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 encapsulation, composition and distributed synchronization 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 applying HTML5 Web Component technologies 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 millions of public and private web sites forming a major part of our information landscape. [CITE?] 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 computing platform have long dreamed of the ability to mix and match independent, reusable

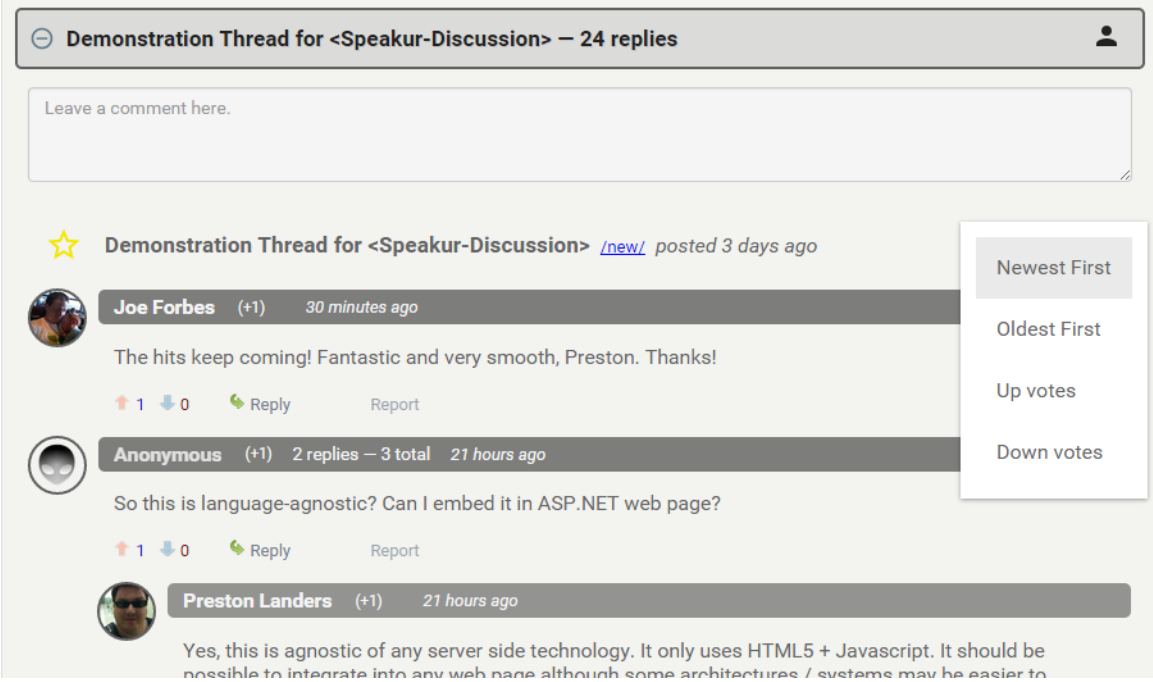

Figure 1.1: A Speakur thread inside a demonstration page.

chunks of functionality — components — in their documents without mutual coupling and interference. The original and current Document Object Model (DOM) browser abstraction provided by HTML 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

[1]Javascript, also rendered as JavaScript or JS, has no significant relation to Sun's (now Oracle's)

for a completely new dimension of dynamic behavior that was not possible before. 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.

Over the years the dynamic behaviors afforded by Javascript grew in importance along with the web, and helped contribute to its success. The flexible, loosely typed nature of Javascript aided the prototyping process and initial development, but the difficulties of maintaining a semblance of coherence in a large sprawling application 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 client side web apps. For many years individual JS frameworks seemed to come and go as ephemerally as teenage pop idols. Interoperability between these completing frameworks was virtually non-existent. Components from one framework typically couldn't be easily 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.

---

popular Java programming language; the name is an unfortunate coincidence at best.

To complicate the naming situation even further, Javascript is officially standardized under the name ECMAScript (ES).

In recent years (roughly 2011 to 2014) Google's Angular [CITE] has emerged as a dominant client framework, due in part to its high perceived quality [CITE] and the fact that it represents an common point for a fragmented industry to rally around. Facebook's React JS 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.

