Modular Responsive Web Design using Element Queries

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

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

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

ABSTRACT

Responsive Web Design (RWD) enables web application 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, severely limiting their reusability. 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.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

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.

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 [3]. 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 it being context-aware, due to media queries only being able to target the viewport. Thus, a responsive module using media queries is layout dependent and has therefore limited reusability. Therefore, 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.

1.1 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 application cannot be built with 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 top menu to being a vertical side menu 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 without element queries.

Additionally, it is popular to define breakpoints relative to the font size. Media queries can only target the font size of the document root, limiting the functionality drastically. With element queries, breakpoints may be defined relative to the font size of the targeted element.

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

1.2 Requirements

The desired behavior of a responsive module is having its inner design responding 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.

- A solution must provide a possibility for an element to change depending on the properties of the parent element. Elements should automatically respond to changes of the parent size so that the correct design can be presented for each size.
- A solution must conform to the syntax of HTML, CSS, and JavaScript so that the compatability of tools, libraries and existing projects is retained.
- A solution must have adequate performance for large applications that make heavy use of responsive modules.
- Extensibility
- Composability

1.3 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. This paper presents a novel implementation of element queries in JavaScript named ELQ, and discusses the new possibilities of GUI design that our implementation enables.

Our approach is implemented as a client side library to be included in applications. 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 is insufficient due to the modules then being limited to applications with static layouts. Also, the generated media queries would not be able to respond to the user changing properties of elements such as layout and font size.

1.4 Contributions

This paper makes the following contributions:

- A new design 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., styling required for RWD is fully encapsulated in the module. As a side effect, responsive modules may also be arbitrarily styled with CSS independently of their context.
- An extensible library architecture that enables plugins to significantly extend the behavior of the library. This makes it possible to create plugins in order to ease integration of ELQ into existing modules or libraries.
- A new implementation technique that offers substantially higher performance than previous approaches. Our technique batch processes DOM operations so that layout thrashing (i.e., forcing the layout engine to perform multiple independent layouts) is avoided.

The rest of the paper is organized as follows:

2. BACKGROUND

- Explain media queries really short (simple example). Maybe explain some terminology.

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

3. OVERVIEW OF ELQ

To enable element queries today, we have developed a novel JavaScript implementation named ELQ. Our implementation satisfied all requirements given in 1.1. The library have been designed to conform to the standards and specifications of the web languages while providing the features required of building advanced constellations of responsive modules. The novelty lies in having significantly improved performance by batch processing DOM operations, and providing an API that enables nesting of modules. ELQ also has excellent browser compatability as it supports Internet Explorer 8 and upwards.

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 cyclic element queries. If a cycle is detected, the new element breakpoint states are not applied in order to avoid an infinite cycle. The cycle detection system is implemented as an conservative algorithm, and may detect false positives.

3.1 The API

As CSS 3 does not support custom at-rules/selectors, responsive elements are annotated in HTML by element attributes. The main idea is to annotate elements with breakpoints of interest so that children can be conditionally styled in CSS by targeting the different element breakpoint states. In this example, we will use the `elq-breakpoints` plugin that observers the element in order to automatically update the breakpoint classes. Although not written in the examples, the library also supports attributes defined with the `data-` prefix to conform to the HTML standard.

The following is example HTML of a an element that has two annotated 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 will always have two classes, one for each breakpoint, that tells if the size of the element is greater or lesser than each breakpoint. For instance, if the element is 400 pixels wide, the element has the two classes `elq-min-width-300px` and `elq-max-width-500`. Similarly, if the element is 200 pixels wide the element the classes are instead `elq-max-width-300px` and `elq-max-width-500`. 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 when using the classes in CSS the API is similar to element queries. However, developers are free to change this API at will as ELQ is plugin-based.

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.

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

.foo.elq-min-width-500px {
  background-color: blue;
}

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

Listing 1: Example usage of the breakpoint state classes in CSS.

3.1.1 Nested modules

This API is sufficient for applications that do not need nested breakpoint elements, and similar features is provided by related libraries. However, using such API in responsive modules still limits the reusability since the modules then may not exist in an outer responsive context. Also, it is customary to compose large/complex modules by smaller modules that may also need to be responsive [3].

The reason this API is not sufficient for nested modules is because there is no way to limit the CSS matching search of the selectors. The selector of the last example given in listing 1 specifies that all paragraph elements should have white text if *any* ancestor breakpoints element is above 500 pixels wide. 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.

