Chapter 2 – Theory

In this chapter we’ll elaborate on the technologies, languages and concepts on which the thesis is based, as well as discuss related work.

# Required knowledge

## Hypertext Markup Language (HTML)

HTML is the main markup language for displaying web pages and other information that can be displayed in a web browser. The language consists of preset tags that form elements with semantic meaning that web browsers use to interpret the content of a web page. These elements can have attributes that primarily exist as name-value pairs. Common attributes include “id”, which defines a unique document-wide identifier for the element, and “class”, which gives the ability to classify similar elements. This can be used for both semantic and presentational purposes.

While earlier HTML standards included presentational tags for manipulating the look of data in a document, they were deprecated in the HTML 4 standard of 1999 and made illegal in HTML 5, giving the responsibility of presentation to Cascading Style Sheets (CSS), and leaving HTML with giving documents structure and semantic meaning.

<!DOCTYPE html>

<html>

<head>

<title>Hello HTML</title>

</head>

<body>

<p>Hello World!</p>

</body>

</html>

Code Snippet : A Minimal HTML document with a title and a paragraph of text.

Web browsers interpret HTML and the document is rendered in a Document Object Model (DOM) hierarchy that defines the structure of a web page. Furthermore scripts embedded in the markup, such as JavaScript (JS), can manipulate this HTML DOM after it’s been rendered to change the behavior and layout of a web page – creating dynamic web pages.

Today the World Wide Web Consortium (W3C) is maintaining the HTML standard. But in the beginning, HTML was largely shaped by various browser vendors who, for the most part, did what they pleased and added tags as they saw fit. Doing this lead to the HTML standard containing many elements and attributes that are either deprecated or gone today, often because they mixed structure with presentation. To be backwards compatible with HTML-documents created in this period, today’s browsers interpret markup in two different modes depending on the document type defined in the HTML: “Standards mode” is the regular parsing mode for modern browsers, where they demand adherence to the HTML standard defined in the document type at the top of the document, such as HTML 4.01 or HTML 5. If no valid document type is defined in a document a browser reverts to “Quirks mode” which is more lenient towards deprecated markup as well as attempting to fix markup that’s flawed in some way – e.g. opening tags that don’t have corresponding closing tags.

Different browsers interpret markup slightly differently, and some are more standards-compliant than others. This leads to developers having to write markup that’s supported by all popular browsers, while using JS code to provide facilities that are not supported by less standards-compliant browsers, such code is know as a “polyfill”. Using polyfills is used heavily in modern web pages and apps to provide support HTML 5 features in browsers such as Internet Explorer (IE) 8 and older, as these don’t support much of the HTML 5 standard.

## Cascading Style Sheets (CSS)

CSS is responsible for describing the presentation semantics in documents written in markup languages. Most commonly it’s used for styling web pages written in either HTML or XHTML (a stricter version of HTML based on XML instead of SGML).

CSS enables the separation of document content from document presentation, which can improve accessibility while providing more flexibility in the presentational characteristics of a web page. It also provides the ability to have multiple pages share formatting through the use of the same style sheet – enabling an easy way to create a uniform presentational profile for web pages.

The syntax of CSS is based on simple English words to define rule sets for various elements in a document described by *CSS selectors*. Selectors can reference either HTML elements directly, or certain element attributes by prefixing attribute-specific characters such as “id”, prefixed by a pound sign (#), and “class”, prefixed by a period (.). Each rule set consists of one or more *properties* that are arranged in name-value pairs, separated by semicolons. Rule sets can also have pseudo-classes appended to them that apply to information from outside the DOM hierarchy of the document, such as *:hover*, which applies to elements that a user hovers the mouse pointer over.

selector [, selector2, ...] [:pseudo-class] {

property: value;

[property2: value2;

...]

}

/\* comment \*/

Code Snippet : A CSS rule set with two selectors, a pseudo-class, two properties and a comment.

These style sheets are called “cascading” because of how they handle situations where multiple rule sets apply to a single element. CSS here specifies a priority scheme to determine which of the overlapping properties should take precedence. This *cascade* goes from most to least specific. So if for instance you have a rule set for anchors, “a”, and a rule set for anchors of the class “emphasized”, “a.emphasized”, the properties in “a.emphasized” will be prioritized over the same properties in “a” for anchor elements with the “emphasized” class attribute.