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 [**WHEN?**] to draft a new standard called **Web Components** that will extend and enhance HTML5 in ways that could have a significant long-term impact. For example, 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 [CITE].

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 stan-

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

[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.

dards 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:

- **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 [CITE]. 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 already have at least some native Web Component support and on these browsers the polyfill is effectively a 'no-op'.

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 encapsulation, modular composition, and the synchronization of distributed 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 that went into 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.

```
<!-- 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:

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

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

`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 simple 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. 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" [26, p. 522]. The point of encapsulation is providing an *abstraction* that consumers of the functionality can rely on without knowing 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. [26, 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 [18].

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 [4]. 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 militates against Bootstrap significantly refactoring the internal structure of the navigation widget because that would require a 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 minimal interference with other components and unintended effects. Even then, conflicts are inevitable when the entire page is treated as a single sandbox and you combine components from many different vendors.

```
<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: A partial example of Twitter Bootstrap navigation bar HTML.

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

```
<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 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. 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 highlighted.

This example illustrates two design principles that are widely followed in other areas of software engineering [CITE]:

- Use **encapsulation** and well defined interfaces, as the <video> element does,

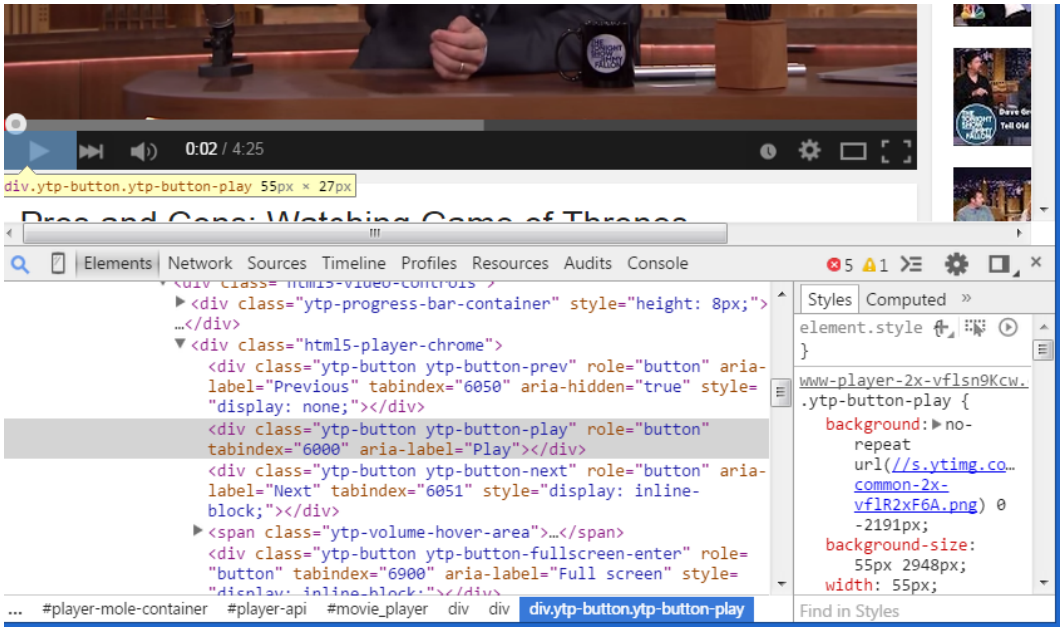

Figure 2.2: Opera's shadow DOM for `<video>` highlighting the Play button

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.

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

The Web Component initiative 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 initiatives 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.

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. Technically, the DOM has long supported cre-

ating custom-named elements, but it was not possible to do much with them because they were treated like an ordinary `<span>`. However the possibility now exists to create custom elements in a standard way that will work consistently across browsers.

TODO: Custom elements:

`http://www.w3.org/TR/custom-elements/`

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 simple 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. As we have seen, browsers use Shadow DOM to encapsulate the private state of standard elements like `<video>` but now this capability is extended to custom-defined elements.

TODO:

`http://www.w3.org/TR/shadow-dom/`

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.

Custom elements can wrap regular text, normal HTML elements, or other custom elements and then project that content into its own internal structure. In the example in figure `??` above, a simple `<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

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

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. But in the case of Web Components, Shadow DOM is 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.

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 found in two different spots on the page but that resource would be requested from the server 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. The one major caveat is that de-duping only happens if the resources are named in exactly the same fashion in each case. Dealing with HTML Imports in a consistent fashion will be discussed in the Implementation section.

TODO: HTML imports:

`http://www.w3.org/TR/html-imports/`

`http://www.html5rocks.com/en/tutorials/webcomponents/imports/`

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 not yet widely used outside of WC. *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.

TODO: Template elements:

`http://www.w3.org/TR/html5/scripting-1.html#the-template-element`

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 template for custom element representation.

However, templates are most useful in combination with 'live' data, not static, unchanging text. Binding data into templates with special operators[2] 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>
```