To solve this problem, we provide a plugin that let us define elements to "mirror" the breakpoints classes of the nearest ancestor breakpoints element. Then, the conditional style of the mirror element may be written as a combinatory selector that is relative to the nearest ancestor breakpoints element. The following is an example usage of the mirror plugin to enable nested modules:

```
<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>
```

As the paragraph elements are mirroring the nearest `.foo` ancestor, we can now write CSS that does not traverse the ancestor tree:

```
.foo {
  /* So that the nested module
     behaves differently */
  width: 50%;
}

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

Since we in the previous examples have annotated elements manually, the power and flexibility of the API have not been properly displayed. Only when combined with JavaScript, things get more interesting.

3.1.2 A high-level API

To further prove the potential of ELQ, we have created a plugin that enables a high-level API for responsive modules. The API enables developers to use responsive grids and utility classes very similar to the ones defined by the CSS Bootstrap framework. The following is an example grid that uses the Bootstrap API:

```
<div class="container">
  <div class="row">
    <div class="col-md-4 col-lg-6">
      ...
    </div>
    <div class="col-md-4" col-lg-6>
      ...
    </div>
    <div class="col-md-4 hidden-lg-up">
      ...
    </div>
  </div>
</div>
```

The example grid is defined to be single columned for small viewports, triple columned for medium viewports, and double columned for large viewports. It should be noted that the last column is hidden for large viewports.

This API uses media queries and supports a fixed set of breakpoints that can be used to determine when the grid should change layout (depending on the viewport size). The column classes defines the behaviour of the grid, and has the syntax `col-[breakpoint]-[size]`. The predefined breakpoints are `xs, sm, md, lg, xl` and the size may be between 1 and 12.

The syntax of our API does not differ much from the Bootstrap API, but the behavior does. The columns of our API responds to the parent `row` instead of the viewport. Therefore, we have altered the column classes to also accept custom breakpoints so that developers have full control of the grid behavior. The `breakpoint` part of the column class can be any positive number, and may optionally be postfixed with a unit. Currently, the supported units are `px, em, rem`. If the units is omitted, `px` is assumed. The following is an example grid that uses our API:

```
<div 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. It should be noted that the last column is hidden when the width is above 700 pixels.

This enables developers to define grid behaviors by pixel precision in nestable modules, while maintaing the familiar Bootstrap API style.

4. EXTENSIONS VIA PLUGINS

One our of contributions is to allow ELQ to be easily extended with plugins. For example, if annotating HTML is undesired it is possible to create a plugin that instead parses CSS. Likewise, if adding breakpoint classes to element is undesired it is possible to create a plugin that does something else when an element breakpoint state of an element has changed. In order to enable such powerful behavior alterings by plugins, extensibility has been the main focus when designing the ELQ architecture.

A plugin is defined by a *plugin definition object* and has the structure shown in listing 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 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 API accessible by ELQ and other plugins.

When necessary, 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 than a breakpoint to being wider).

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 o 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 initializations may occur.
4. If any plugin has requested ELQ to detect resize events of the element, install an 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 all breakpoints of the element.
2. Element breakpoint states are calculated.
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. In additon to being triggered by the activation flow and plugins, it 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 given to a plugin 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` Tells if resize detection should be performed.
- `cycleDetection` Tells if cycle detection should be performed.
- `updateBreakpoints` Tells if the element should be passed through the update flow.
- `applyBreakpointStates` Tells plugins may apply breakpoint states of the element (it is needed fore some elements to only emit element breakpoint states changes, without applying them to the actual element).

4.1 Example Plugin Implementation

The `elq-breakpoints` API that enables developers to annotate breakpoints in HTML, as described in section **??**, 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 (see listing 2) 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 registeres that resize detection is needed for the element and that is should be passed through the update flow. It also enables the element to have its breakpoint states applied and have cycle detection being performed. 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 by the given element breakpoint states.

5. IMPLEMENTATION

5.1 Batch Processing

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. It serves two purposes: to process

batches in different levels to avoid layout thrashing, and to automatically process batches asynchronously for simpler usage.

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

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

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

From this example it is also clear that automated asynchronous processing of batches is important because the function may be called multiple times synchronously, like so:

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

Since one batch for each function call is processed, layout thrashing occurs which results in a severe performance impact. To solve this, each batch is delayed to execute asynchronously so that all synchronous calls of the method is grouped into a pending batch to be executed asynchronously. 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.

