The Report Committee for Preston Brent Landers
certifies that this is the approved version of the following report:

Speakur: leveraging Web Components for composable applications

APPROVED BY

SUPERVISING COMMITTEE:

Adnan Aziz, Supervisor

Christine Julien

Speakur: leveraging Web Components
for composable applications

by

Preston Brent Landers, B.A.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2015

Dedicated to my wife Andrea and to my parents.

More...

Acknowledgments

I wish to thank the multitudes of people who helped me. Time would fail me to tell of the multitudes of individuals ...

Speakur: leveraging Web Components for composable applications

Preston Brent Landers, M.S.E.
The University of Texas at Austin, 2015

Supervisor: Adnan Aziz

This report is a case study of applying abstraction, encapsulation, and composition techniques to web application architecture with the use of Web Components, a proposed extension to the W3C HTML5 document standard. The author presents Speakur, a real-time social discussion plugin for the mobile and desktop web, as an example of using HTML5 Web Components to realize software engineering principles such as encapsulation and interface-based abstraction, and the composition of applications from components sourced from diverse authors and frameworks.

Web authors can add a Speakur discussion to their page by inserting a simple HTML element at the desired spot to give the page a real-time discussion or feedback system. Speakur uses the Polymer framework's implementation of the draft Web Components (WC) standard to achieve encapsulation of its internal implementation details from the containing page and present a simplified, well defined interface (API).

Web Components are a proposed World Wide Web Consortium (W3C) standard for writing custom HTML tags that take advantage of new browser technologies like Shadow DOM, package importing, CSS Flexboxes and data-bound templates. This report reviews Web Component technologies and provides a case study for structuring a real-world WC applet that is embedded in a larger app or system. The major research question is whether W3C Web Components provide a viable path towards the encapsulation and composition principles that have largely eluded web engineers thus far. In other words, *are components really the future of the web*? Subsidiary topics include assessing the maturity and suitability of current Web Components technologies for widespread deployment and the efficient synchronization of component state across distributed networks.

Table of Contents

List of Tables

List of Figures

List of Source Listings

Chapter 1

Introduction

This report is a case study of using W3C Web Components, a proposed HTML5 extension, to implement techniques of encapsulation, abstraction, modularity in web application engineering. The author designed Speakur, a real-time discussion social plugin for the web, as an experiment to determine the viability and maturity of using Web Components to create modern, highly composable web applications. Figure 1.1 shows an example of a Speakur discussion.

Like most web applications Speakur is written in a combination of HTML markup and the JavaScript programming language. HTML is a declarative markup language used to create documents — web pages — which are viewed with the help of a program called the browser. The Hypertext Markup Language (HTML) standard has proven wildly successful since its introduction in 1993 by British computer scientist Tim Berners-Lee, with billions and billions of pages served, and hundreds of millions of public and private web sites forming a major part of our information landscape [39]. More than any other invention, other than perhaps email, the World Wide Web (WWW) has shaped how we see and use the global network.

Those designing and programming applications for the Web as a comput-

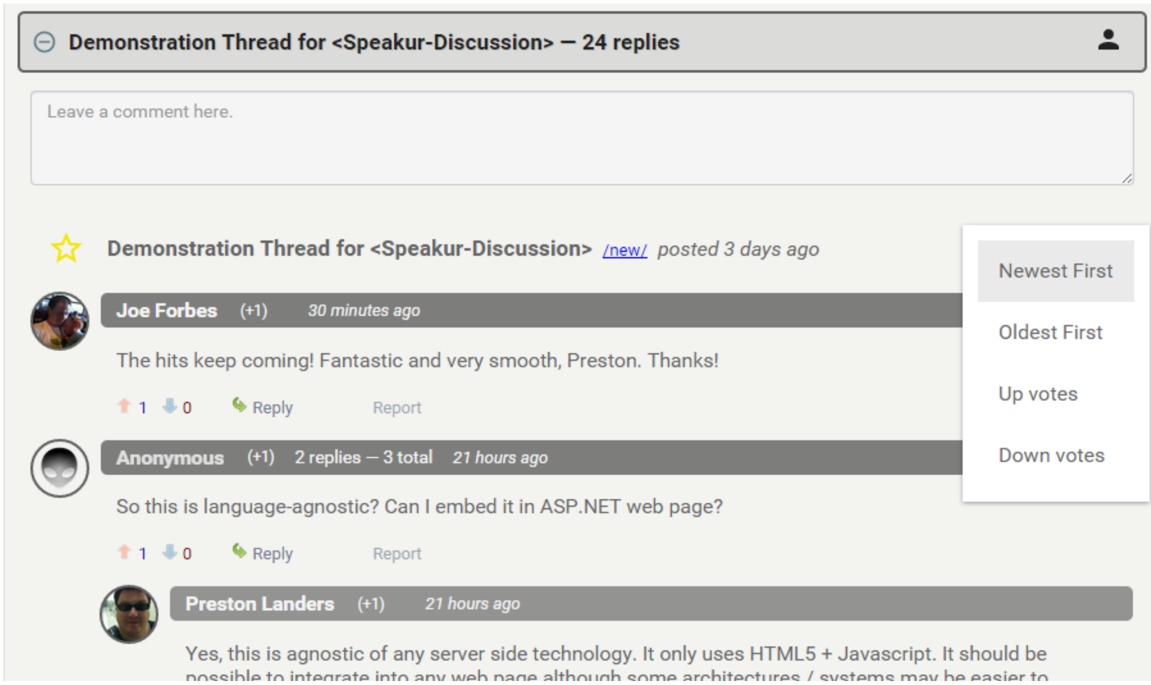

Figure 1.1: A Speakur thread inside a demonstration page.

ing platform have long dreamed of the ability to mix and match independent, reusable chunks of functionality — components — in their documents without mutual coupling and interference. The Document Object Model (DOM) browser abstraction does not allow for significant decoupling; everything lives together on one big page. Hacks like the `<iframe>` tag let you work around some of these restrictions, usually in a limited and inelegant way.

At the time of HTML's introduction the concept of quickly and easily composing a static web page, much less a full-fledged dynamic application, out of Lego-like reusable building blocks seemed like a distant dream at best. The introduction of the JavaScript[1] (JS) programming language to web browsers in 1995 allowed for a completely new dimension of dynamic behavior that was not possible before [15]. Eventually web apps like Gmail and Google Docs, powered by JavaScript, rivaled traditional desktop applications in functionality and usability while being instantly accessible from nearly anywhere. Still, web apps had to be stitched together 'by hand' in ways that carefully ensured the different parts didn't step on each other's toes, else disaster frequently ensued. Each component or area of the system could not help but be coupled to the others at some level as a result of the programming model imposed by the DOM and HTML [37].

Over the years the dynamic behaviors afforded by JavaScript grew in importance along with the web, and helped contribute to its success. JavaScript is now a ubiquitous programming language and has expanded far beyond the desktop web browser. JavaScript interpreters can be found in mobile phones, industrial controllers, and now the embedded devices that comprise the Internet of Things (IoT) [32]. Its flexible, loosely typed nature can be a boon to the prototyping and initial development process, but the difficulties of building a large application in the DOM soon became apparent.