This way of cascading is also used when prioritizing which style sheets should take precedence. CSS gives authors three ways of defining style sheet sources in a document: inline, embedded and external. The priority scheme is as follows:

1. **Inline**, inside the HTML document, specified for a single element inside its “style” attribute.
2. **Embedded**, inside the HTML document, defined in its own block inside a “style” element.
3. **External**, a separate CSS file referenced in the document.

CSS also allows for using different styles depending on the media type from version 2 onwards. E.g. this can allow for differentiating between regular screen versions and printed versions of a document – giving authors the ability to tailor a document’s presentation to different media.

### Media Queries

Media Queries are defined in the CSS 3 standard as an extension of the old media type detection from CSS 2. While the old standard allowed for detecting media types such as *screen* and *print*, media queries give authors the ability to differentiate between media features as well. Features that can be detected with the new standard include *width, height* and *orientation*. Being able to combine these media types and media features into logical expressions is what constitutes a Media Query. Because of this the new standard contains logical operators to give authors the ability to create these expressions. Operators such as AND, NOT and ONLY can be used to define a Media Query.

**@media screen and (min-width: 400px) and (max-width: 700px) { … }**

By using Media Queries authors can specify different presentational properties based not only on what type of medium, but the features of any given medium that’s viewing the document. While the screen type covers everything from a mobile phone to a widescreen TV, specifying the dimensions of screen media, such as max-width, for a given rule set means being able to tailor the presentation more accurately to a specific device.

Code Snippet : An example of a Media Query for a screen media type with a screen width between 400px and 700px

### Fluid Grids

A fluid grid is the term given to web page layouts that are built using relativistic dimensions instead of absolute ones. The crux of this is that elements are given dimensions in the CSS based on percentages and text is sized based on “ems”, instead of pixels. This means that layouts can flow freely outward and inward without breaking on different sized screens than what the author intended to begin with, hence the name “fluid grid”. This method of styling layouts is inherently more flexible than the old method of using pixels to define the dimensions of HTML elements. Even though a layout built as a fluid grid will keep its shape based on screen size, it will still break when the difference between the intended screen dimensions and the viewing screen size is big, or the aspect ratio is significantly altered – such as a web page designed for desktops is viewed on a vertical mobile phone screen. Fluid grids are thus useful for adapting to small changes in the viewport size, but not for significantly different devices.

## JavaScript (JS)

JavaScript is an interpreted, dynamic, weakly typed scripting language that’s commonly implemented in web browsers to support the creation of dynamic web pages. It’s a multi-paradigm language that supports object-oriented, imperative and functional programming styles. JavaScript’s use is primarily client-side, but has gained popularity in server-side applications in later years.

Brendan Eich created JavaScript over the course of 10 days in May 1995 while working at Netscape. The purpose was to create a lightweight interpreted language to appeal to nonprofessional programmers, like Microsoft’s Visual Basic. While it was first known as Mocha and later LiveScript, its name changed to JavaScript to coincide with the addition of support for the Java programming language in the Netscape browser. While JavaScript borrows many names and naming conventions from Java, they are otherwise unrelated and have widely different semantics.

Code Snippet : A small JavaScript example demonstrating anonymous (lambda) functions and closure

var displayClosure = function() {

var count = 0;

return function () { return ++count; };

}

var increment = displayClosure();

increment(); // returns 1

increment(); // returns 2

increment(); // returns 3

Including JavaScript on a web page can be done in two ways:

* **Embedded**, the code is included in the HTML document in a “script” element.
* **External**, the code is in its own JS file and is referenced in the HTML document.

JavaScript was standardized as ECMAScript in 1997, which is still being developed today. Even though the central part of JavaScript is based on the ECMAScript standard now, it supports additional features that aren’t described in the ECMA specification. Many of these additional features are later incorporated into the standard.

JavaScript’s popularity has only increased throughout the years, and it is now commonplace on nearly all web pages. This has been especially apparent after the emergence of AJAX (Asynchronous JavaScript and XML), which kicked off “Web 2.0” where dynamic web pages became the new “thing” to have for both commercial and non-commercial sites alike.

### Polyfills and Shims

On top of being used for creating dynamic, asynchronous web pages, JavaScript is also used to amend compatibility issues in web browsers. As official web standards evolve over time, so do the web browsers who adhere to these standards. This constant evolution causes older browsers to have compatibility issues with new features in specifications created after their release. In a perfect world every user would always use the latest release of their chosen browsers, but for many reasons this is not always a possibility.