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.

2.2.5 Related technologies

There are a number of related W3C initiatives for web standards. Sometimes these are loosely grouped under the label Web Components, and they do

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

help support componentization, but they are separate part of the HTML5 standard.

Mutation observers:

`http://www.w3.org/TR/dom/#mutation-observers`

Model driven views:

`http://mdv.googlecode.com/svn/trunk/docs/design_intro.html`

Pointer events:

`http://www.w3.org/TR/pointerevents/`

Web animations:

`http://www.w3.org/TR/web-animations/`

Selectors (similar to jQuery selectors)

`http://www.w3.org/TR/selectors-api/`

```
// find an element based on 'id' attribute.
var someElem = document.querySelector("#some-id");
```

Placeholder Notes: A significant problem with 'web components' story - scoping! There is just one global scope on the page. This leads to the practice of 'prefixing as poor man's scope'. E.g. instead of facebook providing a <like-button> element, they provide <facebook-like-button>. apparently this may be an area of future development (citation?)

Flex boxes layout: intended for smaller page components...

`https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Flexible_boxes`

CSS Grid layout intended for overall page layout...

`http://www.w3.org/TR/css3-grid-layout/`

2.3 Literature Review

2.3.1 Javascript frameworks

2.3.2 Polymer framework

A team within Google has developed the Polymer [7] framework based on Web Components architecture.

2.4 Speakur

My desire to learn more about modern web development led me to investigate web frameworks like Angular and Meteor. I spent some time building (very) simple 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.

2.4.1 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. 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.

2.4.2 Motivations

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

- Provide a simple API to consumers that hid most implementation details.
- Require minimal server resources. Ideally nothing would need to be "installed" and it could be loaded in a cross-origin fashion from online web developer tools like `https://jsbin.com`.
- Support Markdown formatted comments including syntax highlighting for code snippets.
- Support internationalization (i18n) and localization (l10n) features for a global audience.

- Support distributed event notification similar to the publish-subscribe (pub-sub) design pattern.

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

- 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 [12] and a few other smaller 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

3.1 Functionality

3.2 Architecture Overview

Roy Fielding, the influential author of the REST web architecture, wrote that

> a software architecture is 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. [15]

3.3 Responsive Design

3.4 Polymer and Web Components

3.5 Data store and synchronization

3.5.1 NoSQL and Firebase

'Serverless'

3.5.2 Web Sockets

3.5.3 Security

3.6 Data Flow and Event Handling

3.6.1 Mutation Observers

3.7 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

Lerma's Appendix

The source LaTeX file for this document is no longer quoted in its entirety in the output document. A LaTeX file can include its own source by using the command `\verbatiminput{\jobname}`.

Appendix B

My Appendix #2

B.1 The First Section

This is the first section. This is the second appendix.

B.2 The Second Section

This is the second section of the second appendix.

B.2.1 The First Subsection of the Second Section

This is the first subsection of the second section of the second appendix.

B.2.2 The Second Subsection of the Second Section

This is the second subsection of the second section of the second appendix.

B.2.2.1 The First Subsubsection of the Second Subsection of the Second Section

This is the first subsubsection of the second subsection of the second section of the second appendix.

B.2.2.2 The Second Subsubsection of the Second Subsection of the Second Section

This is the second subsubsection of the second subsection of the second section of the second appendix.