Over time, a bewildering array of frameworks and libraries sprang up around the HTML/JS ecosystem to help manage this complexity and to provide scaffolding and structure for the client side of web apps. For many years individual JS

[1]JavaScript, also rendered as Javascript or JS, has no significant relationship to Sun's (now Oracle's) popular Java programming language. JavaScript is formally standardized under the name ECMAScript (ES).

frameworks seemed to come and go as ephemerally as teenage pop idols [54]. Interoperability between these completing frameworks was virtually non-existent; typically, components from one framework couldn't easily be combined with those from an another. The industry kept searching for the Next Big Thing that would make writing high quality web apps less of bug-ridden, messy chore [54].

In recent years (roughly 2011 to 2014) Google's Angular [25] has emerged as a dominant client framework, due in part to its perceived high quality and the fact that it represents an common point for a fragmented industry to rally around [29]. Facebook's React library with its Virtual DOM is an up-and-comer focused on high performance that is more complementary in nature to Angular than a true challenger, focusing primarily on the "view" part of the common model-view-controller(MVC) design pattern [14].

Yet despite the emergence of the updated HTML5 standard in 2011 and the recent successes of web frameworks like Angular and React in capturing developer attention, a clear picture still did not exist of how web apps could achieve the encapsulated component model that had become prevalent in other areas of software engineering. That is, until engineers from Google[2] and Mozilla[3] and other organizations got together in 2012 under a W3C Web Applications Working Group [61] to draft a new standard called **Web Components** that will extend and enhance HTML5 in ways that could have a significant long-term impact [42]. For example,

[2]As of March 2015, Google's Chrome browser is the most popular desktop browser [63].

[3]The Mozilla Foundation is the sponsor of the popular Firefox web browser. It grew out of Netscape, whose Navigator browser helped bring the web to a mass audience.

as of the time of this writing, a rewrite of Angular called Angular 2.0 is slated to include Web Components as a core architectural element [30].

1.1 Web Components Overview

Fundamentally, the Web Components standard consists of four new core DOM technologies — extensions to the current HTML5 standard. If these standards are accepted by major browser vendors and the World Wide Web Consortium (W3C) which maintains HTML, they will eventually become native browser features and available directly to any web page without needing to use any additional JS frameworks or libraries. The core Web Component technologies are [53]:

- **Custom Elements**: extending HTML with author-created tags
- **Shadow DOM**: encapsulation for the internals of custom elements
- **Templates**: scaffolding for instantiating blocks of HTML from inert templates
- **Imports**: packaging for HTML components

This report also explores several related web standards initiatives that are frequently associated with Web Components but are not formally grouped under them, including mutation observers, model driven views, and the CSS Flexible Boxes and CSS Grid systems. In part because many of these technologies are not yet formally accepted as W3C standards and are not yet widely implemented in typical mobile and desktop browsers, Speakur has been implemented using

Google's experimental Polymer framework [13]. Polymer provides a JavaScript 'polyfill' library to implement many of the new Web Component features in browsers which would otherwise not support them. Eventually this platform polyfill should become unnecessary, in theory, as WC becomes widely adopted in browsers. Some browsers like Google Chrome have already implemented at least some native Web Component support and the polyfill is effectively a 'no-op' in these areas.

The potential componentization of the web is one of the most exciting developments in web engineering in years and follows the overall growth in software-as-a-service (SaaS) and the service oriented architecture model. The conversion of dynamic web logic—not mere snippets of plain HTML—into bundles of reusable, extendable, composable components enables web developers to move to a higher level of abstraction than was previously possible.

The move towards a component-based Web will enable interesting new composite services, mashups, and may help broaden the potential pool of web developers. What previously required a highly integrated, high-overhead development model or lots of tedious glue code can become as simple as importing a custom element and dropping it onto a page.

1.2 Structure of This Report

The goal of this report is to demonstrate the application of software engineering design patterns embodied in the W3C proposed Web Components standard such as encapsulating internal logic behind abstraction layers, modular composition, and the real-time synchronization of web application state. This report discusses many of the goals and principles of the Web Components initiative and how a number of different technologies taken together help raise the overall level of abstraction for content authors, web engineers, and application developers — which I will refer to collectively as (web) authors for short.

The Background section provides an introduction to some of the architectural problems inherent in modern web authoring and how Web Components (WC) address them. It also provides some background the on software engineering design patterns that are embodied in Web Components such as encapsulation and composition. It describes some of the motivations behind the development of Speakur and some of the specific software engineering questions it addresses, such as the ability to provide a hassle-free way to host an embedded discussion forum inside an arbitrary web resource in a way that is fully encapsulated.

The Approach section describes the high level software architecture choices behind Speakur and details the specific structures and techniques used when constructing a Web Component. It describes how Speakur uses WC to implement encapsulated modules whose internals are protected from unintentional outside influence. It also describes how the choice of the Firebase cloud database service and its WebSocket-based event notification system impacts Speakur's architecture.

The Implementation section shows how to apply Web Component principles to the task of creating a flexible and generic discussion forum for both desktop and mobile browsers. It describes the low level architecture, code flow, and synchronization process. An important topic in this section is security: how can we implement a largely client-based system while maintaining some kind of data integrity?

This is followed by an Analysis section which discusses some of the outcomes as compared to the original goals and also looks at the impact of the selection of Web Components, Polymer, Firebase and some of the other architectural choices. A few quantitative results are included, I hope **(TODO)**.

Finally, the Conclusion section is there and wraps up the report with various words **(TODO)**.

1.3 Source Code and Demonstration Resources

The source code for Speakur consists of HTML and JavaScript files located in a Git version control repository. These files constitute an *HTML Import* package that provides a **`<speakur-discussion>`** custom HTML element for the use of web authors in their own pages. An example of using `<speakur-discussion>` is given in Chapter 3 and in briefly in Listing 1.1 below.

```html
<!-- place this on your page
     where you want a discussion -->
<speakur-discussion
  href="http://example.com/news/web-components-in-action"
  allowAnonymous="false">
</speakur-discussion>
```

Listing 1.1: Example of the Speakur custom HTML element

The Speakur source code and component documentation can be found on the social coding site GitHub.com at [44]:

`https://github.com/Preston-Landers/speakur-discussion`

Demonstrations of several web pages which show off embedded Speakur discussions are available at the following location [47]:

`https://preston-landers.github.io/speakur-discussion/components/speakur-discussion/demo.html`

Chapter 2

Background

When the Web was first created by Tim Berners-Lee in 1989, web pages were largely envisioned as static *documents* with a single author or a small group of coordinating authors. The idea of composing a complex web application out of basic components like snapping together Lego blocks seemed like a distant dream at best. Until recently, web authors were limited to using the predefined HTML layout elements or 'tags' that were listed in the W3C standard and understood by browser programs, such as `<title>` and `<video>`. Creating your own *sui generis* HTML elements with unique behaviors seemed beyond the capabilities of the web browsers of the day like Mosaic and Netscape Navigator.

As of early 2015, modern web apps are typically written with a JavaScript framework that provides a cohesive set of structures, design patterns and practices designed to facilitate composing web applications — large or small — from a number of sub-components. Angular, Meteor, and Backbone are three such frameworks [29]. The difference between a 'framework' and a library is somewhat arbitrary, but typically frameworks are more comprehensive than narrowly focused utility libraries. Yet all frameworks must exist within the confines of the programming model provided by the browser and the Document Object Model (DOM). In this model, the entire web page or app belongs to a single 'document', constituent parts are not encapsulated or isolated from each other, and authors are limited to working with the predefined HTML tags. These issues make it difficult to create and share generic, reusable *web components* — in the abstract sense — among different users who may not use the same frameworks or follow the same set of assumptions and conventions.