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);
```

5.2 Element Resize Detection

Unfortunately, there is no standardized resize event for arbitrary elements [**?**]. 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). According to the general use case described in Section **??**, a valid limitation is to only support element resize detection for non-void elements (i.e., elements that may contain content). This limitation is important since most approaches depend on injecting elements into the target element. It is a reasonable limitation since void elements can easily be wrapped with non-void elements without affecting the page visually.

A naive approach to detecting element resize events is to have a script continously checking elements if they have resized given some interval (also known as polling). 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 would force the layout queue to be flushed since the computed style of elements needs to be retrieved in order to know if elements have resized or not. Since the polling is performed all the time the overall page performance is decreased even if the page is idle, which is undesired especially for devices running on battery.

It is desired to instead have an event-based approach that only performs additional computations when an actual element resize has happened. This is achieved by the resize detection subsystem of ELQ by using two independent injecting approaches, both originally presented by [**?**].

The first approach is to inject `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. The main algorithm that is performed when an element e is to be observed for resize events is the following:

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

Step 1 and 3 could theoretically be executed in different batches to avoid layout thrashing. Unfortunately, each `ob-`

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

`ject` element creation in step 4 forces a full layout, which makes the batch processing optimization negligible. Since the creation of `object` elements forms the significant performance penalty, no further optimization attempts were made.

The second approach is to injects an element that contains multiple overflowing elements that listen to scroll events. 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 solution 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 using the leveled batch processor described 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 engine.
3. **The forced layout level:** Step 5 is performed, which forces the layout engine 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 if 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 tree 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.

6. EMPIRICAL EVALUATION

6.1 Performance

The following tests were performed on a computer with a 2.5 GHz processor and 16 GB of memory[2]. The library has been tested in the following browsers: Chrome 42.0.2311.152, Firefox 37.0.1 and Safari 8.0.6. Measurements and graphs show evaluations performed in Chrome unless stated otherwise.

