Allowing Responsive Web Modules

Ben Trovato
Institute for Clarity in Documentation
1932 Wallamaloo Lane
Wallamaloo, New Zealand
trovato@corporation.com

G.K.M. Tobin
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
webmaster@marysville-ohio.com

Lars Thørväld
The Thørväld Group
1 Thørväld Circle
Hekla, Iceland
larst@affiliation.org

ABSTRACT

Categories and Subject Descriptors

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

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

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.

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 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.

In listing 3.1 we see an element that has annotated two 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 by using the classes. See listing 3.1 for an example of how the classes can be used.

```
.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;
}
```

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 3.1 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. See listing 3.1.1 for example markup 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 as shown in listing 3.1.1.

```
.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 resposnive grids and utility classes very similar to the ones defined by the CSS Bootstrap framework. See listing 3.1.2 for an example grid that uses the Bootstrap API. 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.

```
<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>
```

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` 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. See listing 3.1.2 for an example grid that uses our API. 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.

```
<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>
```

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 a 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 4.

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

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:

- `start(element)` Called when an element is requested to be started, in order for plugins to initialize listeners and element properties.
- `getElements(element)` Called in order to let plugins reveal extra elements to be started 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 breakpoint states of an element.

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

- `resize(element)` Emited when an ELQ element has changed size.
- `breakpointStatesChanged(element, breakpointStates)` Emited when an element has changed 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; starting an element and updating an element. When ELQ is requested to start an element, the following flow occurs:

1. The element is intialized by installing properties and a system handling listeners.
2. The `getElements` method of all plugins is called to retrieve any additional elements to start. Additonal elements will go through an own flow.
3. The `start` method of all plugins is called so that plugin specific initialized may occur.
4. If any plugin has requested ELQ to detect resize events of the element, an resize detector is installed.
5. The element is passed through the update flow.

The update flow is as follows:

1. The `getBreakpoints` method of all plugins is called to retrieve all breakpoints of the element.
2. Breakpoint states are calculated.
3. If any state has changed since the previous update:
   (a) Cycle detection is performed.
   (b) The `applyBreakpoints` method of all plugins is called.
   (c) The `breakpointStatesChanged` event is emited.

Of course, there are options to disable some of the steps such as cycle detection and applying breakpoints. In additon to being triggered by the start flow and plugins, it is also triggered by element resize events.

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. One plugin parses the breakpoints of the element attributes and one plugin applies the breakpoint classes. The simplified code of the `make` method of the parsing plugin is presented in listing **??**.

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

    element.elq.resizeDetection = true;
    element.elq.updateBreakpoints = true;
    element.elq.applyBreakpoints = true;
    element.elq.cycleCheck = true;
}

function getBreakpoints(element) {
    return ...
}

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

The applying plugin simply implements the `applyBreakpoints` method to alter the `className` property of the element by the given breakpoint states.

5. IMPLEMENTATION

Batching. Resize listening.

6. EMPIRICAL EVALUATION

6.1 Performance

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

8. RELATED WORK

9. CONCLUSIONS

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

10. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

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.