2.1 Current challenges in web authoring

In object oriented programming, encapsulation is typically defined as a "language mechanism for restricting access to some of the object's components" [51, p. 522]. The point of encapsulation is providing an *abstraction* that consumers of the functionality can rely on without knowing the internals. The goals of encapsulation and abstraction include:

> Identifying the interface of a data structure ... providing *information hiding* by separating implementation decisions from parts of the program that use the data structure ... and allowing the data structure to be used in many different ways by many different components [51, p. 243].

Although techniques of abstraction and encapsulation have been widespread in object oriented programming for decades, the fundamental web client programming model has not allowed for significant encapsulation of things like the DOM structure and CSS style rules [37].

To illustrate how these problems affect the ability of authors to share and reuse code, let's look at an example from the popular Twitter Bootstrap library [7]. Twitter Bootstrap is a collection of Cascading Style Sheet (CSS) rules and JavaScript widgets or components designed to allow web authors to quickly 'bootstrap' an attractive, consistent look-and-feel onto a web page. Bootstrap provides pre-styled user interface (UI) widgets such as menus, buttons, panels, dropdown selectors, alerts, dialogs, and so on, to be used as building blocks to construct web sites or application user interfaces. Because Bootstrap must work within the confines of the DOM and the HTML5 standard, this necessarily exposes a great deal of Bootstrap's internals to its users. For example, to add a Bootstrap site navigation bar to your page, you must essentially copy and paste a large block of HTML and then customize it to your needs as shown in Figure 2.1.

This forces Bootstrap's users to tightly couple the layout of their page with the internal structure required by Bootstrap's navigation bar widget. This coupling hinders a significant refactoring of the navigation widget's internal structure (HTML layout) because that would require the large community of developers to update their applications accordingly. In addition, because CSS rules normally apply across the entire page, the authors of Bootstrap must carefully select the scope and nomenclature of all rules to ensure no unintended side-effects [62]. Even then, conflicts are inevitable when the entire page is treated as a single sandbox and you combine components from many different vendors.

What if instead one could create and share a reusable chunk of functionality — a web component – that hid all of these tedious structural details and encapsu-

```html
<nav class="navbar navbar-default" role="navigation">
  <!-- Brand and toggle get grouped for better mobile display -->
  <div class="navbar-header">
    <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a class="navbar-brand" href="#">Brand</a>
  </div>

  <!-- Collect the nav links, forms, and other content for toggling -->
  <div class="collapse navbar-collapse navbar-ex1-collapse">
    <ul class="nav navbar-nav">
      <li class="active"><a href="#">Link</a></li>
      <li><a href="#">Link</a></li>
      <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown">Dropdown <b class="caret"
        <ul class="dropdown-menu">
          <li><a href="#">Action</a></li>
          <li><a href="#">Another action</a></li>
          <li><a href="#">Something else here</a></li>
          <li><a href="#">Separated link</a></li>
          <li><a href="#">One more separated link</a></li>
        </ul>
```

Figure 2.1: Partial example of Twitter Bootstrap navigation bar HTML.

lated its private, internal state? What if web authors could create their *own* HTML elements? Using Bootstrap's navigation bar could be as easy as replacing the code in figure 2.1 with a custom element like the one in the following example:

```html
<twbs-navbar>
  <a href="#">Home</a>
  <a href="#">About</a>
  <a href="#">Sign In</a>
</twbs-navbar>
```

Listing 2.1: Hypothetical Bootstrap nav bar custom element.

2.1.1 Abstraction, encapsulation and composition

The Web Components working group, consisting of software engineers from several major browser vendors, looked at this situation and found that, in practice, browsers already had a suitable model for encapsulating components that hide complexity behind well-defined interfaces. That model was that one used internally by browsers to implement the newer HTML5 tags like the **`<video>`** element. The `<video>` element presents a simple interface (API) to HTML authors that hides the complexities of playing high definition video. Internally, however, browsers implement `<video>` with a 'shadow' or hidden document inside the object that contains the internal state [40]. For example, an author can write:

```
<video loop src=...> </video>
```

to cause the video to loop repeatedly.

This shadow Document Object Model (DOM) inside the `<video>` tag creates the user interface (UI) needed to control video playback such as the volume controls, the timeline bar, and the pause and play buttons. These inner playback controls are themselves built out of HTML, CSS and JS but these details are not exposed to web authors who simply place a `<video>` element on their page. Figure 2.2 illustrates how this works. It shows the shadow (internal) DOM of a `<video>` element on a page with the Play button `<div>` highlighted.

At the bottom of Figure 2.2 a small `#player-api` tag is visible in the DOM path. This is an example of the container inside `<video>` using the abstraction of a `#player-api` to encapsulate the details of actually controlling playback within the context of the overall `<video>` interface.

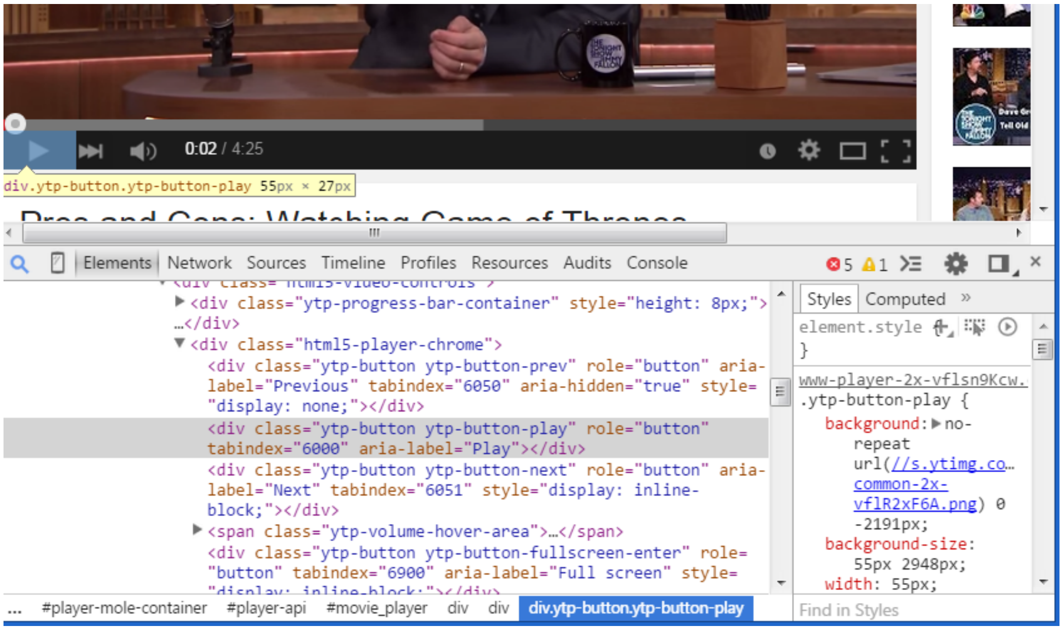

Figure 2.2: Opera's shadow DOM for `<video>` highlighting the Play button

This example illustrates three component design principles that are widely followed in other areas of software engineering [33]:

- Create layers of **abstraction** to represent 'public' details that are relevant to the surrounding code or environment.
- Use **encapsulation** and well defined interfaces to protect private state, hide implementation complexity, and leave implementors free to refactor internals.
- Prefer **composition** or *has-a* relationships over inheritance or *is-a* relationships when building modules to reduce coupling and simplify interface refactoring.