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

The object-based solution performs well when detecting resize events. However, injecting `object` elements is quite a heavy task. See figure 1 for graphs that show the performance of the object-based solution. As shown by the graphs, the injection can be performed with adequate performance as long as the number of elements is low. The solution does not scale well as the number of elements increases.

The scroll-based solution also performs well when detecting resize events. As no `object` elements are injected the memory footprint is reduced significantly, which improves the injection performance. See figure **??** for graphs that show how both solutions perform compared to each other. It is clear that the scroll-based solution both performs and scales better than the object-based solution during injection. The amount of memory used by the scroll-based solution is so low that reliable measurements could not be gathered, as the number of elements was not high enough to affect the memory usage noticeably.

Recall that the scroll-based solution was rewritten and optimized, which is referred to as the ELQ scroll-based solution. By avoiding layout thrashing, the injection performance was improved significantly. See figure 2 for graphs that show how it performs compared to the other solutions. As evident in the figure, the optimized ELQ solution has significantly reduced injection times compared to the other two. It achieves a 32-fold speedup compared to the object-based solution and a 13-fold speedup compared to the scroll-based solution when preparing 500 elements for resize detection. The ELQ solution also scales better, as more clearly shown in figure 3 that includes polynomial regression graphs for all three solutions. Both scroll-based solutions have the same memory footprint (i.e., too low for reliable measurements).

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

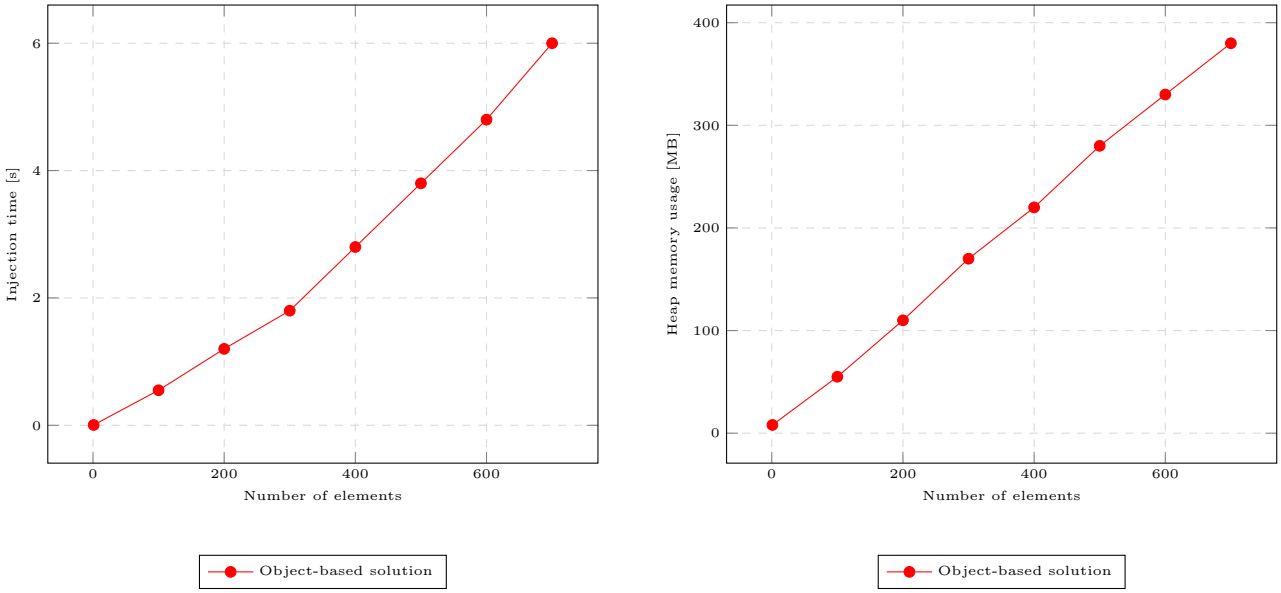

Figure 1: The injection performance of the object-based solution. The left graph shows the injection time. The right graph shows the heap memory used when all `object` elements have been injected.

Firefox and Safari. As shown, great performance can be achieved with the optimized ELQ scroll-based solution in Chrome. Unfortunately, there is no silver bullet to observing element resize events; as the other browser behave differently. See table 1 for the performance of the object-based and scroll-based solutions operating on 100 elements in different browser. The ELQ scroll-based solution is preferred for Chrome, as the injection is 32-fold faster (when operating on 500 elements) than the object-based solution while the resize detection performance is the same for both solutions. In Firefox, the object-based solution detects resize events 2-fold faster than the ELQ scroll-based solution when operating on 100 elements (still, 100 ms for detecting resize events is acceptable). However, the injection time needed for the object-based solution is 5.5-fold of the time needed for the ELQ scroll-based solution. The ELQ scroll-based solution is therefore probably desired in Firefox for the general use case (as described in Section **??