Because of this, authors who use features in newly ratified standards will experience that their web pages don’t always appear as intended, or are broken for users with older browsers. To fix these problems, web developers have looked into the concept of shims. A shim in computing is usually a library between an API and the requestor of said API. A shim intercepts calls to the API and changes either the parameters passed, changes the operation itself or redirects the request elsewhere. This concept is useful when the behavior of an API changes, e.g. when an author starts using features from a new specification and an old web browser doesn’t support it. In modern web development, shims are essential to maintain compatibility with older browsers. JavaScript is used in these cases to check for and intercept requests from older browsers. The features are then either emulated by manipulating the DOM with code and other assets, or the web page is gracefully degraded to not look broken, even though the browser lacks support for the desired features.

Sometimes the concept of shims is expanded to support features that don’t even exist in modern browsers. Certain experimental features that are still being discussed by the W3C or don’t even exist in the HTML working standard can be added to web pages by the use of shims. These intercepting scripts are commonly known as polyfills. While shims have more generalized usage for interception of API’s, polyfills are specifically targeted at compatibility problems in both old and new web browsers. The main difference between polyfills and shims is that while a shim might have its own API that changes the original in some way, a polyfill works silently with no API of its own. The most common use of polyfills today is to silently fill in support for certain HTML 5 features that old browsers don’t support, giving the author the ability to use these features without writing special control structures in his code to account for supported and unsupported browsers.

## Responsive Web Design (RWD)

Making web pages respond to changes in its environment, such as screen size and orientation, has become increasingly important now that mobile devices are taking over as people’s main way to browse the internet. Previously, building a separate site for mobile devices and redirecting to it as the server sees fit has done this. These solutions give the developers two code bases to maintain instead of just one. They are also static in that they are built for desktops and, most commonly, phones. So which one should be served to a tablet, or a TV? Should developers be tasked to create a new, separate version of a web site every time a new device with a different form factor comes along? This would be extremely cumbersome and would increase the complexity of any web development project to an unbearable degree.

RWD is a method that suggests keeping only one code base that has its layout changed as the environment changes. I.e. having the web page’s CSS alter how the page looks based on certain device class definitions, as well as switching from static to fluid layouts that are relative to screen size, instead of fixed width. This will allow the web page to adjust to minor variations in screen size, and not just massive changes such as going from a widescreen desktop monitor to an Android phone.

RWD in itself is largely based on fluid grids, flexible media and media queries . The idea is to use these media queries to alter the CSS used for elements on a web page based on the device’s screen size, and let the fluid grid fill in the smaller gaps between devices of the same class. Often the designer predetermines at which resolutions the layout of the page will use different CSS defined within the media queries. These changes in layout are done on the client side, since it’s all determined in the web page’s CSS. This can lead to performance issues, especially when dealing with larger web pages, scripts and other media.

While the layout is changed and elements may be hidden from users on mobile devices by using media queries, all the content defined in the page’s markup is still downloaded by the browser. Large images and scripts that may never be visible to the user will make a page load slower and spend more bandwidth than necessary. Considering how bandwidth is still at a premium in the mobile context, it’s unfortunate that this widely adopted development method can be detrimental to the user experience. On top of this: media queries aren’t always supported by browsers, both mobile and otherwise, especially older ones. This is a problem considering that unsupported browsers will end up having trouble parsing the CSS of a page using media queries. On certain browsers, such as Internet Explorer 7 and 8, this can be fixed by using polyfills written in JavaScript, but this is not possible in older mobile browsers that don’t have sufficient JavaScript support.

## Mobile First

Many of these issues arise from the fact that developers still use the desktop version of the site as the baseline for the development process. This is understandable considering how the desktop computer has been the main way of accessing the web for more than twenty years. Making the desktop experience gracefully degrade for mobile devices has been the way to go. As mentioned, this leads to the web pages “inheriting” elements from the desktop version that might not even be visible because of media queries, but they’re still downloaded.

Luke Wroblewski’s book “Mobile First” suggests that attacking the problem from the opposite direction might be the way to go. Developing the site for mobile devices first will lead to the desktop benefitting through a more focused design centered on relevant content, he claims. The Mobile First mindset allows developers to embrace the constraints of mobile devices to create services that only deliver what the customer actually wants, cutting out everything that’s not totally necessary. Focusing on the constraints means looking more closely at the performance of a given page to make it function properly even in conditions of low bandwidth and weak computing power. It also means looking at the capabilities of mobile devices and using the platforms to their full potential, using what might not be thought about if the basis of development is a desktop computer. Web-pages or -apps can be made richer by adding context awareness through a mobile device’s sensors and inbuilt services such as GPS, compass, gyroscope, as well as telephone and camera.