Composition helps reduce coupling or structural between modules by forcing them to interact using only public interfaces. In the case of the interface for `<video>`, it's composed of simple block elements and scoped CSS roles and the Volume and Play controls aren't particularly special objects, just `<divs>` with CSS rules and click handlers.

The solution, therefore, to these coupling problems in web authoring is to expose these internal brower APIs for creating elements in a safe and portable fashion. This will allow web authors to create their own rich custom elements using standard portable APIs, encapsulate their internals, and enable easier sharing, composition and integration. The question remains, which specific browser internals must be exposed and standardized in order to support Web Components?

2.2 Web Components

Web Components consists of two main technologies and two supporting features. Custom HTML Elements and Shadow DOM are the two key players while HTML Imports and Templates support these features. One of the central goals of the Web Components initiative is to maintain interoperability across different browsers and frameworks, so that modules which adhere to the Web Components standard can provide a consistent experience no matter what framework the developer chooses or which browser the user selects. For example, an important benefit of Custom Elements is that they *are* standard HTML elements; they live in the DOM and can be accessed by the usual DOM methods and the majority of standard HTML/JS development tools [53]. They also behave somewhat like objects in traditional object oriented programming (OOP) in that they have *methods* and hidden internal *state* (data).

2.2.1 Custom HTML elements

Never before have web authors been able to define their own custom HTML elements that were not found in the official list. Actually, many authors and web frameworks have been doing exactly that for years, primarily for internal purposes where the custom elements are preprocessed and compiled down to standard HTML. The custom elements would not get sent to the end user's browser because it would not know what to do with them. In addition, the DOM has long supported creating custom-named elements, but it was not possible to do much interesting with them because they were treated like an ordinary `<span>` element [17]. However, the possibility now exists to create custom elements in a standard way that will work consistently across browsers with the W3C Custom Element specification [17].

The primary restriction is that all custom elements must have a `-` character (dash) in their name, such as `<my-element>`. This is to avoid a name collision with future built-in HTML elements. To create a new Custom Element, you must first register the element:

```
var MyElement = document.registerElement('my-element');
```

Then you place your new element on the page, either declaratively in HTML:

```
<my-element> hello, world! </my-element>
```

or imperatively with JavaScript:

```
var MyElement = document.registerElement('my-element');
```

```
// instantiate a new instance of the element
var thisOne = new MyElement();
document.body.appendChild(thisOne); // add to the <body>
```

With a quick example like this the result does not look all that different from a `<span>`. To do something more interesting with your custom element you will need to the other features of Web Components: Shadow DOM, templates and imports.

2.2.2 Shadow DOM

Shadow DOM encapsulates the internal structure of an element [21]. As we have seen, browsers already use Shadow DOM to encapsulate the private state of standard elements like `<video>` but now this capability is extended to custom-defined elements.

You can think of shadow DOM like an HTML fragment inside an element that describes its external appearance without exposing these structural details[1]. Typically a custom element definition has a template (more on these in a moment) which produces the shadow DOM necessary to render the element. The actual contents of the shadow DOM are just ordinary elements. Any element (whether custom or not) can have zero, one, or more shadow DOM trees attached.

Custom elements can wrap regular text, normal HTML elements, other custom elements, or nothing at all, and then project that content through its shadow

[1]HTML5 Shadow DOM should not be confused with the React framework's **Virtual** DOM, which is conceptually closer to HTML5 Templates in nature than Shadow DOM.

DOM, therefore its visual representation. In the example in Listing 2.1 above, a **`<twbs-navbar>`** element consumes a set of three **`<a>`** (anchor or link) elements but internally transforms that to something like the example in figure 2.1, projecting the set of links into the nav menu structure with appropriate wrappers.

The `<content>` tag is used inside a custom element's template to indicate the spot where the consumed (wrapped) content should be *projected*. This wrapped content is known as *light DOM*, because it's given by the user and projected through into the shadow. Together the shadow DOM and light DOM form the *logical DOM* of a custom element. It is also possible for elements to have multiple shadow DOM sub-trees. This is used particularly for emulating object-oriented-like inheritance relationships between custom elements.

In languages like C# and Java, the encapsulation of classes and the protection of private object fields are a relatively strong guarantee by the language. JavaScript variables can be protected by *closures* but shadow DOM and CSS are not completely and utterly isolated from the containing page. It is possible to "reach inside" and break encapsulation to at least some degree, but the point is that this must be an intentional act by the developer and not an unexpected side-effect [3].

2.2.3 HTML Imports

One significant problem faced by web developers is the lack of any built-in packaging system for modules in HTML. Prior to Web Components there was no way to import a snippet of HTML or JavaScript from an external location and insert it exactly one time into the current document, similar to an `#include` directive in the C language or the packaging and import systems popular in scripting languages like Python, Go and Ruby. JavaScript could always be loaded with a `<script>` tag like usual, but this did not ensure that resources were loaded and executed exactly once, a process known as *de-duping*. A component that uses a certain JS resource might be needed in two different spots on the page, but that resource would be requested from the server twice and executed twice, degrading application performance.

In order to fix these problems the HTML Imports standard allows for bringing in snippets of HTML, CSS or JavaScript into the current document in a way that ensures automatic de-duping of repeated requests [18]. The one major caveat is that de-duping only happens if the resources are named in exactly the same fashion in each case [2]. Dealing with HTML Imports in a consistent fashion will be discussed in the Implementation section.

2.2.4 Templates

The last major piece of the Web Component puzzle is the native HTML5 `<template>` tag. Unlike the rest of Web Components, `<template>` has already become a standard part of the HTML5 specification, although one that is perhaps not widely used yet outside of WC [22]. *Template* is a frequently overloaded word with different meanings in different programming environments. While HTML5 templates have some similarities to the concept of templates popularized by frameworks like Angular and Django, there are some important differences.

HTML5 templates are inert hunks of HTML embedded in the page that can be instantiated into 'real' elements by JavaScript. Their basic function is to give a source input for the shadow DOM when you create a new (custom) element instance and place it on the page.

However, templates are most useful in combination with 'live' data, not static, unchanging text. Binding data into templates with special operators[2], also known as a Model-Driven View, is **not** a part of the standard HTML5 template spec. The following example of a data bound template is something that does *not* work with plain Web Components alone:

```
<template>
  The temperature is {{ temp }} in {{ city }} right now.
</template>
```

Google engineers have advanced a proposal for standardizing Model-Driven Views but it is not yet part of the standard `<template>` element [27]. Instead this functionality can be handled by a JavaScript framework such as React or Polymer. Data-bound templates are discussed in more detail in the following chapter.

The primary benefit of HTML Templates from a performance perspective is that external resources referenced from the template (images, stylesheets, etc) will not be fetched until the template is actually instantiated. Templates are often used to declare the internal structure (shadow DOM) of custom elements. Therefore the resources needed to use the custom element aren't downloaded until they are actually needed, which is necessary when composing a large application out of numerous distinct elements.

[2] Sometimes called mustaches, handlebars or curly braces.

2.3 Related W3C initiatives

There are a number of related W3C initiatives for web standards. Sometimes these are loosely grouped under the label Web Components, and they do support componentization in web application design, but they are a separate part of the HTML5 standard. These include mutation observers, DOM selectors, and so-called 'responsive' attributes designed to adjust the layout automatically according to screen size. All of these techniques have been used extensively in Speakur.