Appendix C

My Appendix #3

C.1 The First Section

This is the first section. This is the third appendix.

C.2 The Second Section

This is the second section of the third appendix.

Bibliography

[1] T. Berners-Lee. *Hypertext Markup Language (HTML)*. 00000. June 1993. URL: http://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt.

[2] E. Bidelman. *Using Polymer in a WebView - Polymer*. 00000. Mar. 2015. URL: https://www.polymer-project.org/0.5/articles/webview.html.

[3] R. Clark. *Avoiding common HTML5 mistakes*. July 2011.

[4] B. contributors. *Twitter Bootstrap*. Mar. 2015.

[5] B. Contributors. *Bower search*. 2015. URL: bower.io/search.

[6] M. Contributors. *Using CSS flexible boxes*. 2015. URL: https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Flexible_boxes.

[7] P. contributors. *Polymer*. 2015.

[8] W. contributors. *Custom Elements*. 00104. Mar. 2015. URL: http://w3c.github.io/webcomponents/spec/custom/.

[9] W. contributors. *HTML Imports*. 00000. Mar. 2015. URL: http://w3c.github.io/webcomponents/spec/imports/.

[10] W. contributors. *Shadow DOM*. 00002. Mar. 2015. URL: http://w3c.github.io/webcomponents/spec/shadow/.

[11] W. contributors. *Template element*. 00002. Mar. 2015. URL: https://html.spec.whatwg.org/multipage/scripting.html#the-template-element.

[12] X. contributors. *X-Tags*. 2015.

[13] G. Developers. *Easier website development with Web Components and JSON-LD*. 00000. Mar. 2015.

[14] G. Developers. *The Awesome Power of Auto-Binding Templates – Polycasts #08*. Jan. 2015.

[15] R. T. Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[16] E. Gasperowicz. *Creating semantic sites with Web Components and JSON-LD - HTML5 Rocks*. 00000. Mar. 2015. URL: /2015/03/creating-semantic-sites-with-web-components-and-jsonld.

[17] I. 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.

[18] C. Ihrig. *The Basics of the Shadow DOM*. Aug. 2012.
[19] T. 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/.
[20] P. Landers. *Eurgh! translation software*. 00000. Feb. 2015. URL: https://github.com/Preston-Landers/eurgh.
[21] P. 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.
[22] P. Landers. *web component - Conditionally wrapping <content> insertion points in Polymer - Stack Overflow*. Feb. 2015. URL: http://stackoverflow.com/questions/28330000/conditionally-wrapping-content-insertion-points-in-polymer.
[23] M. MacDonald. *HTML5: The Missing Manual*. 2nd. " O'Reilly Media, Inc.", 2013.
[24] N. Marz. *How to beat the CAP theorem - thoughts from the red planet - thoughts from the red planet*. 00000. Oct. 2011. URL: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.
[25] Meligy. *The Longest Write-Up On ng-conf, AngularJS 1.3, 1.4, 1.5 AND 2.0 Yet – Or – Meligy's AngularJS & Web Dev Goodies — Issue10*. 00000. Mar. 2015.
[26] J. C. Mitchell. *Concepts in programming languages*. Cambridge, U.K. ; New York: Cambridge University Press, 2003. ISBN: 0521780985.
[27] P. Precht. *Inheritance and composition with Polymer · Pascal Precht*. July 2014. URL: https://pascalprecht.github.io/2014/07/14/inheritance-and-composition-with-polymer/.
[28] A. Rota. *Complementarity of React and Web Components — WebComponents.org*. 00000. Jan. 2015. URL: http://webcomponents.org/presentations/complementarity-of-react-and-web-components-at-reactjs-conf.
[29] C. Strom. *japh(r) by Chris Strom: I18next, Polymer and Pluralization*. Feb. 2014.
[30] C. Strom. *Patterns in Polymer*. 2014.
[31] L. Vieira. *HTML5 Local Storage Revisited*. Mar. 2015.
[32] E. Zachte. *Wikimedia Traffic Analysis Report - Browsers e.a*. 00001. 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.