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 live synchronization of user interface state.

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 [47]. 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 chunks of

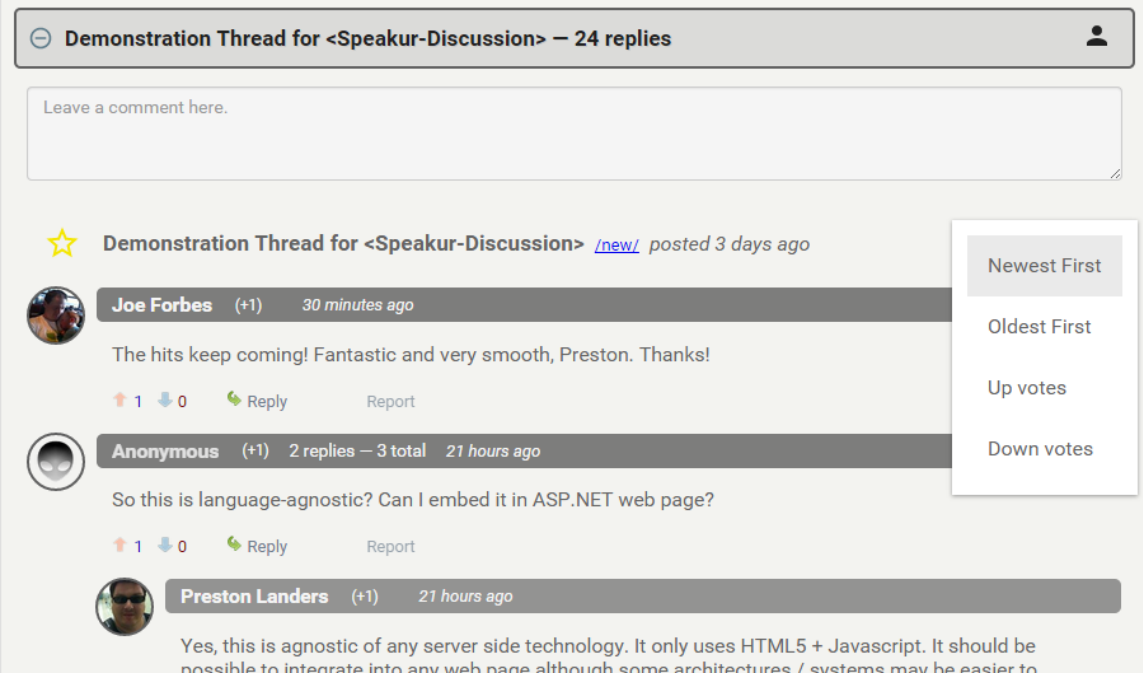

Figure 1.1: A Speakur thread inside a demonstration page.

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 di-

[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).

mension of dynamic behavior that was not possible before [22]. 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 [45].

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) [40]. 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 frameworks seemed to come and go as ephemerally as teenage pop idols [63]. 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 [63].

In recent years (roughly 2011 to 2014) Google's Angular [33] 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 [37]. 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 [21].

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 [71] to draft a new standard called **Web Components** that will extend and enhance HTML5 in ways that could have a significant long-term impact [51]. 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 [38].

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

[2] As of March 2015, Google's Chrome browser is the most popular desktop browser [73].

[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.

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 [62]:

- **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 [19]. 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.

```
<!-- 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 [53]:

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

Demonstrations of several web pages which show off embedded Speakur discussions are available at the following location [56]:

`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 [37]. 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" [60, 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 [60, 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 [45]. 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.

```
<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.

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) be-

cause 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 [72]. 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 encapsulated 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:

```
<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 [49]. 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.

A small `#player-api` tag is visible in the DOM path at the bottom of Figure 2.2. 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.

This example illustrates three component design principles that are widely followed in other areas of software engineering [41]:

- 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.

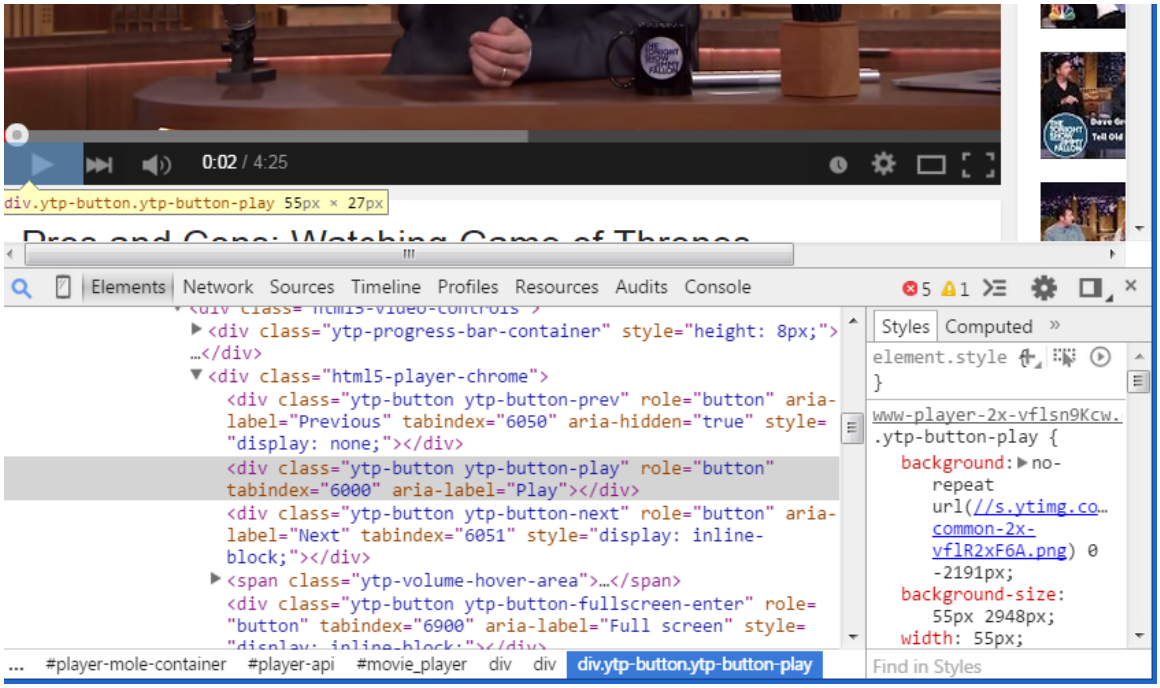

Figure 2.2: Opera's shadow DOM for `<video>` highlighting the Play button

- 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 [62]. 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 element [24]. 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 [24].

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

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

or imperatively with JavaScript:

```
1 var MyElement = document.registerElement('my-element');
2
3 // create a new instance of the element
4 var thisOne = new MyElement();
5 document.body.appendChild(thisOne); // add to the <body>
```

Listing 2.2: Registering a Custom Element in JavaScript.

In a small example like this, the result does not look all that different in the browser from a . 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 [28]. 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 later) 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 DOM, therefore into 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

[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.

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 that are popular in scripting languages like Python, Go and Ruby. JavaScript can always be loaded with a `<script>` tag like usual, but this does not ensure that resources are 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 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 [25]. 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 is discussed in section 3.8.1.

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 [29]. *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:

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

Listing 2.3: An example of a data-bound template.

Google engineers have advanced a proposal for standardizing Model-Driven Views but it is not yet part of the standard `<template>` element [35]. Instead this functionality

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

can be handled by a JavaScript framework such as React or Polymer. Data-bound templates are discussed in more detail in the following chapters.

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 components, not all of which are needed immediately.

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 [26]. This helps decouple components by allowing them to react asynchronously to changes in each

other's public state without directly interconnecting the components with bound variables. The observers "deliver mutations in batches asynchronously at the end of a microtask rather than immediately after they occur" [61]. This improves the 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 [43]. The Selector API was officially standardized in 2013 and browser support is good in current versions [27].

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`:

```
1  // find an element whose id is equal to 'some-id'
2  var someElem = document.querySelector("#some-id");
3  if (someElem) console.log("Found it!");
4  else console.log("Wasn't there.");
```

Listing 2.4: JavaScript query selector example.

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 [23]. 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 [14]. Flex boxes are useful for general user interface (UI) structure, not just device-responsive design, and are widely used in Polymer and in Speakur [18].

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 [19] 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 [31]. 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]. Small 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 built elementary demos with these two frameworks in particular. Although they certainly let you 'get stuff done', 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 to create something that demonstrated common use cases for Web Components and also showed off some of the design possibilities provided by Polymer and Material Design [46]. 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 for other web authors to actually use it.

2.5.2 Motivations

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

1. A simple API and usage pattern that abstracted away most of the 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. Event notification for changes similar to the publish-subscribe (pubsub) design pattern. For example, automatically updating when new replies are posted or existing posts are edited.

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 ruled out the vast majority of frameworks, leaving only Polymer and the less-comprehensive X-Tags project [31] and a few other less ambitious contenders.

The next chapter discusses some of the high level architectural concerns that should be addressed when designing this type of social web plugin, such as creating a scalable database schema, designing a device-responsive user interface, and providing authentication and authorization rules.

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 an embeddable 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 4.8 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 4.8:

1. loading the Web Components polyfill 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 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 available 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 resource-limited database is used instead. Therefore serious users will wish to use their own Firebase account with their own database.

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

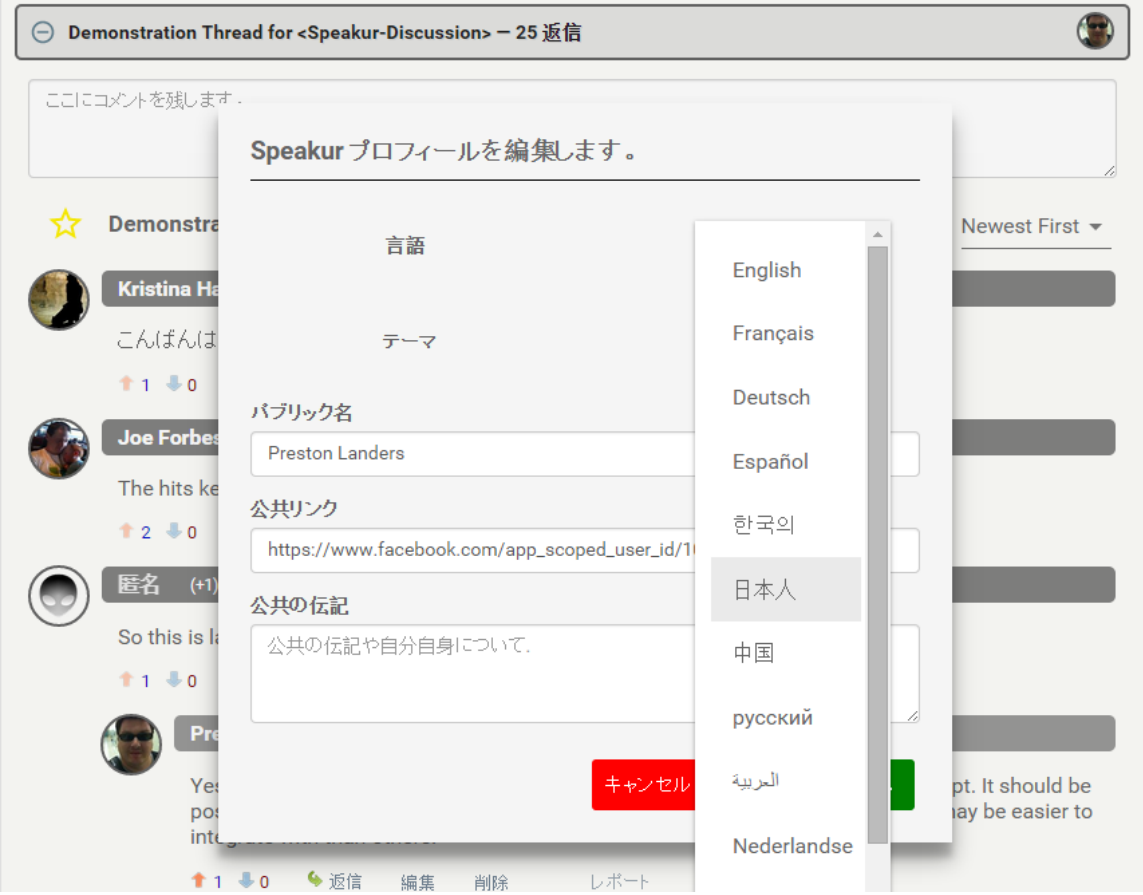

Figure 3.1: Speakur's interface language updates instantly upon selection.

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 [39].

The architecture for Speakur is based on client-side (browser) JavaScript code and HTML layout following the Web Components design principles listed in section 3.4 below. 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 delivery network (CDN) such as `github.io` or your own server [55].

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 [55]. 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`. These internal layers are described in section 4.2.

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 the Firebase authentication and data security rules described below. The simplicity of this arrangement makes it easier to fit Speakur into almost any web application architecture.

3.3 Responsive Design

Although 'native' apps are frequently preferred by mobile users and developers, the mobile web remains an important development platform for the same reason that desktop web apps live alongside native desktop apps; web apps typically do not require installing anything to the device, are always up-to-date, and have a lower barrier to entry overall. 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 two main techniques used in Speakur for responsive design are CSS flexible

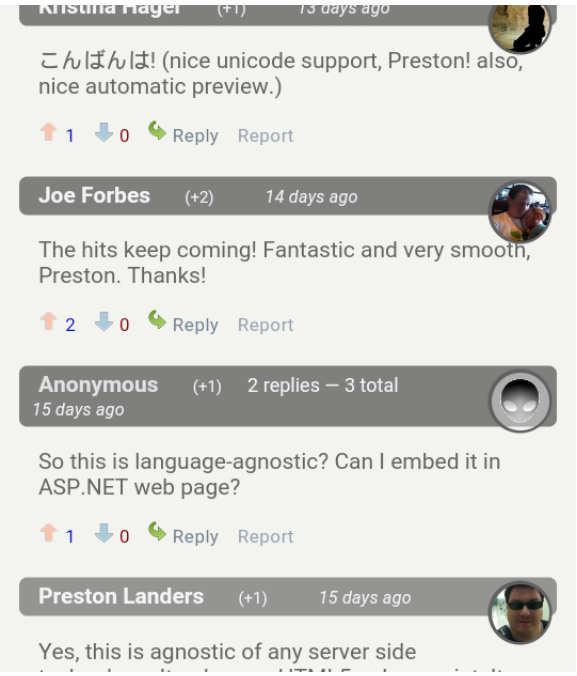

Figure 3.2: Speakur thread on a mobile phone.

(flex) boxes [14] and CSS `@media` rules that apply varied styles to different screen sizes. In addition, Speakur's JavaScript code exposes a `smallScreen` flag to the HTML templates that can trigger layout adjustments, such as moving the user photo to a different location.

The desktop version in Figure 1.1 shows the user photo to the left of the main post area, but the mobile version in Figure 3.2 above shows how the user photo is moved to the right side of the post header. A CSS `@media` rule for small screens disables the indentation of replies shown at the bottom of Figure 1.1. The extensive use of CSS flex box attributes ensures that structural elements like headers and toolbars automatically adjust their layout to available space [18]. This will be described in more detail in section 4.7.

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 polyfill library that is required for Web Component 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. [30]

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 directly, 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 [46]. 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 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 [37].

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 adhere to the principles of "think small" and "be useful right out of the box" from above.

[1] JavaScript Object Notation (JSON) is a popular data format because it's easily human-readable, but its free-form structure compared to XML can be both a blessing and a curse. It mirrors the syntax for writing data structure literals in the JavaScript language.

3.5.1 RESTful API

Firebase can be used by itself 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] [39]. Specifically, 'objects' of whatever type are represented as interlinked hypertext ***resources*** that are operated on by standard HTTP verbs such as PUT and DELETE. For example, in a Speakur thread, a particular user comment (post) is an abstract *resource* located at the following uniform resource locator (URL):

`https://YOUR-DB.firebaseio.com/posts/$ParentId/$PostId`

where `$PostId` is an identifier (id) for the post itself, and `$ParentId` is the id of the post this is a reply to, which could be the top-level post (thread id.) In order to delete this post, one would issue an HTTP request to this URL with the `DELETE` verb, as in this `bash` shell script snippet:

```
1 set DATABASE="https://my-firebase.firebaseio.com"
2 set POST="fake_post_id"  # the post to delete
3 set PARENT="parent_post_or_thread_id"
```

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

```
4
5 curl --request DELETE $DATABASE/posts/$PARENT/$POST
```

Listing 3.1: Deleting a post with the REST API.

Of course, the server would be expected to check your authorization to delete this resource. Viewing (reading) the post is done with a simple `GET` request. Creating a brand new post can be done with `PUT`. Updating an existing post's text is done with the `POST` verb. By default, the resources are *represented* as JSON encoded data, but a client can request an alternative representation such as XML. The storage format for the resource 'at rest' in the server database is irrelevant from the API perspective.

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. Not closely following HTTP semantics, especially using HTTP verbs inappropriately like using `POST` for all actions including deletion.

3. Not using content related HTTP headers appropriately for data representations and API versioning [50].

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 starting from a single root. 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 to enhance performance 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 [15]. Its primary purpose is to avoid the overhead of having the client initiate a new HTTP connection to check on the status of something on the server, also known as polling. Aside from the fact that an HTTP connection can be 'upgraded' (protocol switched) to WebSockets, there is no direct connection or dependency between HTTP and WebSockets. They can be used independently.

Firebase uses WebSockets 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, its security model is tied heavily to Firebase capabilities. All Speakur code runs inside the client's web

browser including the small 'admin' mode. The security architecture of the web is for the most part about protecting the *user* from malicious servers (and other users), not protecting the *server* from the user. It is assumed that servers protect themselves from unauthorized actions. Because 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 must be implemented entirely within Firebase security rules. This is discussed in more detail in section 4.5.

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?" [67]. Confidentiality of in-transit data is handled with Transport Layer Security (TLS), better known as the secure socket layer (SSL) or HTTPS.

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.) Thread owners also have the option of allowing anonymous posts by users who are not signed in.

3.6.2 Authorization

Authorization security rules determine what level of access a user has within the system and what they are allowed to do or see. Speakur's security rules are described in more detail in section 4.5. The set of security rules for a Speakur database is contained in a JSON encoded file that contains expressions that determined whether a given database read or write (change) is allowed. For example, Speakur has defined security rules which prohibit anyone from editing or deleting an existing post unless they are the original post author or an authorized administrator. Here is an excerpt for the `posts` resource from the security rules file:

```
"posts": {
  ".read": true, // anyone can read posts
  ".write": false,  // deny modification by default
  "$parentId": {
    "$childId": {
      // must be admin or post owner to modify a post.
      ".write": "data.exists() ? ( auth.uid === data.child('author').
         child('uid').val() || root.child('admins').child(auth.uid).
         child('scope').val() === '*' ) : true",
      // validate structure of new posts
      ".validate": "newData.hasChildren(['threadId', 'text', 'author'])
         && newData.child('timestamp').val() > 1"
    }
  }
}
```

Listing 3.2: Security rules for the `posts` table (user messages).

This rule structure is dictated by the architecture of Firebase. This setup has certain pros and cons that can be seen from Listing 3.2. On one hand, it's logically designed and the expressions provide a powerful and fairly comprehensive way to specify the security and authorization logic for your database. On the other hand, non-trivial rule expressions can get awkwardly long and in complex cases can devolve into a maze of nested ternary operators. Having one big file with all the JSON formatted security expressions needed for a database works great for simpler cases but may become unwieldy in a more complex application.

3.7 Data Flow and Event Handling

One important area of Speakur's architecture is the flow of data within the system and responding asynchronously to local and remote user events. To understand the general flow of information in Speakur (and Polymer), imagine the component as an inverted tree, with the 'root' at the top being the main `<speakur-discussion>` element, and the nodes and leaves under that being the internal parts such as the headers, the posts, and other user interface controls as shown in Figure 4.3. In this model, information flows *down* through data bindings and bubbles *up* through events.

Data binding is one of the most important extensions to standard DOM that is provided by the Polymer framework. This ties (or 'binds') a variable from the data model to one or more spots in the shadow DOM, or else to an input or output from some other component, such that any change in the variable is reflected in the bound locations. Two primary uses of data bindings are binding a custom element's representation (shadow DOM template) to live data, and sending data to other elements or components [16].

Data binding helps provide separation or decoupling between the user interface or *view* and the underlying data *model*, a design pattern that is commonly known as Model-View-Controller (MVC). A short example can be found in Listing 2.3 and a longer one in section 4.4.1.

Although Polymer does support 2-way bindings, experience has shown that when exchanging data with external components, 1-way (top-down) bindings are easier to reason about and should be preferred. Bound variables should flow in one direction, generally parent to child as mentioned above. So how should a child communicate changes back to the parent? The parent (or any other interested party) can register a DOM mutation observer to be notified when the child value changes, or the child can fire an *event* that the parent can respond to.

As mentioned in section 2.3.1, mutation observers are a standard way to register a callback to be run when some area of the page (DOM) changes. A callback in this case is a function that reacts to a change in some other part of the system. This doesn't necessarily have to be tied to a specific variable like with data-bound templates and can reflect any change in any DOM attribute for standard *and* custom elements.

3.8 Deployment

Speakur is designed to be easy to deploy. Adding a discussion forum to a blog or web app is as simple as altering the page to:

1. load Polymer's Web Components polyfill,
2. import the `<speakur-discussion>` element,

3. then place `<speakur-discussion>` on the page at the desired location.

Because steps 1 and 2 above can load these resources from an external service or content delivery network, Speakur does not require installing any software to the web host serving the blog/app. The HTML, CSS, and JavaScript files, plus associated resources like images and JSON language files, can all be loaded from a different web host. This is known as cross-origin resource sharing, or CORS. Speakur avoids a common security restriction associated with CORS known as Same-Origin Policy, which is used to prevent a class of attacks called Cross-site Request Forgery (CSRF) [13]. It does this by relying entirely on Firebase for live data, which in turn uses WebSockets, which are not subject to Same-Origin Policy restrictions on asynchronous JavaScript HTTP requests (also known as AJAX or `XMLHttpRequest`). This allows Speakur to be used on a site without that site having installed a copy of its files or directly serving them to clients.

3.8.1 Dependencies

Like most applications, Speakur relies heavily on other libraries and frameworks to implement some of its underlying behaviors. Some of these libraries in turn depend on other libraries. Collectively, all of the 3rd party modules required to run Speakur are called *dependencies*. For example, safe HTML rendering of user-supplied Markdown-style text is handled by the excellent Marked library [48]. See Appendix 2 for the full list of Speakur's direct dependencies.

Taking advantage of the de-duping feature of HTML Imports requires that you name the path to your resources and dependencies in a consistent fashion. In other words, if component A and component B both require component Z, both A and B must

use the same URL when requesting (importing) component Z. Furthermore, each of Z's dependencies must also follow this same pattern when importing their own dependencies in order to fully realize de-duping. Therefore Speakur follows the recommendation of the Polymer framework and uses Bower to manage dependencies [8]. Speakur only has to list which packages it needs, either by name or their Git repository address, and Bower handles downloading these to a managed component directory alongside Speakur itself.

3.8.2 Vulcanize

One possible 'problem' with adopting Web Components architecture is that it encourages the partition of a large application into numerous smaller components or files. Of course, this is really a *feature* designed to enhance code organization, encapsulation and productivity, but it has one important side effect: under the HTTP 1.1 protocol that powers the web, each small component requires a separate request which degrades page loading times. Techniques like `Keep-Alive` headers can help, and the forthcoming HTTP 2.0 will address this question more comprehensively.

In the meantime, the Polymer project provides a tool called Vulcanize which concatenates and minifies your web components into a single file. Concatenation puts all of the required text resources (HTML, CSS, and JS) into a single[3] file to reduce the number of HTTP requests. Minification removes all comments, whitespace and non-essential elements from the code to compact it and reduce transfer times. Whether using Vulcan-

[3]Except when Content Security Policy (CSP) requires the enforced separation of HTML, CSS, and JS. In this case, three files are used.

ize results in an overall faster site ultimately depends on a number of factors including how and when the components are used. For that reason, empirical testing is needed to know whether it should be used in production on a real site [20]. Table 4.3 illustrates how Vulcanize helps improve loading performance in three browsers.

Chapter 4

Implementation

4.1 Overview

Speakur is an HTML5 web application that uses the Polymer framework's Web Components architecture. It is primarily a client side application with no dedicated server component other than Firebase. It is delivered as a single Custom Element, `<speakur-discussion>`. You import the tag with `<link rel=import href=...>` to make it available to place on the page. This element has a data-bound `<template>` that provides the element's visual representation (shadow DOM). Actually, a custom element's representation, or logical DOM, may consist of both encapsulated shadow DOM as well as light DOM that is supplied by the *user* of the custom element and *projected* into the shadow DOM as discussed in section 2.2.2. In Speakur's case, the discussion forum is self-contained in terms of content; light DOM is not needed or used in the top-level element. Speakur's options or parameters are controlled with attributes:

```
<speakur-discussion
  firebaseLocation="https://speakur-demo.firebaseio.com"
  xtitle="This is the thread's (initial) title."
  href="demo1"
  initiallyOpen="true"
  allowAnonymous="true">
</speakur-discussion>
```

Listing 4.1: Using HTML attributes to set Speakur options

4.2 Layout and Structure

Structurally, Speakur consists of JavaScript, HTML, and CSS files, along with a few other resource types like images and JSON language files. All of the these internal resources and components are hidden from consumers who only have to import and place the main `<speakur-discussion>` element as discussed in section 4.8 below. The rest of the components are brought in by imports inside `<speakur-discussion>`.

Internally, Speakur is composed of a number of sub-components, a partial list of which is found in Table 4.1. These components are listed in three broad categories: structural containers, logical services, and user interface elements. The structural elements provide a container and abstraction layer for a group of related child elements. For example, `<speakur-card>` provides a Material Design-styled generic 'card' consisting of a styled header, body, and footer. These containers may have certain visible elements such as lines around their border, but most of their representation comes from the components inside.

The logical service elements do not have any visual representation at all. They perform data-related services, provide purely abstract APIs, and wrap external libraries. For example, the Firebase client library is a general purpose JavaScript library, and is not specifically adapted to Web Components or custom elements. Therefore it is often useful to provide a custom element wrapper around a non-WC-aware library. For this reason the Polymer project maintains `<firebase-element>` as a wrapper around the main Firebase library [17]. Within Speakur, each component gets its data either directly from its own internal `<firebase-element>`, or else from a data binding from a parent element that it got from its own `<firebase-element>`.

Name	Function
Structural containers	
`<speakur-discussion>`	Top-level public component.
`<speakur-thread-view>`	Container for the comments displayed in an entire thread.
`<speakur-card>`	Provides a Material Design 'card' container.
`<speakur-post-set>`	A container for a list of posts such as replies to a specific post.
Logical services	
`<firebase-element>`	Web Component wrapper around Firebase.
`<speakur-base>`	Base-class for most other Speakur components.
`<speakur-i18next>`	WC wrapper around i18next library.
`<speakur-profile>`	Users' session data and preferences.
`<speakur-post-vote>`	Controller for voting posts up or down.
User Interface	
`<speakur-compose>`	Composing replies and posts.
`<speakur-post>`	Displays a single user post/comment.
`<speakur-login-button>`	Login/logout button and dropdown.
`<speakur-theme>`	Base class for all themes, mainly CSS rules.
`<speakur-theme-blue>`	A specific theme.
`<speakur-dialog-profile>`	Dialog to edit user preferences.
`<speakur-lang-select>`	Dropdown to choose the UI language.

Table 4.1: Partial list of Speakur's internal components.

Figure 4.1 shows a partial listing of the internal component files. One of these is abstraction layer for an internationalization library. Following the `<firebase-element>` example, I created a simple custom element wrapper around the **i18next** JavaScript library [12]. This wrapper is called `<speakur-i18next>`. It tells i18next where to find Speakur's translation files and creates a Polymer *filter* to perform translations from data-bound templates. See section 4.6 below for details. Logical elements are also used to provide abstract 'controllers' (in the MVC sense), facades, pseudo-OOP base classes and other design patterns.

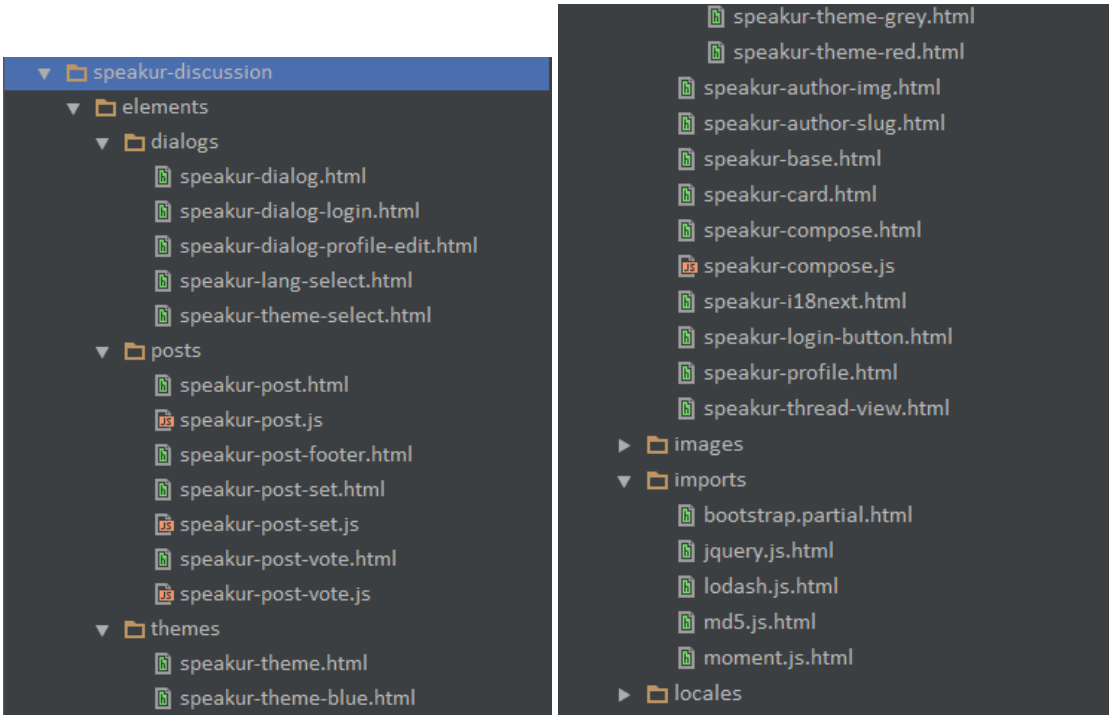

Figure 4.1: Internal elements file layout (partial).

In addition to structural and logical elements, Speakur has a number of user interface (UI) custom elements to do things like display an individual user post or show the

post composition editor. Elements are also reused extensively in different contexts; for example `<speakur-post>` is used for a 'live editor preview' inside `<speakur-compose>` as well as for showing posts in the main `<speakur-thread-view>.`

Some of Speakur's internal components like `<speakur-theme>` are also 'base classes' for other elements in a way that is similar but not identical to classes in object oriented programming (OOP)[1]. Although composition or *has-a* relationships are generally preferred, inheritance or *is-a* relationships can make sense when you truly want to be able to substitute one instance for another interchangeably.

4.3 Database Design

Speakur uses Firebase for NoSQL-style database storage. Traditional relational databases (SQL) have emphasized data normalization – the removal of informational redundancies from databases such that a given *fact* is only recorded in a single place. While this helps ensure data integrity, it can adversely affect query time and scalability as additional joins are required to bring in the necessary facts. One of the defining characteristics of the NoSQL movement is denormalization — that is, the principle of eliminating redundancy is sacrificed in order to gain performance [66]. Facts can be defined in more than one place to speed up queries.

For example, some information about a post's author such as their public username, id, and photo url is saved inside each post they write so that it's not necessary to

[1]Note that JavaScript itself is *prototype*-based and does not have traditional classes or inheritance. Similar behavior is obtained by using regular object instances (instead of classes) as the prototype for instantiating new objects.

Table Name	Function	Key Structure
`admins`	Administrator authorizations	`$uid`
`posts`	User posts	`$parent->$child`
`postvotes`	Public vote counts for posts	`$parent->$child`
`profile`	User data and preferences	`$uid`
`threads`	Discussion thread definitions	`$threadId`
`uservotes`	User votes for posts	`$uid->$parent->$child`

Table 4.2: Speakur's database tables.

look up this information separately just to display the post. One consequence of this is that changing your public name doesn't retroactively change the name on all of your previous posts, just new posts going forward. Certainly, if that name updating was deemed necessary per business requirements it could be accomplished.

The logical interface presented by Firebase resembles a single JavaScript object which can contain other objects and lists nested up to 32 levels deep [9]. A JavaScript object is fundamentally a `key->value` mapping, also known as a hash map or (in Python) as a dictionary. Although this nested object structure doesn't correspond exactly to traditional relational database (SQL) tables, it is convenient to refer to the top (first) level of nested objects as "tables". The leaf node objects are "records" with data fields, and the intermediate objects are keys within the table. Like most NoSQL databases, Firebase is forgiving with regard to structure and fields; a given record is not guaranteed to follow any particular structure unless this is required by the security and validation rules.

The key structure column in Table 4.2 signifies the meaning of the keys underneath a table. For example, the main `posts` table has a `$parent->$child` key structure,

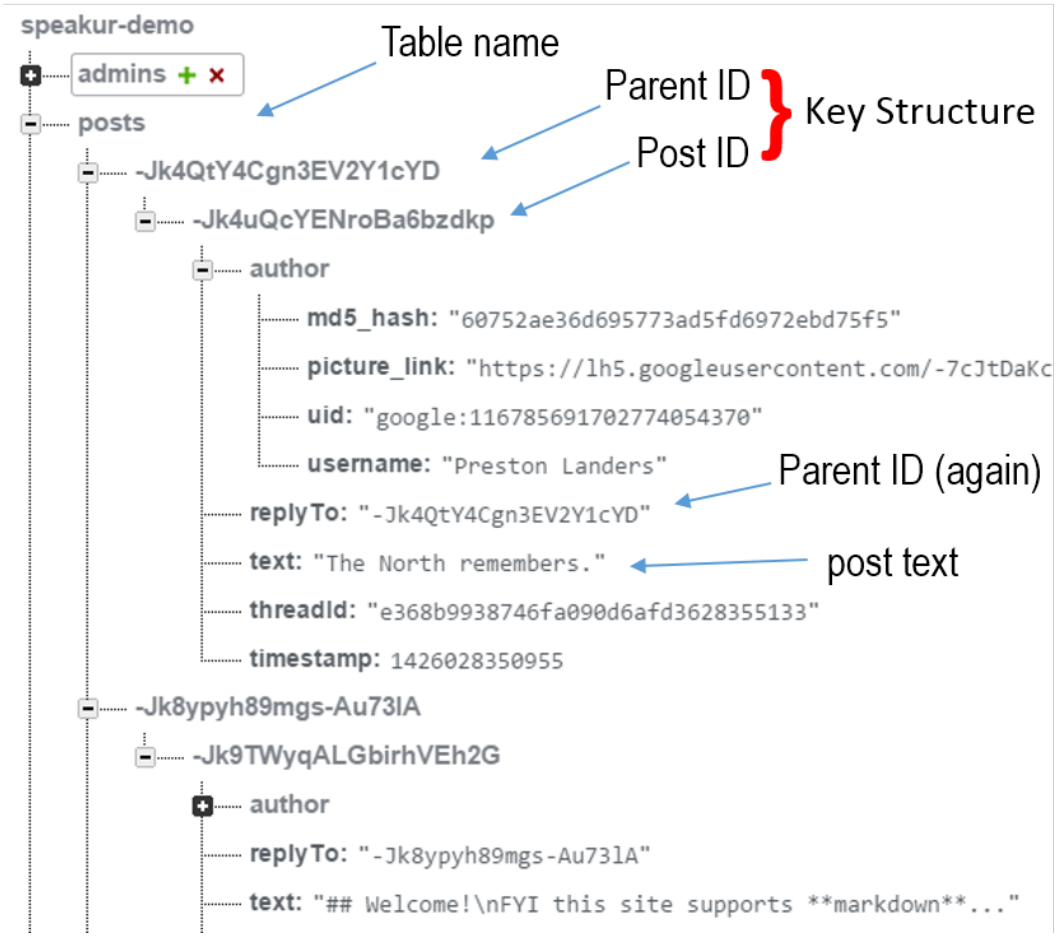

Figure 4.2: Structure of the `posts` table.

which means that the first layer of keys beneath `posts` are post *parent* ids, meaning the id of the post that this post is a reply to, or the thread id in the case of a 'top-level' post. The next nested object down is the *child* id, or the id of the actual post itself, and the keys under that are the 'fields' of the table, such as the `text` and `replyTo` values. Figure 4.2 shows Firebase's administrative view of the database where the `posts` table is expanded, showing the parent and child id layers, and the actual post nested within. Deeply nested

data structures are not good for scalability; relatively flat tables are best. The key structure will become important when we apply security rules in section 4.5.

In addition to denormalizing the data as necessary, sometimes it is convenient for security reasons to keep related bits of data in separate tables in order to apply different security rules. For instance, we want users to be able to modify a post's vote count without being able to modify the post itself. The nested and denormalized structure of the tables helps ensure that we only fetch what we need, when we need it. The flip side is that we need to be aware of these data duplications and update them as necessary, which can be difficult in a more sophisticated application. But in the post vote count example, even if a malicious user sets a false count on the globally-visible value, we can always recalculate the true count later using the individual user vote records.

4.4 Component Synchronization

Speakur consists of a number of Polymer components which break down the problem of presenting a discussion forum into smaller tasks. Some of these components are listed in Table 4.1. They form a hierarchy with `<speakur-discussion>` at the top (root). Figure 4.3 illustrates this, with many components omitted for brevity.

The general data flow is that bindings carry information *down* from parent to child, and changes flow *up* in the form of events and observed mutations. Two-way bindings are possible in that a child element can modify a bound variable and the value would be reflected in the parent and vice-versa, but in some scenarios this makes it difficult to determine an 'authoritative' value or causes other circularity problems. For this reason, bindings should be used one-way across component lines, but two-way bindings are use-

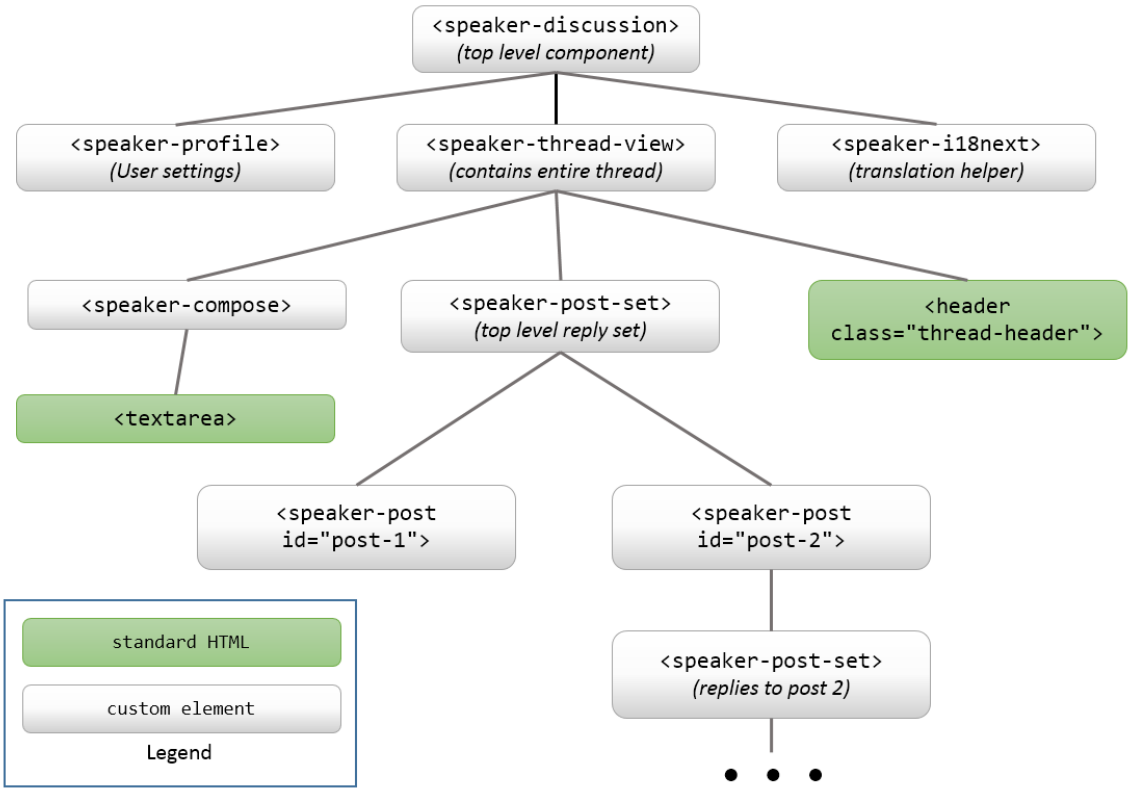

Figure 4.3: Speakur component hierarchy (partial).

ful within a component to update its 'data model' automatically from user input [16].

Each component keeps its own `<firebase-element>` to pull data from Firebase as needed, or in some cases the relevant object is passed in as a binding from a parent element that has its own `<firebase-element>`. This keeps database specific knowledge localized to each component, helping it "be composable". Speakur as a whole doesn't know or care what storage solution its individual components use, whether Firebase or another API.

4.4.1 Data Bindings and Events

From an MVC viewpoint, a Polymer element with instance variables *is* the 'model'. The element's data-bound `<template>` is the 'view', and the JavaScript code is the 'controller'. Data bindings carry data information down from parent to child while DOM events propagate upwards.

Using `<firebase-element>` isolates each component from direct interaction with the general purpose Firebase JavaScript library and ties the database directly into the Polymer data binding system. Persisting changes is incredibly easy; writing a new value to a variable not only saves it to the database but pushes the change to all other 'subscribers' — whoever is viewing that post or other object at the time. The `data=` attribute on `<firebase-element>` allows binding a variable in a component to a remote database object:

```
<!-- from speakur-post.html -->
<firebase-element
   data="{{ post }}"
   location="{{ fbLocation(firebaseLocation, 'posts', parentId,
      postId) }}">
</firebase-element>
```

Listing 4.2: Binding the `post` variable to a database record.

In the example above, the data retrieved by the `<firebase-element>` is bound by the `{{ }}` operators to the component's `post` variable. Any changes to this database object, whether locally or remotely initiated, will be reflected in the `post` variable and any associated data-bound `<template>`s, and vice-versa. Note that the *source* URL of this `<firebase-element>` is itself a `{{ }}` bound variable (`location`) which finds the location with the `fbLocation` method. Any changes in the variables `firebaseLocation`,

`parentId` or `postId` will cause `fbLocation` to be recomputed, which could in turn cause `post` to pull from a different location.

DOM events carry state changes up from children to elements above it. Polymer provides the `fire()` method to trigger a named event with extra data attached. This is a convenience wrapper since it can be done with pure JavaScript. An example of using `fire()` can be found in the language selection dialog from Figure 3.1. The language selection UI is handled in the `<speakur-lang-select>` component, which is only concerned with presenting the language chooser and not with handling the actual localization, reinforcing separation of concerns. When the user selects a new language it fires an event that bubbles up to the top level `<speakur-discussion>` element:

```
// from speakur-lang-select.js
domReady: function () {
    // listen for core-select event fired by the dropdown menu
    this.$.menu.addEventListener('core-select', function (e) {
        if (e.detail.isSelected) {
            var oldLocale = this.locale;
            var newLocale = e.detail.item.id;

            // Tell container(s) the language selection changed
            this.fire('speakur-locale-change',
               {oldLocale: oldLocale, newLocale: newLocale});
        }
    });
}
```

Listing 4.3: Firing a language change event.

A handler in the top level container (`<speakur-discussion>`) responds to this named event (`speakur-locale-change`) and tells the internationalization component to update the language. This in turn sets the **`lc`** (locale) property in the base element. This property is then used in translation expressions as discussed in section 4.6 below.

4.4.2 Data-Bound Templates

Speakur translates internal variables (the model) into visible output with the use of data-bound templates. This is similar in effect to the model-view-controller design pattern. As mentioned above, the custom element itself is the 'model' and its `<template>` is the view. This allows embedding variables directly in the HTML representation of a component as shown in Listing 2.3 and here:

```
<!-- from speakur-post-footer.html -->
<div on-click="{{editPost}}" hidden?=
    "{{!canEditPost(post, globals.profile, globals.isAdmin)}}">
  {{ 'Edit' | $$(lc) }}
</div>
```

Listing 4.4: User interface control with data bindings (edit post link).

There are three data bindings with `{{ }}` in Listing 4.4:

1. The `hidden?=` attribute (provided by Polymer) binds to `canEditPost` to determine if the Edit link should be shown under a post. In other words, is this user allowed to edit this post? If not, hide this Edit `<div>`.

2. The `<div>` binds its click event to `editPost` so it functions as a link. The `editPost` method of this component will be called when the Edit link is clicked.

3. The inner content of the `<div>` (its light DOM) is bound to a Polymer filter expression that translates the word "Edit" to the current language.

The `{{ }}` operators are not limited to binding to simple variables. Polymer supports a rich expression language within binding operators. That said, keeping `<tem-`

`plate>` expressions simple and moving more complex logic to methods in the JavaScript controller helps uphold separation of concerns between controller and view.

4.5 Security

Because Firebase is Speakur's only 'server', most of its security rests in Firebase's authentication and authorization rules. Obviously the user interface should be crafted such that undesirable actions are not presented as options for the user, but one also has to consider malicious users stepping outside of the allowed interface and/or directly accessing the REST API. Authentication ensures that we know the identity of anyone making database changes, and the Firebase security rules enforce authorization requirements.

4.5.1 Authentication

Speakur currently supports signing in (authenticating) users with Google and Facebook profiles. This avoids the need to create a dedicated user registration and password system by using single sign-on with popular Internet identity providers. In this scenario, Google or Facebook is an OAuth identity provider (IdP), and Firebase (standing in for Speakur) is a service provider (SP). The identity provider gives the service provider a limited selection of personal information, generally their name, profile picture, and email address. The user authenticates directly to the chosen IdP (e.g. Facebook) and their password is never sent to Firebase or Speakur. Future versions may support additional identity providers like GitHub and Twitter as well as Firebase's own simple user registration system.

Inside Speakur, authentication is handled by the `<firebase-login>` element,

provided by `<firebase-element>` [17]. A binding is set up between that and the `user` variable in `<speakur-discussion>` so that when user authentication is completed, the `user` variable contains an object with user properties like their authentication ID and other data. The authentication session is automatically used by any `<firebase-element>` anywhere on the page provided their `location=` points to the same database URL.

4.5.2 Firebase Security Policy Rules

Authorization to perform any given action in the Speakur database is determined by consulting the nested rules in the JSON-formatted security object, whose documentation can be found at [10]. These rules can be entered directly into the Firebase administrative console and a reference copy can be found in the Speakur source code repository under the file `security-rules.json` [56].

The structure of the security rule object mirrors the nested hierarchy of the Firebase database itself. Each top level key maps to rules for a particular database "table" like `posts`. These fall into three categories: read, write, and validation rules. These are each structured as boolean expressions (predicates) that determine whether the given action is allowed.

1. `.read` rules determine whether a given record can be retrieved by the current user.
2. `.write` rules authorize the creation, modification and deletion of a record.
3. `.validate` rules check whether a record is correctly *structured*.

Write rules determine whether a given create, modify or delete is allowed. Validate rules are run after a write rule has passed in order to check whether the new version

is correctly structured per business rules or requirements. Each expression has access to several pre-defined variables including snapshots of the database as it exists both before and after the change would be applied. For example, Listing 3.2 shows rules for the `posts` table. The `.write` rule asserts that anyone can create a new post, but modifying an existing post requires that you (`auth.uid`) are either the post owner or an administrator. The `.validate` rule ensures that all posts contain at least 4 keys: `threadId`, `text`, `author`, and `timestamp`. More sophisticated rules and validations are possible, and rules can be nested, but it can be unwieldy to fit a large set of rules into a single predicate expression, making them more difficult to understand and maintain.

4.6 Internationalization

The process of customizing an application for local languages and customs is called internationalization (i18n) and localiation (l10n)[2]. Speakur offers user interface translations for 15 languages. The user can select a new language from the Profile dialog and the interface updates instantly. The actual substitution of strings using the currently selected language is handled by the i18next JavaScript library, which is abstracted behind the custom element wrapper `<speakur-i18next>` [12]. This wrapper provides a Polymer filter named `$$` which is used in `{{ }}` expressions in data-bound templates to translate user-visible UI strings. Of course, the users' posts themselves are not translated; just the application's interface strings.

The i18next library loads translation strings out of JSON files, one for each sup-

[2] *Internationalization* refers to the process of **enabling** an app to be localized, and localization refers to adapting it to a **specific** locale.

ported language. The current translations were obtained by taking the English language file and passing it through the Microsoft Translation API using 'Eurgh!', a Python package that I wrote for another project [53]. The i18next library includes the ability to handle variable interpolation insertion points and pluralization rules which are important for producing grammatically correct text. Here is a short excerpt from the English file followed by the Russian version:

```
// from speakur-en.json
{
  "View Comments": null,
  "replies_count": "__count__ reply",
  "replies_count_plural": "__count__ replies"
}
// from speakur-ru.json
{
  "View Comments": "Посмотреть комментарии",
  "replies_count": "ответ __count__",
  "replies_count_plural": "__count__ ответы"
}
```

Listing 4.5: String resources for internationalization.

This example shows that with the `'View Comments'` key, the corresponding value can be null if the key itself can serve as the text. The other two keys illustrate variable interpolation and pluralization logic where different grammatical forms can be selected based on a numeric variable, in this case the number of replies that a post has. These translation strings would be used in a template like:

```
<!-- from speakur-post.html -->
<div class='post-replies-count'>
  {{ 'replies' | $$({count: replyCount}, lc) }}
</div>
```

Listing 4.6: Internationalizing a data-bound template.

`i18next` sees that a `count` variable is provided to `$$` and looks for that key (`replies`) with either a `_count` or a `_count_plural` suffix depending on the cardinality of `count`. In this expression the `lc` variable (the second parameter to the `$$` call) reflects the user's current language setting, such as `lc=en` for English or `lc=ru` for Russian. Interestingly, the `lc` variable is *not* required by the `$$` filter because it already knows the current locale. Instead, `lc` is included in the expression merely to register a value dependency with Polymer such that when the locale is changed elsewhere, all dependent expressions are recalculated, or in this case, the strings are re-translated to the current locale[3]. In this case it would output something like "1 reply" or "2 replies" as needed. Otherwise Polymer wouldn't bother to recalculate this expression when the locale (`lc`) changed because none of its direct clauses had acquired different values. Adding `lc` to this expression ensures that the text is updated when the language changes.

Another powerful internationalization feature is provided by the moment.js library, a general purpose date/time library that also provides localizing services. The date/time localizations are not provided by Speakur but are included in moment.js itself. In addition to formatting dates and times (like the time of a post) according to local conventions, it also provides a `fromNow` function that outputs a localized "ago" string — the amount of time elapsed since a certain point such as "10 seconds ago" or "3 weeks ago". Speakur uses this in post headers to show how long ago they were posted, and ties them to a timer-based value dependency such that the "ago time" updates automatically on the page as time passes. For example, immediately after posting a new reply, its header will say "posted a few moments ago." After a minute it will say "posted a minute ago", and so

[3] Polymer expression dependencies are implemented internally with DOM mutation observers [16].

on in idiomatic fashion appropriate to the current language. This text has a tooltip which gives the localized full date such as “Wednesday, March 11th 2015 7:53 pm”.

4.7 Responsive Design

As mentioned in section 3.3, Speakur uses several techniques to achieve a design that automatically responds to different devices and screen sizes, including mobile phones. The three main responsive design techniques used in Speakur are:

1. Using CSS `@media` rules to apply different styles such as sizes and margins to different screen sizes.

2. Using CSS Flexible boxes (a.k.a ‘flex’) layout with `wrap` options to create structures like toolbars while allowing them to gracefully rearrange on smaller screens.

3. Using a JavaScript variable (`smallScreen`) to control the behavior of data-bound templates and provide a different experience on smaller devices.

In Speakur’s style sheets, a small screen is treated as the base case and larger tablet or desktop-style devices are given extra rules with `@media`. For example, the following rule gives the main Speakur container more padding on large screen devices:

```
<!-- from speakur-discussion.css -->
@media only screen and (min-device-width : 800px) {
  :host .speakur-body-container {
    margin: 4px;
    padding: 8px;
  }
}
```

Listing 4.7: CSS @media rule for large screen devices.

CSS Flexible boxes are little too complex to explain here in full, but the gist is that `div`s, and nearly any other element, can be given 'flex layout' attributes that provide guidance to browsers about how to align the components inside and adjust for available space. Polymer provides convenient layout shortcuts for the standard CSS attributes [18].

4.8 Publishing and Deployment

Speakur is published to `github.io`, a static web host and content delivery network for GitHub projects. This allows users to import the component without hosting it themselves. Web authors can also host Speakur's files on their own server if desired. This makes it very easy to deploy Speakur in a wide variety of system architectures. Speakur is provided in two versions; the 'development' version where each resource is provided as a separate, raw, non-minified file, and a 'production' or Vulcanized version where all necessary resources are combined into a single file for faster loading speeds [20]. In a relatively small application like Speakur the difference is not huge, especially on Google Chrome which already supports Web Components natively, because the HTML component transfer times are largely dwarfed by the cost of loading the actual discussion from the database. Future protocol enhancements like HTTP 2 may eliminate this discrepancy. That said, vulcanization does offer current benefits, as shown in Table 4.3 which compares the time (in milliseconds) to initially display the Speakur component and then fully load a thread with 25 comments.

State	Raw	Vulcanized
Google Chrome		
visible	2200	1150
complete	4200	2900
Mozilla Firefox		
visible	7400	6050
complete	12000	10500
Opera		
visible	2800	1100
complete	7000	2800

Table 4.3: Loading time (in milliseconds) in three browsers.

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

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 [19]:

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

- Firebase database service and client library [9]
- Marked library [48]
- highlight.js library
- i18next library [12]
- 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: http://bower.io/search.

[9] Firebase contributors. *Firebase: Understanding Data*. Mar. 2015. URL: https://www.firebase.com/docs/web/guide/understanding-data.html.

[10] Firebase contributors. *Firebase: Understanding Security*. Feb. 2015. URL: https://www.firebase.com/docs/security/guide/understanding-security.html.

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

[12] i18next contributors. *i18next library*. Feb. 2015. URL: http://i18next.com/.

[13] Mozilla contributors. *Same Origin Policy*. Feb. 2015. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

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

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

[16] Polymer contributors. *Data binding overview*. Feb. 2015. URL: https://www.polymer-project.org/0.5/docs/polymer/databinding.html.

[17] Polymer contributors. *Firebase Element*. Mar. 2015. URL: https://github.com/polymer/firebase-element/.

[18] Polymer contributors. *Layout Attributes*. Mar. 2015. URL: https://www.polymer-project.org/0.5/docs/polymer/layout-attrs.html.

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

[20] Polymer contributors. *Vulcanize*. Mar. 2015. URL: https://github.com/polymer/vulcanize.

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

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

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

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

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

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

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

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

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

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

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

[32] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. ISBN: 9780596554873.

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

[34] 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.

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

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

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

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

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

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

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

[42] 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.

[43] 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.

[44] 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.

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

[46] 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/.

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

[48] Christopher Jeffrey. *Marked library*. Dec. 2014. URL: https://github.com/chjj/marked.

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

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

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

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

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

[54] 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.

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

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

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

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

[59] 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.

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

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

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

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

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

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

[66] Pramod J Sadalage and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012. ISBN: 978-0321826626.

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

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

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

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

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

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

[73] 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.