2.3.1 Mutation Observers

For example, *mutation observers* allow a component to register a 'callback' function to observe and react to any changes in the state of an area of interest in the DOM, whether that change was generated by a user action or another component [19]. This helps decouple components by allowing them to react asynchronously to changes in each other's public state without directly interconnecting the components. The observers "deliver mutations in batches asynchronously at the end of a micro-task rather than immediately after they occur" [52]. This aids performance and user experience (UX) by ensuring that the callback is only called as-needed rather than once for each small change.

2.3.2 Selectors

Another standardization initiative is in the area of DOM / CSS *selectors*, as popularized by the highly successful jQuery library written by John Resig, which as of 2012 was used in half of all major websites [35]. The Selector API was officially standardized in 2013 and browser support is good in current versions [20].

The JS selector API allow you to quickly find DOM elements based on a rule-based description string (the selector) to perform further operations on their state such as reading or modifying data, observing future changes, attaching animations, and so on. The following example shows how to populate a JS variable with the element that has the `id` attribute equal to `some-id`:

```
// find an element whose id is equal to 'some-id'
var someElem = document.querySelector("#some-id");
```

2.3.3 Responsive Layout for Mobile

The rise of smartphones and other mobile devices has accelerated the need for web developers to make their applications responsive to the type of client used to access it. Creating a web site or application that adjusts to the smaller screen of a mobile phone is often known as *responsive design*. Before the addition of responsive design features to the HTML spec, it was always possible to make manual layout adjustments in JavaScript but this was sometimes brittle and error-prone.

Besides JavaScript, there are three main CSS techniques that can be used to make a responsive design: the `@media` rule, the Grid layout, and Flexible (Flex) boxes. The CSS `@media` rule allows one to restrict the scope of CSS rules such that they only apply to certain media types including different screen sizes.

The CSS Grid layout is primarily intended to control the overall page layout on different device sizes [16]. For example, a 'sidebar' might normally run alongside the main content down the side of the page in a desktop layout, but in a mobile layout the sidebar might come down at the end after the main content. This includes different layouts for portrait vs. landscape device orientation. The CSS Flexible Boxes model (Flex) is intended for adjusting the flow of smaller page components [11]. Flex boxes are useful for general user interface (UI) structure, not just device-responsive design, and are widely used in Polymer and in Speakur.

2.4 Polymer Framework

The exciting possibilities offered by Web Components seemed to call for a new framework engineered from the ground-up to take advantage of it. In light of that, a team within Google developed the Polymer framework both to embody the Web Components architecture [13] and to provide a suite of components and user interface widgets useful for building applications. Because Web Components are a bleeding-edge feature not yet natively implemented in most browsers, the Polymer developers opted to create a *polyfill*, library to fill in these missing features to the degree possible. This polyfill is used in several other Web Components related projects such as X-Tags [24]. Unfortunately polyfills cannot provide a 100% complete Web Components implementation; only native browser support can.

Because Polymer and to a lesser extent Web Components are still experimental and under development, they are both subject to frequent changes. The version of Speakur described in this report is based on Polymer release 0.5. As of the time of this writing, Polymer 0.8 is planned and will contain significant breaking changes to application structure [5]. Portions of this report pertaining to specific Polymer practices or interfaces are likely to be out of date by the time you read this.

2.5 Speakur

My desire to learn more about modern web development led me to investigate web frameworks like Angular and Meteor. I spent a few weeks building elementary demos with these two frameworks in particular. Although they are expressive and powerful, and are used every day to power high-traffic applications, I was unhappy with the non-standard and idiosyncratic nature of these frameworks. They relied on 'proprietary' (even if open source) extensions that were not native to HTML and not easily transportable across different frameworks and architectures. This dissatisfaction led me to learn about the Web Components initiative.

2.5.1 Origin

Learning about Web Components quickly led me to the Polymer project. I wanted a component that demonstrated common use cases for Web Components and also showed off some of the design possibilities provided by Polymer and Material Design [38]. I was also intrigued by the possibilities of a server-free design afforded by Firebase. Some kind of 'live' social plugin seemed like a natural fit for the capabilities of Polymer and Firebase, so this led to a discussion plugin for blogs and other articles. My hope was that it would required little or nothing in the way of dedicated server resources in order to actually use it.

2.5.2 Motivations

I wanted my discussion component to have some of the following attributes:

1. A simple API that abstracted away most implementation details.

2. Require minimal server resources. Ideally nothing would need to be "installed" and it could be loaded in a cross-origin fashion from online developer tools like `https://jsbin.com`.

3. Support distributed event notification similar to the publish-subscribe (pubsub) design pattern.

   In essence, 'live' data updates: when someone replies to a post it should become instantly available to anyone viewing that thread.

4. Support Markdown formatted comments including syntax highlighting for code snippets.

5. Support internationalization (i18n) and localization (l10n) features for a global audience.

6. If any framework was used at all, it should be based on Web Components.

   This instantly ruled out the vast majority of frameworks, leaving only Polymer and the less-comprehensive X-Tags project [24] and a few other less ambitious contenders.

In the next chapter we will discuss some of the high level architectural concerns that should be addressed when designing such a component.

Chapter 3

Approach

This chapter discusses the architectural approach behind Speakur, a social discussion plugin for the desktop and mobile web based on Web Components and the Polymer framework. Web authors can use Speakur to easily add a comment section to their articles or blog posts. Visitors can leave feedback about the article and engage in discussion with each other. Discussions are grouped into topics or 'threads', and within these threads, users can reply to the main article or to each other.

3.1 Functionality

Speakur is a Custom Element (`<speakur-discussion>`) that provides a drop-in discussion forum or comment hosting service for a blog, web page or other web application. Examples of Speakur's user interface can be found in Figures 1.1 and 3.1. Placing Speakur inside a web resource is very straightforward. As shown in Listing 1.1, and in section 4.7 below, you simply place the `<speakur-discussion>` element in your page's HTML at the desired spot. This requires two supporting steps detailed in section 4.7:

1. loading the Web Components polyfill library script

2. importing the `<speakur-discussion>` element.

Once an author imports the element and places it on their page, that site now has an integrated discussion forum that works equally well for desktop and mobile users. All forum data including user profiles and comment text is stored in an online cloud database called Firebase [9]. The messy details of structuring a discussion forum are abstracted away from the web page author.

`<speakur-discussion>` presents a simplified interface (API) to users. There are only a few options to set including the URL of the Firebase instance and the thread target URL or `href`. If you do not provide your own a Firebase URL, by default, my own resource-limited database is used instead. Therefore serious users will wish to use their own Firebase account and instance.

In addition to basic commenting features, Speakur offers the ability to vote comments up or down, custom profiles, the ability to leave comments in Markdown syntax [10] with syntax highlighting for common programming languages, and (rough) user interface translations (localizations) in 15 languages as shown in Figure 3.1.