**). In Safari, the ELQ scroll-based solution detects resize events in 800 ms while the object-based solution detects them in 25 ms, which of course is unacceptable. Unfortunately, the injection time needed for the object-based solution is 3-fold slower than the ELQ scroll-based solution. Since a delay of 800 ms when detecting resize events is undesired in most use cases, the object-based solution is preferred for Safari. Recall from Section **??** that this is due to WebKit and Gecko not being able to queue the scroll mutation operations as Blink does.

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

Table 1: Performance of the object-based and ELQ scroll-based solutions in different layout engine when operating on 100 elements.

MB.

6.2 Case studies

Plugins makes it easy to integrate.

7. DISCUSSION

- Performance, APIs, Features.
- The mirror functionality of ELQ makes it uniquely suitable for nested modules.

7.1 Limitations

(Drawbacks) No good alternative. Mention RICG (perhaps standard that solves the issue in the future).

7.2 Standardization

It is stated on the W3C's www-style mailing list [4] by Zbarsky of Mozilla, Atkins of Google and Sprehn of 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 requires significant effort to implement [1]. Atkins argues that a correct and full implementation 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.

We want RICG to standardize the following: bla bla bla

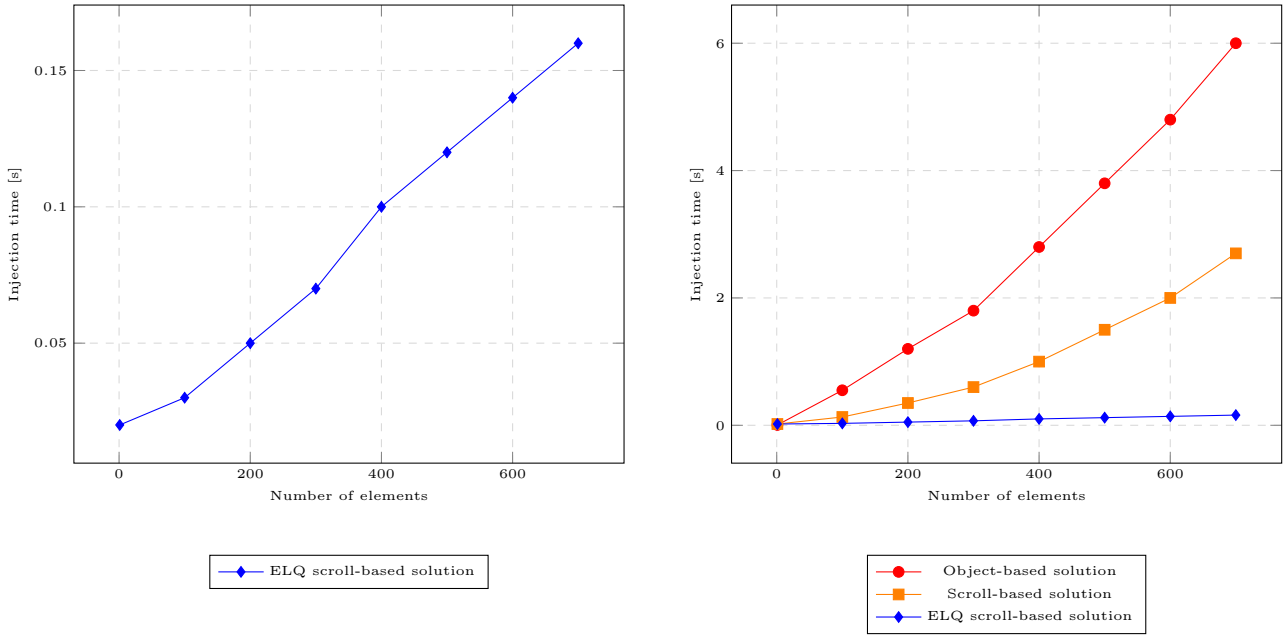

Figure 2: The injection performance of the optimized ELQ scroll-based element resizing detection solution. The left graph shows the injection time of the ELQ scroll-based solution. The right graph shows all three solutions for comparison. The heap memory usage graph has been omitted as the memory usage of the scroll-based solutions is too low for reliable measurements.

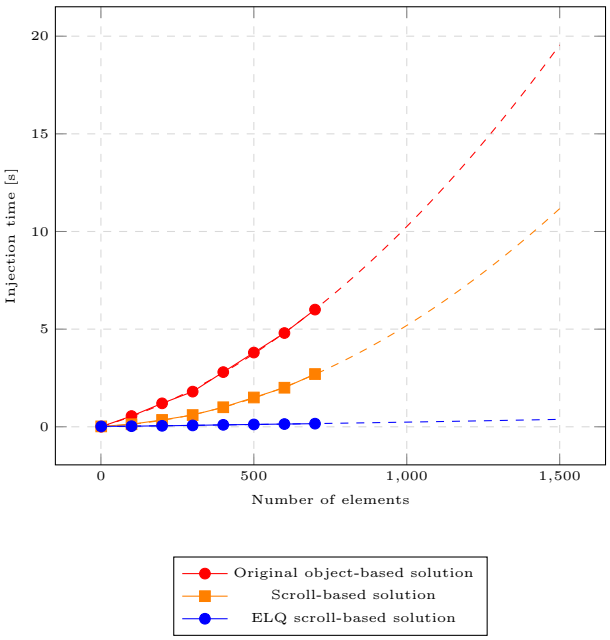

Figure 3: The injection performance of all three solutions including graph predictions by polynomial regression.

8. RELATED WORK

9. CONCLUSIONS

- Production ready.
- Probably no standard (or not in a long time).

10. ACKNOWLEDGMENTS

The authors would like to thank EVRY AB for sponsoring the ELQ project, and the supporting projects for element resize detection and batch processing.

11. REFERENCES

[1] Css containment draft. An issue that discusses why a special element viewport element is needed and why a standard for CQ might be hard to push.

[2] Ricg irc log. The log where the element query limitations were discussed.

[3] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[4] W3C. W3C public mail archive: The :min-width/:max-width pseudo-classes. `https://lists.w3.org/Archives/Public/www-style/2013Mar/0368.html` accessed 2015-04-28.