Making the desktop site from a mobile baseline means progressively enhancing it as the capabilities and screen size of the device improves. This means doing the polar opposite of “graceful degradation”. This way of thinking will probably sort a lot of performance issues itself simply because the designs and the use of scripts and media will be more limited due to lack of screen space on a mobile device. This approach also helps with regards to making images and other media more bandwidth-friendly. It will make browsers download mobile-optimized media first – only downloading HD desktop-optimized media if the device can handle it or the user wants it.

## RESS: Responsive Design + Server Side Components

RESS is a concept suggested by Luke Wroblewski. It combines the flexibility of RWD with the capability of having the server decide what markup to serve the client. The idea is to have a single template that defines the layout of the page, but have selected components adapted with different implementations on the server. This way we get the flexibility of RWD with the performance issues fixed by tailoring the markup on the server, leaving the client to adapt the layout without having to download unnecessary media and scripts. The server can optimize things like source order, URL structure, media and application design before sending any content to the browser, but without relying on user agent (UA) redirects to entirely device class specific code templates. The problem with these standalone full-page code templates is that they’re almost guaranteed to contain duplicate code that also exists in the implementations for other device classes. Being able to reduce or completely eliminate duplicate code will make life easier for developers as well as increasing the maintainability of the system.

Using this method we can have a server side solution without relying on redirects. The same URL can be kept for each implementation since the server generates the whole adapted page. Fluid grids and media queries will still handle the layout, but the components themselves will be optimized according to the capabilities and features of the device sending the request. Known problems with RWD such as flexible images can be easily fixed using this method as image components on a page can have their markup changed to reference mobile-friendly images if the server detects such a device.

There’s one problem with having detection done on the server, though: it relies on being able to accurately detect what kind of UA is making the request. The reliability of server side user agent detection has been debated for many years, and several excellent solutions exist for accurately interpreting user-agent strings on the server. But as mentioned earlier this solution falls flat as soon as someone fakes the user-agent string or otherwise blocks the server from making what amounts to an educated guess. Being reliant on the user-agent string alone means that UA’s that don’t send a meaningful UA string will be subject to receiving a default “fallback” page for unknown UA’s.

# Related work

## RESS with Detector

Detector is a flexible server side UA feature detection system created by Dave Olsen and written about in his own blog (Olsen, 2012). He’s a programmer and project manager at West Virginia University. The idea behind it is to never assume you know everything based on just a UA string, and that you should never run feature tests unless you don’t know which UA you’re running in. Based on this he skips the predefined database altogether and makes the system learn the features each new UA it encounters through a client side test suite.

The system is split into two parts: the client side test suite, based on Modernizr, and the server side detection logic that stores and looks up UA strings and decides what to do with them. The client side part is simply a Modernizr test page that’s sent to the client if the UA string has not been encountered, or if certain tests have been set to run on a per-session or per-request basis. The results from these tests are saved in a cookie and stored on the server upon a page reload. The server side logic is responsible for storing the information gathered by the client side tests as well as gathering useful information from the UA string itself, such as operating system and device name. This method of user agent detection (or more accurately: *feature detection*) not only removes the need for a central database, it makes the system *future friendly* by allowing it to figure out an unknown device’s features from the device itself, instead of a database that’s solely relying on the UA string for information. Of course, this means that a new device that’s not been encountered by the system will be subject to a barrage of tests that will slow down the loading time of a page and take up valuable resources. But it will only happen once: the very first time this new UA string is detected. Each subsequent request from the same user agent will get served the appropriate markup straight from the server, without having to run the tests on the client.

Deciding which markup to serve a given UA is done by the implementation of Detector on a given server. Detector gives the ability to define device *families* that decide what features uniquely identifies a type of device you want to tailor the page to. Families are defined in a JSON file that’s loaded by Detector and used to look up which family is best suited to the requesting UA. Attributes for a family can take two forms: Boolean attributes gotten from the ua-parser.php and a *features array* that can contain any features detected by the Modernizr client side tests. All attributes and features in the feature array have to be evaluated to *true* for a family to be selected. The JSON file containing the family definitions are read sequentially, and as such have the most general case defined last and most specific case first to find the best fitting family for any given UA. These device families are central to creating a RESS solution by using Detector, as they’re what we use to differentiate between different device classes.