From a technical perspective, one of the more interesting features in Speaker is the use of Polymer's data-bound templates, also known as Model-Driven Views, to allow components to automatically reflect changes in far-flung areas of the application. Also, Firebase's event notification architecture allows all web clients to instantly and transparently reflect any changes in another remote client. Another example of the power of data-bound templates is that all of the user-visible text in

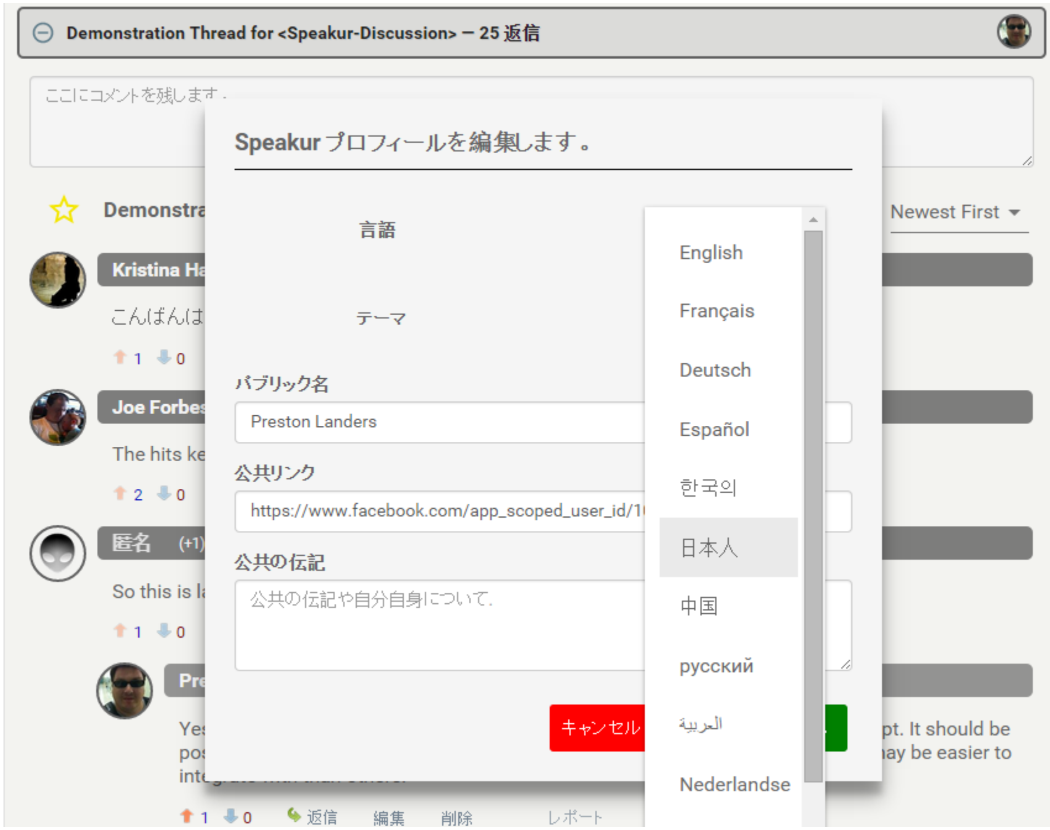

Figure 3.1: Speakur's interface language updates instantly upon selection.

the application updates immediately as soon as the user changes his or her locale preference in the dialog in Figure 3.1.

3.2 Architecture Overview

It has been said that "any problem in computer science can be solved with another layer of indirection" (usually attributed to Wheeler). The key to understanding a software package is learning its architecture, which is really a map of these layers of indirection and abstraction. Roy Fielding, the author of the influential REST web architecture, described a software architecture as:

> ...an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.
>
> At the heart of software architecture is the principle of abstraction: hiding some of the details of a system through encapsulation in order to better identify and sustain its properties. A complex system will contain many levels of abstraction, each with its own architecture [31].

The architecture for Speakur is based on client-side (browser) JavaScript code and HTML layout following Web Components design principles. There is no dedicated server component except for the Firebase cloud database service [9]. Speakur is built entirely from plain HTML, JS and CSS files that can be served from a content distribution network (CDN) such as `github.io` or your own server [46].

This low-overhead design allows Speakur to be used on your own website without actually 'installing' any software; just load Speakur directly from `github.io` with an HTML *import* and then insert a tiny bit of HTML markup into your document [46]. This helps fulfill the ease of use requirements #1 and #2 in section 2.5.2.

Most of the user interface elements in Speakur — things like dropdown menus and dialogs as well as invisible functional components like expand/collapse elements — are implemented with Polymer's Core and Paper custom element libraries. Speakur presents a simple API through `<speakur-discussion>`, but internally it consists of a number of internal abstraction layers or custom elements.

In turn, these internal layers consist of still more focused layers, other Polymer components, simple HTML templates, and wrappers around external JavaScript libraries like `moment.js`.

As previously mentioned, in order to make the `<speakur-discussion>` element available for use, you must first load the Web Components polyfill and then *import* the Speakur element, either from `github.io` or your own server. It is recommended to create your own Firebase instance for data security and resource limitation reasons, but even this is not required. The author's demonstration database will be used by default.

Security for web clients engaging in data manipulation (i.e, posting, editing or deleting comments) is handled entirely through Firebase authentication and data security rules described below. The simplicity of this arrangement means that it is extremely easy to fit Speakur into almost any web application architecture.

3.3 Responsive Design

Because Speakur is not a standalone application but rather a plug-in designed to be embedded into other web pages or apps, the full document is not under its control. This can affect the mobile user experience, but within these limits, Speakur strives to present a responsive interface to different screen sizes.

The primary way it does this is...

3.4 Polymer and Web Components

As described in Chapter 2, Web Components are a W3C initiative to expose certain native browser features in a public, standardized way. Polymer is a Google web framework built from the ground up around Web Components. Polymer also provides the crucial Web Component polyfill library that is required for these features to work on most current browsers.

In general, Speakur tries to adhere to the core principles laid out by the Web Components developers for general purpose components. It's worth quoting those here in full, because they are applicable to software engineering in general. They are:

1. Address a common need.
2. Do one job really well.
3. Work predictably in a wide variety of circumstances.
4. Be useful right out of the box.
5. Be composable.
6. Be styleable.
7. Be extensible.
8. Think small.
9. Adapt to the user and device.
10. Deliver the key benefit to HTML authors, not just coders. [23]

Explain how I follow those guidelines... [TODO]

The Polymer project provides two (optional) libraries of Custom Elements for use in your own projects. **Core Elements** includes both user interface widgets and invisible functional elements. Core Elements are minimally styled and can be used standalone, but they are also used as 'base classes' for **Paper Elements**, which implement the so-called 'Material Design' look and feel (design language) used by apps for Google's Android mobile operating system [38]. Speakur's layout is composed of native HTML5 elements, Core and Paper Elements, and Speaker's own custom elements that are used to abstract internal implementation details.

3.5 Data store and synchronization

All persistent data in Speakur is stored in a cloud database called Firebase [9]. Anyone who wants to use Speakur can register for a free account on `firebase.com` and create a database instance to hold Speakur data. No other server component is required. Firebase is a NoSQL-style key-value data store of the kind that has been popularized with the growth of Node.js, a server-side JavaScript environment, and the MongoDB NoSQL database [29].

Firebase provides WebSockets-based event notification and synchronization as well as a security rule description format based on JSON[1] for securing and validating user actions. The use of Firebase as the only server component allows for easy deployment of Speakur with minimal dependencies, helping to keep the component small and focused.

[1]JavaScript Object Notation (JSON) is a popular alternative to XML because it is more easily human-readable. It mirrors the syntax for writing data structure literals in the JavaScript language.

3.5.1 Firebase API

Firebase can be used 'standalone' as the sole provider of data services to an application, as Speakur does, or else it can be used as an auxiliary to other services or REST APIs. One of the key architectural benefits of using Firebase, besides its ease of deployment, is that its data binding and event notification system allows for applications to respond to changes in real-time while remaining performant.

