/stowball/Class-Query.
28. W3C. Document object model events. Retrieved March 14, 2015 from http://www.w3.org/TR/DOM-Level-2/events.html.
29. W3C. HTML 5.1. Retrieved October 10, 2015 from http://www.w3.org/html/wg/drafts/html/master/dom.html.
30. W3C. Media queries. Retrieved April 19, 2015 from http://www.w3.org/TR/css3-mediaqueries/.
31. W3C. Selectors level 3. Retrieved March 19, 2015 from http://www.w3.org/TR/css3-selectors/.
32. W3C. W3C public mail archive: The :min-width/:max-width pseudo-classes. Retrieved April 28, 2015 from https://lists.w3.org/Archives/Public/www-style/2013Mar/0368.html.
33. L. Wiener. ELQ: Extensible Element Queries for Modular Responsive Web Components. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2015.
34. C. Worrell. Responsive Elements. Retrieved April 29, 2015 from https://github.com/coreyworrell/responsive-elements.

A Appendix

A.1 Example Plugin Implementation

The `elq-breakpoints` API that enables developers to annotate breakpoints in HTML, as described in Section 2, is implemented as two plugins. This shows that even the core functionality of ELQ is implemented in terms of plugins. The first plugin parses the breakpoints of the element attributes. The second plugin applies the breakpoint states as classes.

The following is a simplified implementation of the `make` method of the parsing plugin:

```
function activate(element) {
  if (!element.hasAttribute("elq-breakpoints")) {
    return;
  }

  element.elq.resizeDetection       = true;
  element.elq.updateBreakpoints     = true;
  element.elq.applyBreakpointStates = true;
  element.elq.cycleDetection        = true;
}

function getBreakpoints(element) {
  // Parse the "elq-breakpoints-*" attributes
  // and retrieve their breakpoints.
  return ...;
}

// Return the plugin API
return {
  activate: activate,
  getBreakpoints: getBreakpoints
};
```

In the `activate` method the plugin registers that resize detection is needed for the element and that it should be passed through the update flow. It also enables the application of breakpoint states and run-time cycle detection. Although not shown in the simplifed implementation, `applyBreakpointStates` and `cycleDetection` are in some cases disabled.

The plugin that applies the element breakpoint states simply implements the `applyBreakpointStates` method to alter the `className` property of the element using the given element breakpoint states.

A.2 Implementing element resize detection

Polling. A naive approach to detecting element resize events is to have a script continously check elements if they have resized given some interval. This approach is appealing because it does not mutate the DOM, supports arbitrary elements, and it provides excellent compatibility. However, in order to prevent the responsive elements lagging behind the size changes of the user interface, polling needs to be performed quite frequently. The problem is that each poll forces the layout queue to be flushed since the computed style of elements needs to be retrieved in order to know if elements have resized or not [33]. Since the

polling is performed all the time the overall page performance is decreased even if the page is idle, which is undesired especially for mobile devices running on battery.

Injecting Objects. Only documents emit resize events in modern browsers and therefore such events can only be observed for frame elements (since a frame element has its own document). This approach injects `object` elements into the target element, which can be listened to resize events since `object` elements are frames. The `object` is styled so that it always matches the size of the target element and so that it does not affect the page visually. This approach has good browser compatability and excellent resize detection performance, but imposes severe performance impacts during injection since `object` elements use a significant amount of memory as shown in Section 5.1.

A.3 Batch processing in ELQ

Batch processing is the foundation of the performance gains of our approach, and is therefore used by several subsystems. ELQ uses a leveled batch processor, which is implemented as a stand-alone project.[5] It serves two purposes: to process batches in different levels to avoid layout thrashing, and to automatically process batches asynchronously to enable multiple synchronous calls being grouped into a pending batch.

Being able to process a batch in levels is important when different types of operations, that are to be processed in a specific order (usually to avoid layout thrashing), needs to be grouped together in a batch. For example, a function that doubles an element's width and reads the new calculated height benefits by being batch processed in three levels: reading the width, mutating the width, and reading the height. The following is an example implementation of such function that uses the leveled batch processor:

```
var batchProcessor = ...;

function doubleWidth(element, callback) {
  // First level: reading the width.
  var width = element.offsetWidth;
  var newWidth = (width * 2) + "px";

  // Second level: mutating the width.
  // This is executed in level 0 of the batch.
  batchProcessor.add(0, function mutateWidth() {
    element.style.width = newWidth;
  });

  // Third level: reading the height.
  // This is executed in level 1 of the batch,
  // after level 0. Changing the level number
  // from "1" to "0" results in layout thrashing.
  batchProcessor.add(1, function readHeight() {
    var height = element.offsetHeight;
    callback(height);
  });
}
```

[5] `https://github.com/wnr/batch-processor`

It should be noted that the first level is executed asynchronously by the function, and not handled by the actual batch processor. Since each batch is delayed to execute asynchronously, all synchronous calls of the method is grouped into a pending batch. If the batch would not automatically be delayed, layout thrashing would occur when the method is called multiple times. This results in a 45-fold speedup, when applied to 1000 elements, of the function compared to not processing the batch in levels. It also results in a simple API that allows multiple synchronous calls without causing layout thrashing, like so:

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

The `activate` method of ELQ is implemented similarly so it may also be called multiple times synchronously, without performance penalties, like the following example:

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

A.4 A grid API

In this section we present a plugin that defines an API that enables developers to use responsive grids consisting of twelve columns and utility classes very similar to the ones defined by the CSS Bootstrap framework [19]. The goal of the API is to provide an abstraction of element queries, so that developers may focus on responsivity using classes instead of the syntax presented in previous sections. The following is an example grid:

```
iv class="container">
<div class="row">
  <div class="col-500-4 col-700-6">
    ...
  </div>
  <div class="col-500-4" col-700-6>
    ...
  </div>
  <div class="col-500-4 hidden-700-up">
    ...
  </div>
</div>
div>
```

The example grid is defined to be single columned when the width of the grid is below 500 pixels, triple columned when the width is between 500 and 700 pixels, and double columned for when the width is above 700 pixels. The last column is hidden when the width is above 700 pixels.

The column classes define the behaviour of the grid, and have the syntax `col-[breakpoint]-[size]`. The `[breakpoint]` part of a column class is relative to the parent `row` and can be any positive number including an optional unit. Currently, the supported units are `px, em, rem`. If the unit is omitted, `px` is assumed. Grids may also be nested.

The plugin traverses the grid structure to initialize all columns and possible nested grids. It also generates and applies the CSS needed for each column breakpoint automatically to the document. This enables developers to easily create responsive grids in nestable modules.