To build a RESS solution by using Detector there’s one extra thing needed: a template framework, such as Mustache, Handlebars or Jade. Without a template system RESS is not possible, as you would have to serve an entire page defined by the device class, instead of tailoring specific components while keeping the base markup common between all classes. Which template system is used is not as important as arranging the templates into a well structures file-hierarchy. The reason for this is to allow for easy switching between template-partials after having Detector decide which family a UA belongs to. Following Luke Wroblewski’s example of having a page with different navigation for different devices, we could do the following using Mustache for templates with the given file-hierarchy for partials to the right:

The markup has the header and footer set as Mustache partials. This means that whenever the page is loaded the partials will be filled by whatever markup is in the templates in the folder of the family chosen by Detector. In the base case we have preset definitions of both the header and footer sections. For the mobile family of devices we have overwritten both the header and footer – perhaps to move a lot of navigation from the header to the footer for more comfortable one-handed use. In the case of the tablet the header will be overwritten while the footer is left untouched. In the case of both desktop and TV no changes are made as the templates in the base folder are meant for use with larger screens. This way of organizing the templates lets the system look in the folder of the chosen device family first, while filling in the remaining partials with what’s in the base folder – removing a lot of code that would otherwise be repeated throughout each family-specific markup. Falling back on a base directory is not supported in Mustache by default, but Dave Olsen has made the small tweaks necessary for this functionality in his own fork of Mustache’s loader. While this kind of structure is invisible to the user, it makes for a lot more readable code, and keeps the code base maintainable even if it grows in size by virtue of its modular design. Changing the order and layout of different components is just a matter of changing the base html file, leaving the device-specific code untouched. It also allows for adding family-specific components later by simply adding new partials in a specific folder. E.g. if the base navigation in the header doesn’t utilize the screen space of a 50” TV well enough; it’s just a matter of adding a *header.mustache* file to the TV folder and it will replace the base markup with TV-tailored navigation whenever Detector gets a request from a TV-family UA.

Not only presentational elements can be turned into components using the RESS with Detector. Includes such as scripts and CSS markup can be put into partials that are specific to each device family. Doing this will ensure that devices only ever download files and markup that’s useful and visible to the user; skipping all the fluff that’s present in most RWD solutions and solving the problem with using “display:none” as the only way of hiding elements from users on unsupported devices. This way the system supports both *progressive enhancement* and *graceful degradation* of a web page without much hassle, as the system will detect which partials – and thus scripts, CSS, and media – to download for any given device.

Since the family system is completely independent of the base markup of a page, it opens for increased adaptability from the developer’s standpoint. If a certain mobile device falls outside of the scope of the normal “mobile” family definition, a new family can be created to cover this. While modern “smart phone” browsers today support almost as much JavaScript and CSS as a desktop computer, many older phones do not, and many of these are still prevalent in many parts of the world. These “feature phones” have browsers, but they often lack much CSS support and rarely support JavaScript to any useful degree. Assuming these phones fall outside of the “mobile” definition used above, the server will fall back to the default components, which are designed with desktop in mind. This is definitely not desirable on these types of devices. Creating a family to cover them is needed, and easily done by e.g. creating a family called “mobile-basic” that covers the features present on such devices. This requires nothing more than adding this new definition to the family JSON file and tailoring the necessary partials for these types of devices, the rest of the code base doesn’t need to be touched.

Detector offers a fairly straightforward solution to the problem of making the server more involved in RWD. With a simple tweak to make the template system able to *failover* to a base directory in the case of missing partials is all that’s needed to create a RESS system. It’s also flexible with the family system allowing for any kind of template system to be used, which gives developers increased freedom in building their web pages. The family system also separates the logic of detecting which partials to use from the presentational markup in the html and template files, keeping the code base clean and maintainable while being *future friendly*. While it doesn’t solve the problem of dumb content (which should fall to the CMS) and context aware coding (which should be done by the editor), it goes a long way to create a possible solution for the server-side of things. Detector is currently in production on the West Virginia University home page.

# Summary

In this chapter we’ve looked at the technologies, languages and concepts underlying this thesis: HTML, CSS, JavaScript, Responsive Web Design, Mobile First and RESS. We’ve also looked into related work in the same field, focusing on the Detector project by Dave Olsen.