Firebase itself provides a REST API for data access by programs like Speakur. The term REST or RESTful is sometimes misunderstood, but in Roy Fielding's original 2000 Ph.D. thesis, REST refers to transferring *representations* of application state and using hypertext as the engine of application state[2]. Specifically, 'objects' of whatever type are represented as interlinked hypertext ***resources*** that are operated on by standard HTTP verbs such as PUT and DELETE.

Areas where typical web APIs fall short of being truly "REST-ful" include:

1. Treating URLs as endpoints for remote procedure calls (RPC) instead of hypertext resources that are interlinked in exactly the same way a website is like a tree that starts from the home page and links to various resources.

2. Using HTTP verbs inappropriately, such as using POST for all actions including deletion.

3. Not using content related HTTP headers appropriately for data representation and API versioning [41].

[2]Known under the somewhat awkward acronym of HATEOAS.

The Firebase API follows typical RESTful patterns in data access, allowing the database and its metadata to be addressed as a set of linked HTTP resources. In addition, Firebase client libraries use WebSockets, a lower-level TCP/IP protocol, to perform event notification and distributed synchronization without the overhead of polling or high-overhead HTTP 1.1 requests. WebSockets are used for performance reasons but all of the data in Firebase (and hence, in Speakur) is accessible from the RESTful HTTP API outside of a WebSocket.

3.5.2 WebSockets

The WebSockets protocol, formally known as RFC 6455, is a TCP/IP protocol that can be used alongside HTTP for persistent data connections between web clients and servers [12]. The primary purpose is to avoid the overhead of initiating a new HTTP connection to check on the status of something on the server, also known as polling. Firebase uses Web Sockets rather than traditional high-overhead HTTP requests to move data back and forth to the client. This always-on connection allows for sending nearly instant event notifications to all currently active clients with minimal overhead. In practice, this allows the application to update its state in real-time as different users read and write values the database. The Firebase client library 'subscribes' to an area of interest in the database, such as the replies to a particular thread, and receives notifications when these areas change and updates the local representation as appropriate.

3.6 Security

Because Speakur relies entirely on Firebase for data persistence, that means its security model is largely built around Firebase. All Speakur code runs inside the client's web browser including the small 'admin' mode. The security architecture of the web is largely based around protecting the *user* from malicious servers and other users, not protecting the *server* from the user. The server has to take its own steps to protect itself from unauthorized actions. Speaker has no 'server' as such other than the Firebase cloud service. That means the security mechanisms that do things like prevent users from deleting each other's posts are implemented entirely within Firebase security rules as discussed in section 4.4.2. Confidentiality of data in-transit is handled with the same Transport Layer Security (TLS) better known as the secure socket layer (SSL) or HTTPS.

Firebase implements the two major categories of access control: authentication and authorization. Authentication (sometimes abbreviated *authn*) answers the question "who are you?" while authorization (*authz*) asks "what are you allowed to see and do?" [57].

3.6.1 Authentication

Speakur's authentication and sign-in system is handled through Firebase, specifically Google and Facebook OAuth single-sign on (SSO). Users can sign into a Speakur discussion thread, and hence Firebase, through their Facebook or Google identity. Firebase supports other authorization schemes, including account registration ("simple password"), Twitter and GitHub identities. Site owners who use

Speakur can also designate certain threads to allow anonymous commenting.

Every user who registers within a Speakur instance by signing in with one of those identity providers gets a unique identifier — the `uid` or user ID. This `uid` is used extensively in the database to refer to the user, including in the security (authorization) rules that are external to the actual database (i.e, security metadata.)

3.6.2 Authorization

Authorization rules determine what level of access a user has within the system, or what they are allowed to do or see.

3.7 Data Flow and Event Handling

3.7.1 Mutation Observers

3.8 Dependencies and Deployment

Bower

Vulcanize

CORS

Chapter 4

Implementation

4.1 Layout and Structure

4.2 Database Design

4.3 Distributed Synchronization

4.3.1 Data Bindings

4.3.2 Data-Bound Templates

4.4 Security

4.4.1 Authentication

4.4.2 Firebase Security Policy Rules

4.5 Internationalization

4.5.1 Architecture

4.5.2 Libraries

4.6 Responsive Design

4.6.1 Mobile Users

4.6.2 Accessibility

4.7 Publishing and Deployment

Chapter 5

Analysis

5.1 Using Speakur

5.2 Lessons Learned

5.3 Future of Web Components

Chapter 6

Conclusions

Concluding remarks here...

Summary of lessons learned.

Is this *the future*?

Appendices

Appendix A

Speakur source code example

This index may contain an example of Speakur's source code...

Appendix B

Open Source Credits

Speakur would not have been possible without the following open source components. The components listed here are the direct Bower dependencies. Some of these may install other dependencies in turn.

- Polymer framework [13]:

  Web Components polyfill, core library, Core Elements and Paper Elements.

- Firebase database service and client library [9]
- Marked library
- highlight.js library
- i18next library
- moment.js library
- lodash.js library
- jQuery.js library
- js-md5 library
- pvc-globals custom element

Index

Bibliography

[1] Tim Berners-Lee. *Hypertext Markup Language (HTML)*. June 1993. URL: `http://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt`.

[2] Eric Bidelman. *HTML Imports*. Dec. 2013. URL: `http://www.html5rocks.com/en/tutorials/webcomponents/imports/`.

[3] Eric Bidelman. *Shadow DOM 201: CSS and Styling - HTML5 Rocks*. Apr. 2014. URL: `http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom-201/`.

[4] Eric Bidelman. *Using Polymer in a WebView*. Mar. 2015. URL: `https://www.polymer-project.org/0.5/articles/webview.html`.

[5] Michael Bleigh. *Sneak Peak of Polymer 0.8*. Feb. 2015. URL: `https://divshot.com/blog/web-components/polymer-0-8-sneak-peek/`.

[6] Richard Clark. *Avoiding common HTML5 mistakes*. July 2011. URL: `http://html5doctor.com/avoiding-common-html5-mistakes/`.

[7] Bootstrap contributors. *Twitter Bootstrap*. Mar. 2015. URL: `http://getbootstrap.com/`.

[8] Bower Contributors. *Bower*. 2015. URL: `bower.io/search`.

[9] Firebase contributors. *Firebase*. Mar. 2015. URL: `https://www.firebase.com/`.

[10] GitHub contributors. *GitHub Flavored Markdown*. 2015. URL: `https://help.github.com/articles/github-flavored-markdown/`.

[11] Mozilla contributors. *Using CSS flexible boxes*. 2015. URL: `https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Flexible_boxes`.

[12] Mozilla contributors. *WebSockets*. Feb. 2015. URL: `https://developer.mozilla.org/en-US/docs/WebSockets`.

[13] Polymer contributors. *Polymer*. 2015. URL: `https://www.polymer-project.org/0.5/docs/start/everything.html`.

[14] React contributors. *React*. 2015. URL: http://facebook.github.io/react/.

[15] W3C contributors. *A Short History of Javascript*. Feb. 2012. URL: https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript.

[16] W3C contributors. *CSS Grid Layout Module*. Mar. 2015. URL: http://www.w3.org/TR/css3-grid-layout/.

[17] W3C contributors. *Custom Elements*. Mar. 2015. URL: http://w3c.github.io/webcomponents/spec/custom/.

[18] W3C contributors. *HTML Imports*. Mar. 2015. URL: http://w3c.github.io/webcomponents/spec/imports/.

[19] W3C contributors. *Mutation Observers*. July 2014. URL: http://www.w3.org/TR/dom/#mutation-observers.

[20] W3C contributors. *Selectors API*. Feb. 2013. URL: http://www.w3.org/TR/selectors-api/.

[21] W3C contributors. *Shadow DOM*. Mar. 2015. URL: http://w3c.github.io/webcomponents/spec/shadow/.

[22] W3C contributors. *Template element*. Mar. 2015. URL: https://html.spec.whatwg.org/multipage/scripting.html#the-template-element.

[23] Web Components contributors. *Ten Principles for Great General Purpose Web Components*. Nov. 2014. URL: https://github.com/basic-web-components/components-dev.

[24] X-Tags contributors. *X-Tags*. 2015. URL: http://www.x-tags.org/.

[25] Google Developers. *AngularJS*. Mar. 2015. URL: https://angularjs.org/.

[26] Google Developers. *Easier website development with Web Components and JSON-LD*. Mar. 2015. URL: http://googlewebmastercentral.blogspot.com/2015/03/easier-website-development-with-web.html.

[27] Google Developers. *Model-Driven Views*. July 2014. URL: http://mdv.googlecode.com/svn/trunk/docs/model.html.

[28] Google Developers. *The Awesome Power of Auto-Binding Templates – Polycasts #08*. Jan. 2015. URL: https://www.youtube.com/watch?v=82LfXCeuaOo.

[29] J. Dickey. *Write Modern Web Apps with the MEAN Stack: Mongo, Express, AngularJS, and Node.js*. Develop and Design. Pearson Education, 2014. ISBN: 9780133962376.

[30] Santiago Esteva. *AngularJS 2 Status Preview*. Mar. 2015. URL: http://ng-learn.org/2014/03/AngularJS-2-Status-Preview/.

[31] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[32] Flaki. *The JavaScript World Domination*. Mar. 2015. URL: https://medium.com/@slsoftworks/javascript-world-domination-af9ca2ee5070.

[33] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012. ISBN: 9780133065213.

[34] Ewa Gasperowicz. *Creating semantic sites with Web Components and JSON-LD*. Mar. 2015. URL: http://updates.html5rocks.com/2015/03/creating-semantic-sites-with-web-components-and-jsonld.

[35] Matthias Gelbmann. *Why jQuery is the Most Popular JavaScript Library*. Aug. 2012. URL: http://w3techs.com/blog/entry/jquery_now_runs_on_every_second_website.

[36] Ian Hickson et al. *HTML5 A vocabulary and associated APIs for HTML and XHTML*. W3C Recommendation 28 October 2014. http://www.w3.org/TR/2014/REC-html5-20141028/. W3C, Oct. 2014.

[37] Colin Ihrig. *The Basics of the Shadow DOM*. Aug. 2012. URL: http://www.sitepoint.com/the-basics-of-the-shadow-dom/.

[38] Tomomi Imura. *Creating a Polymer Chat App with Material Design*. Jan. 2015. URL: http://www.pubnub.com/blog/creating-a-polymer-chat-app-with-material-design/.

[39] InternetLiveStats.com. *Total Number of Websites*. Mar. 2015. URL: http://www.internetlivestats.com/total-number-of-websites/.

[40] Eiji Kitamura. *Introduction to Shadow DOM*. Oct. 2014. URL: http://webcomponents.org/articles/introduction-to-shadow-dom/.

[41] Steve Klabnik. *Nobody Understands REST or HTTP*. July 2011. URL: http://blog.steveklabnik.com/posts/2011-07-03-nobody-understands-rest-or-http.

[42] Yves Lafon et al. *Web Applications Working Group Charter*. 2015. URL: http://www.w3.org/2014/06/webapps-charter.html.

[43] Preston Landers. *Conditionally wrapping <content> insertion points in Polymer*. Feb. 2015. URL: http://stackoverflow.com/questions/28330000/conditionally-wrapping-content-insertion-points-in-polymer.

[44] Preston Landers. *Eurgh! translation software*. Feb. 2015. URL: https://github.com/Preston-Landers/eurgh.

[45] Preston Landers. *Polymer: adding implicit arguments to function calls in expressions - Stack Overflow*. Feb. 2015. URL: http://stackoverflow.com/questions/28530725/polymer-adding-implicit-arguments-to-function-calls-in-expressions.

[46] Preston Landers. *Speakur Demo*. Mar. 2015. URL: https://preston-landers.github.io/speakur-discussion/components/speakur-discussion/demo.html.

[47] Preston Landers. *Speakur-Discussion*. Mar. 2015. URL: https://github.com/Preston-Landers/speakur-discussion.

[48] Matthew MacDonald. *HTML5: The Missing Manual*. 2nd. O'Reilly Media, Inc., 2013.

[49] Nathan Marz. *How to beat the CAP theorem*. Oct. 2011. URL: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[50] Meligy. *The Longest Write-Up On ng-conf, AngularJS 1.3, 1.4, 1.5 AND 2.0 Yet*. Mar. 2015. URL: http://gurustop.net/newsletter/10.

[51] John C. Mitchell. *Concepts in programming languages*. Cambridge, U.K. ; New York: Cambridge University Press, 2003. ISBN: 0521780985.

[52] Addy Osmani. *Detect, Undo And Redo DOM Changes With Mutation Observers*. June 2014. URL: http://addyosmani.com/blog/mutation-observers/.

[53] Soledad Penades. *An Introduction to Web Components*. Jan. 2015. URL: http://webcomponents.org/presentations/an-introduction-to-web-components-at-web-components-london.

[54] Allen Pike. *A JS Framework on every table*. Feb. 2015. URL: http://www.allenpike.com/2015/javascript-framework-fatigue/.

[55] Pascal Precht. *Inheritance and composition with Polymer*. July 2014. URL: https://pascalprecht.github.io/2014/07/14/inheritance-and-composition-with-polymer/.

[56] Andrew Rota. *Complementarity of React and Web Components*. Jan. 2015. URL: http://webcomponents.org/presentations/complementarity-of-react-and-web-components-at-reactjs-conf.

[57] William Stallings. *Cryptography and network security: principles and practice*. 5th. Boston: Prentice Hall, 2011. ISBN: 9780136097044.

[58] Chris Strom. *I18next, Polymer and Pluralization*. Feb. 2014. URL: http://japhr.blogspot.com/2014/02/i18next-polymer-and-pluralization.html.

[59] Chris Strom. *Patterns in Polymer*. 2014. URL: http://patternsinpolymer.com/.

[60] Luis Vieira. *HTML5 Local Storage Revisited*. Mar. 2015. URL: http://www.sitepoint.com/html5-local-storage-revisited/.

[61] W3C. *Web Applications Working Group*. 2015. URL: http://www.w3.org/2008/webapps/.

[62] Phillip Walton. *Web Components and the Future of CSS*. Nov. 2014. URL: http://webcomponents.org/presentations/web-components-and-the-future-of-css/.

[63] Erik Zachte. *Wikimedia Traffic Analysis Report - Browsers e.a.* Mar. 2015. URL: https://stats.wikimedia.org/wikimedia/squids/SquidReportClients.htm.

Vita

Preston Brent Landers was born in Texas and attended high school on the Nevada side of Lake Tahoe. He received his Bachelor of Arts in English from the University of Texas at Austin. He works as a web and mobile software engineer for Journyx, Inc.[*] in Austin, Texas and began graduate studies in Software Engineering at the University of Texas at Austin in August 2012.

Permanent address: `planders@utexas.edu`

This report was typeset with LaTeX[†] by the author.

[*] `http://www.journyx.com`

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.